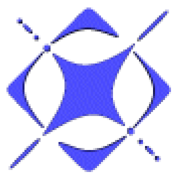
Fachhochschule
Münster University of
Applied Sciences

Java *Generics*

Ein kleines Tutorium zur Verwendung von *Generics*.

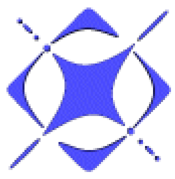
Prof. Dr. Nikolaus Wulff

Java Generics



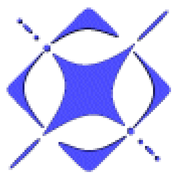
- Seit 2004 stehen mit der Auslieferung von J2SE 5.0 (dem JDK 1.5) ***Generics*** zur Verfügung.
- Es lassen sich typsichere Methoden und mit Typen parametrisierte Klassen und Schnittstellen entwickeln.
- Die Java Collection und Reflection APIs wurde konsequent auf Generics umgestellt und die Gefahr eines falschen Cast ist deutlich geringer geworden.
- Oberflächlich ist die Schreibweise ähnlich zu C++ Templates. Die Umsetzung jedoch grundverschieden.
- Die Implementierung von eigenem Generics Code erfordert einige Umsicht.

Generischer Typbezeichner



- Generische Methoden, Klassen und Schnittstellen enthalten einen oder mehrere formale Typbezeichner.
- Typ Informationen werden vom Compiler während des Übersetzungsvorgangs ausgewertet und erlauben eine strikte Überprüfung der Typumwandlungen.
- Formale Typbezeichner $T_1,\ldots,T_n$ einer Klasse ***C*** werden mit spitzen Klammern notiert als C<$T_1,\ldots,T_n$>.
  - Beispiele sind List<T> oder Map<K,V>.
- Eine konkrete Realisierung wird mit dem aktuellem, tatsächlichem Typ instantiiert.
  - Map<String,Double> amap = new HashMap<String,Double>();

Generische Klassen



- Es lassen sich sowohl generische Klassen und Schnitt-stellen als auch Vererbungsbeziehungen definieren.

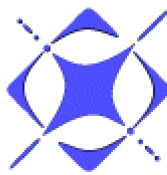
```
interface MyInterface<A,B> {
    A foo(B b);
}
```

```
class MyClass<C,D> {
    C   any;
    D[] many;
}
```

```
class ImplementingClass<E,F> implements MyInterface<E,F> {
    MyClass<F,Integer> aMember;
    public E foo(F f) {
        E e = null;
        // calculate e = foo(f)
        return e;
    }
}
```

- Die so definierten generischen Typen gelten für den gesamten Scope der Klasse und Schnittstelle.

Generische Methoden



- Auch einzelne Methoden lassen sich gezielt mit Generics beschreiben.

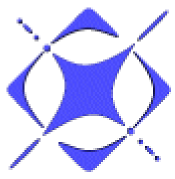
```
class StandardClass {

    public <V> V bar(V x) {
        V y = null;
        // calculate y = bar(x)
        return y;
    }
    public Double foo(Double x) {
        return bar(x); // compiler checks V == Double
    }
}
```

Generischer Typ V der Methode

- Auch unterschiedliche Typen sind möglich:

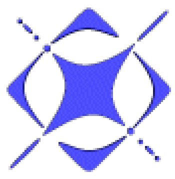
```
    public <V,R> R bar(V x) {
        R y = null;
        // calculate y(x)
        return y;
    }
```

Gebundener Typ



- Gibt es Restriktionen an den Typen T, so können diese explizit angegeben werden: <T extends A>.
- Diese Art der Restriktion heisst ***Upper Bound***.
- T ist vom Typ A oder eine Subklasse von A oder T implementiert die Schnittstelle A .
- Es ist auch möglich mehrere Restriktionen per **&** zu verknüpfen. Hierbei muss eine Klassenvererbung als erstes stehen, dann folgen mögliche Schnittstellen.
- Es seien A eine Klasse und B und C ein Interface dann wäre die Deklaration <T extends A & B & C> zulässig, jedoch <T extends B & A & C> unzulässig.

Universelle Algorithmen



- Anstatt einen Algorithmus für unterschiedliche Typen mehrfach zu duplizieren, wird eine Implementierung für alle passenden Typen angestrebt:

```
public void sort(Long[] a) {
    // sort long array
}
public void sort(Float[] a) {
    // sort float array
}
public void sort(Double[] a) {
    // sort double array
}
```

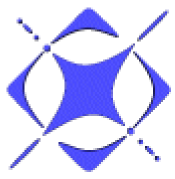
Copy & Paste Programmierung durch Umbenennen der Typen!

```
Integer[] i = {2,3,1};
Double[] d  = {2.0,3,5};
sort(i);
sort(d);
```

Noch kein Generic, aber eine Idee!

```
public void sort(Number[] a) {
   // sort all kinds of numbers:
   // Integer,Long, Float, Double
}
```

Universelle Rückgabewerte?



- Ein Problem entseht, wenn der Algorithmus statt void ein typabhängiges Ergebnis liefern soll:

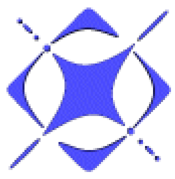
```
public Long[] sort(Long[] a) {
    // sort array
    return a;
}
public Float[] sort(Float[] a) {
    // sort array
    return a;
}
...
```

Es kann nur der „kleinste gemeinsame Nenner“ als Typ zurückgeliefert werden...

```
public Number[] sort(Number[] n) {
    // sort all kinds of numbers:
    // Integer,Long, Float, Double
    return n;
}
```

```
Integer[] i = {2,3,1};
Number[]  n;
n = sort(i);                // OK
i = sort(i);                // compiler error
i = (Integer[]) sort(i);    // ok but dangerous cast!!!!
```

Generischer Algorithmus



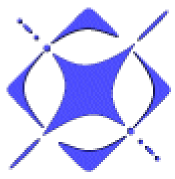
- Mit Generics lässt sich die Aufgabe elegant lösen.

```
public <T extends Number> T[] sort(T[] n) {
   // sort all kinds of numbers:
   // Integer,Long, Float, Double
   return n;
}
```

- <T `extends` Number> spezifiziert alle Unterklassen von Number als mögliche Belegungen für T.
- Der Compiler sichert zu, dass immer der richtige Ergebnistyp vorliegt – ohne Casts!

```
Integer[] i = {2,3,1};
Number[]  n = ...;
n = sort(n);  // OK sort Numbers
i = sort(i);  // OK sort Integers
n = sort(i);  // OK sort Integers assign Numbers
```

Generische Schnittstellen

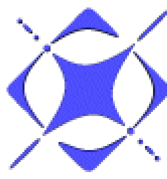


- Lassen sich nur Subklassen von Number sortieren?
- Eine der obersten Regeln der Objektorientierung lautet „programmiere gegen Schnittstellen“!

```
public <T extends Comparable<T>> T[] sort(T[] n) {
    // sort all kinds of comparables:
    // Integer,Long, Float, Double, Intervals, Strings
    return n;
}
```

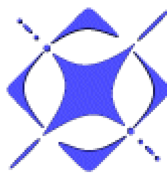
- `<T extends Comparable<T>>` erlaubt alle Typen mit einer Ordnungsrelation $x<y$ als Belegungen für T.
- Diese generische Funktionsschnittstelle ist wesentlich universeller einsetzbar.
- Aber Achtung: Sie hat noch einen Schönheitsfehler...

Typ Erasure



- Die Typ Informationen existieren nur zur Übersetzungszeit zum Überprüfen der Zuweisungen.
- Generics der Form <T> werden im Byte-Code als *java.lang.Object* referenziert und Generics der Form <T extends ***C***> als Realisierung von ***C***.
- Dies bedingt, dass keine generischen Arrays zur Laufzeit erzeugt werden können.
  - **T**[] array = new **T**[10] macht keinen Sinn, was ist **T**?
    - **T**[] array = (**T**[]) new Object[10] funktioniert mit Warnung!
  - Statt Arrays generische Container Klassen verwenden.
- Aus den fehlenden Typinformationen folgen einige Konsequenzen bei der Programmierung mit Generics.

Alle <T> sind gleich...

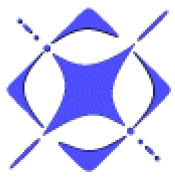


- Alle Typinformationen sind im Byte-Code gelöscht. Eine Disassemblierung mit Hilfe von javap zeigt dies.

```
public void unsaveList() {
    List<Integer> iList = new ArrayList<Integer>();
    List<Number>  nList = new ArrayList<Number>();
    iList.add(25);
```

```
public void unsaveList();
  Code:
   0:    new     #65; //class java/util/ArrayList
   3:    dup
   4:    invokespecial   #67; //Method java/util/ArrayList."<init>":()V
   7:    astore_1
   8:    new     #65; //class java/util/ArrayList
   11:   dup
   12:   invokespecial   #67; //Method java/util/ArrayList."<init>":()V
   15:   astore_2
   16:   aload_1
   17:   bipush  25
```

Seiteneffekte



- Generics ermöglichen Typsicherheit,

```
List<Integer> iList = new ArrayList<Integer>();
List<Number>  nList = new ArrayList<Number>();
iList.add(2);                          // OK autoboxing adds Integer
nList = iList;                         // compile error
nList = (List<Number>)iList;           // compile error
```

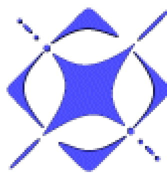
- solange sie nicht explizit umgangen

```
Object ref = iList;
nList = (List<Number>)ref;             // warning but compiles
nList.add(3.14);                       // OK adds a Double!!!
```

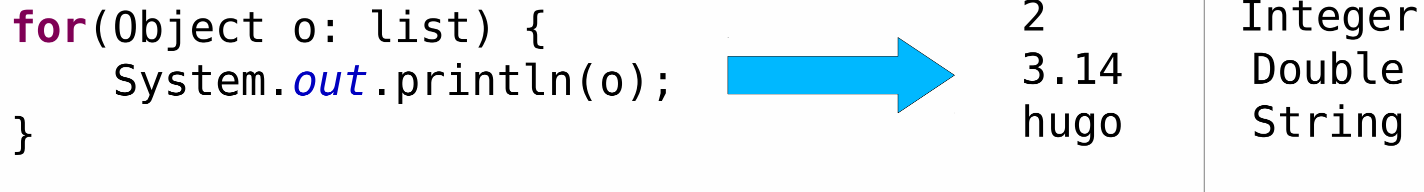
- oder mit „altem Code“ gemischt werden!

```
List   list;
list = iList;                          // NO warnings!
list.add("hugo");                      // warning
```

@SuppressWarnings

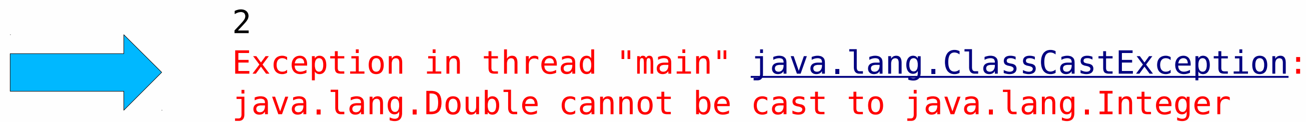


- Durch das Einfügen eines Double in die Number Liste und einer Zeichenkette in die untypisierte Liste, ist die Integer Liste korrumpiert worden!

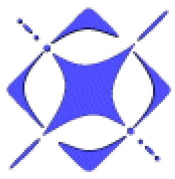


- Das Auslesen der Integer Liste schlägt fehl...

```
for(Integer i: iList) {
    System.out.println(i);
}
```



Wildcards

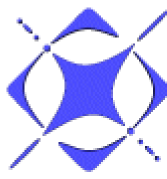


- Wenn nicht bekannt ist, von welchem generischem Typ z.B. eine Liste ist, so wird dies mit List<?> angegeben. Die resultierende Liste ist dann „read-only“ und nicht mehr beschreibbar.

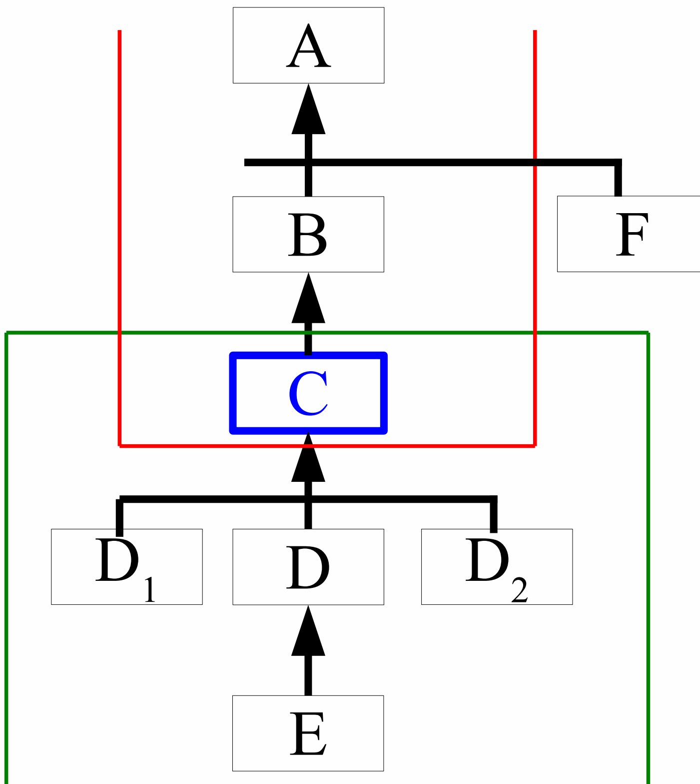
```
List<Integer> iList = new ArrayList<Integer>();
List<?>       wlist;
// ...
wlist = iList;                  // OK, no warnings
wlist.add(1);                   // compile error
wlist.add(3.4);                 // compile error
wlist.add("hugo");              // compile error
wlist.add(null);                // erlaubt!
```

- „null“ ist der einzige Typ der immer passt.
  - Möglicherweise wird eine NullPointerException geworfen.

Wildcards und Vererbung



- Es gibt zwei mögliche Vererbungseinschränkungen.
- Kovariant: <? extends *C*> „*upper bound*“
- Kontravariant: <? super *C*> „*lower bound*“

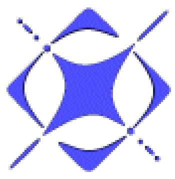


covariant erasure: => C

contravariant erasure: => A => Object

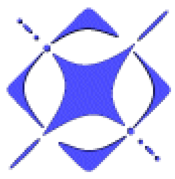
```
List<Object>  oList = new ArrayList<Object>();
List<Integer> iList = new ArrayList<Integer>();
List<Double>  dList = new ArrayList<Double>();
List<Number>  nList = new ArrayList<Number>();

List<? extends Number> covar = iList;
List<? super Number>  contra = oList;

contra.add(1);
contra.add(3.5);
contra.addAll(dList);

contra.add(new Obect()); // compile error
covar.add(1)             // compile error
```

Wildcard Input Output



- Kovariante Wildcards <? extends ***C***> bieten sich an bei **lesendem Zugriff** .
  - Beim Lesen gilt als erasure der upper bound ***C***.
  - Schreiben (außer null) ist nicht möglich.
- Kontravariante Wildcards <? super ***C***> bieten sich an bei **schreibendem Zugriff**.
  - Alle Typen unterhalb des lower bound ***C*** sind einfügbar.
  - Beim Lesen ist als erasure der Typ immer Object.
- Die typische Frage bei der Aufteilung kovariant oder kontravariant richtet sich danach soll generisch gelesen oder geschrieben werden.

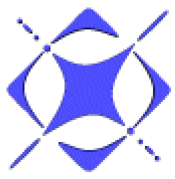
Producer-Consumer Beispiel



- Produzenten sollen „Dinge“ einer Menge hinzufügen und Konsumenten diese verbrauchen können.
- P benötigt schreibenden und C lesenden Zugriff:

```
public interface Producer<T> {
    /**
     * Write objects of type T to the given set.
     * @param set to fill
     */
    void produce(Collection<? super T> set);
}

public interface Consumer<T> {
    /**
     * Consume objects of type T from the set.
     * @param set to consume
     */
    void consume(Collection<? extends T> set);
}
```

Zufälliger Produzent

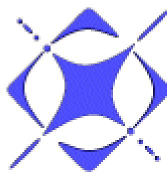


- Ein Produzent erzeugt per Zufall Kreise, Rechtecke, etc. Diese werden dann an ein Display übergeben...

```
public class ShapeRandomizer implements Producer<Shape>  {
     private Random rnd = new Random();
    @Override
    public void produce(Collection<? super Shape> shapes) {
        Shape shape;
        int x,y,w,h,r,type;   // fill x,y,w,h random not shown here...
        for(int i=0;i<10;i++) {
            type = (int) Math.rint(0.5+NUM_TYPES  *rnd.nextDouble());
            switch(type) {
            case 1:
                shape = new Circle();
                break;
            case 2:
                shape = new Rect();
                break;
            case 3:
                shape = new Triangle();
                break;
             default:
                throw new IllegalArgumentException("unknown type " + type);
            }
            shapes.add(shape);
        }
}
```

Producer kann beliebige Shapes einfügen...

Konkreter Konsument



- Ein Display zeigt beliebige Shape Objekte an.

```
public interface Shape {
    int getX();
    int getY();
    void setX(int x);
    void setY(int y);
    void draw(Display display);
}
```

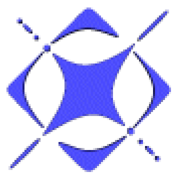
- Es gibt viele Ausprägungen, Circle, Triangle, Rect,...

```
public class Display implements Consumer<Shape> {

    /* (non-Javadoc)
     * @see de.lab4inf.shapes.Consumer#consume(java.util.Collection)
     */
    @Override
    public void consume(Collection<? extends Shape> shapes) {
        for(Shape s: shapes) {
            System.out.printf("Display shape at (%3d,%3d) -> draw ",
                              s.getX(),s.getY());
            s.draw(this);
        }
    }
```

Konsument kann beliebige Shapes entnehmen und verwenden...

```
}
```

Type Erasure <? extends C>

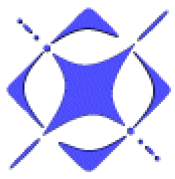


```
public void consume(java.util.Collection);
  Code:
   0:   aload_1
   1:   invokeinterface #20,  1; //InterfaceMethod java/util/Collection.iterator:()Ljava/util
   6:   astore_3
   7:   goto    64
   10:  aload_3
   11:  invokeinterface #26,  1; //InterfaceMethod java/util/Iterator.next:()Ljava/lang/Objec
   16:  checkcast       #32; //class de/lab4inf/shapes/Shape
   19:  astore_2
   20:  getstatic       #34; //Field java/lang/System.out:Ljava/io/PrintStream;
   23:  ldc     #40; //String Display shape at (%3d,%3d) -> draw
   25:  iconst_2
   26:  anewarray       #3; //class java/lang/Object
   29:  dup
   30:  iconst_0
   31:  aload_2
   32:  invokeinterface #42,  1; //InterfaceMethod de/lab4inf/shapes/Shape.getX:()I
   37:  invokestatic    #46; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
```

```
for(Shape s: shapes) {
```

- Der disassemblierte Code zeigt, dass bei kovarianten Wildcards nicht der Typ Objekt, sondern der *Upper Bound* ***C*** im Bytecode verwendet wird.

More Lower Bounds



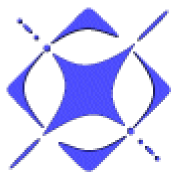
- Die Schnittstelle von sort wurde definiert als:

```
public <T extends Comparable<T>> T[] sort(T[] n)
```

- Leider ist dies zu restriktiv. Was ist mit einer einer Klassenhierarchie von vergleichbaren Foo-Objekten?

```
class Foo implements Comparable<Foo> {
        …
}
class Bar extends Foo {
        …
}

List<Foo> foos = ...;
List<Bar> bars = ...;

foos = sort(foos);   // Ok will be sorted
bars = sort(bars);   // compile error
```

Generic Subtyping

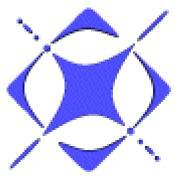


- Die Klass Bar erbt oder überlädt Comparable<Foo> der Elternklasse. Jedoch List<Bar> passt nicht zur vorgegeben ***sort*** Signatur List<Bar>≠ List<Foo>!
- Eine zusätzliche Schnittstelle Comparable<Bar> kann nicht für Bar deklariert werden, da wg. Type erasure die ***compareTo*** Methoden nicht unterscheidbar sind...
- Eine kontravariante Comparable<? super T> Signatur für die ***sort*** Methode erlaubt es auch Subklassen von Foo als gültige Typen mit einzubeziehen:

```
public <T extends Comparable<? super T>> T[] sort(T[] n)
```

- Daher immer `Comparable<? super T>` codieren...

Zusammenfassung



- Zwar ist ein Circle ein Shape, aber eine List<Circle> oder eine List<Rect> darf nicht „überall“ verwandt werden, wo eine List<Shape> erwartet wird, der Compiler kann u. U. keine Typsicherheit zusichern.
- Für jeder Subklasse eine eigene *display* Methode zu schreiben ist kein Ausweg, daher wird der Typ mit ko- und kontravarianten Wildcards geöffnet.
- Der Einsatz von Generics kostet einiges an Aufmerksamkeit, aber wenn sie erst einmal verstanden sind, so bieten sie eine mächtige Möglichkeit für allgemeine, modulare Softwareentwürfe.