

## **Das erweiterte Typsystem (ab Java 5)**

Mit Java 5 wurde eine ganze Reihe von Spracherweiterungen eingeführt. Alle betreffen nur den Compiler. Sie erlauben vereinfachte Schreibweise und bessere Lesbarkeit. Der deklarative Anteil von Java und die Typprüfung durch den Compiler wurden erheblich verstärkt.

- Annotationen
- Generische Typen und Methoden
- Enum-Klassen\*

## Annotationen

Annotation bedeutet “Anmerkung”. Annotation sind daher ein Mittelding zwischen Kommentar und Deklaration. Sie können sich auf Typen, Felder und Methoden beziehen.

Im Unterschied zu einem Kommentar sind sie sehr formal aufgebaut und können durch den Compiler oder auch zur Laufzeit untersucht werden.

Im Unterschied zu einer Deklaration haben sie keinen Einfluss auf die Korrektheit eines Programms.

Ihre Bedeutung liegt in:

- Lesbarkeit und Überprüfbarkeit von Zusatzinformationen
- Optionen für die Übersetzung
- Steuerung von Frameworks und Mechanismen (Web-Services, Unit-Tests)

## Beispiele für Annotationen

Der Compiler soll sicherstellen, dass tatsächlich die Methode der Oberklasse überschrieben wurde.

```
@Override  
public String toString() { .. }
```

Bestimmte Compilerwarnungen sollen für das folgende Element unterdrückt werden.

```
@SuppressWarnings ("unchecked")  
public Stack() { ...}
```

Die folgenden Annotationen sind selbstdefiniert (für später)

Die folgende Klasse ist threadsicher

```
@ThreadSafe  
public class ConcurrentStack {
```

Die folgende gemeinsame Variable wird durch die angegebene Sperre geschützt

```
@GuardedBy ("this")  
private int counter = 0;
```

## Annotationen in JUnit

In JUnit 4 kennzeichnen Annotationen Testmethoden. Durch Tags können weitere Eigenschaften festgelegt werden.

**@Before**

```
public void initialisierung() { .. }
```

**@Ignore("ist noch nicht fertig")**

**@Test**

```
public void testMethode() { ... } // wird ignoriert
```

Timeout:

```
@Test(timeout=1000)
```

Test ob eine Exception geworfen wird:

```
@Test(expected = ArithmeticException.class)
```

```
public void illegalConstruction() {
```

```
    new Bruch(1, 0);
```

```
}
```

## Annotationen definieren

Der Compiler soll sicherstellen, dass tatsächlich die Methode der Oberklasse überschrieben wurde.

```
@Documented                erscheint in JavaDoc
@Target(ElementType.TYPE)   darf nur bei Klassen/Interfaces stehen
@interface ThreadSafe {
}

@Target(ElementType.FIELD)  steht bei Klassen-/Instanzvariablen
@interface GuardedBy {
    String value()           benötigt zwingend eine Stringangabe
}

@Target({ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
@interface Example {
    String name()
    Class using()
    double cost() default 100.0
    String[] friends()
}

@Example(name="abc", using=String.class, friends={"fritz", "paul"})
```

## Generische Datentypen in Scala und Java

### Java:

Array:

```
int[] a = new int[n];
```

Containerobjekte:

```
List<Integer> a = new ArrayList<Integer>();
```

### Scala:

Array:

```
val a: Array[Int] = new Array[Int](n)
```

Containerobjekte:

```
var a: List[Int] = List[Int]()
```

**Problem:** mit Ausnahme von Arrays sind die konkreten Typargumente zur Laufzeit nicht bekannt (*Typlöschung, type erasure*) !

## Generische Methoden

Syntax:

*Modifikatoren* *<Typparameter>* *Typ Name (Parameter) ...*

(Modifikatoren sind **public**, **static**, **abstract**, ...)

```
public static <T> List<T> asList(T... a) {  
    return new ArrayList<T}  
List<String> stringList = asList("hello", "world");
```

Die notwendige Information über Typparameter wird beim Aufruf der Methode automatisch ermittelt (Typinferenz).

**Scala:**

```
def methode[T] (x: T): T
```

*Scala kennt generell die Typinferenz*

## Nach oben beschränkte Typparameter

Syntax:

```
<Parameter extends InterfOderKlasse & Interface .. >
```

Gibt an, dass der Typparameter sich auf eine Unterklasse einer Klasse bezieht und die angegebenen Interfaces implementiert. Wenn keine Oberklasse sondern nur Interfaces angegeben sind, darf der Typparameter auch ein Interface sein, das die anderen Interfaces erweitert.

Auswirkung:

- Einschränkung für mögliche konkrete Typparameter
- Möglichkeit alle deklarierten Methoden aufzurufen

(schlechtes) Beispiel:

```
public static <T extends Number> double abs (T x) {  
    double v = x.doubleValue ();  
    return Math.abs (v);  
}
```

(Scala:  $T <: \text{Number}$ , von unten beschränkt:  $\text{Int} <: T$ )

## Probleme mit Generics (type erasure) \*

= *keine Laufzeitinformation über Typparameter vorhanden!*

### Erzeugung eines Arrays:

```
class Stack<T> {  
    private T[] data = new T[100]; // falsch !!!  
    //richtig:  
    private T[] data = (T[]) new Object[100];  
}
```

**Begründung:** `new` wird zur Laufzeit ausgeführt und benötigt eine Klassenreferenz. Wegen type erasure kann das kein Parameter sein!

**Die richtige Variante ist aber ebenfalls nicht perfekt, da der Cast nichts tut! Die strenge Typprüfung der Übersetzung wird ausgehebelt: *Warnung!***

Solange keine Warnung erfolgt, garantiert Java Typsicherheit. Bei dem Vorliegen von Warnungen kann man den Compiler „überlisten“. Es gibt aber immer noch die Typprüfung zur Laufzeit (ClassCastException).

*Der Kompromiss ist nötig, wegen der (vorläufigen) Aufwärtskompatibilität zu altem Code.*

In der Java-Literatur werden Alternativen unter dem Schlagwort *Reification* beschrieben. (*Scala hat eine halb-automatische Reification-Methode*)

## Grundprobleme der Typsicherheit \*

- Jede von **mehreren Referenzen** auf ein Objekt kann einen anderen Typ haben. Jede Referenz muss (zur Übersetzungszeit) einen **Obertyp** des Objekttyps haben.
- Bei **Behältern** (z.B. Array) muss gewährleistet sein, dass **keine falschen Objekte gespeichert** werden.
- Der Compiler kann das nur unzureichend prüfen. Typsicherheit von **Arrays** geht nur über die automatische **Laufzeitprüfung**.
- Generische Typen haben stehen zur Laufzeit nicht zur Verfügung (**Typlöschung**).
- Typverträglichkeiten müssen bei **Generischen Typen strenger** als bei Arrays geprüft werden.
- Ausnahmen werden vom Programmierer über Typanpassungen formuliert. Allerdings: sind das nur Hinweise für den Compiler! Anpassungen mit Typparametern bewirken nichts, keine Typprüfung! (warning)

*Viele Programmiersprachen (insbesondere Skriptsprachen) kennen nur dynamische Typinformation. Viele Probleme verschwinden damit von selbst – allerdings auch alle Vorteile der Information für den Compiler!*

*Scala versucht den Mittelweg, möglichst viel zu raten. (+ ein paar Workarounds)*

## Vererbung bei Array

```
Super[] s;  
Derived[] d = new Derived[n];
```

Wenn `Super` ein Obertyp von `Derived` ist, dann ist auch `Super[]` ein Obertyp von `Derived[]`.

Damit ist die Zuweisung `s = d` erlaubt (die umgekehrte Zuweisung nicht).

Das kann funktionieren oder zu Problemen führen, die der Compiler nicht erkennt:

```
s = d;  
s[i].methode();           // funktioniert  
Super x = s[i];          // funktioniert  
Derived y = (Derived) s[i]; // funktioniert  
  
s[i] = new Super();      // ArrayStoreException
```

**Die Typsicherheit des Arrays beruht nur auf der Typinformation im Arrayobjekt!**

*Es ist daher auch wichtig, wenn ein Array möglichst die richtige (dynamische) Typinformation trägt. Das genau ist auch das Thema von **Reification**.*

## Probleme mit Generischen Typen (Vererbung)

```
class Super<T> ...  
class Derived<T> extends Super<T> ...
```

$T$ ,  $O$  und  $U$  sollen beliebige konkrete Typen bezeichnen,  $O$  soll Obertyp von  $U$  sein.

1. Es gilt: `Super<T>` ist Obertyp von `Derived<T>`
2. Aber es gilt **nicht**, dass `Super<O>` Obertyp von `Super<U>` ist (genausowenig wie `Derived<O>` Oberklasse von `Derived<U>` ist).

Umgangssprache:

Ein Eimer mit Nüssen ist ein Behälter mit Nüssen und für Nüsse (Fall 1)

Ein Eimer für Nüsse ist kein Eimer für Objekte!! (Fall 2)

(er ist allerdings ein Eimer von Objekten, das gilt aber nur solange man da nichts hineintun kann! -- Spezialfall, erfordert in Java besonderen Hinweis)

## Begründung: Zuweisungsanomalie

In einer Zuweisung  $L = R$  hat die linke Seite den Typ  $L$  und die rechte Seite den Typ  $R$ .

Die Zuweisung ist erlaubt, wenn der Typ  $R$  **gleich  $L$  ist oder ein Untertyp von  $L$**  ist. D.h. auf der rechten Seite der Zuweisung kann man immer auch speziellere Untertypen verwenden (*kovariantes Verhalten*, vgl. *Substitutionsprinzip*).

Umgekehrt lautet dieser Satz:

Die Zuweisung ist erlaubt, wenn der Typ  $L$  **gleich  $R$  ist oder ein Obertyp von  $R$**  ist. D.h. auf der linken Seite der Zuweisung kann man immer auch allgemeinere Obertypen verwenden (*kontravariantes Verhalten*).

Dieses Szenario hat direkte Auswirkungen auf die Regeln der Vererbung bei generischen Typen. Und zwar hängt das Verhalten davon ab, ob eine Zuweisung oder eine Verwendung eines Objektes erfolgt.

Man kann nicht alles haben!! (*there is no free lunch*)

Klartext: ist es ein Eimer von Nüssen oder für Nüsse?

***Funktionale Datentypen verhalten sich immer kovariant (es gibt keine Zuweisung).***

## Beispiel

```
class Example<T> {  
    private T storage;  
  
    public put(T x) { storage = x; }  
  
    public T get() { return storage; }  
}
```

angenommen, es wäre erlaubt:

```
Example<Object> st = new Example<Integer>();
```

dann wäre auch `st.put("Hallo")` erlaubt. Fehler !!!

wenn umgekehrt sowas wie

```
Example<Integer> st = new Example<Object>();
```

zulässig wäre, hätten wir kein Problem mit `put`, aber ein Problem mit `get` (zu genauer Rückgabetyt) !!!

**Wir dürfen die Vererbung im Typparameter nicht zulassen, da wir zur Laufzeit nicht mehr die für den sicheren Gebrauch nötige Typinformation haben!**

## Typeinschränkungen und Wildcards

Die vorangehende Diskussion zeigt, dass Typparameter und Vererbung sich nicht immer vertragen. Da es keine Lösung gibt, die immer richtig ist, brauchen wir spezielle Varianten.

Es gibt 4 denkbare Möglichkeiten:

- **Invarianz.** Die Vererbung der Parameter stellt keine Typverwandschaft her.
- **Kovarianz.** Die Typverwandschaft der parametrisierten Typen entspricht der Vererbungsverträglichkeit der Parameter.
- **Kontravarianz.** Die Verträglichkeit der Generischen Typen ist umgekehrt zu der der Parameter.
- **Bivarianz.** Der Typparameter schränkt die Verwandschaft nicht ein.

Je nach Wahl der Vererbungsregel ergeben sich andere Einschränkungen in der Verwendung der Objekte.

***Während man bei Java die Varianzregel beim Gebrauch festlegt, erfolgt dies in Scala meist bei der Deklaration der Datenstruktur.***

## Festlegung bei Klassendefinition in Scala

### Invarianz:

```
class Stack[T] {  
    def T pop() =  
    def push(T x)  
}
```

stringStack = stringStack  
~~objectStack~~ = ~~stringStack~~  
~~stringStack~~ = ~~objectStack~~

### Kovarianz:

```
class Stack[+T] {  
    def T pop() =  
    def push(T x)  
}
```

stringStack = stringStack  
objectStack = stringStack  
~~stringStack~~ = ~~objectStack~~

### Kontravarianz:

```
class Stack[-T] {  
    def T pop()  
    def push(T x) =  
}
```

stringStack = stringStack  
~~objectStack~~ = ~~stringStack~~  
stringStack = objectStack

## Wildcards

Wildcards dienen der ungenauen Angabe von aktuellen Typparametern. Damit lassen sich alle vier Möglichkeiten der Verträglichkeit ausdrücken.

- Der unbeschränkte Wildcard `?` sagt nichts aus:  
Kaum Operationen aber *Bivarianz*
- Von unten beschränkter Wildcard (`? super Typ`): *Kontravarianz*.
- Von oben beschränkter Wildcard (`? extends Typ`): *Kovarianz*.
- Exakte Typangabe (kein Wildcard): *Invarianz*.

## Beispiel für unbeschränkten Wildcard:

```
public static void printList(List<?> lst) {  
    for (Object x : lst)  
        System.out.println(x);  
}
```

Da der Typparameter unbekannt ist, kann man nur davon ausgehen, dass der wirkliche Objekttyp sich mit `Object` verträgt. -- Die Methode kann mit beliebigen Listen aufgerufen werden.

Anmerkung: wenn man ein Objekt erzeugt, muss man den Typ kennen. Der Wildcard tritt nur in Deklarationen auf.

### **Scala (kein Wildcard):**

```
def printList[+T](lst: List[T]) {  
    for (x <- lst) println(x)  
}
```

## Beispiel für von oben beschränkten Wildcard:

```
public static double summe(List<? extends Number> lst) {  
    double s = 0.0;  
    for (Number x : lst)  
        s += x.doubleValue();  
    return s;  
}
```

Hier weiß der Compiler, dass alle Listenobjekte von der abstrakten Klasse `Number` abgeleitet sind. Es ist möglich, die Listeninhalte entsprechend zu verwenden. Es ist aber nicht möglich, der Liste neue Inhalte zuzuweisen (damit könnte ja dann die Regel verletzt werden, dass eine konkrete Liste nur `Double` enthalten darf).

Die Methode kann mit `List<X>` aufgerufen werden, wenn `X` ein (konkreter) Untertyp von `Number` ist.

## Beispiel für von unten beschränkten Wildcard:

```
public interface Comparator<T> {  
    public int compare(T x, T y)  
}
```

Die Verwendung eines passenden Objekts wird wie folgt deklariert:

```
public static <T> T maxObject(List<T> lst, Comparator<? super T> c) {  
    T max = null;  
    for (T x : lst) {  
        if (max == null || c.compare(x, max) > 0) max = x;  
    }  
    return max;  
}
```

Die Methode `maxObject` kann mit einer beliebigen Liste aufgerufen werden; das `Comparator`-Objekt muss für die Listenelemente funktionieren, d.h. es muss für deren Typ oder einen Obertyp davon geschrieben sein..

## Es kann kompliziert werden:\*

```
public static <T extends Comparable<? super T>>
    T maxObject(List<T> lst) {
    T max = null;
    for (T x : lst) {
        if (max == null || x.compareTo(max) > 0) max = x;
    }
    return max;
}
```

### Anmerkungen:

Die extends-Angabe zu dem Parameter gibt dem Compiler notwendige Information.

Bei dem Typparameter für `Comparable` steht ein Wildcard.

Wildcards werden, wenn möglich, bevorzugt.

## Gültigkeitsbereich für Typparameter \*

- komplette Parameterdeklaration (auch vor erstem Auftreten):  
<T **extends** S, S ...>
- komplette Klasse/Methode
- auch innerhalb innerer Klassen
- **nicht:** in statischen Variablen, Methoden und statischen Klasse !!!

statische Methoden und Klassen müssen ihre eigenen Parameter haben!  
(Grund: für die gesamte Menge von generischen Objekten, gibt es nur eine einzige Klasse!)

```
class Abc<X> {  
    static class Nested<Y> {  
        Y variable;  
    }  
  
    Abc Nested<X> instanzVariable;  
  
    static <Z> void staticMethod(Z argument) { ... }  
}
```

## Iteration und Typparameter

In der Klassenbibliothek sind viele Klassen und Interfaces mit Parametern versehen worden:

```
interface Iterator<T> {  
    T next();  
    boolean hasNext();  
    T remove();  
}
```

```
interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

```
interface List<T> extends Iterable<T> {  
    ...  
}
```

## Einfache Eigenschaften der Aufzählung\*

```
public enum Tag {  
    MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG,  
    FREITAG, SAMSTAG, SONNTAG  
    // hier darf aber auch ein Klassenkörper stehen!  
}
```

Da jedes Tag-Objekt einmalig ist, genügt der Vergleich mit `==`

```
if (t == Tag.MONTAG) ...
```

Den Objekten ist eine Ordnungszahl zugeordnet: `t.ordinal()`

Die Methode `toString()` gibt den lesbaren Namen zurück.

Man kann ein Objekt aus dem Namen konstruieren:

```
Tag t = Tag.valueOf("MONTAG");
```

Die Methode `Tag.values()` liefert eine Array der Elemente.

Es gibt ein typsicheres switch:

```
switch (t) {  
    case MONTAG: // nicht Tag.MONTAG  
        ...  
}
```

## Anbindung an die Java-Bibliothek \*

Jede Enum Klasse ist abgeleitet von

```
abstract class Enum<E extends Enum<E>>  
    implements Comparable<E>, Serializable
```

Es gibt eine besonders effiziente Map-Implementierung:

```
class EnumMap<K extends Enum<K>, V>  
    implements Map<K, V> ...
```

und eine besonders effiziente Set-Implementierung:

```
class EnumSet<E extends Enum<E>> implements Set<E> ...
```

sinnvolle Methode:

```
EnumSet<Tag> x = EnumSet.range(Tag.MONTAG, Tag.MITTWOCHE);
```

HashSet/HashMap funktioniert, aber EnumSet/EnumMap sind effizienter.

Grund: Es gibt eine feste Menge von Objekte, so dass hier die Hashzahl = der Ordinalzahl ist und die Arraygröße = der Anzahl der Objekte ist (bei einem Set kann es sogar eine Bitmenge sein).

## Erweiterungen von Enum-Klassen \*

Enum-Klassen können wie andere Klassen auch über einen Konstruktor und über Methoden verfügen (der Konstruktor-Aufruf erfolgt nur bei der Definition der Konstanten).

```
enum Tag { // ordnet jedem Tag eine konstante Arbeitszeit zu
    MONTAG(6), DIENSTAG(7), ..., SONNTAG(0);

    private int std;

    private Tag(int std) { // muss private sein (warum?)
        this.std = std;
    }

    public int getStd() {
        return std;
    }
}
```

## Polymorphe Enum-Konstanten\*

Bei der Deklaration einer Enum-Konstanten kann für diese eine die Enum-Klasse erweiternde Anonyme Klasse deklariert werden. Dort können Methoden überschrieben werden.

```
public enum State {  
    S0 {  
        public State nextState() { return S1; }  
    },  
    S1 {  
        public State nextState() { return S0; }  
    };  
  
    public abstract State nextState();  
}
```

Natürlich kann diese Variante mit den anderen kombiniert werden. Die Methoden der anonymen Klassen müssen extern deklariert sein. Dies kann auch in einem Interface geschehen:

```
public enum State extends Interface { . . . }
```