

Kapitel 4: Generische Datentypen

- Generischer Stack als Beispiel
- Typparameter
- Generische Methoden
- Compilierung von generischen Typen: Type Erasure und Raw Types
- Generische Felder
- Generischer Stack mit Interface und Klassen
- Typbeschränkung und Comparable
- Wildcards (Platzhalter) und Subtyping

Motivation: polymorpher Stack

- Gegeben ist der bekannte **int-Stack** als einfach verkettete Liste (top- und toString-Methode ist weggelassen).
- Ziel:
polymorpher (= vielgestaltiger) Stack, der für unterschiedliche Elementtypen verwendbar ist.
Also: String-Stack, Complex-Stack, Double-Stack, etc.
- Ansätze:
 - polymorpher Stack über Vererbung
 - generischer Stack
- Bemerkung: für jede Stack-Variante eine eigene Klasse zu definieren, wäre zu mühselig, zu fehleranfällig und zu wartungsaufwendig.

```
public class LinkedStack {  
  
    public LinkedStack() {top = null;}  
  
    public boolean empty() {  
        return top == null;  
    }  
  
    public void push(int x) {  
        top = new Node(x, top);  
    }  
  
    public int pop() {  
        if (top == null)  
            throw new EmptyStackException();  
        int x = top.data;  
        top = top.next;  
        return x;  
    }  
  
    private static class Node {  
        int data;  
        Node next;  
        Node(int x, Node p) {  
            data = x;  
            next = p;  
        }  
    }  
  
    private Node top;  
}
```

Polymorpher Stack über Vererbung (1)

- **Object** als Elementtyp
- Damit können Elemente beliebigen Typs abgespeichert werden.
- Man erhält einen **polymorphen** (vielgestaltigen) Stack.
- Kontrolle, dass nur Elemente des gewünschten Typs abgespeichert werden - z.B. String-Elemente - geschieht über Programmierdisziplin.
- **Ansatz daher fehleranfällig.** Fehler werden erst zur Laufzeit erkannt (schlecht!).
- Java-Historie: So wurden in Java polymorphe Datentypen realisiert, bevor generische Datentypen in Java 5.0 in 2004 eingeführt wurden.

```
public class LinkedStack {  
  
    public LinkedStack() {top = null;}  
  
    public boolean empty() {  
        return top == null;  
    }  
  
    public void push(Object x) {  
        top = new Node(x, top);  
    }  
  
    public Object pop() {  
        if (top == null)  
            throw new EmptyStackException();  
        Object x = top.data;  
        top = top.next;  
        return x;  
    }  
  
    private static class Node {  
        Object data;  
        Node next;  
        Node(Object x, Node p) {  
            data = x;  
            next = p;  
        }  
    }  
  
    private Node top;  
}
```

Polymorpher Stack über Vererbung (2)

```
public class StackApplication {  
  
    public static void main(String[] args)  
  
        // strStack ist ein Stack fuer Strings  
        LinkedStack strStack = new LinkedStack();  
  
        strStack.push("eins");  
        strStack.push("zwei");  
        strStack.push("drei");  
        System.out.println(strStack);  
  
        String s = (String) strStack.pop();  
        System.out.println(s);  
  
        strStack.push(123);  
        s = (String) strStack.pop();  
    }  
}
```

Der Elementtyp für den Stack wird nur über Kommentar und Namensgebung festgelegt.

eins, zwei, drei

String-cast notwendig!

drei

OK. 123 ist ein Integer und damit vom Typ Object

ClassCastException zur Laufzeit!

- Fehler erst zur Laufzeit zu erkennen ist grundsätzlich schlechter als Fehler bereits zur Compilierungszeit zu erkennen.
- Durch generische Datentypen werden Fehler bereits zur Compilierungszeit erkannt.

Generischer Stack (1)

- `class LinkedStack<T>` ist eine sogenannte **generische Klasse**. Ebenso `class Node<T>`.
- `T` ist ein **formaler Typparameter** und steht stellvertretend für einen beliebigen Referenztyp wie z.B. `String`.
- Konvention für Typparameter ist ein großer Buchstabe:
T für Type,
E für Element Type,
K für Key Type,
V für Value Type, usw.
- Überall, wo vorher der Typ `LinkedStack` bzw. `Node` stand, steht jetzt `LinkedStack<T>` bzw. `Node<T>`.
- Es gibt auch **generische Interfaces**. Oberbegriff ist dann **generischer Typ**.

```
public class LinkedStack<T> {  
  
    public LinkedStack() {top = null;}  
  
    public boolean empty() {  
        return top == null;  
    }  
  
    public void push(T x) {  
        top = new Node<T>(x,top);  
    }  
  
    public T pop() {  
        if (top == null)  
            throw new EmptyStackException();  
        T x = top.data;  
        top = top.next;  
        return x;  
    }  
  
    private static class Node<T> {  
        T data;  
        Node<T> next;  
        Node(T x, Node<T> p) {  
            data = x;  
            next = p;  
        }  
    }  
  
    private Node<T> top;  
}
```

Generischer Stack (2)

- String ist nun **aktueller Typparameter** und wird eingesetzt für den formalen Typparameter T im generischen Typ `LinkedList<T>`
- `LinkedList<String>` wird auch **parameterisierter Typ** oder auch **Instantiierung** des generischen Typs `LinkedList<T>` genannt.
- **Das Abspeichern von Elementen falschen Typs wird bereits zur Compilierungszeit erkannt.**

```
public class StackApplication {  
    public static void main(String[] args)  
  
        LinkedList<String> strStack  
            = new LinkedList<String> ();  
  
        strStack.push("eins");  
        strStack.push("zwei");  
        strStack.push("drei");  
        System.out.println(strStack);  
  
        String s = strStack.pop();  
        System.out.println(s);  
  
        // strStack.push(123);  
    }  
}
```

Instantiierung von
`LinkedList<T>`.

eins, zwei, drei

Kein cast notwendig!

drei

Würde der Compiler
verboten.

Anmerkungen (1)

- Analogie: generischer Typ entspricht einer Methoden-Definition; parameterisierter Typ entspricht einem Methoden-Aufruf.

```
public class LinkedStack<T> {  
  
    public void push(T x) {  
        top = new Node<T>(x, top);  
    }  
  
    // ...  
  
    private static class Node<T> {  
        T data;  
        Node<T> next;  
        Node(T x, Node<T> p) {  
            data = x;  
            next = p;  
        }  
    }  
    private Node<T> top;  
  
    public static void main(String[] args) {  
        LinkedStack<String> strStack  
            = new LinkedStack<String>();  
        // ...  
    }  
}
```

```
public class A {  
  
    public void f(int x) {  
        g(x);  
    }  
  
    public void g(int x) {  
        // ...  
    }  
  
    public static void main(...) {  
        f(12);  
    }  
}
```

Anmerkungen (2)

- In der Java-Version 7.0 wurde der **Diamond-Operator** `<>` eingeführt.
- Bei einem Konstruktoraufbau für eine parameterisierte Klasse muss die Typinformation nicht angegeben werden.
Die Information leitet der Compiler durch Typinferenz ab.
- Beispiel:

statt

```
Stack<String> strStack =  
    new LinkedStack<String>();  
  
Map<String, List<String>> dictionary =  
    new TreeMap<String, List<String>>();
```

kürzer

```
Stack<String> strStack =  
    new LinkedStack<>();  
  
Map<String, List<String>> dictionary =  
    new TreeMap<>();
```


Anmerkungen (3)

- Bei einem generischen Typ ist der Typparameter nur für die aktuelle Typdefinition gültig und könnte auch umbenannt werden.
- Geschachtelte generische Typen (z.B. Node<T>) können den Typparameter überlagern.

```
public class LinkedStack<T> {  
  
    public void push(T x) {  
        top = new Node<T>(x, top);  
    }  
  
    // ...  
  
    private static class Node<T> {  
        T data;  
        Node<T> next;  
        Node(T x, Node<T> p) {  
            data = x;  
            next = p;  
        }  
    }  
  
    private Node<T> top;  
}
```



```
public class LinkedStack<T> {  
  
    public void push(T x) {  
        top = new Node<T>(x, top);  
    }  
  
    // ...  
  
    private static class Node<XYZ> {  
        XYZ data;  
        Node<XYZ> next;  
        Node(XYZ x, Node<XYZ> p) {  
            data = x;  
            next = p;  
        }  
    }  
  
    private Node<T> top;  
}
```

- Linke Schreibweise (mit gleichbenannten Typvariablen) ist jedoch üblicher.

Kapitel 4: Generische Datentypen

- Generischer Stack als Beispiel
- **Typparameter**
- Generische Methoden
- Compilierung von generischen Typen: Type Erasure und Raw Types
- Generische Felder
- Generischer Stack mit Interface und Klassen
- Typbeschränkung und Comparable
- Wildcards (Platzhalter) und Subtyping

Hüll-Klassen (Wrapper-Klassen) (1)

Vorsicht

- Typparameter bei generischen Typen dürfen nur mit Referenztypen instantiiert werden.
- Basisdatentypen wie `int`, `double`, etc. dürfen nicht zur Instantiierung verwendet werden.

```
LinkedList<int> intStack = new LinkedList<int>();
```

nicht erlaubt!

Hüll-Klassen (Wrapper-Klassen)

- Es gibt zu jedem primitiven Datentyp eine entsprechende Hüllklasse, die einen primitiven Datenwert einfach in eine Klasse einhüllt.
- Hüll-Klassen: `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short` (und weitere).
- Alle Hüllklassen haben die gemeinsame abstrakte Oberklasse `Number`.

```
LinkedList<Integer> intStack = new LinkedList<Integer>();  
  
LinkedList<Double> doubleStack = new LinkedList<Double>();
```

Hüll-Klassen (Wrapper-Klassen) (2)

Boxing und Unboxing

- Zum **Einpacken (Boxing)** von Datenwerten in entsprechende Hüll-Objekte gibt es Konstruktoren bzw. statische `valueOf()`-Methoden.

Der Konstruktor legt immer ein neues Objekt an.

Die `valueOf()`-Methode ist eine Fabrikmethode, die auch bereits vorkonstruierte Objekte zurückliefern kann (Cache-Mechanismus). `Integer.valueOf()` macht dies für den Wertebereich -128 bis +127.

```
Integer x = new Integer(12);  
Integer y = Integer.valueOf(12);
```

- Zum **Auspacken (Unboxing)** gibt es entsprechende `xxxValue()`-Methoden.

```
int i = x.intValue();
```

Hüll-Klassen (Wrapper-Klassen) (3)

Autoboxing und Autounboxing

- Das lästige Ein- und Auspacken kann auch vom Java-Compiler übernommen werden (**Auto-Boxing** und **Auto-Unboxing**):

```
Integer x = 12;
```

statt

```
Integer x = Integer.valueOf(12);
```

```
int i = x;
```

statt

```
int i = x.intValue();
```

- Damit lässt sich ein Integer-Stack sehr angenehm verwenden:

```
Stack<Integer> intStack = new LinkedStack<>();  
intStack.push(1);  
intStack.push(2);  
int i = intStack.pop();
```

Generische Typen mit Elementgleichheit (1)

- In einem generischen Typ kann der Typparameter nur durch einen Referenztyp – wie z.B. Integer, String, Circle, etc. – instantiiert werden.
- Dabei ist zu beachten, dass die Operationen `x == y` und `x != y` auf Referenzen und nicht auf Wertebene stattfinden.
- Damit ein Vergleich auf Wertebene durchgeführt wird, muss stattdessen die Methode `equals` verwendet werden.
- Jedoch muss `equals` für den instantiierten Typ geeignet überschrieben sein bzw. werden. Sonst würde die Methode `Object.equals()` benutzt werden, die nur einen Referenzvergleich durchführen würde.

Generische Typen mit Elementgleichheit (2)

- Beispiel: linear verkettete Liste

```
public class LinkedList<T> {  
  
    // ...  
  
    public void add(T x) {...}  
  
    public boolean hasElement(T x) {  
        for (Node<T> p = head; p != null; p = p.next)  
            if (p.data.equals(x)) // statt p.data == x  
                return true;  
        return false;  
    }  
  
    public static void main(String[] args) {  
        LinkedList<Integer> list = new LinkedList<>();  
        list.add(500);  
        list.add(100);  
        list.add(300);  
        System.out.println(list.hasElement(300)); // true  
        System.out.println(list.hasElement(200)); // false  
    }  
}
```

hasElement prüft, ob x in der linear verketteten Liste vorkommt.

Für Integer ist equals geeignet überschrieben.

Generische Typen mit Elementgleichheit (3)

- Beispiel: linear verkettete Liste mit Circle-Elementen

```
public class Circle{
    private double radius;
    public Circle(double r) {radius = r;}

    @Override public boolean equals(Object o) {
        if (this == o)
            return true;
        if (!(o instanceof Circle))
            return false;
        Circle c = (Circle) o;
        return radius == c.radius;
    }

    public static void main(String[] args) {
        LinkedList<Circle> circleList = new LinkedList<>();
        circleList.add(new Circle(5));
        circleList.add(new Circle(1));
        circleList.add(new Circle(3));
        System.out.println(circleList.hasElement(new Circle(3)));    // true
        System.out.println(circleList.hasElement(new Circle(2)));    // false
    }
}
```

Für Circle wird equals
geeignet überladen

Kapitel 4: Generische Datentypen


- Generischer Stack als Beispiel
- Typparameter
- **Generische Methoden**
- Compilierung von generischen Typen: Type Erasure und Raw Types
- Generische Felder
- Generischer Stack mit Interface und Klassen
- Typbeschränkung und Comparable
- Wildcards (Platzhalter) und Subtyping

Generische Methoden

- Eine Methode lässt sich ebenfalls generisch definieren, indem ähnlich wie bei Typen ein Typparameter <T> (direkt vor der Methoden-Signatur) eingeführt wird.
- Der Aufruf einer generischen Methode ist wie bei einer nicht-generischen Methode.

```
public static boolean hasElement(int x, int[] a) {  
    for (int i = 0; i < a.length; i++)  
        if (x == a[i])  
            return true;  
    return false;  
}
```

```
public static <T> boolean hasElement(T x, T[] a) {  
    for (int i = 0; i < a.length; i++)  
        if (x.equals(a[i]))  
            return true;  
    return false;  
}  
  
public static void main(String[] args) {  
    Double[] a = {5.1, 3.2, 1.1, 7.3, 6.0, 4.0, 8.1};  
    System.out.println(hasElement(2.0, a));  
}
```



Überführung
in eine
generische
Methode

Kapitel 4: Generische Datentypen

- Generischer Stack als Beispiel
- Typparameter
- Generische Methoden
- Compilierung von generischen Typen: Type Erasure und Raw Types
- Generische Felder
- Generischer Stack mit Interface und Klassen
- Typbeschränkung und Comparable
- Wildcards (Platzhalter) und Subtyping

Compilierung von generischen Typen durch type erasure

- Bei der Compilierung von generischen Typen wird `<...>` weggelassen (**type erasure**) und der Typparameter durch `Object` (bzw. erste Typschranke) ersetzt.
- Der durch type erasure entstandene Typ wird auch **raw type** genannt.
- Beim Holen von Daten aus dem raw type werden **casts** eingebaut.

```
public class LinkedList<T> {  
  
    public void push(T x) {  
        top = new Node<T>(x, top);  
    }  
  
    private static class Node<T> {  
        T data;  
        Node<T> next;  
        Node(T x, Node<T> p) {...}  
    }  
    // ...  
  
    static void main() {  
        LinkedList<String> strStack =  
            new LinkedList<String>();  
        strStack.push("abc");  
        String s = strStack.pop();  
    }  
}
```

type
erasure →

```
public class LinkedList {  
  
    public void push(Object x) {  
        top = new Node(x, top);  
    }  
  
    private static class Node {  
        Object data;  
        Node next;  
        Node(Object x, Node p) {...}  
    }  
    // ...  
  
    static void main() {  
        LinkedList strStack =  
            new LinkedList();  
        strStack.push("abc");  
        String s =  
            (String) strStack.pop();  
    }  
}
```

raw types

Cast beim Holen
von Daten

Vorsicht mit Raw Types!

- Wird in einem Programm ein generischer Typ definiert, dann kann prinzipiell auch sein Raw Type im Programm verwendet werden.
- Dann kann jedoch vom Compiler keine Typsicherheit gewährleistet werden.
- Daher sollten **Raw Types** (bis auf ganz wenige Ausnahmen) **vermieden** werden!

```
public class LinkedStack<T> {  
  
    // ...  
  
    public static void main() {  
        LinkedStack strStack = new LinkedStack();  
        strStack.push(123);  
        String s = (String) strStack.pop();  
    }  
}
```

Generischer Typ

Raw Type

ClassCastException zur Laufzeit!

Besonderheit der Type-Erasure-Technik

- Durch die type-erasure-Technik wird **jede generische Klasse in genau eine Klasse in Byte-Code übersetzt** – unabhängig davon, wie viele unterschiedliche Instantiierungen der generischen Klasse im Programm vorkommen.
- Es gibt auch andere Ansätze:
In C++ wird jede Instantiierung einer generischen Klasse (in C++ template class genannt) in eine eigene Klasse übersetzt. D.h. `Stack<String>` und `Stack<Integer>` ergeben zwei getrennte Klasse. Es wird dadurch mehr Code erzeugt, der aber in der Regel effizienter ist.
- Um einige Eigenheiten der generischen Typen in Java besser verstehen zu können, ist das Verständnis der type-erasure-Technik hilfreich. Wir werden gleich sehen, dass generische Felder nicht direkt möglich sind.

Generische Felder (1)

- Es dürfen in Java keine generischen Felder erzeugt werden.

```
public class ArrayStack<T> {  
  
    private static final int n = 16;  
    private int size;  
    private T[] data;  
  
    public ArrayStack() {  
        size = 0;  
        data = new T[n]; // nicht erlaubt!  
    }  
  
    // ...  
}
```

- Hierfür verantwortlich sind **zwei Eigenschaften** von Java, die bei generischen Feldern nicht gleichzeitig erfüllt werden können:
 - Felder sind **reifiziert** (sprich re – ifiziert), d.h. Felder kennen zur Laufzeit ihren Elementtyp
 - **Type-Erasure-Technik**

Generische Felder (2)

- **Reifikation:**

(= Vergegenständlichung) bedeutet im Zusammenhang mit Typen in Java, dass Typinformationen zur Laufzeit zur Verfügung stehen. Felder kennen zur Laufzeit ihren Elementtyp. Damit lassen sich Typfehler zur Laufzeit erkennen.

```
Object[] arr = new Integer[10];  
arr[0] = "Hallo";
```

Zuweisung ist OK, da Felder kovariant.

ArrayStoreException zur Laufzeit!
Das Feld arr kennt zur Laufzeit seinen Elementtyp Integer.

- **Type-Erasure-Technik:**

Durch die Type-Erasure-Technik werden bei der Compilierung die Typ-Informationen beseitigt. Damit verträgt sich diese Technik nicht mit reifizierten Typen.

```
public class ArrayStack<T> {  
    private T[] data;  
    ...  
    data = new T[n];  
}
```

Nicht erlaubt.
Fehler zur Compilierungszeit.

- In J. Bloch, *Effective Java*, wird in *Item 28* mit einem Beispiel gezeigt, dass generische Felder nicht typsicher sind.
Eine ausführliche Diskussion dieser Problematik finden Sie auch in Naftalin & Wadler, *Java Generics*, Kapitel 6 *Reification*.

Generische Felder (3)

- Durch Erzeugen eines Object-Felds mit Typ-Cast von Object[] nach T[] und gleichzeitiger Unterdrückung einer Compiler-Warnung mit @SuppressWarnings("unchecked") kann dennoch ein generisches Feld angelegt werden:

```
public class ArrayStack<T> {  
  
    private static final int n = 16;  
    private int size;  
    private T[] data;  
  
    @SuppressWarnings("unchecked")  
    public ArrayStack() {  
        size = 0;  
        data = (T[]) new Object[n];  
    }  
    // ...  
}
```

- Dieser Trick sollte nur sparsam verwendet werden, da Typsicherheit nicht mehr vom Compiler garantiert werden kann. Der Programmierer ist stattdessen verantwortlich. Das ist aber bei Containern wie Stack kein Problem, da nur mit push neue Elemente in den Keller abgespeichert werden können, wobei der korrekte Typ gewährleistet ist. Außerdem wird das generische Feld nicht nach außen verfügbar gemacht.

Kapitel 4: Generische Datentypen

- Generischer Stack als Beispiel
- Typparameter
- Generische Methoden
- Compilierung von generischen Typen: Type Erasure und Raw Types
- Generische Felder
- **Generischer Stack mit Interface und Klassen**
- Typbeschränkung und Comparable
- Wildcards (Platzhalter) und Subtyping

Generischer Keller – mit Interface und Klassen (1)

```
public interface Stack<T> {  
    public void push(T n);  
    public T pop();  
    public T top();  
    public boolean empty();  
}
```

```
public class ArrayStack<T>  
implements Stack<T> {  
  
    private static final int N = 16;  
    private int size;  
    private T[] data;  
  
    @SuppressWarnings("unchecked")  
    public ArrayStack() {  
        size = 0;  
        data = (T[]) new Object[N];  
    }  
  
    public boolean empty() {  
        return size == 0;  
    }  
    // ...  
}
```

```
public class LinkedStack<T>  
implements Stack<T> {  
  
    // wie zuvor  
}
```

Generischer Keller – mit Interface und Klassen (2)

```
public class ArrayStack<T> implements Stack<T> {  
    // ...  
  
    public void push(T x) {  
        if (data.length == size)  
            data = Arrays.copyOf(data, 2*size);  
        data[size++] = x;  
    }  
  
    public T top() {  
        if (size == 0) throw new EmptyStackException();  
        return data[size-1];  
    }  
  
    public T pop() {  
        if (size == 0) throw new EmptyStackException();  
        T x = data[--size];  
        data[size] = null;  
        return x;  
    }  
}
```

Referenz wird nicht mehr benötigt.
Daher auf null setzen.

Generischer Keller – mit Interface und Klassen (3)

```
public class StackApplication {  
  
    public static void main(String[] args) {  
  
        Scanner in = new Scanner(System.in);  
        int d = in.nextInt();  
  
        Stack<Double> s;  
  
        if (d == 0) {  
            s = new ArrayStack<>();  
        }  
        else {  
            s = new LinkedStack<>();  
        }  
  
        s.push(3.1);  
        s.push(2.3);  
        s.push(1.7);  
  
        System.out.println(s);  
    }  
}
```

Kapitel 4: Generische Datentypen

- Generischer Stack als Beispiel
- Typparameter
- Generische Methoden
- Compilierung von generischen Typen: Type Erasure und Raw Types
- Generische Felder
- Generischer Stack mit Interface und Klassen
- **Typbeschränkung und Comparable**
- Wildcards (Platzhalter) und Subtyping

Motivation: generische Minimumsfunktion

- Es soll eine Methode zum Berechnen des Minimums eines int-Feldes in eine generische Methode überführt werden.

```
public static int min(int[] a) {  
    int min = a[0];  
    for (int i = 1; i < a.length; i++)  
        if (a[i] < min)  
            min = a[i];  
    return min;  
}
```

- Der Ausdruck `a[i] < min` ist nur für primitive Typen erlaubt.
- Durch welchen Methodenaufruf soll also `a[i] < min` in einer generischen Methode ersetzt werden?
- Wie wird gewährleistet, dass der instantiierte Typ auch tatsächlich eine geeignete Methode anbietet?

Typbeschränkung

- Durch

`<T extends Type>`

wird erzwungen, dass für T nur Subtypen von *Type* eingesetzt werden dürfen.

- Beispiel:

```
public class NumberStack<T extends Number> {  
    ...  
}  
  
public static void main(...) {  
    NumberStack<Integer> ns = new NumberStack<>();  
    NumberStack<String> strs = new NumberStack<>();  
}
```

OK!
Integer ist Subtyp von Number.

Nicht OK!
Würde Compiler verbieten, da String
kein Subtyp von Number ist.

Typbeschränkung mit Interface Comparable<T>

- Mit Hilfe des Interface `java.lang.Comparable` kann erzwungen werden, dass `T` eine Vergleichsoperation `compareTo` anbietet.

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

```
public static <T extends Comparable<T>> T min(T[] a) {  
    T min = a[0];  
    for (int i = 1; i < a.length; i++)  
        if (a[i].compareTo(min) < 0) // a[i] kleiner als min  
            min = a[i];  
    return min;  
}
```

- Die Spezifikation für `Comparable` in der Java API ist einzuhalten.
- Insbesondere soll `x.compareTo(y)` eine negative Zahl, 0, bzw. eine positive Zahl zurückliefern, falls `x` kleiner, gleich bzw. größer als `y` ist.
- Es wird empfohlen, dass `x.compareTo(y) == 0` und `x.equals(y)` denselben Wert liefern.

Verwendung der generischen Minimumsfunktion (1)

- Viele Klassen in der Java-API implementieren bereits das Interface Comparable: Integer, Double, String, ...
- Diese Klassen können daher bei der Minimumsfunktion eingesetzt werden.

```
public static void main(String[] args) {  
  
    Integer[] a = {5, 1, 7, 3};  
    System.out.println(min(a));  
  
    String[] sf = {"fuenf", "eins", "zwei", "drei"};  
    System.out.println(min(sf));  
  
}
```

1

drei

Verwendung der generischen Minimumsfunktion (2)

- Damit beispielsweise ein Feld mit Circle-Objekten mit der generischen Minimumsfunktion bearbeitet werden kann, muss die Klasse Circle das Interface Comparable implementieren.

```
class Circle implements Comparable<Circle> {  
    private double radius;  
    public Circle(double r) {radius = r;}  
    // ...  
    public int compareTo(Circle c) {  
        if (this.radius < c.radius)  
            return -1;  
        else if (this.radius == c.radius)  
            return 0;  
        return +1;  
    }  
}
```

```
public static void main(String[] args) {  
  
    Circle[] cArr = {new Circle(5), new Circle(1), new Circle(3)};  
    System.out.println(min(cArr).getRadius());  
}
```

1

Interface Comparable<T> in der Java API

interface Comparable<T>

This interface imposes a **total ordering** on the objects of each class that implements it. This ordering is referred to as the class's **natural ordering**, and the class's `compareTo` method is referred to as its natural comparison method.

It is strongly recommended (though not required) that natural orderings be **consistent with equals**.

Eine Relation \leq ist eine **totale Ordnung** (lineare Ordnung), falls \leq **reflexiv**, **transitiv**, **antisymmetrisch** und **total** ist.

int compareTo(T o)

- (1) Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- (2) The implementor must ensure $\text{sgn}(x.\text{compareTo}(y)) == -\text{sgn}(y.\text{compareTo}(x))$ for all x and y .
- (3) The implementor must also ensure that the relation is transitive: $(x.\text{compareTo}(y) > 0 \ \&\& \ y.\text{compareTo}(z) > 0)$ implies $x.\text{compareTo}(z) > 0$.
- (4) Finally, the implementor must ensure that $x.\text{compareTo}(y) == 0$ implies that $\text{sgn}(x.\text{compareTo}(z)) == \text{sgn}(y.\text{compareTo}(z))$, for all z .
- (5) It is strongly recommended, but not strictly required that $(x.\text{compareTo}(y) == 0) == (x.\text{equals}(y))$. Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

Aus der Definition von `compareTo` ergibt sich die **natürliche Ordnung**:

$$x \leq y \text{ gdw. } x.\text{compareTo}(y) \leq 0$$

(1) – (4) erzwingen, dass die natürliche Ordnung auch eine totale Ordnung ist.

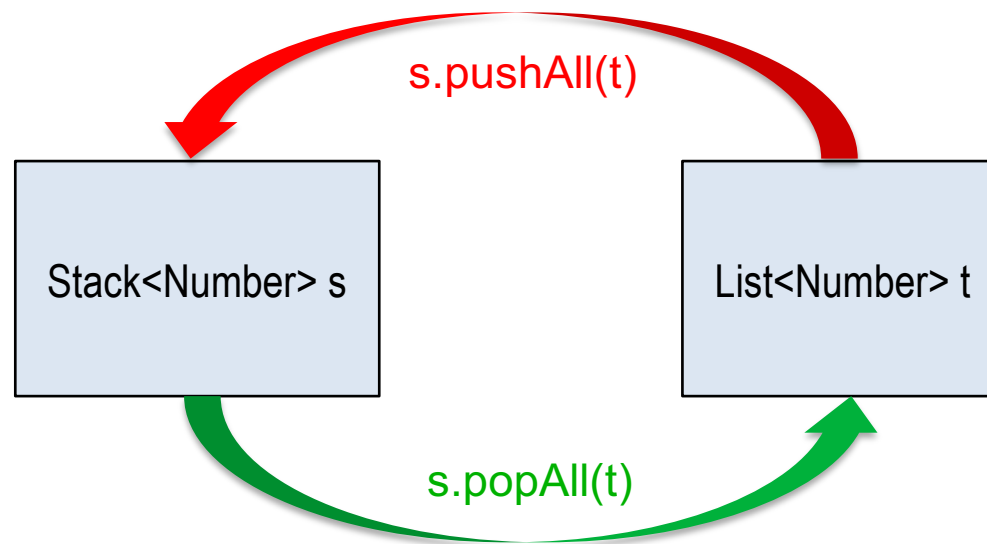
Siehe auch *Item 14* in J. Bloch, *Effective Java*.

Kapitel 4: Generische Datentypen

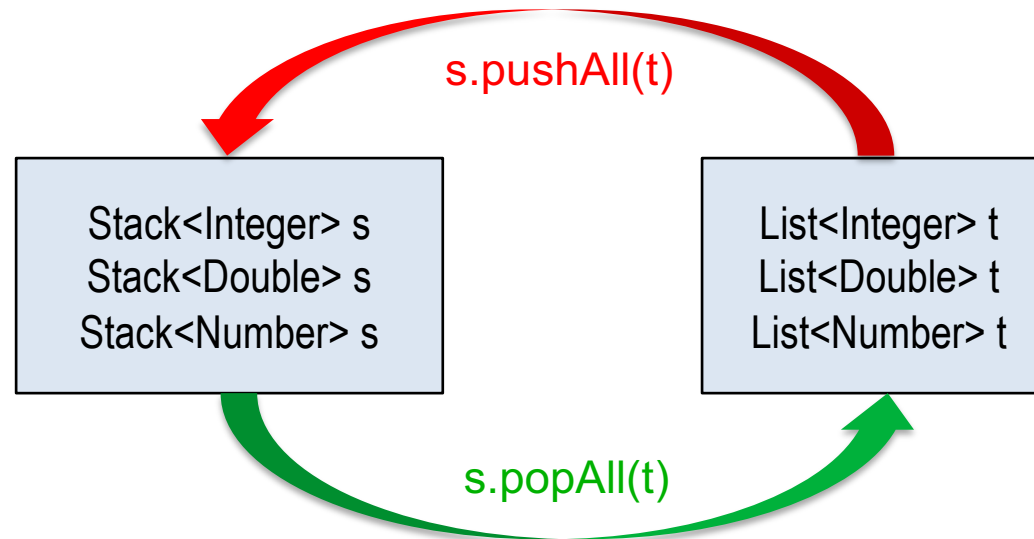
- Generischer Stack als Beispiel
- Typparameter
- Generische Methoden
- Compilierung von generischen Typen: Type Erasure und Raw Types
- Generische Felder
- Generischer Stack mit Interface und Klassen
- Typbeschränkung und Comparable
- Wildcards (Platzhalter) und Subtyping

Motivation: folgende Problemstellung

- Betrachte Operationen mit parameterisierten Container als Parameter;
z.B. Number-Liste.
- **s.pushAll(t):**
Einfügen aller Elemente aus der Liste t in den Stack s.
t ist Eingabe-Parameter.
- **s.popAll(t):**
Entfernen aller Elemente aus dem Stack s und Abspeichern in die Liste t.
t ist Ausgabe-Parameter.



Wünschenswert: möglichst viel Flexibilität bei voller Typsicherheit

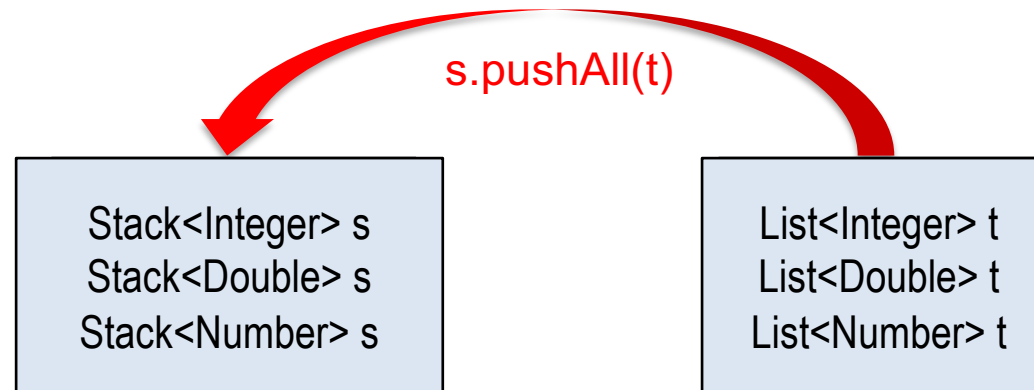


	List<Integer> t	List<Double> t	List<Number> t
Stack<Integer> s			
Stack<Double> s			
Stack<Number> s			

+ zulässig bei pushAll
- nicht zulässig bei pushAll

+ zulässig bei popAll
- nicht zulässig bei popAll

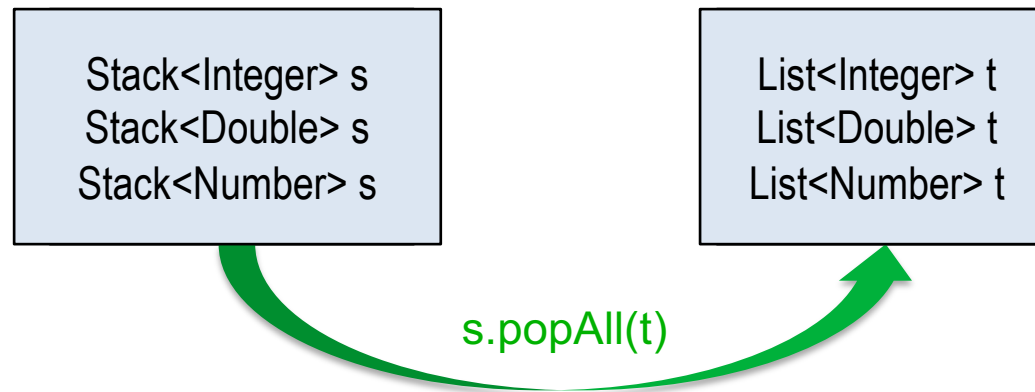
Wünschenswert: möglichst viel Flexibilität bei voller Typsicherheit



	List<Integer> t	List<Double> t	List<Number> t
Stack<Integer> s	+	-	-
Stack<Double> s	-	+	-
Stack<Number> s	+	+	+

+ zulässig bei pushAll
- nicht zulässig bei pushAll

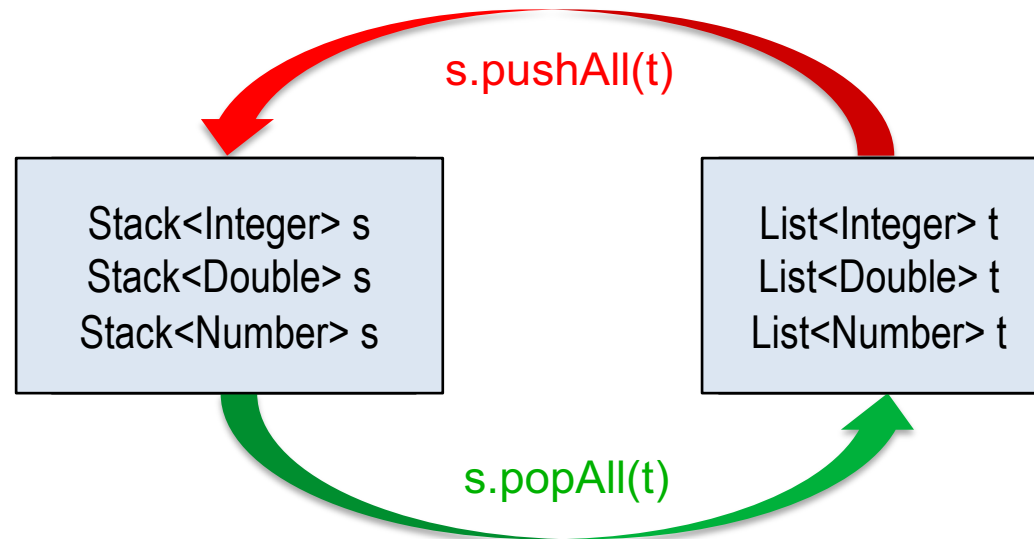
Wünschenswert: möglichst viel Flexibilität bei voller Typsicherheit



	List<Integer> t	List<Double> t	List<Number> t
Stack<Integer> s	+	-	+
Stack<Double> s	-	+	+
Stack<Number> s	-	-	+

+ zulässig bei popAll
- nicht zulässig bei popAll

Wünschenswert: möglichst viel Flexibilität bei voller Typsicherheit



	List<Integer> t		List<Double> t		List<Number> t	
Stack<Integer> s	+	+	-	-	-	+
Stack<Double> s	-	-	+	+	-	+
Stack<Number> s	+	-	+	-	+	+

+ zulässig bei pushAll
- nicht zulässig bei pushAll

+ zulässig bei popAll
- nicht zulässig bei popAll

Felder als Transportcontainer: unflexibel und unsicher

```
public class Stack<T> {
    public void push(T x) {...}
    public T pop() {...}
    public boolean isEmpty() {...}
    public int size() {...}
    // ...

    public void pushAll(T[] arr) {
        for (T x : arr)
            push(x);
    }

    public void popAll(T[] arr) {
        int i = 0;
        while (!isEmpty())
            arr[i++] = pop();
    }

    public static void main(...) {
        Stack<Number> s = new Stack();
        Number[] t = new Number[...];

        s.pushAll(t);
        s.popAll(t);
    }
}
```

- Wegen der Kovarianz von Feldern funktioniert **pushAll** wie gewünscht.
- Dagegen ist **popAll** unsicher und unflexibel.

☐ unflexibel

☒ unsicher

	Integer[] t		Double[] t		Number[] t
Stack<Integer> s	+	+	-	-	- <input checked="" type="radio"/>
Stack<Double> s	-	-	+	+	- <input checked="" type="radio"/>
Stack<Number> s	+	<input checked="" type="radio"/>	+	<input checked="" type="radio"/>	+

Felder als Transportcontainer: Beispiele

```
public static void main(String[] args) {
```

```
    Stack<Number> s = new Stack<>();
```

```
    Integer[] intArr = {1, 3, 5};  
    s.pushAll(intArr);
```

Number-Stack

```
    Double[] dbArr = {1.5, 3.2, 3.1};  
    s.pushAll(dbArr);
```

```
    Number[] nbArr = {7, 7.5, 8};  
    s.pushAll(nbArr);
```

```
    System.out.println(s);
```

8, 7.5, 7, 3.1, 3.2, 1.5, 5, 3, 1

```
    intArr = new Integer[s.size()];  
    // s.popAll(intArr);
```

Laufzeitfehler: ArrayStoreException

```
    nbArr = new Number[s.size()];  
    s.popAll(nbArr);  
    System.out.println(java.util.Arrays.toString(nbArr));
```

```
}
```

8, 7.5, 7, 3.1, 3.2, 1.5, 5, 3, 1

(Ohne wildcards) parameterisierte Listen als Transportcontainer: Sicher aber unflexibel

```
public class Stack<T> {  
    // ...  
    public void pushAll(List<T> list) {  
        for (T x : list)  
            push(x);  
    }  
  
    public void popAll(List<T> list) {  
        while (!isEmpty())  
            list.add(pop());  
    }  
  
    public static void main(...) {  
        Stack<Number> s = new Stack<>();  
        List<Number> l = new LinkedList<>();  
        l.add(1); l.add(3.1); l.add(5);  
        s.pushAll(l);  
        s.popAll(l);  
    }  
}
```

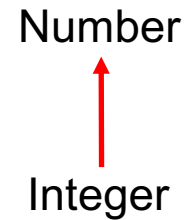
- (ohne wildcards) parameterisierte Typen sind **invariant**.
- Obwohl z.B. Integer ein Subtyp von Number ist, ist List<Integer> kein Subtyp von List<Number> und umgekehrt.

○ unflexibel

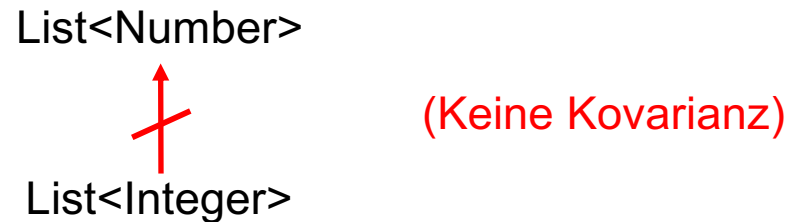
	List<Integer> t		List<Double> t		List<Number> t	
Stack<Integer> s	+	+	-	-	-	⊖
Stack<Double> s	-	-	+	+	-	⊖
Stack<Number> s	⊖	-	⊖	-	+	+

(Ohne wildcards) parameterisierte Listen sind invariant

- Zwar gilt:



- Jedoch keine Subtyp-Beziehungen zwischen `List<Number>` und `List<Integer>`



Parameterisierte Listen mit extends- und super-wildcards

- `List<? extends A>`

bedeutet `List<X>`, wobei `X` irgendein **Subtyp** von `A` ist.
„Liste für irgendein Subtyp von `A`“

- `List<? super A>`

bedeutet `List<X>`, wobei `X` irgendein **Supertyp** von `A` ist.
„Liste für irgendein Supertyp von `A`“

- `?` wird auch **wildcard (Platzhalter)** genannt.

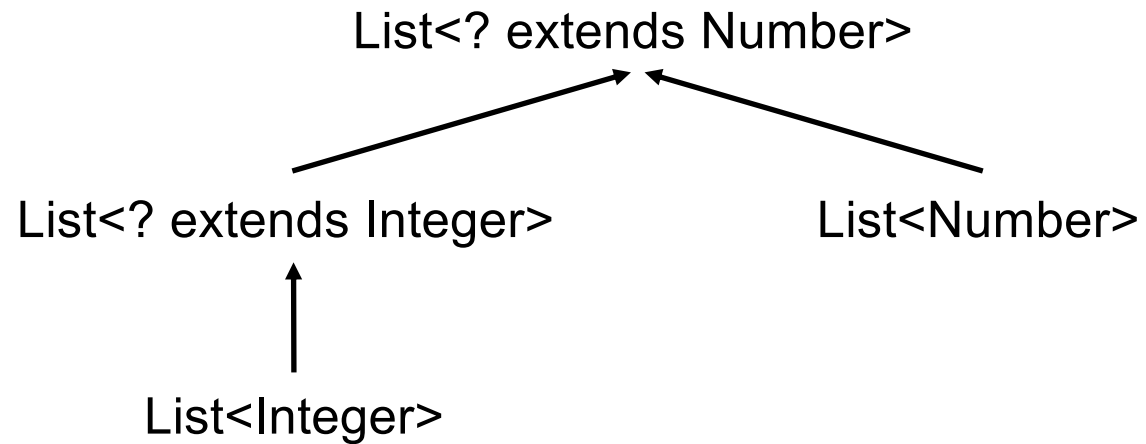
Parameterisierte Listen mit extends-Wildcards sind kovariant

(1) Falls $\begin{array}{c} B \\ \uparrow \\ A \end{array}$ dann $\begin{array}{c} \text{List}<? \text{ extends } B> \\ \uparrow \\ \text{List}<? \text{ extends } A> \end{array}$ Kovarianz

(2) Für jeden Typ A : $\begin{array}{c} \text{List}<? \text{ extends } A> \\ \uparrow \\ \text{List}<A> \end{array}$

(3) $\text{List}<?>$ steht für $\text{List}<? \text{ extends Object}>$

Beispiel

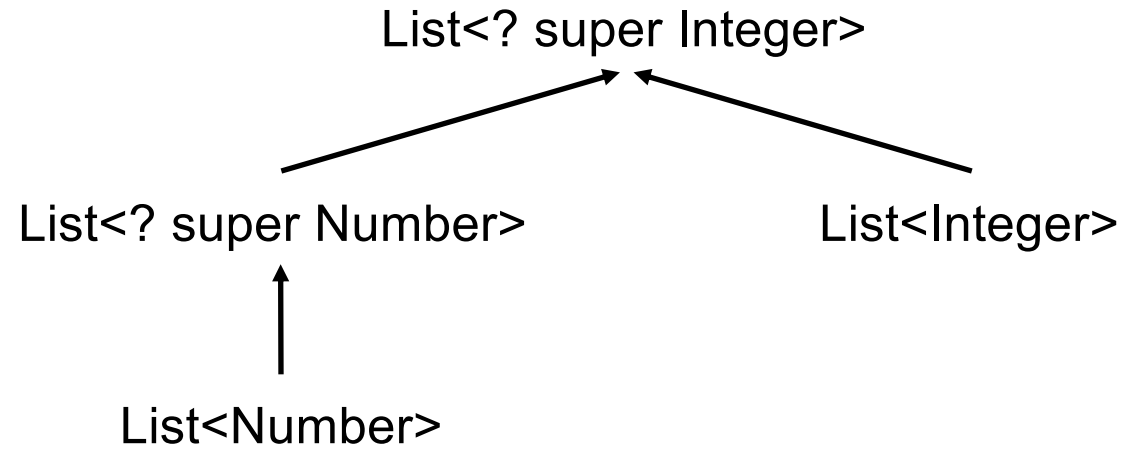


Parameterisierte Listen mit super-Wildcards sind kontravariant

(1) Falls $\begin{array}{c} B \\ \uparrow \\ A \end{array}$ dann $\begin{array}{c} \text{List}<? \text{ super } A> \\ \uparrow \quad \downarrow \\ \text{List}<? \text{ super } B> \end{array}$ **Kontravarianz**

(2) Für jeden Typ A : $\begin{array}{c} \text{List}<? \text{ super } A> \\ \uparrow \\ \text{List}<A> \end{array}$

Beispiel



Liste mit Wildcards: flexibel und sicher

```
public class Stack<T> {  
    // ...  
    public void pushAll(List<? extends T> l) {  
        for (T x : l)  
            push(x);  
    }  
  
    public void popAll(List<? super T> l) {  
        while (!isEmpty())  
            l.add(pop());  
    }  
  
    public static void main(String[] args) {  
        Stack<Number> s = new Stack<>();  
        List<Number> l = new LinkedList<>();  
        l.add(1); l.add(3.1); l.add(5);  
  
        s.pushAll(l);  
        s.popAll(l);  
    }  
}
```

	List<Integer> t		List<Double> t		List<Number> t	
Stack<Integer> s	+	+	-	-	-	+
Stack<Double> s	-	-	+	+	-	+
Stack<Number> s	+	-	+	-	+	+

Liste mit Wildcards: Beispiele (1)

```
public static void main(String[] args) {
```

```
    Stack<Number> nbStack = new Stack<>();
```

```
    List<Integer> intList = new LinkedList<>();  
    intList.add(1); intList.add(3); intList.add(5);  
    nbStack.pushAll(intList);
```

```
    List<Double> dbList = new LinkedList<>();  
    dbList.add(1.5); dbList.add(3.2); dbList.add(3.1);  
    nbStack.pushAll(dbList);
```

```
    List<Number> nbList = new LinkedList<>();  
    nbList.add(7); nbList.add(7.5); nbList.add(8);  
    nbStack.pushAll(nbList);
```

```
    System.out.println(nbStack);
```

```
    List<Number> nbListRes = new LinkedList<>();  
    nbStack.popAll(nbListRes);  
    System.out.println(nbStack);  
    System.out.println(nbListRes);
```

```
}
```

Integer-Liste in
Number-Stack
einkellern

Double-Liste in
Number-Stack
einkellern

Number-Liste in
Number-Stack
einkellern

Number-Stack
auskellern und in
Number-Liste
abspeichern

Liste mit Wildcards: Beispiele (2)

```
public static void main(String[] args) {
```

```
    Stack<Integer> intStack = new Stack<>();
```

```
    List<Integer> intList = new LinkedList<>();  
    intList.add(1); intList.add(3); intList.add(5);  
    intStack.pushAll(intList);
```

```
    List<Number> nbListRes = new LinkedList<>();  
    intStack.popAll(nbListRes);
```

```
    System.out.println(intStack);  
    System.out.println(nbListRes);
```

```
}
```

} Integer-Liste in
Integer-Stack
einkellern

} Integer-Stack
auskellern und in
Number-Liste
abspeichern

PECS-Prinzip

- PECS = Producer Extends and Consumer Super
- Dieses aus [Bloch, Effective Java] stammende Mnemonic hilft bei der Auswahl, ob eine extends oder eine super-Wildcard benutzt werden soll.
- Ein Container, aus dem Daten gelesen werden (= **Producer**), wird mit einer **extends-Wildcard** versehen.
- Ein Container, in dem Daten abgespeichert werden (= **Consumer**), wird mit einer **super-Wildcard** versehen.

```
public class Stack<T> {  
    // ...  
    public void pushAll(List<? extends T> list) {  
        for (T x : list)  
            push(x);  
    }  
  
    public void popAll(List<? super T> list) {  
        while (!isEmpty())  
            list.add(pop());  
    }  
  
    // ...  
}
```

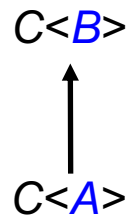
Aus list werden Daten gelesen.
list ist daher **Producer** und mit einer **extends-Wildcard** zu versehen.

In list werden Daten geschrieben.
list ist daher **Consumer** und mit einer **super-Wildcard** zu versehen.

Zusammenfassung der Typisierungsregeln für Wildcards

Parameterisierte Typen (ohne Wildcards) sind invariant

- Seien $C<A>$ und C beliebige parameterisierte Typen mit $A \neq B$.
- Dann gilt weder

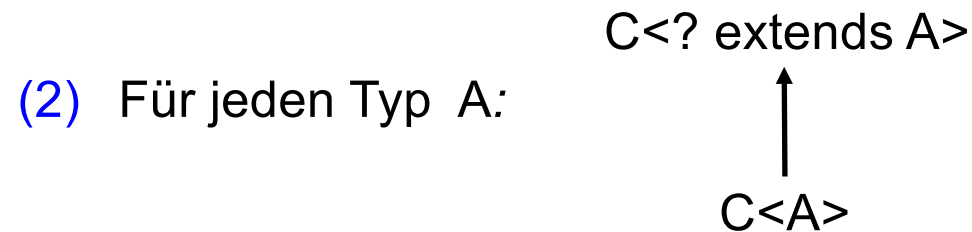
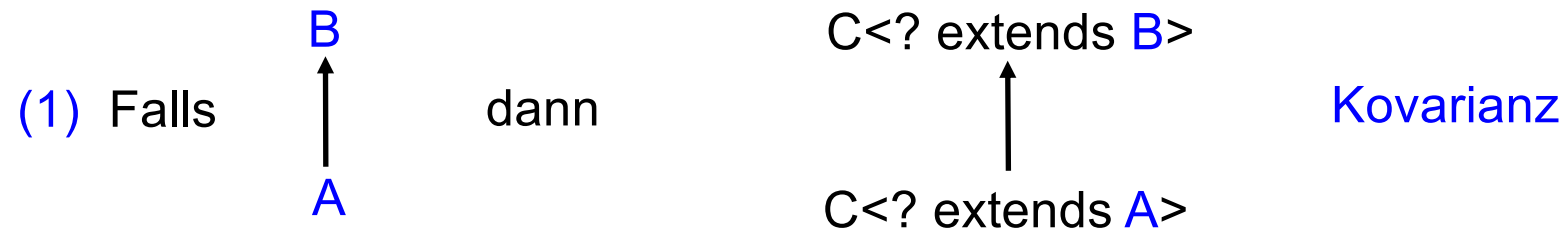


- noch



- Man sagt auch: parameterisierte Typen sind **invariant**.

Parameterisierte Typen mit extends-Wildcards sind kovariant



(3) $C<?>$ steht für $C<? \text{ extends Object}>$

Parameterisierte Typen mit super-Wildcards sind kontravariant

(1) Falls $\begin{array}{c} B \\ \uparrow \\ A \end{array}$ dann $\begin{array}{c} C<? \text{ super } A> \\ \uparrow \\ C<? \text{ super } B> \end{array}$ Kontravarianz

(2) Für jeden Typ A : $\begin{array}{c} C<? \text{ super } A> \\ \uparrow \\ C<A> \end{array}$