



# Generisches Programmieren

Thomas Röfer

Generische Klassen und Interfaces

Generische Typen

Typebounds

Wildcard-Typen

Übersetzung generischer Klassen

Grenzen generischer Typen

Polymorphe Methoden

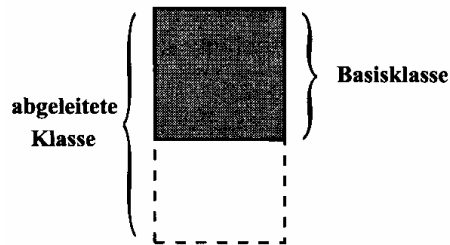


# Rückblick „Vererbung“

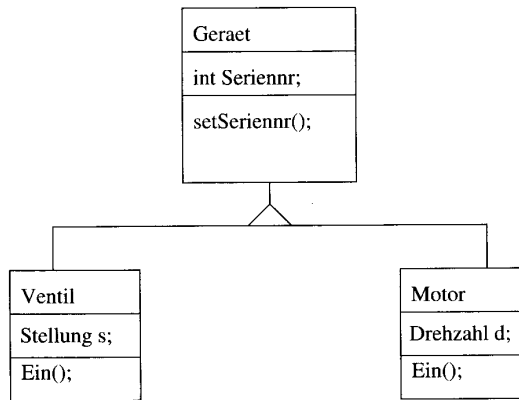
## Pakete

```
java.lang.System.out.println("Hallo")
Paket Paket Klasse Attribut Methode
```

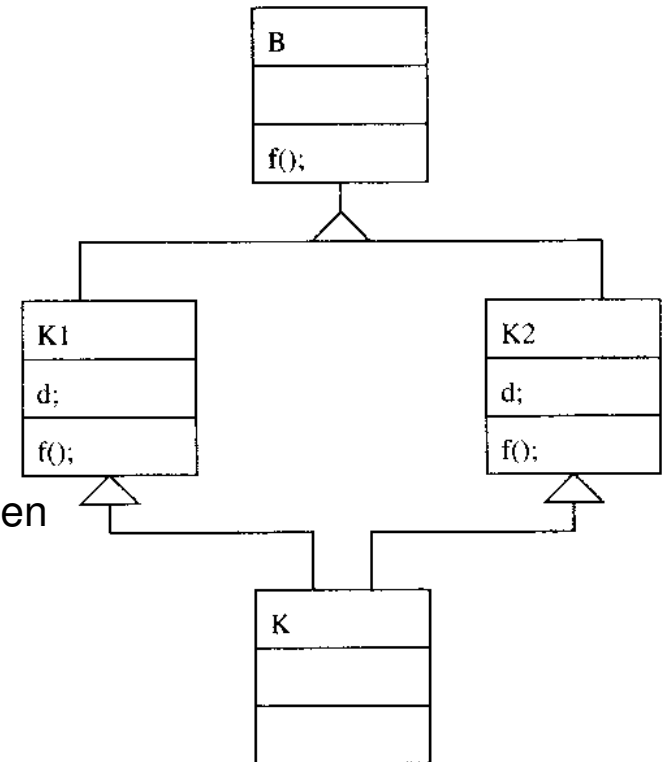
## Speicherlayout



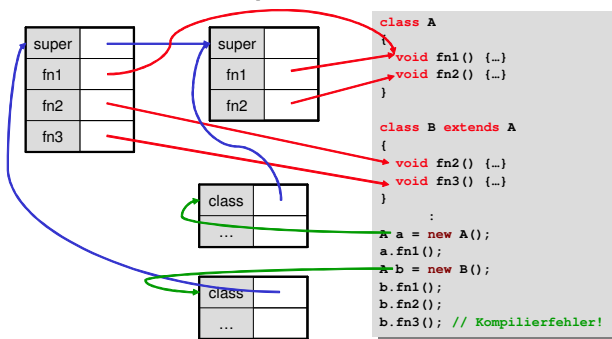
## Vererbung



## Mehrfaches Erben



## Frühes/spätes Binden



## Abstrakte Klassen / Schnittstellen

```

abstract class Gerat {
  int seriennummer;
  abstract void ein();
}

interface Printable {
  void print();
}
  
```

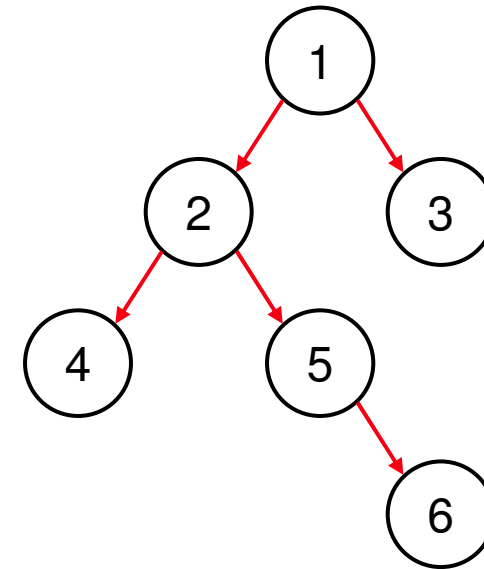


# Motivation

```
class Node
{
    private Object info;
    private Node left, right;

    Node(Object i) {info = i;}
    Node(Object i, Node l, Node r)
        {info = i; left = l; right = r;}
    Object getInfo() {return info;}
    Node getLeft() {return left;}
    Node getRight() {return right;}
    void setInfo(Object i) {info = i;}
}
```

Problem: Zugriff nicht Typ-sicher!



```
Node n1 = new Node("foo");
Node n2 = new Node(3.14);
Node n0 = new Node(23, n1, n2);

String s = (String) n1.getInfo();
```



# Generische Klassen

## ▸ Definition

- Generische Klassen haben Typvariablen, die im Klassenrumpf verwendet werden können
  - `class Node<T> { T info; ... }`
- Typvariablen können (fast) wie ein normaler Typ benutzt werden
- Mehrere Typvariablen sind möglich

```
class Pair<T, U>
{
    private T first;
    private U second;
    Pair(T t, U u) {first = t; second = u;}
    T getFirst() {return first;}
    U getSecond() {return second;}
}
```

```
class Node<T>
{
    private T info;
    private Node<T> left, right;

    Node(T i) {info = i;}
    Node(T i, Node<T> l, Node<T> r)
        {info = i; left = l; right = r;}
    T getInfo() {return info;}
    Node<T> getLeft() {return left;}
    Node<T> getRight() {return right;}
    void setInfo(T i) {info = i;}
}
```



# Generische Typen

## ▶ Generische Interfaces

```
interface Taggable<T>
{
    void setTag(T tag);
    T getTag();
}
```

```
class A implements Taggable<String>
{
    private String tag;
    public void setTag(String t) {tag = t;}
    public String getTag() {return tag;}
}
```

## ▶ Begriffe

- ▶ Eine *generische Klasse* ist eine Klassendefinition, in der unbekannte Typen durch Typvariablen vertreten sind. Für *generische Interfaces* gilt entsprechendes
  - ▶ `class Node<T> { T info ...`
- ▶ Ein *generischer Typ* ist eine Typangabe, in der eine generische Klasse mit einem konkreten Typargument versehen wird
  - ▶ `Node<Integer> n = new Node<Integer>(23);`
  - ▶ `Pair<Integer, String> p = new Pair<Integer, String>(28359, "Bremen");`



# Typebounds

## ▶ Motivation

- ▶ Manchmal sollen nicht alle Typen für die Belegung der Typvariablen zulässig sein, z.B. wenn es spezielle Anforderungen an die Typargumente gibt
- ▶ Diese können durch *Typebounds* festgelegt werden
  - ▶ `class Node<T extends Number> { ...`

## ▶ Definition

- ▶ Ein Typebound bedeutet „ist kompatibel zu“
- ▶ Typebounds können Klassen und auch Interfaces sein
  - ▶ *Auch bei Interfaces wird hier extends verwendet*
- ▶ Es kann mehrere Typebounds pro Typvariable geben
  - ▶ *Sie werden durch & aufgezählt werden, z.B.*  
`class Node<T extends A & B & C> { ...`
  - ▶ *Erster Typebound kann Klasse oder Interface sein, weitere können nur Interfaces sein*
- ▶ Ein Typebound legt die Mindestanforderungen für eine Typvariable fest
- ▶ Dadurch wird auch definiert, was man von diesem Typ an Funktionalität erwarten kann
- ▶ Typebounds können wiederum die Typvariable enthalten
  - ▶ `class Node<T extends Comparable<T>> { ...`



# Wildcard-Typen

## ▶ Fragestellung

- ▶ Jede generische Klasse erzeugt viele generische Typen
  - ▶ *Node<T>* erzeugt *Node<Number>*, *Node<Integer>*, *Node<Double>*...
- ▶ Wie stehen die von einer generischen Klasse erzeugten Typen zueinander?

## ▶ Vererbung generischer Typen

- ▶ Eine Ableitungsbeziehung zwischen Typargumenten überträgt sich nicht auf die generischen Typen
  - ▶ *Node<Number> n = new Node<Integer>(23); // Fehler!*
- ▶ Dies wird als *Invarianz* bezeichnet

## ▶ Wildcard (Joker) bei Typangaben

- ▶ Generische Typen können unbestimmte Typargumente nennen
  - ▶ *Node<?> nx;*
  - ▶ *nx = new Node<String>("foo");*
  - ▶ *nx = new Node<Integer>(1);*
  - ▶ *nx = new Node<Double>(3.14);*

```
int count (Node<?> n)
{
    if (n == null)
        return 0;
    else
        return 1
            + count (n.getLeft ())
            + count (n.getRight ());
}
```



# Covarianz generischer Typen

## ▶ Motivation

- ▶ Zu einem Wildcard-Typ mit Typargument ? sind alle generischen Typen der betreffenden generischen Klasse kompatibel
- ▶ Manchmal möchte man dies durch einen sog. *Upper-Typebound* einschränken
  - ▶ `Node<? extends Number> nb;`
  - ▶ `nb = new Node<Integer>(23);`
  - ▶ `nb = new Node<Object>(new Object()); // Fehler!`

## ▶ Definition

- ▶ Mit *Upper-Typebounds* wird *Covarianz* für generische Typen eröffnet
- ▶ Allgemein gilt
  - ▶  $C<A>$  ist kompatibel zu  $C<? \text{ extends } B>$ , wenn  $A$  ist kompatibel zu  $B$

## ▶ Problem

- ▶ Auch Arrays kennen Covarianz, aber statische Typprüfung versagt
  - ▶ `Number[] a = new Integer[23];`
  - ▶ `a[0] = new Double(3.14); // ArrayStoreException`
- ▶ Lösung bei generischen Typen: nur Lesen erlaubt
  - ▶ `Node<? extends Number> nb = new Node<Integer>(23);`
  - ▶ `Number n = nb.getInfo();`
  - ▶ `nb.setInfo(3.14); // Fehler!`



# Contravarianz generischer Typen

## ▶ Motivation

- ▶ Zu einem Wildcard-Typ mit Typargument `?` sind alle generischen Typen der betreffenden generischen Klasse kompatibel
- ▶ Manchmal möchte man dies durch einen sog. *Lower-Typebound* einschränken
  - ▶ `Node<? super Number> nb;`
  - ▶ `nb = new Node<Object>(new Object());`
  - ▶ `nb = new Node<Integer>(23); // Fehler!`

## ▶ Definition

- ▶ Mit *Lower-Typebounds* wird *Contravarianz* für generische Typen eröffnet
- ▶ Allgemein gilt
  - ▶  $C\langle A \rangle$  ist kompatibel zu  $C\langle ? \text{ super } B \rangle$ , wenn  
 $B$  ist kompatibel zu  $A$

## ▶ Nur Schreibzugriff

- ▶ `Node<? super Number> nb = new Node<Object>(new Object());`
- ▶ `nb.setInfo(1);`
- ▶ `Number n = nb.getInfo(); // Fehler!`



# Zusammenfassung Varianzen

- ▶ **Invarianz**
  - ▶ Verschiedene generische Typen sind zueinander inkompatibel, unabhängig von der Kompatibilität ihrer Typargumente
- ▶ **Bivarianz**
  - ▶ Wildcard-Typen ohne Einschränkung ( $C<?>$ ) sind immer zueinander kompatibel
- ▶ **Covarianz**
  - ▶ Zu Wildcard-Typen mit Upper-Typebound ( $C<? \text{ extends } B>$ ) sind alle generischen Typen kompatibel, deren Typargument zu  $B$  kompatibel ist
- ▶ **Contravarianz**
  - ▶ Zu Wildcard-Typen mit Lower-Typebound ( $C<? \text{ super } B>$ ) sind alle generischen Typen kompatibel, zu deren Typargument  $B$  kompatibel ist

	Typ	Lesen	Schreib.	Kompatible Typargumente
<b>Invarianz</b>	$C<T>$	ja	ja	T
<b>Bivarianz</b>	$C<?>$	nein	nein	Alle
<b>Covarianz</b>	$C<? \text{ extends } B>$	ja	nein	B und abgeleitete Typen
<b>Contravarianz</b>	$C<? \text{ super } B>$	nein	ja	B und Basistypen



# Übersetzung generischer Klassen

## ▶ Ansatz

- ▶ Generische Datentypen werden in Java ausschließlich vom Compiler verarbeitet
- ▶ Das Laufzeitsystem weiß nichts von generischen Datentypen

## ▶ Übersetzung

- ▶ Mit *Type-Erasure* wird „generischer Code“ mit Typvariablen und Typargumenten auf normalen, nicht-generischen Java-Quelltext reduziert
- ▶ Der nicht-generische Java-Quelltext wird weiterverarbeitet wie bisher
- ▶ Aus jeder generischen Klasse wird eine nicht-generische Klasse generiert und in eine .class-Datei übersetzt
  - ▶ *In C++ wird dagegen jede Instanziierung einer Klasse mit Typargumenten getrennt übersetzt*
  - ▶ *Dadurch langsames Übersetzen und größere Kompilate, aber bessere Optimierungsmöglichkeiten und weniger Einschränkungen*



# Type-Erasure

- ▶ **Bei der Type-Erasure generischer Klassen werden**
  - ▶ Typ-Variablen in spitzen Klammern gelöscht
  - ▶ Vorkommen von Typvariablen mit einem oder mehreren Typebounds durch den einzigen bzw. ersten Typebound ersetzt
  - ▶ Vorkommen von Typvariablen ohne Typebounds durch *Object* ersetzt
- ▶ **Bei der Type-Erasure der Verwendung generischer Klassen werden**
  - ▶ die Typ-Korrektheit statisch geprüft (d.h. zum Übersetzungszeitpunkt)
    - ▶ *Typargumente müssen allen Typebounds genügen*
    - ▶ *generische Typen müssen auch untereinander korrekt verwendet werden, insbesondere bei Wildcard-Typen*
  - ▶ Typargumente, einschließlich Wildcards, in spitzen Klammern gelöscht
  - ▶ Typecasts eingeschoben, wo der Wert eines Typarguments benutzt wird



# Type-Erasure – Beispiel

## Generische Klasse

```
class Node<T>
{
    private T info;
    private Node<T> left, right;

    Node(T i) {info = i;}
    Node(T i, Node<T> l, Node<T> r)
        {info = i; left = l; right = r;}
    T getInfo() {return info;}
    Node<T> getLeft() {return left;}
    Node<T> getRight() {return right;}
    void setInfo(T i) {info = i;}
}
```

```
Node<String> n = new Node<String>("foo");
String s = n.getInfo();
```

## Nach Type-Erasure (Rawtype)

```
class Node
{
    private Object info;
    private Node left, right;

    Node(Object i) {info = i;}
    Node(Object i, Node l, Node r)
        {info = i; left = l; right = r;}
    Object getInfo() {return info;}
    Node getLeft() {return left;}
    Node getRight() {return right;}
    void setInfo(Object i) {info = i;}
}
```

```
Node n = new Node("foo");
String s = (String) n.getInfo();
```

# Type-Erasure – Beispiel

## Generische Klasse

```
class Node<T>
{
    private T info;
    private Node<T> left, right;

    Node(T i) {info = i;}
    Node(T i, Node<T> l, Node<T> r)
        {info = i; left = l; right = r;}

    T getInfo() {return info;}
    Node getLeft() {return left;}
    Node getRight() {return right;}
    void setInfo(T i) {info = i;}
}
```

Rawtypes lassen sich auch  
direkt benutzen, aber die  
Typsicherheit geht verloren  
(Compiler-Warnung)

```
Node<String> n = new Node<String>("foo");
String s = n.getInfo();
```

## Nach Type-Erasure (Rawtype)

```
class Node
{
    private Object info;
    private Node left, right;

    Node(Object i) {info = i;}
    Node(Object i, Node l, Node r)
        {info = i; left = l; right = r;}

    Object getInfo() {return info;}
    Node getLeft() {return left;}
    Node getRight() {return right;}
    void setInfo(Object i) {info = i;}
}
```

```
Node n = new Node("foo");
String s = (String) n.getInfo();
```



# Grenzen generischer Typen (1)

## ▶ Primitive Typargumente

- ▶ `Node<int> ni = new Node<int>(23); // Fehler!`
- ▶ Aber:  
`Node<Integer> ni = new Node<Integer>(23); // Autoboxing`

## ▶ Statische Elemente

- ▶ `class Broken<T> { static T data; } // Fehler!`
- ▶ Grund: Alle Klassen `Broken<T>` teilen sich das Klassenattribut `data`. Welchen Typ soll es haben?

## ▶ Dynamische Typprüfung

- ▶ `class Node<T> { boolean isCompatible(Object o) { return o instanceof T; } } // Fehler!`
- ▶ Type-Erasure: `o instanceof T` → `o instanceof Object`

## ▶ Typecasts

- ▶ `class Node<T> { T info; void setInfo(Object o) { info = (T) o; } } // Sinnlos`
- ▶ Type-Erasure: `(T) o` → `(Object) o`
- ▶ Compiler erzeugt: „warning: unchecked cast of type T“



## Grenzen generischer Typen (2)

### ▶ Konstruktoraufrufe

- ▶ `class Node<T> { T info; Node() { info = new T(); } } // Fehler`
- ▶ Woher soll Java wissen, dass `T` einen Standard-Konstruktor hat?
- ▶ Beispiel:  
`Node<Integer> ni = new Node<Integer>();`
- ▶ Ausweg:  
`class Node<T> { T info; Node(T i) { info = i; } }`  
`Node<Integer> ni = new Node<Integer>(23);`

### ▶ Generische Basistypen

- ▶ `import java.util.Date;`  
`class Timestamped<T> extends T { Date timestamp = new Date(); } // Fehler`
- ▶ Generische Klasse muss Konstruktor der Basisklasse aufrufen können. Dieser kann hier aber nicht zur Kompilierzeit ermittelt werden!

### ▶ Exceptions

- ▶ `class UniversalException<T> extends Exception { T reason; } // Fehler`
- ▶ Generische Typen können nicht für Exceptions verwendet werden, da das Fangen mit `catch` auf dem Ermitteln des Typs des geworfenen Objekts basiert. Dieser geht aber bei der Type-Erasure verloren.
- ▶ Compiler erzeugt: „a generic class may not extend java.lang.Throwable“



## Grenzen generischer Typen (3)

### ▶ Arrays von Typvariablen

- ▶ `class Container<T> { T[] a = new T[100]; }`
- ▶ Ausweg:  
`class Container<T> { T[] a = (T[]) new Object[100]; } // Warnung`
- ▶ Compiler erzeugt „uses unchecked or unsafe operations“
- ▶ Man kann trotzdem typsichere Klassen erstellen:

```
class Container<T>
{
    private T[] a = (T[]) new Object[100]; // Warnung

    void set(int i, T t) {a[i] = t;}

    T get(int i) {return a[i];}
}
```



# Generische (polymorphe) Methoden

## ▶ Definition

- ▶ Polymorphe Methoden sind unabhängig von generischen Typen
- ▶ Sie können auch in nicht-generischen Klassen definiert werden
- ▶ Klassen- und Objektmethoden sowie Konstruktoren können polymorph sein

## ▶ Aufruf

- ▶ *String s = this.<String>vote("foo", "foo", "bar");*
- ▶ *int i = this.<Integer>vote(1,2,2);*
- ▶ *s = this.<String>vote(1,2,2); // Fehler!*

## ▶ Typ-Inferenz

- ▶ *String s = vote("foo", "foo", "bar");*
- ▶ Typparameter werden automatisch mit dem „untersten“ gemeinsamen Typ belegt
- ▶ *Double d = vote(1, 3.14, 1); // Fehler!*
- ▶ entspricht  
*Double d = vote(new Integer(1), new Double(3.14), new Integer(1)); // Fehler!*
- ▶ *Number n = vote(1, 3.14, 1); // ok*

```
<T> T vote(T x, T y, T z)
{
    if(x.equals(y))
        return x;
    else if(y.equals(z))
        return y;
    else if(z.equals(x))
        return z;
    else
        return null;
}
```



# Funktionen höherer Ordnung

## ▶ Definition

- ▶ Funktionen höherer Ordnung bekommen selbst Funktionen als Parameter
- ▶ In Java sind Funktionen als Parameter nicht möglich, wohl aber Objekte von Klassen, die bestimmte Interfaces implementieren

## ▶ Beispiel: Faltung

- ▶ Eine Faltung „faltet“ eine Liste (oder ein Array) zu einem einzigen Wert zusammen
- ▶ Dazu wird eine binäre Funktion der Reihe nach auf alle Elemente angewendet
- ▶ In Haskell:  
 $foldl (+) 0 [1,2,3,4,5]$   
 $= (((((0 + 1) + 2) + 3) + 4) + 5)$

```
interface FoldlFn<A, B> { A fn(A a, B b); }
```

```
<A, B> A foldl(FoldlFn<A, B> f, A a, B[] b)  
{  
    for(B b0 : b)  
        a = f.fn(a, b0);  
    return a;  
}
```

```
String concat(Object[] objects)  
{  
    return foldl(new FoldlFn<String, Object>()  
    {  
        public String fn(String s, Object o)  
            {return s + o.toString();}  
    }, "", objects);  
}
```