

# Grundlagen der Programmierung

## VL17: „Advanced Topics“

Prof. Dr. Samuel Kounev  
Jóakim von Kistowski  
Norbert Schmitt



- Enums
- Container & Collections
- Generics
- Javadoc
- Innere Klassen
- Anonyme Klassen
- Exceptions





Auflistungen

# ENUMS

# Aufgelistete Werte

- Manchmal möchte man eine Variable definieren, die nur eine gewisse Menge an aufgelisteten Werten halten soll
- Beispiele:  
`dayOfWeek: SUNDAY, MONDAY, TUESDAY, ...`  
`month: JAN, FEB, MAR, APR, ...`  
`gender: MALE, FEMALE`  
`title: MR, MRS, MS, DR, PROF`
- Erinnerung: Konstanten werden in ALLCAPS geschrieben
- Was ist der eigentliche Typ dieser Konstanten?

- In der Vergangenheit wurden solche Auflistungen mit Integers realisiert:

```
public final int SPRING = 0;  
public final int SUMMER = 1;  
public final int FALL = 2;  
public final int WINTER = 3;
```

- Nervig und fehleranfällig

```
int season = WINTER; ...; season = season + 1;  
int now = WINTER; ... ; month = now;
```

- Besser: enum-Typ nutzen

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

- Ein **enum** ist eigentlich ein neuer Klassentyp
  - Kann auch optional als innere Klasse (s. später) deklariert werden
  - Liefert Typsicherheitsprüfungen zur *Compile-Zeit*
    - Jeder deklarierte Wert ist eine eigene Instanz der enum-Klasse
    - Enums sind implizit **public**, **static** und **final**
    - Enums können mit **equals** oder **==** verglichen werden
  - Enums erben von **java.lang.Enum** und implementieren **java.lang.Comparable** (d.h., Enums sind sortierbar)
  - Enums überschreiben **toString()** and bieten **valueOf()**
  - Beispiele:

```
Season season = Season.WINTER;  
System.out.println(season);           // schreibt WINTER  
season = Season.valueOf("SPRING");    // setzt season zu Season.SPRING
```

```
public void tellItLikeItIs(DayOfWeek day) {  
    switch (day) {  
        case MONDAY:  
            System.out.println("Mondays are bad.");  
            break;  
        case FRIDAY:  
            System.out.println("Fridays are better.");  
            break;  
        case SATURDAY:  
        case SUNDAY:  
            System.out.println("Weekends are best.");  
            break;  
        default:  
            System.out.println("Midweek days are so-so.");  
            break;  
    }  
}
```

Quelle: <http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>

- Enums bieten Typsicherheit zur Compile-Zeit
  - `int` enums bieten überhaupt keine Typsicherheit, z.B. `int season = 43;`
- Enums haben lesbare Namen für ihre Zustände
  - Bei `int` enums muss man die Zugehörigkeit der Konstanten mit Präfixen darstellen (Beispiel: `seasonWINTER` oder `S_WINTER`).
- Enums sind robust
  - Hinzufügen, Entfernen oder Umordnen von Konstanten erfordert Rekompilierung
- Weil Enums Objekte sind, können sie auch zu Collections (s. später) hinzugefügt werden
- Weil Enum-Typen Klassen sind, können sie auch Attribute und Methoden enthalten



- **String toString()**
  - gibt den Namen des Enum-Objektes als Konstante zurück
- **boolean equals(Object other)**
  - vergleicht die enum-Konstanten
- **int compareTo(E o)**
  - gibt eine Ordnung zum Sortieren zurück  
(-1 für kleiner, 0 für gleich, 1 für größer)
- **static enum-type valueOf(String s)**
  - gibt das Enum-Objekt mit dem Namen **s** zurück
- **static enum-type[] values()**
  - gibt ein Array der Konstanten Enum-Objekte zurück

- Konstruktoren von `enums` weichen von der Standard-Darstellung ab
- Jeder aufgelistete Name im `Enum` ist ein eigener Konstruktor-Aufruf

- Beispiel:

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

Erstellt 4 Objekte mit dem Standard-Konstruktor

- Beispiel 2

```
public enum Coin {  
    private final int value;  
    Coin(int value) { this.value = value; }  
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);  
}
```

- Enum- Konstruktoren sind nur im Enum selbst verfügbar
  - Enum wird als unveränderlich angenommen

- C Enums werden ähnlich wie in Java deklariert
  - Sind aber Aliase für Integer-IDs

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};

int main(void) {
    colors_t mycolor;
    mycolor = blue;
    if (mycolor == green) mycolor = red;
}
```

- C++ erlaubt Enum-Klassen
  - Keine Integer Aliase; Zugriff auf Werte mit Scope Resolution Operator

```
enum class Colors {black, blue, green, cyan, red, purple, yellow, white};

int main(void) {
    Colors mycolor;
    mycolor = Colors::blue;
    if (mycolor == Colors::green) mycolor = Colors::red;
}
```



# CONTAINERS & COLLECTIONS

- Arrays
  - Eigenschaften
    - Lineare Folge fester Größe
    - `length`-Attribut
    - Kann primitive Datentypen enthalten oder Referenzen auf Objekte
    - Nur Objekte eines Typs
  - Vorteile
    - Sehr effizient
    - Typüberprüfung zur Compile-Zeit
  - Nachteile
    - Feste Größe
    - Nur ein Typ
    - Wenig Methoden

→ **Container**

- Ein Container ist ein Objekt das eine Sammlung von Referenzen auf andere Objekte enthält
- Ein Gruppierungsmechanismus mit komplexen Methoden zum Verwalten und Ändern der Sammlung
- Zwei unterschiedliche Konzepte von Containern

## 1. Collection

- Enthält einzelne Objekte
- z.B. in Java: Set, ArrayList, LinkedList

## 2. Map

- Enthält Schlüssel-Wert Paare (key-value pairs)
- Jeder Schlüssel wird auf einen Wert abgebildet
- Jeder Schlüssel darf nur einmal vorkommen
- z.B. in Java: HashMap, HashTable

- Eigenschaften
  - Können verschiedene Typen beinhalten
  - Keine feste Größe, automatische Größenanpassung
  - Enthalten nur Referenzen auf Objekte
  - Keine primitiven Typen
  - Package `java.util`
- Beispiele von Methoden
  - `add(Object o)`
  - `remove(int index)`
  - `size()`
  - `get(int index)`
  - `put(Object key, Object value) // für Map`

```
import java.util.ArrayList;
```

```
// Constructs an empty list with the specified  
// initial capacity:
```

```
ArrayList songs = new ArrayList(20);
```

```
// Add a Song:
```

```
songs.add(new Song());
```

```
// Set the artist:
```

```
// (the cast is needed, to make clear it is a Song,
```

```
// you could do movies inside ☺ but ...)
```

```
((Song) songs.get(0)).setArtist("Abba");
```

ArrayList.get(i) returns a reference to an Object. The type cast (Song) is needed to tell the compiler that it is actually an instance of the Song class. This way, we can then call the method setArtist of Song.



```
/**
 * This method returns a String description of all Songs
 * @return String - a String containing all Song names
 */

public String getAllSongs() {
    int i = 0;
    String result = "";

    while (i < songs.size()) {
        result = result + ((Song) songs.get(i)).toString() + "\n";
        i++;
    }

    if (i == 0)
        return ("Keine Lieder vorhanden!");
    else
        return result;
}
```

- Iteratoren: Objekte die die folgenden zwei Methoden bieten um einen Container zu durchlaufen

```
boolean hasNext()
```

```
// Returns true if the iteration has more elements
```

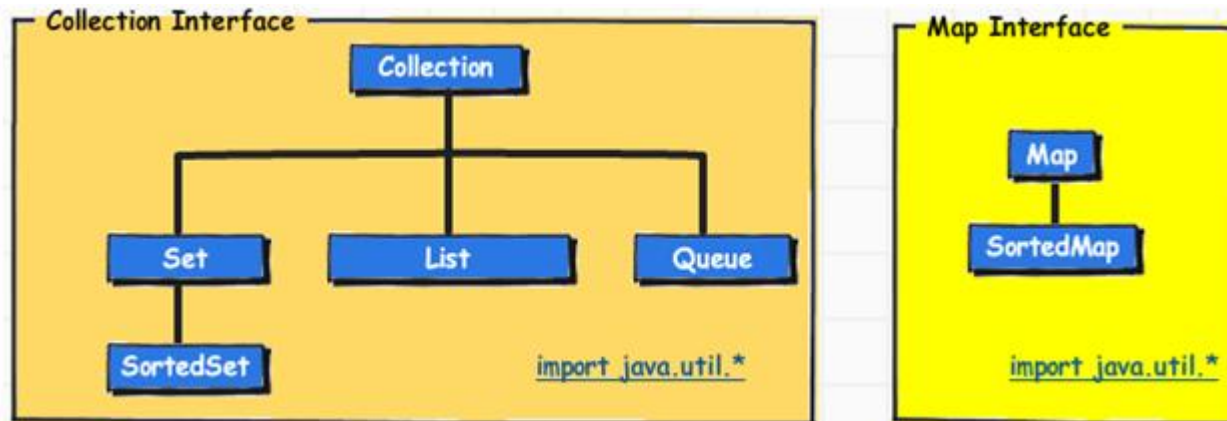
```
Object next()
```

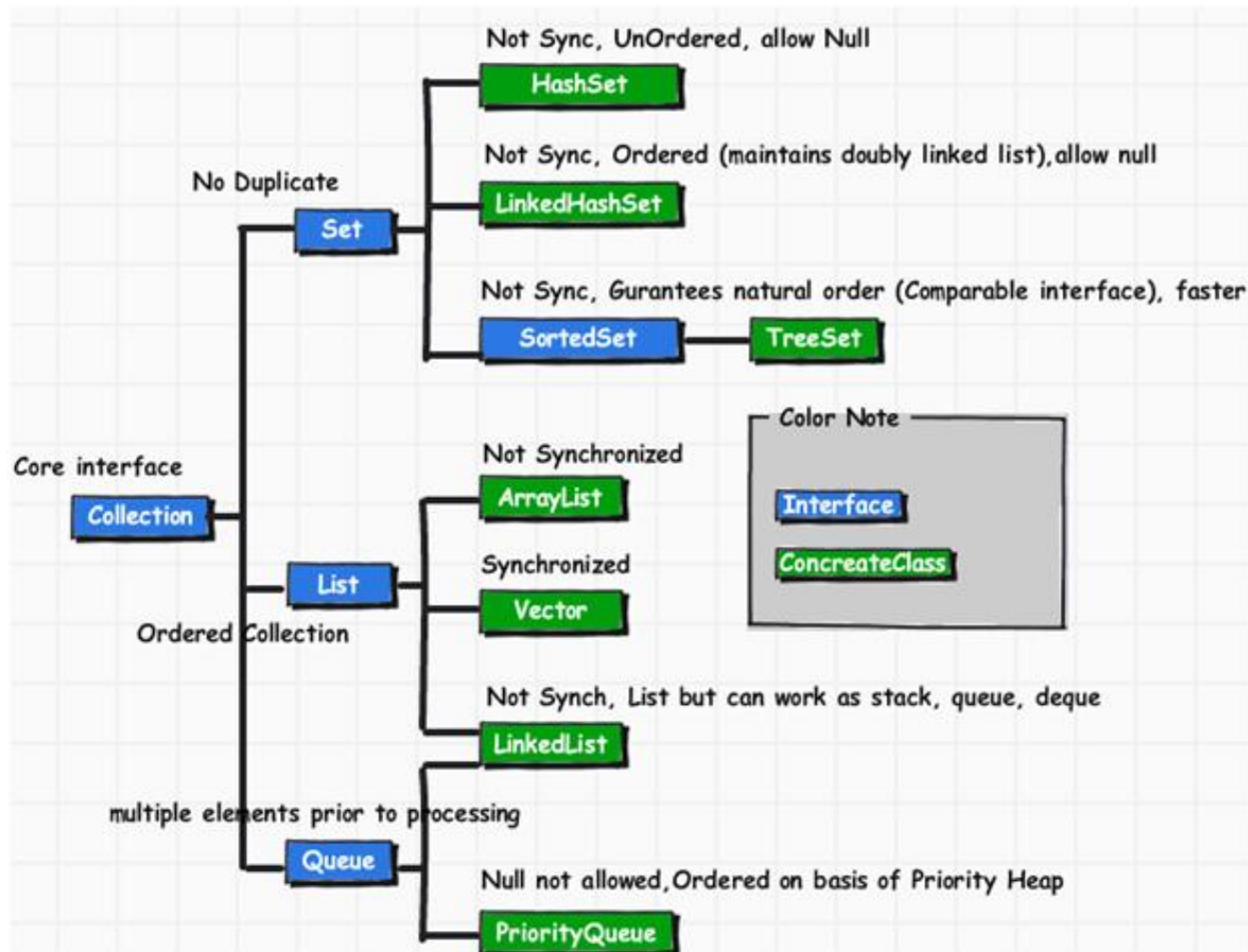
```
// Returns the next element in the iteration
```

- Wozu Iteratoren?
  - Höhere Abstraktion
  - Generischer Code
    - Änderung von z.B. `ArrayList` in `LinkedList` oder `Set` möglich
- Alle drei Klassen bieten `iterator()` Methode an

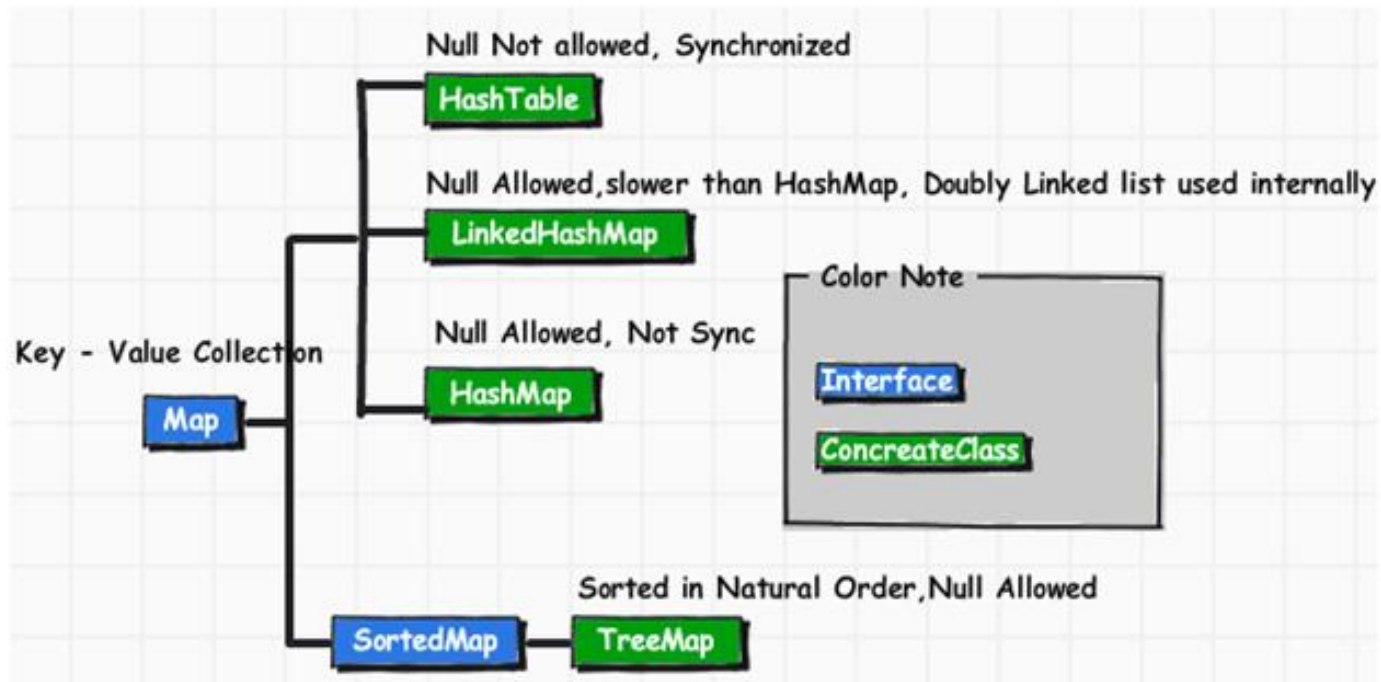
```
public String getAllSongs(ArrayList songs) {  
    String result = "";  
  
    Iterator e = songs.iterator();  
    while(e.hasNext())  
        result = result + ((Song) e.next()).toString() + "\n";  
  
    if (result.equals(""))  
        return ("Keine Lieder vorhanden");  
    else  
        return result;  
}
```

- Statt mit konkreten Klassen wird meistens mit Interfaces gearbeitet  
z.B. `List songs = new ArrayList(20);`  
`List` ist ein Interface in `java.util`  
`ArrayList` ist eine konkrete Klasse die das `List`-Interface implementiert  
`LinkedList` ist eine andere Klasse die das `List`-Interface implementiert
- Die Verwendung generischer Interfaces erleichtert den Austausch von den konkreten Klassen, z.B. von `ArrayList` zu `LinkedList`
- Wichtige Interfaces: `Collection`, `List`, `Set`, `Queue`, `Map`





Quelle: <http://www.jitendrazaa.com/blog/java/complete-java-collection-tutorial-for-the-beginner/>



Quelle: <http://www.jitendrazaa.com/blog/java/complete-java-collection-tutorial-for-the-beginner/>

- Nachteile der Standard-Container (d.h. ohne Verwendung von Generics - s. später)
  - Weniger effizient als Arrays
  - Keine Typüberprüfung; manuelle Typumwandlungen (casts) notwendig
- Bsp. für **unterschiedliche Performance** (Anzahl für die Implementierung benötigter elementarer Operationen wie addieren, subtrahieren,...)
  - klassisches Array → Get:172; Iteration:516; Insert: na; Remove: na
  - ArrayList → Get:281; Iteration:1375; Insert:328; Remove:30484
  - LinkedList → Get:5828; Iteration:1047; Insert:109; Remove:16
- Wichtige funktionale Unterschiede

	HashSet	LinkedHashSet	TreeSet	ArrayList	Vector	LinkedList	HashTable	LinkedHashMap	HashMap	TreeMap
Null	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓
Duplicate	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗
Sorted Result	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓
Retrieval order same as insertion	✗	✓	✗	✓	✓	✓	✗	✓	✗	✗

Quelle: <http://www.jitendrazaa.com/blog/java/complete-java-collection-tutorial-for-the-beginner/>

- C++ bietet viele Container-Klassen als Teil der Standard Template Library an
- Alle C++ Container sind Templates
  - Verwendung sehr ähnlich zu Java Generics (s. später)
- Beispielcontainer

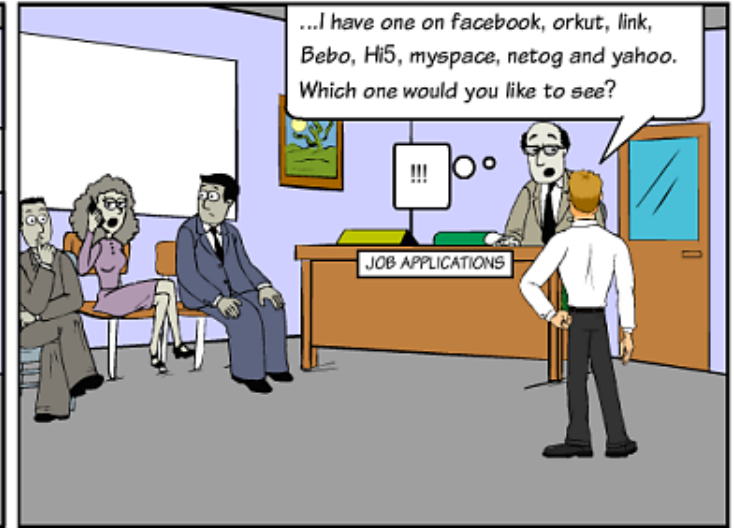
C++ Container	Java Collection	Beschreibung
Vector	ArrayList	Array mit veränderbarer Länge
List	LinkedList	Doppelt verkettete Liste
Set	SortedSet	Sortierte und iterierbare Menge in der jedes Element maximal ein mal vorkommt
Map	SortedMap	Sortierte und iterierbare Tupelsammlung mit Schlüssel/Wert-Tupeln



HIGH PROFILED - BY MEERASAPRA



www.toondoo.com



Variable Typisierung


# GENERICIS

- **Hintergrund:** Alte Versionen der Java Collections (vor Java 1.5) waren *Objekt*-basiert und erforderten das Benutzen von hässlichen Typecasts
  - Typ der Elemente einer Collection kann nicht spezifiziert werden
  - Beim **Zugriff** braucht man immer **Typecast** zur spezifischen Klasse  
Beispiel: `((Song) songs.get(0)).setArtist("Abba");`
- **Java Generics**
  - Ermöglichen mehr Sicherheit und Lesbarkeit
  - Vor allem für allgemeine Datenstrukturen nützlich
- **Generische Programmierung** = Programmierung von Klassen und Methoden mit parametrisierten Typen

- Generische Klassen und Methoden sind ein Werkzeug zur Erstellung von wiederverwendbaren objektorientierten Typen und Bibliotheken
- Volles Verständnis erfordert Einarbeitung in Typ-Theorie
  - Insbesondere die Prinzipien der *Kovarianz* und *Kontravarianz*
- In dieser Vorlesung illustrieren wir das Prinzip hinter generischer Typisierung mit Hilfe von einigen Beispielen

- Beispiel einer **generischen Klasse**

```
class Paar<T> {  
    public T first;  
    public T second;  
    public Paar(T f, T s) { first = f; second = s; }  
    public Paar() { first = null; second = null; }  
}
```



Die Klasse Paar ist parametrisiert nach Typ T

- Die Klasse wird mit der Substitution des Typs T durch einen konkreten Typ instantiiert. Beispiel: **Paar<String>**
- Eine instantiierte generische Klasse ist wie eine ganz normale Klasse benutzbar (fast):

```
Paar<String> pair = new Paar<String>("1","2");
```

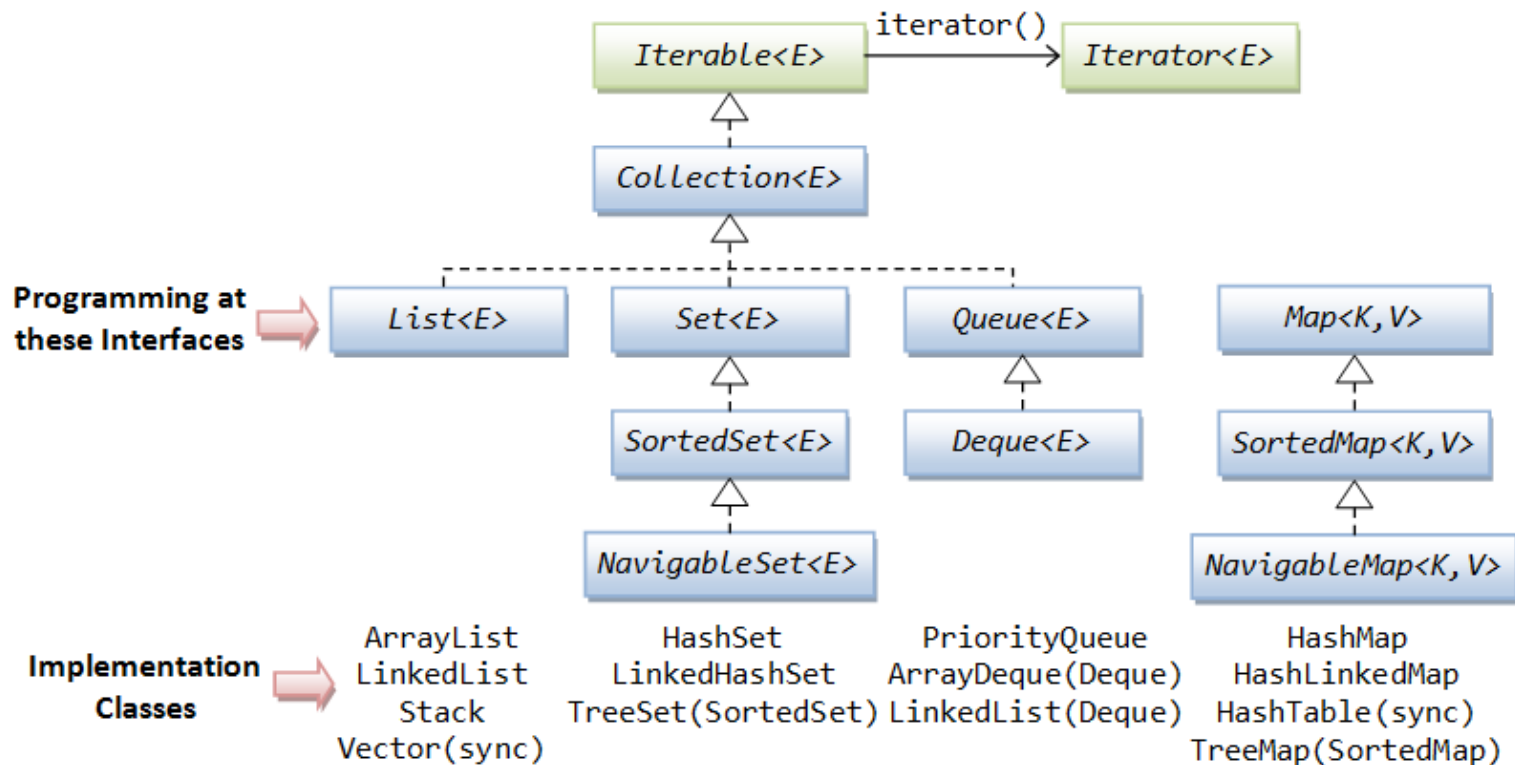
- Mehrere Typ-Parametrisierungen:

```
class Paar<T, U> {  
    public T first;  
    public U second;  
    public Paar (T x, U y) { first = x; second = y; }  
    public Paar () { first = null; second = null; }  
}
```

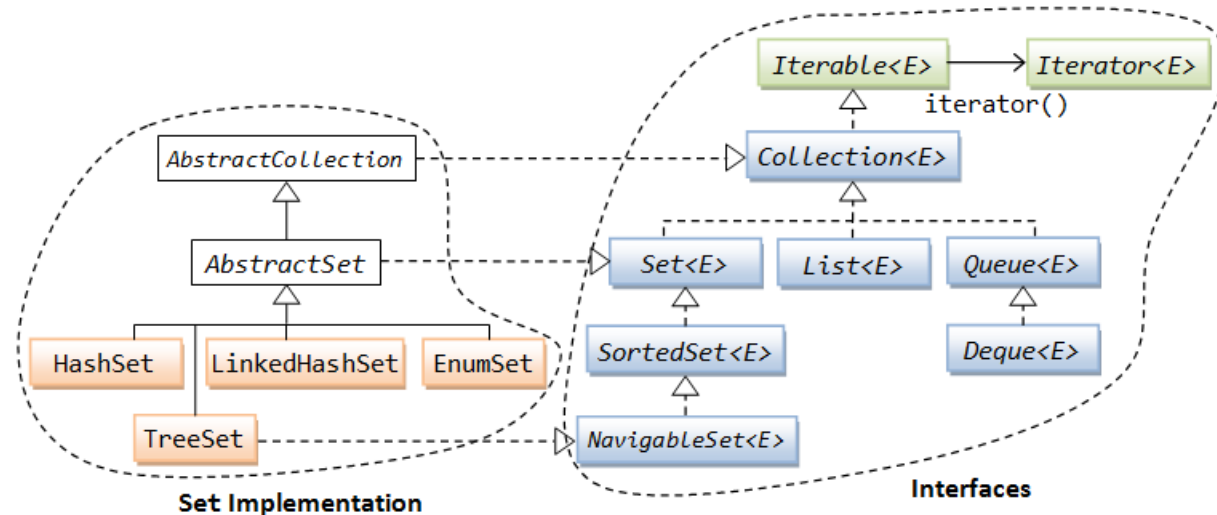
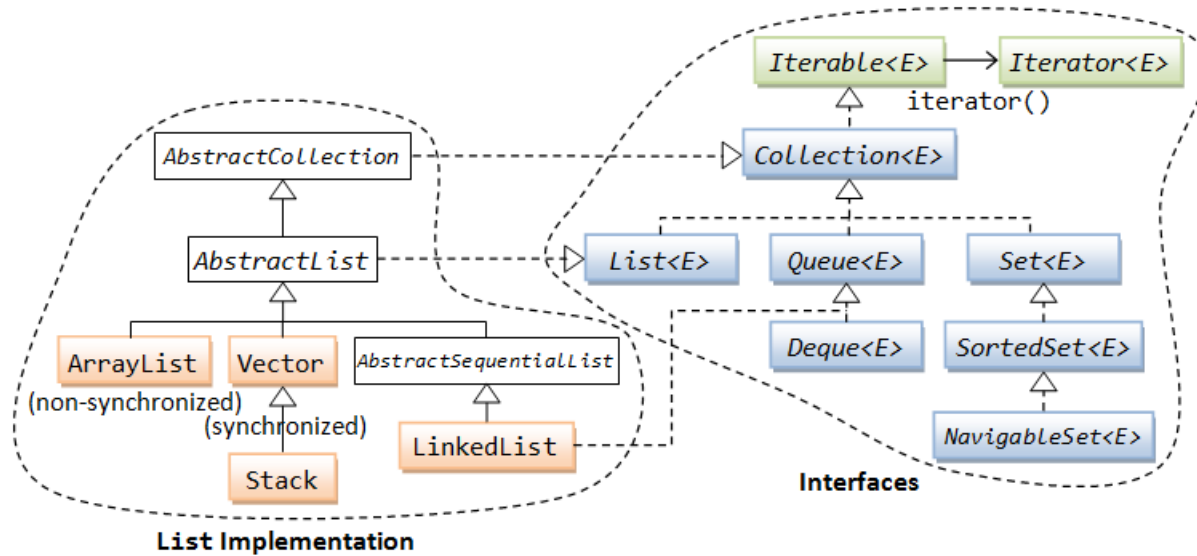
- Zur Instantiierung

`Paar<String, Number>`

- Generics werden am häufigsten verwendet um mit typisierten Containern zu arbeiten
- Ab Java 5 (v1.5) werden typisierte Container-Klassen unterstützt



Quelle: [http://www.ntu.edu.sg/home/ehchua/programming/java/J5c\\_Collection.html](http://www.ntu.edu.sg/home/ehchua/programming/java/J5c_Collection.html)



Quelle: [http://www.ntu.edu.sg/home/ehchua/programming/java/J5c\\_Collection.html](http://www.ntu.edu.sg/home/ehchua/programming/java/J5c_Collection.html)

```
import java.util.ArrayList;

// Constructs an empty list with the specified
// initial capacity:
ArrayList<Song> songs = new ArrayList<Song>(20);

// Add a Song:
songs.add(new Song());

// Set the artist:
songs.get(0).setArtist("Abba");
```

songs is of type ArrayList<Song> and therefore songs.get(i) returns a reference to a Song object. No type cast is needed!



- Unterstützen *statisch typisierte* Datenstrukturen
  - *Frühe Compile-Zeit Erkennung* von Typverletzungen
    - Beispiel: Man kann keinen **String** in **ArrayList<Song>** einfügen
  - Versteckt automatisch generierte Typumwandlungen
  - Keine Typecasts notwendig
- Generische Typen sind Fabriken für *Compile-Zeit*-Abkapselungen generisierter Klassen und Methoden
- *Oberfächlich betrachtet* ähnlich wie C++ Templates

- **Generische Methoden** können in normalen und generischen Klassen definiert werden

```
class Algorithms { // irgendeine Utility-Klasse
    public static <T> T getMiddle(T[] a) {
        return a[a.length / 2];
    }
    ...
}
```

- Beim Aufruf der generischen Methode kann **der Typ** spezifiziert werden

```
String s = Algorithms.<String>getMiddle(names);
```

- Meist wird der Typ automatisch vom Compiler abgeleitet:

```
String s = Algorithms.getMiddle(names);
```

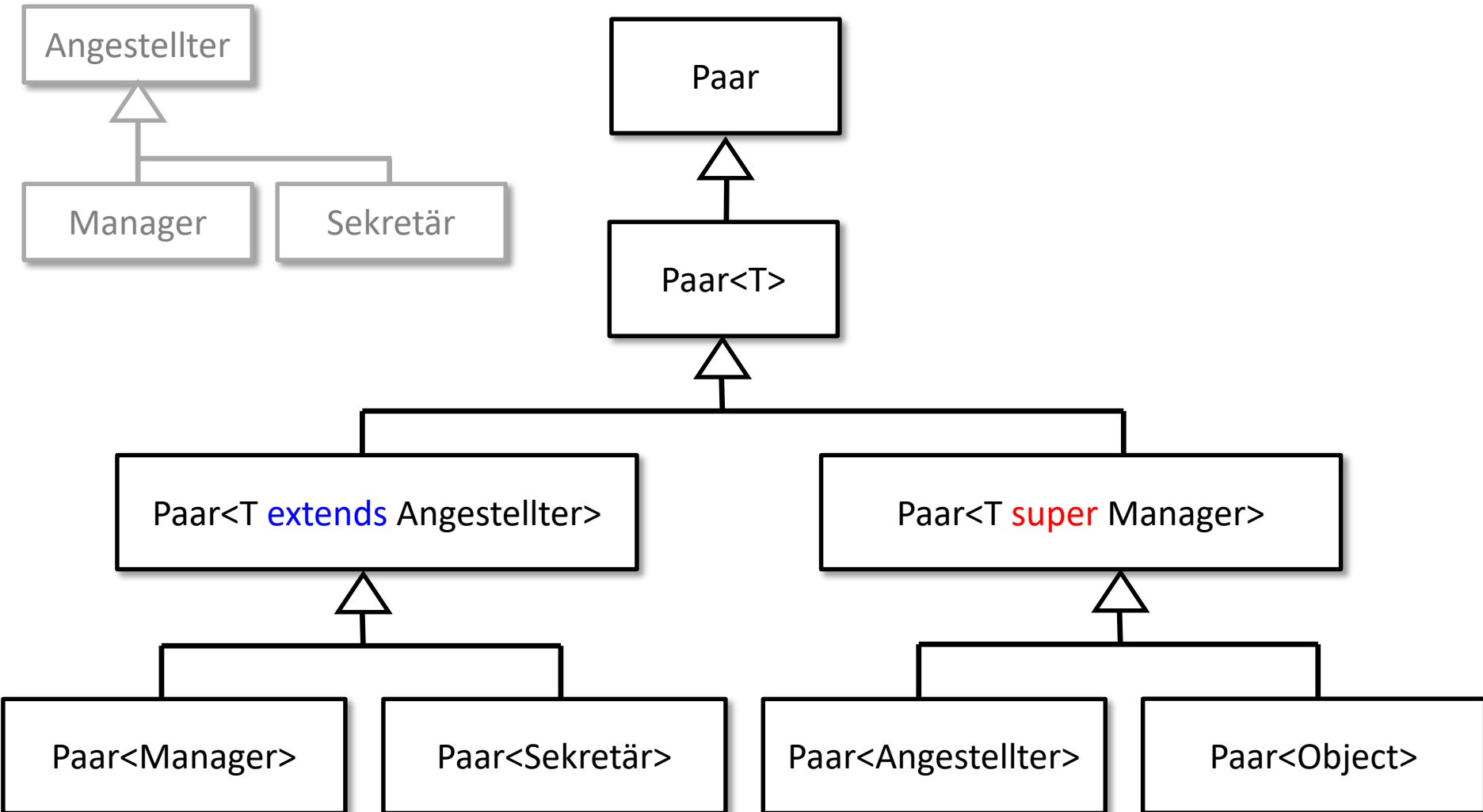
```
class Garage<T extends Vehicle> { ... }
```

- T darf hier nur eine Subklasse von der Klasse **Vehicle** sein

```
class Car extends Vehicle {...}
```

```
Garage<Car> myGarage = new Garage<Car>();
```

- Anstelle von **Vehicle** kann eine beliebige Klasse oder ein beliebiges Interface stehen
- Die Klasse **Garage** darf nur für einen Typ **T** instanziiert werden, der die entsprechende Klasse erweitert (Subklasse), oder im Falle von Interface, das entsprechende Interface implementiert
- Ähnlich gibt es:  
**Class<T super X>** → T darf nur eine Superklasse von **x** sein



- `Paar<Manager>` passt auf `Paar<T extends Angestellter>`  
→ Subtyprelation (*kovariante* Typisierung)
- `Paar<Object>` passt auf `Paar<T super Angestellter>`  
→ Supertyprelation (*kontravariante* Typisierung)
- `Paar<Angestellter>` kann nur *Angestellte* enthalten, aber `Paar<Object>` erlaubt alles (z.B. *Numbers*)  
→ *keine* Subtyprelation
- `Paar<T> <= Paar<?> <= Paar`  

```
List<String> sl = new LinkedList<String>();  
List x = sl; // OK  
x.add(new Integer(5)); // Typsicherheitswarnung  
..  
String str = sl.get(0); // wirft ClassCast
```

- C++ erlaubt generische Programmierung mit Klassen Templates

- Erlauben Typ-spezifische Versionen generischer Klassen

- Format:

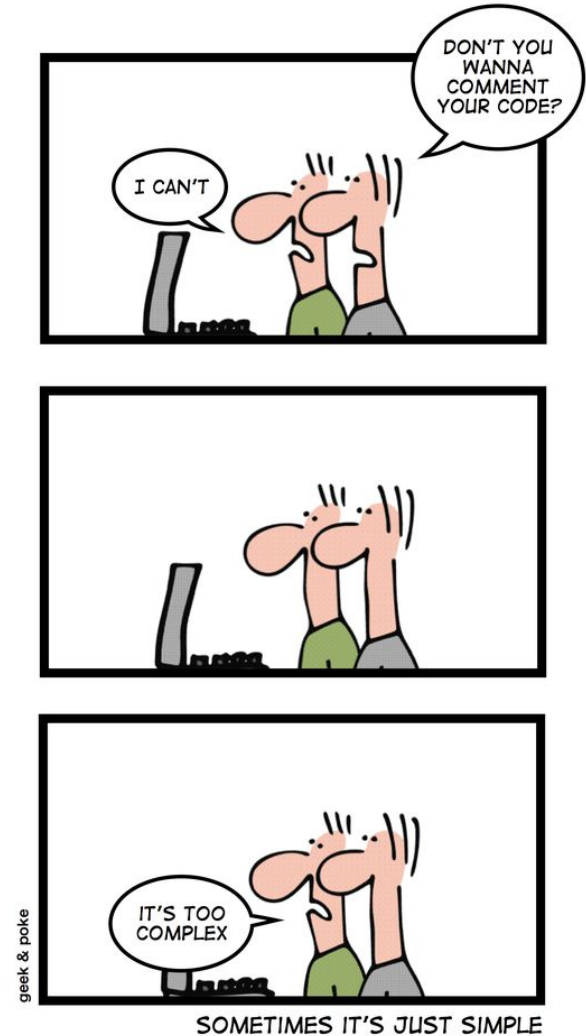
```
template <class T>
class MyClass {
    // class definition
}
```

- "T" kann ein beliebiger Typ sein (z.B. `int`, `double`, `struct`)
  - Objektinstanziierung:  
`MyClass<T> myObject;`

- Template Klassen-Methoden
  - Werden normal deklariert, `template <class T>` muss vor die Deklaration geschrieben werden
    - Generische Daten in der Klasse werden mit `T` angezeigt
- Konzeptioneller Unterschied zu Java Generics:
  - Der Compiler ermittelt jede Benutzung des Templates
  - Kompiliert separate Klasse für jedes `Template<T1>, ..., Template<Tn>`
    - ➔ Templates werden zur Compilezeit zu Klassen transformiert!
  - Java Generics werden zur Compilezeit auf Typsicherheit geprüft, aber erst zur Laufzeit zu Klassen transformiert

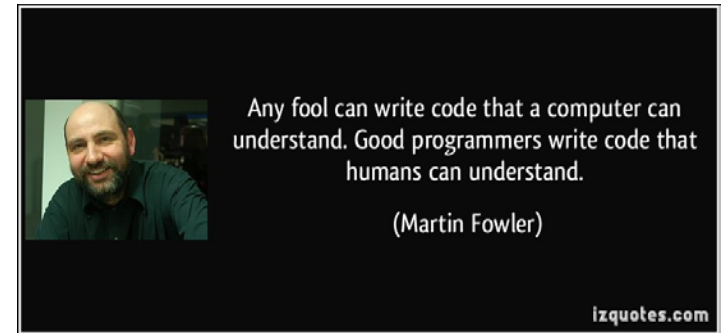
Codedokumentation

# JAVADOC





- Wozu Code kommentieren?



- Problem beim Dokumentieren

- Aktualisierung der Dokumentation
- Code und Dokumentation oft in separaten Dateien

- Lösung: javadoc

- Code und Dokumentation in einer Datei
- javadoc tags z.B. @param, @return
- Teil der JDK Installation
- Generiert HTML Dokumentation

- Klassen, Methoden und Variablen
- Beispiel

```
/** A class comment */
public class MediaDataBase {
    /** A variable comment */
    public int i;

    /** A method comment */
    public void addSong(Song song) {
        ...
    }
}
```

- Beispiele

`@author author-information`

`@version version-information`

`@param parameter-name description`

`@return description`

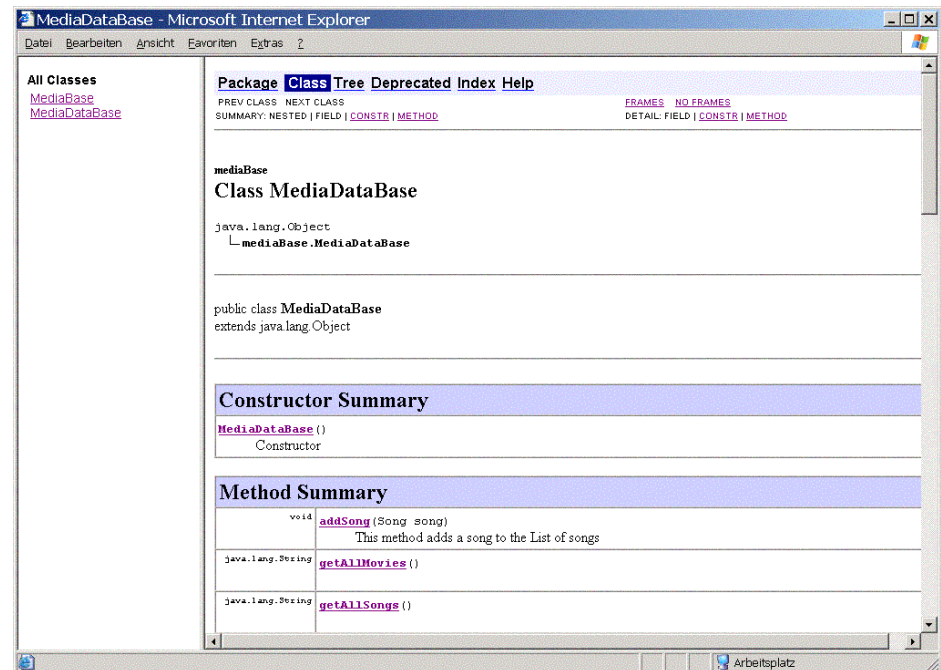
`@throws fully-qualified-class-name description`

```
/**  
 * This method adds a song to the List of songs  
 * @param song the Song to add  
 * @return void  
 */  
public void addSong(Song song) {  
    int i;  
    songs.add(song);  
}
```

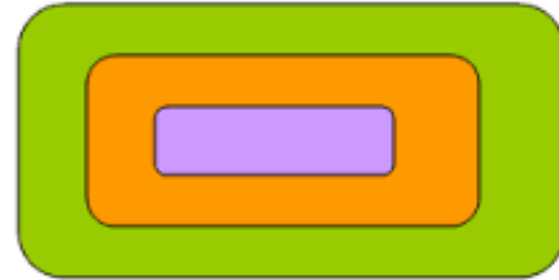
- Zum Erzeugen der HTML-Dokumentation von mediaBase:

```
javac MediaDataBase -d doc mediaBase mediaBase.ui  
mediaBase.model
```

- Siehe javadoc -help



*Java Inner Classes*



Klassen in Klassen

# GESCHACHTELTE, INNERE UND ANONYME KLASSEN

- Klassen können deklariert werden
  - innerhalb einer anderen Klasse (**geschachtelte / innere Klasse**)
  - außerhalb einer Klasse, aber in eigenständiger Datei (**bevorzugt!**)
  - außerhalb einer Klasse, aber in der gleichen Datei

```
public class <MAINCLASS> {  
    // Geschachtelte Klasse  
    public static class <KLASSENNAME> {  
        // hier die einzubindenden Variablen angeben  
    }  
    // Das Hauptprogramm kann diese geschachtelten Klassen nutzen  
    public static void main(String[] args) {  
        // hier steht das eigentliche Hauptprogramm  
    }  
}
```

```
public class BeispielGeschachtelteKlasse {  
    public static class Person {  
        public String vorname;  
        public String nachname;  
        public int alter;  
        public double gehalt;  
        public String abteilung;  
        public int personalnummer;  
    }  
    public static void main(String[] args) {  
        Person otto;  
        otto = new Person();  
        otto.vorname = "Otto";  
        otto.nachname = "Meier";  
        otto.alter = 22;  
        otto.gehalt = 2345.0;  
        otto.abteilung = "F&E";  
        otto.personalnummer = 1234;  
        System.out.println(otto.alter);  
        System.out.println(otto.personalnummer);  
        // u.s.w.  
    }  
}
```

- Sondertyp von geschachtelten Klassen

```
class Außen {  
    class Innen {  
    }  
}
```

Ohne Schlüsselwort **static**!

- Compiler erzeugt für **innere** Klassen eine eigene class-Datei:  
`Außen$Innen.class`
- Vorteil ist im wesentlichen eine reinere Objektorientierung
  - z.B. lässt sich von einer Klasse Tree ein Such-Algorithmus abspalten
- Auch Klassen innerhalb von Methoden sind nach gleichem Muster möglich
  - sog. **Lokale Klassen**



- Innere Klassen können nur im Zusammenhang einer Instanz der äußeren Klasse instanziiert werden
- Die Instanz der inneren Klasse ist immer mit der Instanz der entsprechenden äußeren Klasse assoziiert
- Innere Klassen können auf Komponenten der sie umfassenden Klasse zugreifen
- Eine innere Klasse ist außerhalb des sie definierenden Blockes, außer mit Hilfe des voll qualifizierenden Namens (z.B. Außen.Innen), nicht sichtbar

- In Java sind auch unbenannte lokale Klassen möglich, die dann **anonyme Klassen** genannt werden, diese Klassen müssen ein Interface oder eine Elternklasse implementieren
- Beispiel, wir wollen folgendes Interface inline implementieren:

```
/* MyInterface*/  
public interface MyInterface {  
    // Methodensignatur  
    public int myMethod();  
}
```

## Konstruktor

Bei Interfaces muss hier immer der Standard-Konstruktor verwendet werden. Bei Klassen der jeweilige Konstruktor der Klasse.

## Geschweifte Klammer

Hier können wir Methoden überschreiben und neue Methoden einfügen.

## Überschriebene Methode

Hier überschreiben wir *myMethod*.  
Bei Interfaces müssen alle Methoden überschrieben werden.

## Ausgabe

5

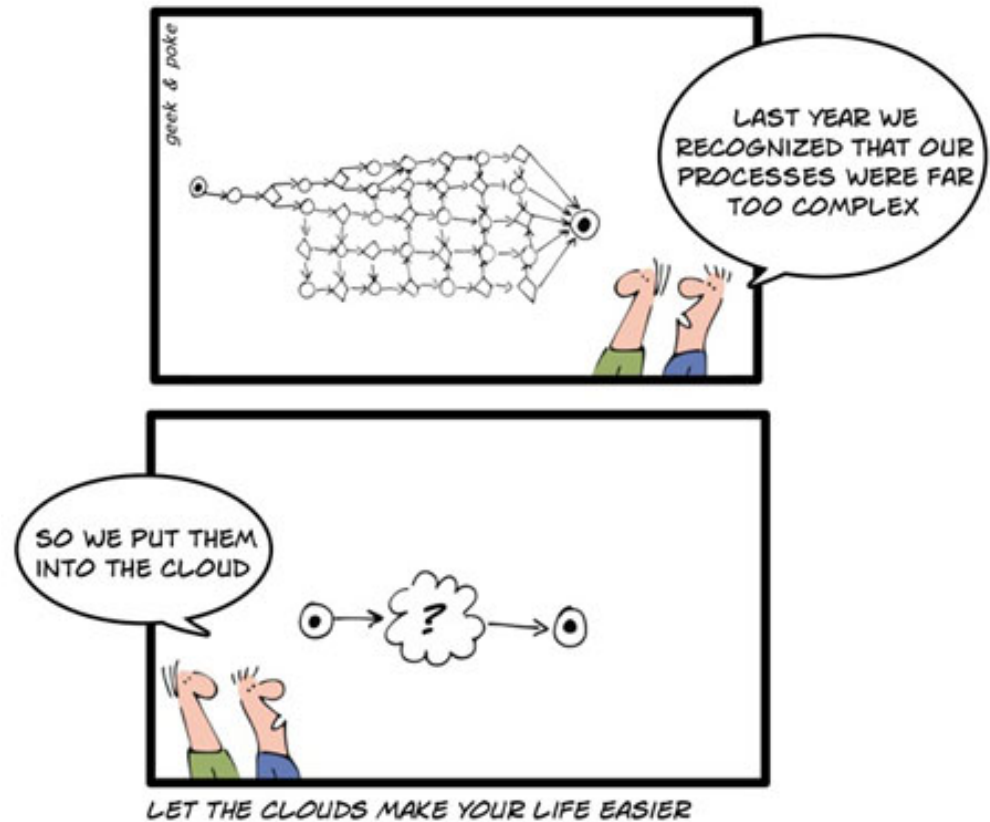
```
/* main-Methode */  
public static void main(String[] args) {  
    // Deklaration und Implementierung in Einem  
    MyInterface five = new MyInterface() {  
        public int myMethod() {  
            return 5;  
        }  
    }  
    System.out.println(five.myMethod());  
}
```

- Anonyme Klassen müssen nicht direkte Kinder des Interfaces (der Elternklasse) sein
- Beispiel: Unsere Implementierung erbt von *LinkedList*, wird aber in eine Variable vom Typ *List* geschrieben. Dies funktioniert, da *LinkedList* auch von *List* erbt.

```
class L {  
    private List list = new LinkedList() {  
        public boolean add(Object o) {  
            System.out.println("Objekt " + o + " hinzugefügt.");  
            return super.add(o);  
        }  
    };  
  
    public List getList() {  
        return list;  
    }  
}  
Aufruf: new L().getList().add("Türklinke");
```

Neue Konstrukte

# JAVA 7



## Java SE Version History (Green: Major; Blue: Minor)



```
public class BinaereLiterale {  
    public static void main(String[] args) {  
        // Binaere Literale  
  
        int dezWert = 42;           // Die Zahl 42, dezimal  
  
        int oktWert = 052;          // Die Zahl 42, oktal  
  
        int hexWert = 0x2A;         // Die Zahl 42, hexadezimal  
  
        int binWert = 0b101010;    // Die Zahl 42, binaer  
  
        System.out.println(dezWert + ", " + oktWert + ", " +  
                             hexWert + " und nochmal " + binWert);  
    }  
}
```

```
public class Unterstrich {  
    public static void main(String[] args) {  
        // Unterstrich in numerischen Literalen  
        long kreditKartenNummer = 1234_5678_9012_3456L;  
        long versicherungsNummer = 999_99_9999L;  
        float pi = 3.14_15F;  
        double e = 2.71_82_81_82_84_59;  
        long hexBytes = 0xFF_EC_DE_5E;  
        long hexText = 0xCAFE_BABE;  
        long maxLong = 0x7fff_ffff_ffff_ffffL;  
        long binBytes = 0b11010010_01101001_10010100_10010010;  
        int ok1 = 4_2;  
        int ok2 = 4_____2;  
        int ok3 = 0x2__a;  
        int ok4 = 0_42;  
        int ok5 = 04_2;  
    }  
}
```

```
public class UnterstrichFalsch {  
    public static void main(String[] args) {  
        // UNZULAESSIGE Verwendung des Unterstriches  
        float badPi1 = 3_.1415F; // _ direkt vor dem Dezimalpunkt  
        float badPi2 = 3._1415F; // _ direkt hinter d. Dezimalpunkt  
        long badNr = 99_99_99_L; // _ direkt vor dem L-Anhang  
        int badNr1 = _42; // _ am Anfang (waere Bezeichner)  
        int badNr2 = 42_; // _ am Ende der Ziffernfolge  
        int badNr3 = 0_x42; // _ innerhalb des 0x-Praefix  
        int badNr4 = 0x_42; // _ am Anfang der Ziffernfolge  
        int badNr5 = 0x42_; // _ am Ende  
        int badNr6 = 042_; // _ am Ende  
    }  
} // nicht compilierbar!
```



```
public static int tageImMonat(String monat, int jahr) {
    . . .
    switch (monat.toLowerCase()) {
        case "februar":
            if ((jahr%4!=0) || ((jahr%100==0) && (jahr%400!=0))) {
                tage = 28;
            } else {
                tage = 29;
            }
            break;
        case "april":      case "juni":
        case "september": case "november":
            tage = 30;
            break;
        case "januar": case "maerz": case "mai": case "juli":
        case "august": case "oktober": case "dezember":
            tage = 31;
            break;
    }
    return tage;
}
```

```
public class GPaar<T> {  
    private T l, r;  
  
    public GPaar(T l, T r) {  
        this.l = l;  
        this.r = r;  
    }  
    public GPaar() {  
        this.l = null;  
        this.r = null;  
    }  
    public T getL() {  
        return l;  
    }  
    public String toString() {  
        return "(l,r) = (" + l + ", " + r + ")";  
    }  
}
```

```
GPaar<Integer> paar1 = new GPaar<>(33,3333);  
GPaar<String> paar2 = new GPaar<>("l","r");  
GPaar<Integer> paar3 = new GPaar<>();  
Integer i3 = paar3.getL();  
GPaar<String> paar4 = new GPaar<>();  
String s4 = paar4.getL();  
int i = new GPaar<>(42,1111).getL();  
String s = new GPaar<>("ich","du").getL();
```

```
public class GPaar<T> {  
    private T l, r;  
  
    public GPaar(T l, T r) {  
        this.l = l;  
        this.r = r;  
    }  
    public GPaar() {  
        this.l = null;  
        this.r = null;  
    }  
    public T getL() {  
        return l;  
    }  
    public String toString() {  
        return "(l,r) = (" + l + "," + r + ")";  
    }  
}
```

Nicht compilerbar!

```
GPaar<String> paar6 = new GPaar<>(1,2);
```

```
GPaar<Integer> paar7 = new GPaar<>("a","b");
```

```
public class ListDiamondDemo {  
    public static void main(String[] args) {  
        List<String> liste1 = new ArrayList<>();  
        liste1.add("Hallo!");  
        List<? extends String> liste2 = new ArrayList<>();  
        liste1.addAll(liste2);  
    }  
}
```

**Nicht compilerbar!**

```
public class ListDiamondFalsch {  
    public static void main(String[] args) {  
        List<String> liste1 = new ArrayList<>();  
        liste1.add("Hallo!");  
        liste1.addAll(new ArrayList<>());  
    }  
}
```

```
import java.nio.file.*;
import java.io.*;
public class PathBeispiele {
    public static void main (String[] args) {
        Path td;
        td = Paths.get("testdatei.txt");
        System.out.println("Path: " + td);
        System.out.println("FileName: " + td.getFileName());
        for (int i=0; i<td.getNameCount(); i++)
            System.out.println("Name: " + td.getName(i));
        System.out.println("Parent: " +
            td.getParent());
        System.out.println("Root: " +
            td.getRoot());
        System.out.println();
    }
}
```

```
Path: testdatei.txt
FileName: testdatei.txt
Name: testdatei.txt
Parent: null
Root: null
```

```
td = Paths.get("c:/Eigene Dateien/ws/java7/testdatei.txt");
System.out.println("Path: " + td);
System.out.println("FileName: " + td.getFileName());
for (int i=0; i<td.getNameCount(); i++)
    System.out.println("Name: " + td.getName(i));
System.out.println("Parent: " + td.getParent());
System.out.println("Root: " + td.getRoot());
}
}
```

```
Path: c:\Eigene Dateien\ws\java7\testdatei.txt
FileName: testdatei.txt
Name: Eigene Dateien
Name: ws
Name: java7
Name: testdatei.txt
Parent: c:\Eigene Dateien\ws\java7
Root: c:\
```

```
import static java.nio.file.StandardOpenOption.*;
import static java.nio.file.StandardCopyOption.*;
import java.nio.file.*;
import java.io.*;

public class DateiOperationen {
    public static void main (String[] args) {

        Path txtVersion = Paths.get("testdatei.txt");
        Path altVersion = Paths.get("testdatei.alt");
        Path oldVersion = Paths.get("testdatei.old");
        Path quark       = Paths.get("quark.txt");
```

```
try {
    Files.deleteIfExists(quark);
    Files.copy(txtVersion, altVersion, REPLACE_EXISTING);
    Files.move(altVersion, oldVersion, REPLACE_EXISTING);
} catch (IOException ioe) {
    ioe.printStackTrace();
}
try (PrintWriter p =
    new PrintWriter(
        Files.newOutputStream(txtVersion, CREATE, APPEND))) {
    p.println("Eine weitere Zeile.");
    p.println("Und noch eine weitere Zeile!");
} catch (IOException ioe) {
    ioe.printStackTrace();
}
}
```



Ausnahmebehandlung

# EXCEPTIONS

```
public class Hallo {  
    public static void main ( String[] args ) {  
        System.out.println("Hallo!");  
    }  
}
```

- Compilieren

Compiliere ...

Programm kompiliert.

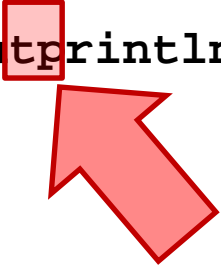
- Interpretieren

Programm wird ausgeführt ...

Hallo!

Programm beendet

```
public class Hallo {  
    public static void main ( String[] args ) {  
        System.out.println("Hallo!");  
    }  
}
```



- Compilieren

## Fehlermeldungen

Tipp: Beheben Sie stets die oberste Fehlermeldung zuerst. Klicken Sie auf eine Fehlermeldung, um zur Zeile zu springen.

Hallo.java (3,11): cannot find symbol  
Methode 'println(java.lang.String)' in Klasse 'java.lang.System'

- Interpretieren
  - **Nicht möglich!**

```
public class Feld
{
    public static void main (String [] args)
    {
        int[][] field = new int[5][];

        System.out.print(field[0][0]);
    }
}
```

- Compilieren

Compiliere ...

Programm compiliert.

- Interpretieren

Programm wird ausgeführt ...

Exception in thread "main" java.lang.NullPointerException  
at Feld.main(Feld.java:9)

Es sind Laufzeitfehler aufgetreten.

- Prinzipiell: Eine Ausnahmesituation
- Konkret in Java: Eine Instanz der Klasse `java.lang.Exception`
  - extends `Throwable`
- Auszug API:

Overview Package **Class** Use Tree Deprecated Index Help

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#)

---

java.lang

## Class Exception

[java.lang.Object](#)

- ↳ [java.lang.Throwable](#)
- ↳ [java.lang.Exception](#)

All Implemented Interfaces:

[Serializable](#)

- Manchmal ist es schwer, im Vorherein zu erkennen, welche Probleme mit den eigenen Programmen auftreten können
- Instanzen der Klasse `java.lang.Exception` helfen uns also beim Verstehen von Problemen in unserem Programm, statt es einfach „abstürzen“ zu lassen.
- Exceptions können durch den Entwickler abgefangen und damit **behandelt** werden. Dann kann das Programm wie vorgesehen weiter ausgeführt werden.
  - Dieses Behandeln von Ausnahmen nennt man **Fangen (catching)**

```
Programm wird ausgeführt ...  
Exception in thread "main"  
java.lang.NullPointerException  
    at Feld.main(Feld.java:9)
```

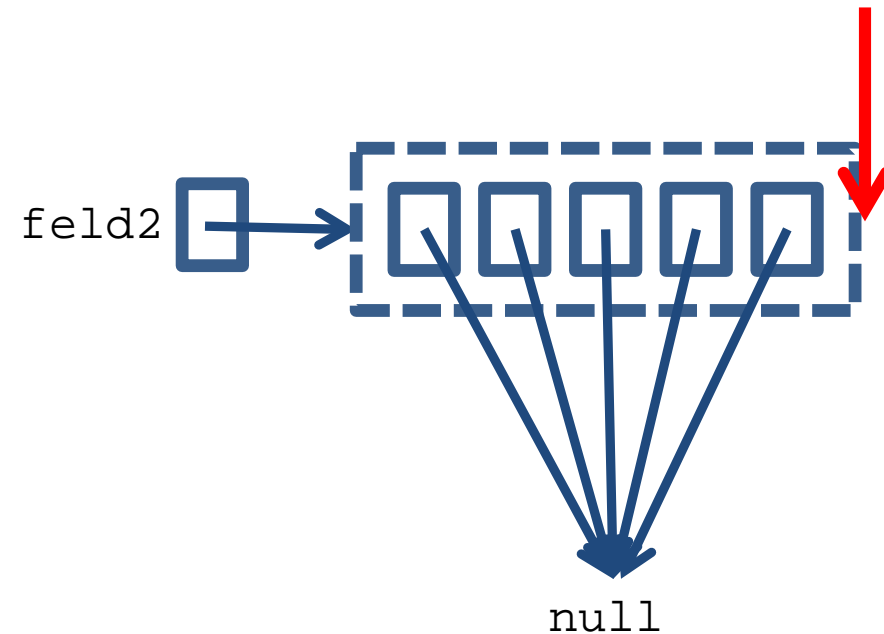
Es sind Laufzeitfehler  
aufgetreten.

## ArrayIndexOutOfBoundsException

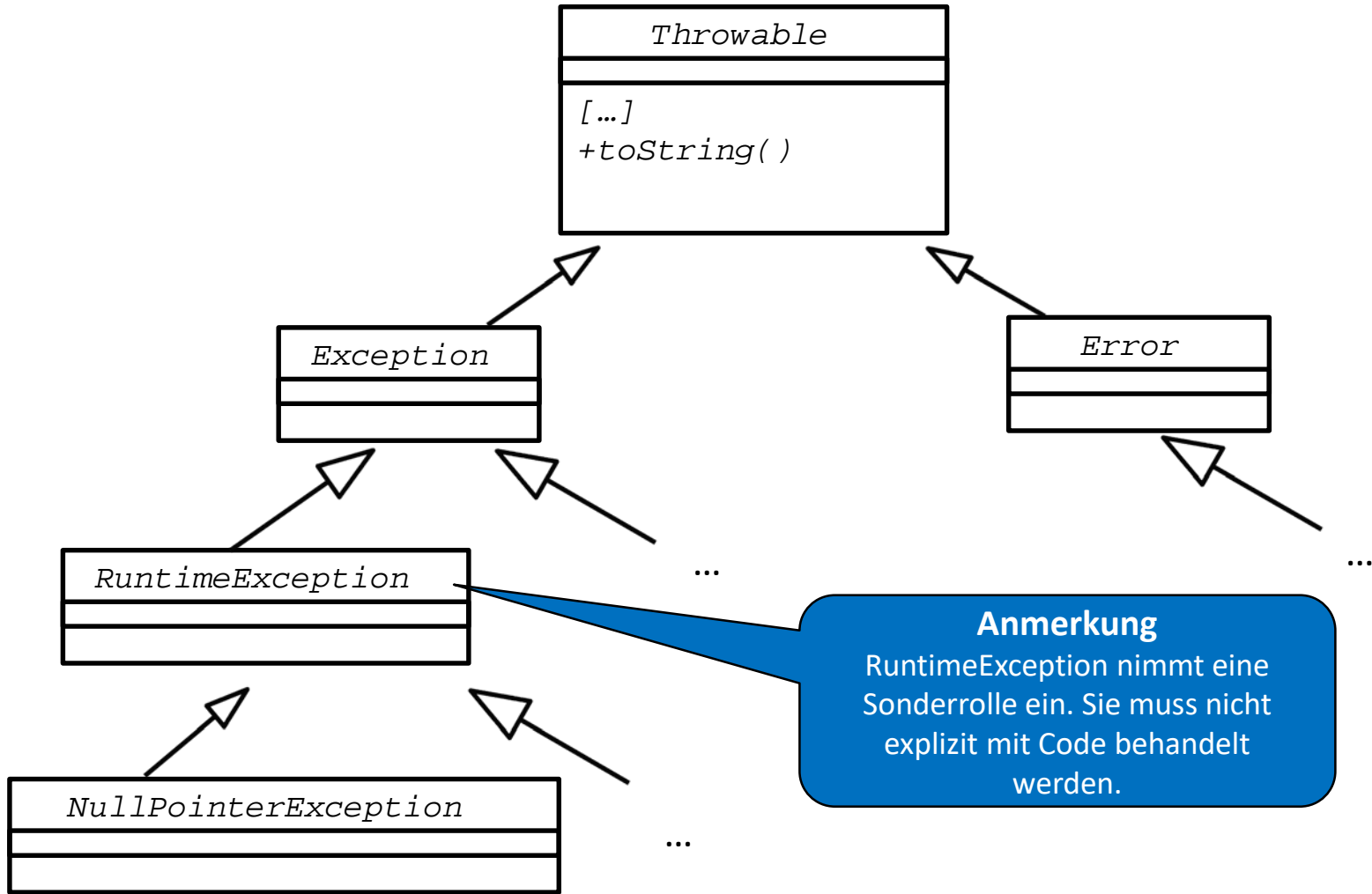
Code

```
int[] feld2 = new int[5];  
System.out.print(feld2[5]);
```

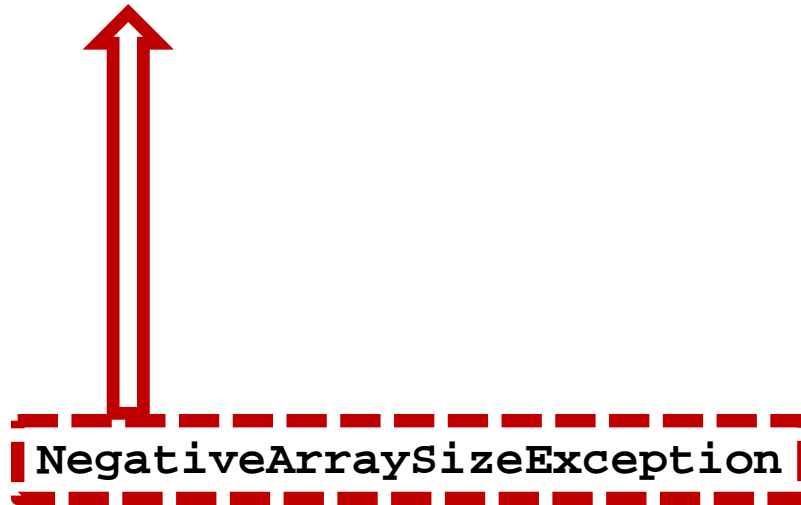
Speicher







```
int[] zfeld;  
Scanner in = new Scanner(System.in);  
System.out.print("Wie lang soll das Feld sein? ");  
zfeld = new int[in.nextInt()];  
.  
.  
.
```



```
int[] zfeld;  
Scanner in = new Scanner(System.in);  
System.out.print("Wie lang soll das Feld sein? ");  
try {  
    zfeld = new int[in.nextInt()];  
} catch (NegativeArraySizeException e) {  
    System.out.print("Bitte nur positive Werte! Wie lang soll das Feld sein? ");  
    zfeld = new int[in.nextInt()];  
}  
.  
.  
.
```

```
int[] zfeld;

Scanner in = new Scanner(System.in);

System.out.print("Wie lang soll das Feld sein? ");

try {

    zfeld = new int[in.nextInt()];
    for (int i = 0; i < zfeld.length; i++) {
        zfeld[i] = i * i;
    }

    System.out.print(" Welches El. soll ausgegeben werden? ");
    System.out.println(zfeld[in.nextInt()]);

} catch (NegativeArraySizeException nase) {
    System.out.print("Nur positive Werte erlaubt.");
} catch (Exception e) {
    System.out.print("Eine Ausnahme ist aufgetreten.");
} finally {
    System.out.println(" Bitte beim nächsten Programmstart beachten. Programm wird
beendet...");
}
```

```
int[] zfeld;  
Scanner in = new Scanner(System.in);  
System.out.print("Wie lang soll das Feld sein? ");  
try {  
    zfeld = new int[in.nextInt()];  
    for (int i = 0; i < zfeld.length; i++) {  
        zfeld[i] = i * i;  
    }  
    System.out.print(" Welches El. soll ausgegeben  
    System.out.println(zfeld[in.nextInt()]);  
} catch (NegativeArraySizeException nase) {  
    System.out.print("Nur positive Werte erlaubt.");  
} catch (Exception e) {  
    System.out.print("Eine Ausnahme ist aufgetreten."  
} finally {  
    System.out.println("Beim nächsten Programm  
beendet...");  
}
```

**Potentieller Fehler 0**

Negative Eingabe =>  
NegativeArraySizeException

**Potentieller Fehler 1**

Negative/zu Große Eingabe =>  
ArrayIndexOutOfBoundsException

**Catch Block 0**

Wird zuerst überprüft. Eine Exception,  
die hier gefangen wird, löst späteres  
Catch nicht aus.

**Catch Block 1**

Wird danach überprüft. Hier eine  
allgemeinere Exception, die u.a.  
ArrayIndexOutOfBoundsException mit  
fängt.

**Finally**

Wird immer am Ende ausgeführt.  
Kann auch ohne Exceptions nötig sein,  
um Verhalten am Ende zu garantieren.  
Beispiel: Geöffnete Dateien schließen.

# Zusammenfassung

- **Exceptions** helfen beim Aufdecken von Lücken und Problemen im eigenen Programm
- **Exceptions** können durch den Nutzer/die Nutzerin abgefangen werden (catching)
- Beim Abfangen ist die Vererbungslinie zu beachten, um die spezifisch-gewollten **Exceptions** geeignet abzufangen

# Fragen?



**"It's the latest innovation in office safety.  
When your computer crashes, an air bag is activated  
so you won't bang your head in frustration."**

# Danksagung

- Vorlesungsmaterialien von Prof. Dr. Detlef Seese wurden als Basis verwendet
- Unterstützung bei der technischen und inhaltlichen Gestaltung des Vorlesungsmaterials leisteten:

Jóakim v. Kistowski

Dietmar Ratz, Joachim Melcher, Roland Küstermann, Jana Weiner, Hagen Buchwald, Matthes Elstermann, Oliver Schöll, Niklas Kühl, Tobias Diederich