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.

- Typparameter, Generische Typen und Methoden
- Annotationen*
- Enum-Klassen*

Notwendigkeit für Typparameter

Historie: „Reine" Objektorientierung ermöglicht die Formulierung polymorpher Behälterklassen. Polymorphie ermöglicht hohe Flexibilität, hat aber den Nachteil, dass statische Typinformation fehlt.

Beispiel: Man kann in Java entweder einen Stack für Objekte schreiben (was ist da drin?) oder man schreibt einen Stack für Strings (dann muss man für jeden Datentyp einen Stack schreiben). Generische Klassen ermöglichen, es für Objekte eine beliebig einstellbare Klasse zu schreiben: Stack<T>.

```
interface Stack {
    void push(Object x); x kann alles sein
    Object pop();        die Rückgabe kann alles sein
}
interface StringStack {
    void push(Object x); x muss ein String sein
    Object pop();        die Rückgabe ist ein String
}
interface <T> {          beschreibt Stack vom Typ T
    void push(T x);      x muss T sein
    T pop();             die Rückgabe ist T
}
Stack<String> stringStack; verhält sich wie StringStack
```

Templates in C++

C++ kennt das Template-Konzept. Das erlaubt eine mächtige Parametrisierung von Klassen und Funktionen. Es ist aber genau genommen nichts anderes als ein eleganter Präprozessormechanismus. Eines der Probleme sind schwer lesbare Fehlermeldungen.

```
template<class T, int S>  class Stack {
private:
  T data[S];
  int top = 0;
public:
  void push(T x) {
    if (top == S) throw "stack full";
    data[top++] = x;
  }
  T pop() {
    if (top == 0) throw "stack empty";
    return data[--top];
  }
}

Stack<int, 100) intStack();
```

In C++ wird für jeden Datentyp ein eigenes Objekt erstellt. Als Parameter sind auch elementare Typen und Zahlen erlaubt.

Generische Datentypen in Java und Scala

In Java wollte man wohl zunächst das komplizierte C++-Erbe vermeiden. Wie in den klassischen Programmiersprachen sind nur Arrays mit dem Elementtyp parametrisierbar.

Erst in Java 5 hat man dann das Konzept der generischen Typen eingeführt. Anstelle des komplexen Template-Mechanismus bezieht sich die Parametrisierung nur auf die Typprüfung und nicht auf die Codegenerierung.

Das Java-Konzept hat einige Vorteile. Seine Einführung erfolgte allerdings unter den Einschränkungen der Aufwärtskompatibilität:

- Prüfung nur im Compiler (**Typlöschung**)
- **Varianzangabe bei Verwendung**
- Explizite „**Reification**“ (= Übergabe des Typs als Referenz zu Klassenobjekt)

In Scala war ein Neuanfang möglich. Allerdings musste die Kompatibilität zu JVM und Java-Bibliothek berücksichtigt werden:

- Prüfung nur im Compiler (**Typlöschung**)
- **Varianzangabe bei Deklaration** (bei Verwendung auch möglich)
- Implizite „**Reification**“ (bei geeigneter Deklaration automatische Typspeicherung)

Generische Datentypen in Scala und Java

Java:

Array:

```
int[] a = new int[n];
Object[] b = a;                 // erlaubt
```

Containerobjekte:

```
List<Integer> a = new ArrayList<Integer>();
List<Object> b = a;             // verboten
```

Scala:

Array:

```
val a: Array[Int] = new Array[Int](n)
val b: Array[Object] = a        // verboten
```

Containerobjekte:

```
val a: List[Int] = List[Int]()
val b: List[Object] = a         //verboten
```

Probleme:

- Java-Arrays sind besonders parametrisiert und habe eine besondere Schreibweise
- Grundsätzlich ist List<Obertyp> kein Obertyp von List<Untertyp> !
- Es ist nicht möglich `T[]` zu erzeugen, da Arrays den Elementtyp kennen müssen!

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>(a);
}
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

Probleme mit Generischen Typen (Vererbung)

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

Oben soll Obertyp von *Unten* sein.

1. Es gilt: `Super<T>` ist Obertyp von `Derived<T>`

2. Aber es gilt **nicht**, dass `Super<Oben>` Obertyp von `Super<Unten>` ist (genausowenig wie `Derived<Oben>` Oberklasse von `Derived<Unten>` 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.

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.

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!!

- ***Rückgabetypen sind kovariant***
- ***Funktionsparameter sind kontravariant***
- ***Funktionale Datentypen sind kovariant (es gibt keine Zuweisung).***

Kovarianz und Kontravarianz in Scala

Funktions-Parameter/-Resultate verhalten sich wie linke/rechte Seite der Zuweisung:

- Funktionsresultate verhalten sich kovariant
- Funktionsparameter verhalten sich kontravariant

Definition (Scala):

- Ein kovarianter Typparameter (**+T**) steht nur in kovarianten Positionen.
- Ein kontravarianter Typparamert (**-T**) steht nur in kontravarianten Positionen.

Beispiel:

```
trait Function1[-T,+R] { def apply(arg: T): R }
```

Mit einer Funktion g: T => T ist erlaubt:

```
val f1: T => O = g  wir können das Ergebnis einem Obertyp zuordnen
val f2: U => T = g  wir können die Parametertypen einschränken
```

```
val f: U1 => O2 = g: O1 => U2
```
mit U1 Unterkl. von O1, U2 Unterkl. von O2

Ein und derselbe Parameter kann nicht ko- und kontravariant zugleich sein!

In diesem Fall ist der Parameter nichtvariant (T)

Unterschiedliche Verwendung je nach Varianz

Invarianz:

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

Kovarianz:

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

Kontravarianz:

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

Beispiele

```
abstract class Stack[T] {
    def push(T x): Unit
    def pop(): T
}

abstract class Option[+T] {
    def get: T
}

abstract class OutputChannel[-Msg] {
    def send(msg: Msg)
}

abstract class Function1[-T,+R] {
    def apply(x: T): R
}
```

Zu beachten:

```
class List[+A] {
    def map[B](f: A => B): List[B] // Umdrehung von +/-
```

Der Ergebnistyp von f steht in kontravarianter Position!
Lösung: Von A losgelöster Parameter. Er kann beliebig sein, und steht damit in keiner Beziehung zu A!

Untere Typgrenze

B >: A bedeutet, dass der neue Parameter B ein Obertyp von A ist

Verwendung: Beschreibung eines Ergebnistyps

Hier muss das Ergebnis ein gemeinsamer Obertyp von A und B sein.1

```
class List[+A](val head: A, val tail: List[A]) {
    def cons[B >: A](h: B): List[B] = new List(h, this)
```

Obere Typgrenze

B <: A bedeutet, dass der neue Parameter B ein Untertyp von A ist

Verwendung: Dem Compiler wird Typinformation mitgegeben.

```
def maxObject[A <: Ordered[A]](xs: List[A]): A = {
   if (tail.isEmpty) head
   else {
       val m = maxObject(tail)
       if (m >= head) m else head
   }
```

(in Scala gibt es noch weitergehende Möglichkeiten)

Typparameter in Java

Wie in Scala, sind die Typparameter in Java nichtvariant.

In Java gibt es keine Varianzdeklaration in der Klassendeklaration. Stattdessen findet diese beim Gebrauch (Deklaration von Variablen, Vererbung) statt.

Diese Lösung ist leistungsfähiger als die Scala-Variante (Scala bietet diese Möglichkeit aber auch).

Aber sie verlagert den Umgang mit Typparametern vom Bibliotheksentwickler zum Bibliotheksnutzer.

Ergebnis: die meisten Java-Entwickler fühlen sich überfordert und ignorieren die Parameter!

Nach oben beschränkte Typparameter

Syntax:

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

Gibt an, dass der Typparameter sich auf eine Unterklasse einer Klasse bezieht und die angegebenen Inferfaces 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 <: Numeric[T]` *)*

Wildcards

Mit Wildcards wird die Varianz bei der Verwendung festgelegt.
Wildcards dienen der ungenauen Angabe von aktuellen Typparametern. Damit lassen sich alle vier Möglichkeiten der Verträglichkeit ausdrücken.

- Der unbeschränkte Wildcard `?` sagt nichts über den Typ 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..

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..

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) { ... }
}
```

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]; // Warnung
```

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

Ebenso nicht möglich / Warnung:

Java:

```
if (x instanceof T)  a = (T) x;
```

Scala:

```
case x: T => ...
```

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.

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

Bestimmte Compilerwarnungen sollen für das folgende Element unterdrückt werden.

```
@IgnoreWarnings("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"})
```

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-Implentierung:

```
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.MITTWOCH);
```

`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 { . . . }
```