Hilfreiche Erweiterungen

Local-Variable Type Inference

Beispiel

```
public class TestKlasse {
  public static void main(String[] args) {
    var var = new Manager("Bill Gates", 100000.00, 50000.00);
    var a = new Angestellter("Meier", 5000.00);
    var b = a;
    var c = new Angestellter("Meier", 5000.00);
```

Local-Variable Type Inference

Seit Java 10 ist es erlaubt, bei der Deklaration von Variablen anstelle des Typs einfach nur `var` zu schreiben.

Local-Variable Type Inference

- Funktioniert nur in lokalen Scopes, also innerhalb von Methoden oder Blöcken
- Wichtig: Funktioniert nicht bei Klassenvariablen oder Methodenparametern
- `var` ist kein neues Keyword!

$\Rightarrow$ Rückwärtskompatibilität ist also gewährleistet

$\Rightarrow$ Java bleibt eine streng typisierte Sprache, keine dynamische Typisierung wie in JavaScript

Beispiel

```
var var = "a variable named var";
System.out.println(var);
```

Beispiele, was nicht geht ...

Beispiel

```
var nil = null;
// kompiliert nicht, von null kann kein Typ abgeleitet werden
var uninitialized;
// kompiliert nicht, die Initialisierung muss immer zusammen mit der Deklaration erfolgen
var x = 0, y = 1;
//kompiliert nicht, Mehrfachdeklaration und -initialisierung ist nicht moeglich
```

Pattern Matching bei `instanceof` (Vorsicht)

Beispiel

```
if (obj instanceof Person) {
  final Person person = (Person) obj;
  // ... Zugriff auf person...
}
```

Seit Java 16 durch sog. „Binding-Variable“

```
if (obj instanceof Person person) {
  // Hier kann man auf die Variable person direkt zugreifen
}
else {
  // Hier kein Zugriff auf person
  System.out.println(obj.getClass());
}
```

Generics - Klassen und Methoden

Idee und Motivation

Idee

Die Grundidee generischer Klassen bzw. Schnittstellen ist es, einzelne Datentypen bei der Programmierung einer Klasse nicht von vornherein festzulegen.

Beispiel

Speicherung eines Datentriplets (a,b,c) in unterschiedlichen Kontexten:

- TripletPoint-Klasse
- TripletRectangel-Klasse
- TripletString-Klasse

$\Rightarrow$ Generische Klasse, die den Datentyp der zu verwaltenden Daten variabel lässt

Beispiel

```
public class Triplet<T> {
  public T a, b, c;
  public Triplet(T data1, T data2, T
      data3)
  {
    a = data1;
    b = data2;
    c = data3;
  }
}
```

```
import java.awt.Point;
public class TestTriplet {
  public static void main(String[] args)
  {
    var t1 = new Triplet<>("12", "abc",
        "...");
    var t2 = new Triplet<>(new Point(1,
        1), new Point(2, 2), new Point
        (3, 3));
  }
}
```

Wichtig

Generische Klassen und Methoden können nur Objekte verarbeiten
$\Rightarrow$ Nutzung von Wrapper-Klassen bei elementaren Datentypen

Beispiel

```
var t3 = new Triplet<>(1.0, 2.8, 83.2);
Double tst3 = t3.a;
```

Deklaration generischer Klassen und Schnittstellen

Konvention

```
class MyClass<T> {}
class MyDictionary<E, N> {}
```

T für Typ, E für Element, N für Number, aber auch beliebige Großbuchstaben möglich
$\Rightarrow$ Bitte im Praktikum und Klausur verwenden

T kann als Klassenname verwendet werden

Beispiel

```
public class MyClass<T> {
  private T var; // Variable vom Typ T
  // Konstruktor, erwartet einen Parameter vom Typ T
  public MyClass(T data) {
  }
  // Methode, gibt ein Ergebnis vom Typ T zurueck
  public T calc(){
    T result = null;
    return result;
  }
}
```

Typeinschränkungen

Beispiel (einfach)

```
class MyClass<T extends Comparable<T>> {...}
```

Generischer Typ akzeptiert nur Klassen, die von einer Basisklasse abgeleitet werden bzw. Schnittstellen implementieren (hier Schnittstelle Comparable<T>)

Beispiel (mehrfach)

```
class MyPolygon <T extends Point & Comparable > {...}
```

Auch mehrere Typeinschränkungen sind möglich.
Immer zuerst die Basisklasse angeben, dann die Schnittstelle.

Generische Schnittstellen und Vererbung

Interfaces

```
interface MyInterface<T> {...}
```

Es gelten dieselben Vererbungsregeln wie für normale Klassen

```
public class HashMap<K, V> extends AbstractMap <K, V> implements Map<K, V>, Cloneable,
    Serializable{...}
```

Deklaration generischer Methoden

Auch in nicht-generischen Klassen möglich

- Einschränkung des Typ möglich (extend)
- Typparameter muss vor Rückgabewert bzw. void angegeben werden
- Rückgabewert kann auch generisch sein
- Beim Konstruktor wird der generische Typ direkt vor dem Methodennamen angegeben

Generische Methode

Beispiel

```
public class TestGenericMethods {
  public static void main(String[] args) {
    String s = max("abc", "efg");
    Integer n = max(123, 456);
    System.out.println(s);
    System.out.println(n);
  }
  public static <T extends Comparable<T>> T max(T a, T b) {
    if (a.compareTo(b) > 0)
      return a;
    else
      return b;
  }
}
```

Ok?

Beispiel

```
  var t1 = new Triplet<>("12", "abc", "...");
  var t2 = new Triplet<>(new Point(1, 1), new Point(2, 2), new Point(3, 3));
  var t3 = new Triplet<>(1.0, 2.8, 83.2);
  outputTriplet(t1);
  outputTriplet(t2);
  outputTriplet(t3);
}
public static void outputTriplet(Triplet<String> t) {
  System.out.format("[%s, %s, %s]\n", t.a, t.b, t.c);
}
```

Weil ...

Problem

Explizite Angabe des Typs bedeutet Verlust der Flexibilität

```
public static void outputTriplet(Triplet<String> t) {
```

Problem

Generische Typinformation gehen beim Kompilieren verloren. Daher nicht erlaubt:

```
public static void outputTriplet(Triplet<String> t) {
  System.out.format("[%s, %s, %s]\n", t.a, t.b, t.c);
}
public static void outputTriplet(Triplet<Integer> t) {
  System.out.format("[%s, %s, %s]\n", t.a, t.b, t.c);
}
```

Wildcard-Parameter

Lösung

```
public static void outputTriplet(Triplet<?> t) {
  System.out.format("[%s, %s, %s]\n", t.a, t.b, t.c);
}
```

Neues Problem

Wildcards mit Regeln

- Große Freiheit durch Wildcard-Parameter
- Problem: Methoden müssen mit jedem denkbaren generischen Typ zurechtkommen

Wildcards einschränken durch

- Upper Bounded Wildcards
- Lower Bounded Wildcards

Upper Bounded Wildcards

Syntax

```
<? extends Xxx>
```

Nur Typen, die der Klasse Xxx bzw. der Schnittstelle Xxx entsprechen

Beispiel

```
MyClass<? extends A>
```

Nur Typen der Klasse A bzw. davon abgeleiteten Klassen

Beispiel

```
MyClass<? extends Comparable<?>>
```

Klassen, welche die generische Schnittstelle Comparable implementieren

Lower Bounded Wildcards

Syntax

```
<? super Xxx>
```

Als Datentyp nur Xxx sowie deren Basisklassen

Beispiel

```
MyClass<? super Double>
```

Nur Typ Double sowie allgemeiner (Number, Object)

Beispiel

Methode zur Berechnung der Summe eines Triplets mit dem Typ Double, Integer, BigDecimal

```
public static double sumTriplet(Triplet<? extends Number> t) {
  return t.a.doubleValue() + t.b.doubleValue() + t.c.doubleValue();
}
```

Beispiel

Methode zur Veränderung der Triplet-Werte mit dem Typ Double, Number, Object

```
public static void changeTriplet(Triplet<? super Double> t, Double x) {
  t.a = x;
  t.b = x * x;
  t.c = x * x * x;
}
```

Arrays

Nicht erlaubt

```
obj = new Classname<?>
```

Aber: Erzeugung eines Arrays zur Speicherung von Objekten einer Klasse mit unterschiedlichen Typen zulässig

```
Triplet<?> trip = new Triplet<?>[3];
trip[0] = new Triplet<>("1", "2", "x");
trip[1] = new Triplet<>(new Point(0, 0)), new Point(1, 1), new Point(1, 2));
trip[2] = new Triplet<>(1.2, 2.7, Mathi.PI);
for (var t: trip)
  outputTriplet(t)
```

Generics-Beispiel zu Comparable

Übung

Alle Objekte der Geometrie-Schnittstelle (siehe letzte Praktikumsaufgabe sollen verglichen und sortiert werden)
$\Rightarrow$ Nutzung der Comparable-Schnittstelle

```
public interface Comparable<T> {
  public int compareTo(T obj);
}
```

Lösung

Erweiterung Interface

```
public interface Geometrie extends
    Comparable<Geometrie>{
  double berechneUmfang();
  double berechneFlaeche();
}
```

Klasse Kreis

```
@Override
public int compareTo(Geometrie other) {
double flaecheThis = this.
    berechneFlaeche();
double flaecheOther = other.
    berechneFlaeche();
if(flaecheThis == flaecheOther)
  return 0;
else if(flaecheThis > flaecheOther)
  return 1;
else
  return -1;
}
```

Lösung

Rechteck

```
@Override
public int compareTo(Geometrie other) {
  return Double.compare(this.
      berechneFlaeche(), other.
      berechneFlaeche());
}
```

Main

```
Arrays.sort(geos);
for (Geometrie g : geos) {
  System.out.println("Flaeche:" + g.
      berechneFlaeche());
}
```

Comparable versus Comparator

flexibler Vergleich, keine Festlegung auf EIN Sortierkiterium

```
Arrays.sort(geos, new Comparator<Geometrie>() {
  @Override
  public int compare(Geometrie g1, Geometrie g2) {
    return ((Double) g1.berechneUmfang()).compareTo(g2.berechneUmfang());
  }
});
```

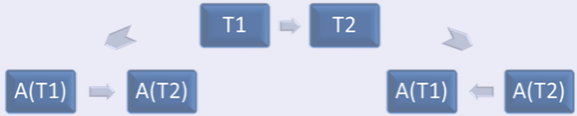
Zusammenfassung

- Generics zur Definition von Klassen, bei denen der Typ von manchen Feldern variieren kann.
- Generics können auch in Interfaces auftreten (vgl. Collection Framework)
- Einschränkung von generischen Typen
- Comparable Interface zur Definition von Methoden und Klassen, bei denen die Typen generisch sind, aber eine Vergleichsoperation verwendet werden soll.
- Comparable ist oft eine Vorbedingung im Collection Framework.

Theorie

Kovarianz, Kontravarianz, Invarianz

- Kovarianz und Kontravarianz ist die von Typen abhängige Kompatibilität
- Ein von T abhängiges $A(T1)$ ist kovariant, wenn aus der Kompatibilität von $T1$ zu $T2$ die (Typ)Kompatibilität von $A(T1)$ zu $A(T2)$ folgt
- Wenn aus der Kompatibilität von $T1$ zu $T2$ die Kompatibilität von $A(T2)$ zu $A(T1)$ folgt, dann ist der Typ $A(T)$ kontravariant
- Wenn aus der Kompatibilität von $T1$ zu $T2$ keine Kompatibilität zwischen $A(T1)$ und $A(T2)$ folgt, dann ist $A(T)$ invariant



Theorie

Kovarianz bei Arrays

Weil Arrays kovariant sind und Integer von Number abgeleitet ist, ist Integer[] eine Ableitung von Number[]

```
Number[] a = new Integer[2];
a[0] = new Integer( 1 );
a[1] = new Double( 3.14 ); //Laufzeitfehler
```

Theorie

Invarianz bei Arrays

- Obwohl Integer von Number abgeleitet ist, ist ArrayList<Integer> keine Ableitung von ArrayList<Number>

```
ArrayList<Number> a = new ArrayList<Integer>(); //Kompilierfehler
```

- Die Ableitungsbeziehung zwischen Typargumenten überträgt sich nicht auf generische Klassen, es gibt keine Kovarianz bei Generics.
  $\Rightarrow$ Typsicher bei Generics gewährleistet, anders als bei Arrays
  $\Rightarrow$ Mehr nach der Vorlesung zu Collections

Theorie

Wie werden Generics „übersetzt“

- Java-Compiler übersetzt Generics direkt in Bytecode durch „type erasure“
- Bei der Übersetzung per Type Erasure werden die Typparameter eines Typs oder einer Methode entfernt, so dass zur Laufzeit parametrisierte Typen nicht mehr von regulären Typen unterschieden werden können

Wie werden Generics „übersetzt“

Original

```
public class Box<T> {
  private T contents;
  public Box(T cont) {
    contents = cont;}
  public T getContents() {
    return contents;}
  public void setContents(T o) {
    contents = o;}
  public static void main(String[] args)
      {
    Box<String> b = new Box<String>("
        Apple");
    String s = b.getContents();
    System.out.println(s);
    Box<Integer> b1 = new Box<Integer>(
        new Integer(3));
    int i = b1.getContents();
    System.out.println(i);}}
```

Type Erasure

```
public class BoxWG {
  private Object contents;
  public BoxWG(Object cont) {
    contents = cont;
  }
  public Object getContents() {
    return contents;
  }
  public void setContents(Object o) {
    contents = o;
  }
  public static void main(String[] args)
      {
    BoxWG b = new BoxWG("Apple");
    String s = (String)b.getContents();
    System.out.println(s);}}
```