Generische Programmierung mit Generics

Marcel Lüthi
Departement Mathematik und Informatik

Agenda

- Motivation

Einfache Konzepte

- Generische Klassen
- Generische Methoden

Fortgeschrittene Nutzung

- Vererbung, Kovarianz und Kontravarianz
- Type erasure
- Wildcards

Motivation für Generische Programmierung

Stack für Ints

```
class IntStack {
  int[] data = new int[1000];
  private int nElements = 0;

  public void push(int element) {
    data[nElements] = element;
    nElements += 1;
  }

  public int pop() {
    if (nElements <= 0) {
       return null;
    } else {
       nElements -= 1;
       return data[nElements];
    }
  }
}
```

Stack für Strings

```
class StringStack {
  String[] data = new String[1000];
  private int nElements = 0;

  public void push(String element) {
    data[nElements] = element;
    nElements += 1;
  }

  public String pop() {
    if (nElements <= 0) {
       return null;
    } else {
       nElements -= 1;
       return data[nElements];
    }
  }
}
```

Generisches Programmieren mit Objekten

Lösungsansatz: Nutzung von gemeinsamen Supertyp `Object`

```
class Stack {

    private Object[] data = new Object[1000];
    private int nElements = 0;

    public void push(Object element) {
        data[nElements] = element;
        nElements += 1;
    }

public Object pop() {
        if (nElements <= 0) {
            return null;
        }
        else {
            nElements -= 1;
            return data[nElements];
       }
    }
}
```

Probleme

- Wir verlieren Typinformation
  - Intention nicht klar
    - Speichern wir Listen, Zahlen oder Strings?
  - Wir brauchen explizite downcasts
    `Integer i = (Integer) stack.pop()`
  - Fehlende Typsicherheit
    - Fehler zur Laufzeit - Keine Hilfe von Compiler
- Klassen müssen in Hierarchie angeordnet sein.

Beispiel

Folgendes gibt zur Laufzeit einen Fehler

In [ ]:
```
Stack stack = new Stack();
stack.push("abc");
stack.push(5);

String s = (String) stack.pop();
```

Generische Klassen

Lösung: Java Generics

Typ wird als Parameter der Klasse angegeben

In [ ]:
```
class Stack<E> {

    private E[] data = (E[])new Object[1000];
    private int nElements = 0; // Anzahl Elemente im Stack


    public void push(E element) {
        data[nElements] = element;
        nElements += 1;
    }

    public E pop() {
        if (nElements <= 0) {
            return null;
        }
        else {
            nElements -= 1;
            return data[nElements];
       }
    }
    public int size() { return this.nElements; }
}
```

Nutzung des Generischen Stacks

- Beim benutzen der Stack Klasse wird Typ angegeben.
- Beim entfernen des Elements ist kein Cast nötig

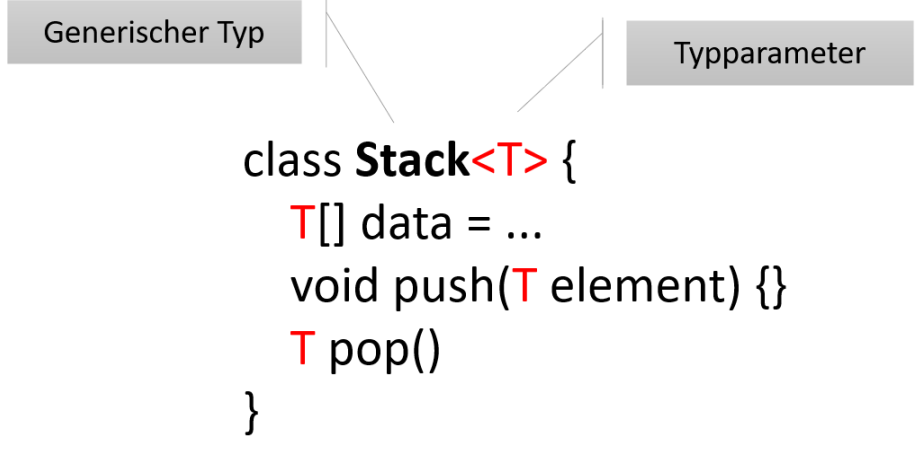
```
Stack<String> stringStack = new Stack<String>();

stringStack.push("abc");
String s = stringStack.pop();
```

Miniübung

- Erzeugen Sie einen Stack von Integern
- Versuchen Sie ein Objekt vom falschen Typen auf den Stack zu legen
- Können Sie einen Stack von `int` erzeugen?

In [ ]:
```
Stack<int> intstack = new Stack<int>();
```

Terminologie



Vereinfachte Syntax

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

- Typ auf rechter Seite wird von Compiler erzeugt.

Mehrere Typparameter

- Wir können beliebig viele Typparameter einführen

Beispiel: Tuple Klasse

In [ ]:
```
class Tuple<T, U> {

    private T first;
    private U second;

    public Tuple(T first, U second) {
        this.first = first;
        this.second = second;
    }


    public T getFirst() { return this.first;}
    public U getSecond() { return this.second; }

}
```

Anwendung von Tuple

In [ ]:
```
Tuple<String, Date> t = new Tuple<String, Date>("Tom", new Date(1970, 12, 31));

String s = t.getFirst();
Date d = t.getSecond();

System.out.println("the first element is: " +s);
System.out.println("the second element is: " +d);
```

Klassen der Java Standardbibliothek

Viele Klassen der Java Standardbibliothek sind via Generics implementiert

- Wir können diese nun nutzen

Beispiel: LinkedList

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Class LinkedList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.AbstractSequentialList<E>
        java.util.LinkedList<E>

Type Parameters:

`E` - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the `List` and `Deque` interfaces. Implements all optional list operations, and permits all elements (including `null`).

Beispiel: LinkedList

In [ ]:

```
import java.util.LinkedList;

LinkedList<String> ll = new LinkedList<>();
ll.add("first Element");
ll.add("second Element" );

for (String s : ll) {
    System.out.println(s);
}
```

Typeinschränkungen

Comparable in Java

- In Java implementieren Typen die vergleichbar sind das `Comparable` Interface.

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

Typeinschränkung (Bounded type parameters)

- Wir können Typeinschränkung auf generische Parameters mittels der `extends` Klausel definieren

Beispiel:

- Generischer Typ sollte vergleichbar sein:

```
class A<E extends Comparable<E> >
```

- Konsequenz: Nur Subtypen von Comparable können als generische Typen benutzt werden.

Anwendungsbeispiel

- Sortierte Liste, die für alle vergleichbaren Typen funktionieren sollte:

In [ ]:
```
class SortedList <E extends Comparable<E> > {
      E[] data = (E[]) new Comparable[1000];
      int nElements = 0;

      void add(E elem) {
          int i = nElements - 1;

          // Methode compareTo ist definiert für Typ E !!
          while (i >= 0 && elem.compareTo(data[i]) < 0) {
              data[i+1] = data[i];
              i -= 1;
          }
          data [i+1] = elem;
          nElements++;
     }


    void print() {
        for (int i = 0; i < nElements; i++) {
            System.out.println(data[i] + " ");
        }
    }
}
```

Anwendung

Folgendes funktioniert:

In [ ]:
```
SortedList<Integer> intList = new SortedList<Integer>();
intList.add(5);
intList.add(1);
intList.add(9);
intList.print();
```

Anwendung

Folgendes funktioniert:

In [ ]:
```
SortedList<Integer> intList = new SortedList<Integer>();
intList.add(5);
intList.add(1);
intList.add(9);
intList.print();
```

Folgendes funktioniert nicht:

In [ ]:
```
new SortedList<Object>();
```

Grund: `Stack` und `Object` sind nicht `Comparable`

Generische Methoden

Generische Methoden

- Methoden, die mit unterschiedlichen Parametertypen arbeiten können
  - Syntax `<T> void methodeName(T t) {}`
- Besprochene Regeln gelten auch für Methoden

Beispiel

In [ ]:
```
static <T>  void copyElements(LinkedList<T> source, LinkedList<T> destination) {

    for (T element : source) {
        destination.add(element);
    }
}
```

Generische Methoden

- Methoden, die mit unterschiedlichen Parametertypen arbeiten können
  - Syntax `<T> void methodeName(T t) {}`
- Besprochene Regeln gelten auch für Methoden

Beispiel

In [ ]:
```
static <T>  void copyElements(LinkedList<T> source, LinkedList<T> destination) {

    for (T element : source) {
        destination.add(element);
    }
}
```

Anwendung:

In [ ]:
```
LinkedList<Integer> list = new LinkedList<>();
list.add(5);
list.add(7);
list.add(12);
System.out.println("Liste: " + list);

LinkedList<Integer> listCopy = new LinkedList<>();
copyElements(list, listCopy);
System.out.println("Kopie: " +listCopy);
```

Übung

- Implementieren Sie eine generische Methode `max` welche für zwei übergebene Elemente vom Typ `T` das grössere Element zurückgibt.

In [ ]:
```
<T extends Comparable<T> > T max(T e1, T e2) {
    if (e1.compareTo(e2) < 0 ) {
        return e2;
    } else {
        return e1;
    }
}

System.out.println(max(7, 3));
```

Generische Methoden: Typsicherheit

Nicht Typsicher

- Deklaration von Methode mit gemeinsamen Supertyp.
  `Comparable max(Comparable a, Comparable b)`
- a muss nicht von demselben Typ wie b sein
- Drückt nicht aus was wir wollen

Typesicher

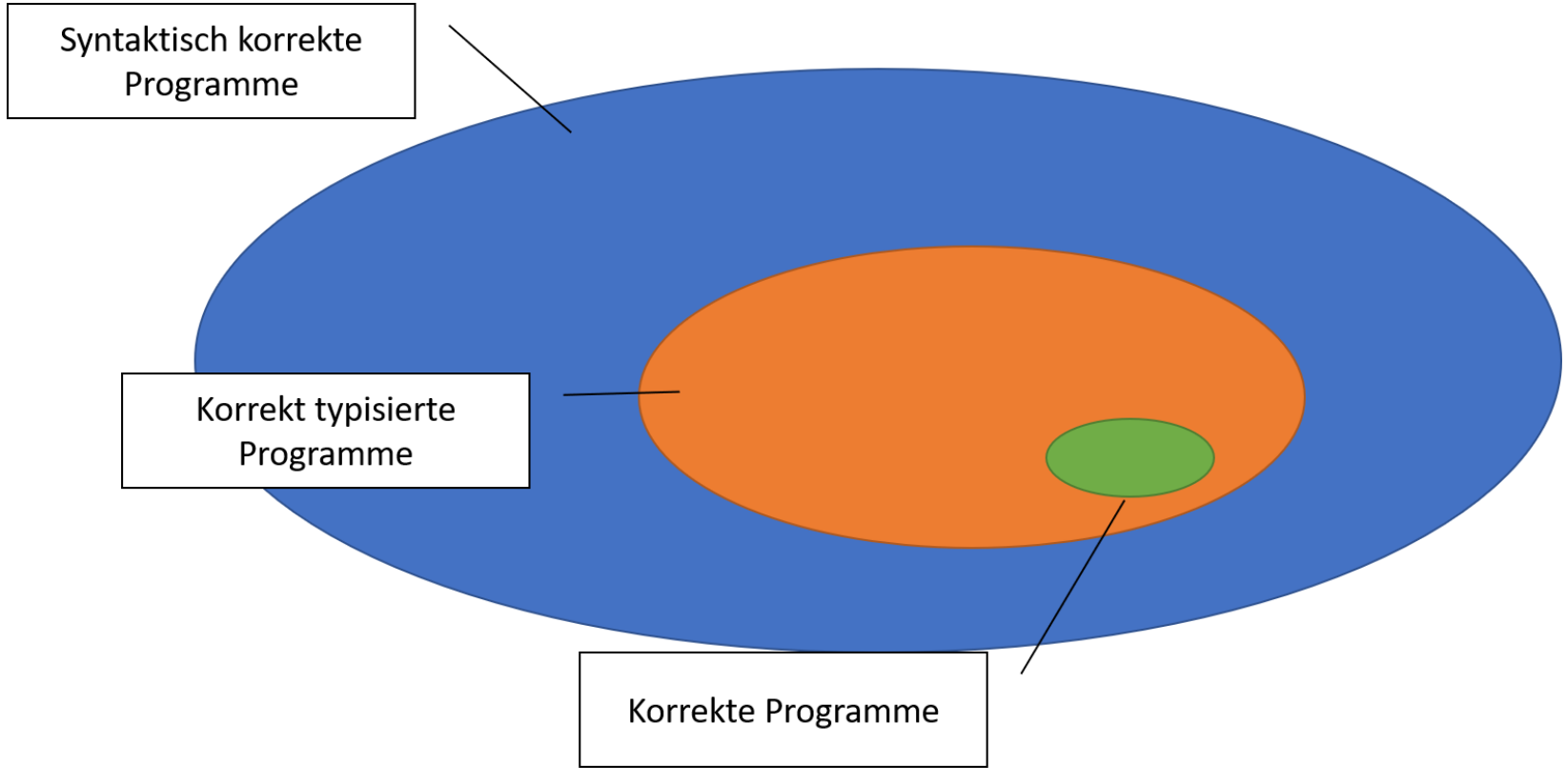
- Deklaration mit Generics
  `<T extends Comparable> max(T a, T b)`
- a und b haben denselben Typ
- Drückt genau aus was wir wollen

Zusammenfassung

Generics lösen die Probleme der generischen Programmieren über Objektehierarchien

- Kein Verlust von Typinformation
  - Intention wird klar ausgedrückt
  - Keine expliziten downcasts mehr, da Typ bekannt ist
  - Typfehler werden zur Kompilation und nicht Laufzeit erkannt.
- Funktioniert für beliebige Klassen

Zusammenfassung



Vererbung

Vererbung

Wir können von generischen Klassen ganz normal erben.

```
abstract class A<T> {

    abstract T aMethode(T t);
}
```

- Unterscheidung: Typ bleibt generisch oder wird konkretisiert

Fall 1: Vererbung mit generischem Typ:

```
class B<T> extends A<T> {

    T aMethode(T t) {
        return t;
    }
}
```

- T bleibt Typparameter

Fall 2: Vererbung mit konkretem Typ

```
class C extends A<Integer> {

    Integer aMethode(Integer t) {
        return t;
    }
}
```

- Typ `T` wurde hier durch `Integer` ersetzt

Kovarianz, Kontravarianz und Invarianz

- Gegeben eine generische Klasse `class A<T> {}`

Kovarianz

Falls `S` ein Supertyp von `T` ist, dann ist `A<S>` ein Supertyp von `A<T>`

Kontravarianz

Falls `S` ein Supertyp von `T` ist, dann ist `A<S>` ein Subtyp von `A<T>`

Invarianz

Falls `S` ein Supertyp von `T` ist, dann ist `A<S>` weder ein Subtyp von `A<T>` noch ein Supertyp

Kovarianz, Kontravarianz und Invarianz

- Gegeben eine generische Klasse `class A<T> {}`

Kovarianz

Falls `S` ein Supertyp von `T` ist, dann ist `A<S>` ein Supertyp von `A<T>`

Kontravarianz

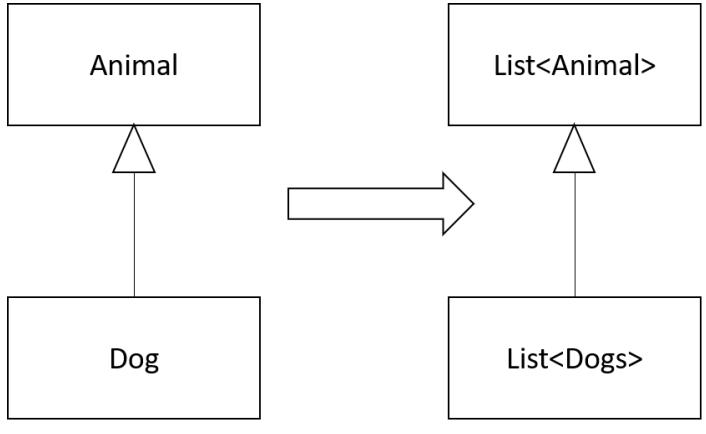
Falls `S` ein Supertyp von `T` ist, dann ist `A<S>` ein Subtyp von `A<T>`

Invarianz

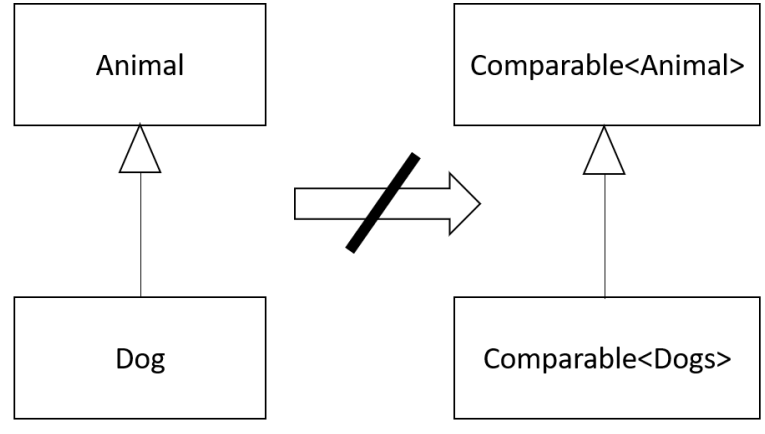
Falls `S` ein Supertyp von `T` ist, dann ist `A<S>` weder ein Subtyp von `A<T>` noch ein Supertyp

- *Kontravarianz tritt nur in seltenen Fällen auf*
  - *Kann in diesen Kurs ignoriert werden*

Kovarianz und Invarianz



Die Klasse List<T> ist kovariant



Die Klasse Comparable<T> ist invariant

Kovarianz und Invarianz in Java

Alle generischen Typen sind in Java invariant

Konsequenz:

- Folgendes funktioniert nicht:

In [ ]:
```
LinkedList<Object> o = new LinkedList<String>();
```

Kovarianz und Invarianz in Java

Arrays sind kovariant in Java

- Folgendes funktioniert:

In [ ]:
```
Object[] objectArray = new String[100]
```

Kovarianz und Invarianz in Java

Arrays sind kovariant in Java

- Folgendes funktioniert:

In [ ]:
```
Object[] objectArray = new String[100]
```

Grund

- Arrays gabe es bevor Generics eingeführt wurde
  - Viele nützliche Funktionen auf generischen Arrays könnten nicht programmiert werden.
  - Beispiel: `void arrayEquals(object[] array1, object[] array2)`

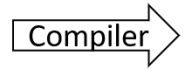
Type erasure

Type erasure

- Zur Kompilationszeit werden alle Typparameter ersetzt
  - Typinformation ist zur Laufzeit nicht vorhanden

Generischer Typ

```
class Stack<T> {
   T[] data = ...
   void push(T element) {}
   T pop()
}
```



Rohtyp

```
class Stack {
   Object[] data = ...
   void push(Object element) {}
   Object pop()
}
```

Erzeugen von Objekten von Generischem Typ

Nicht möglich!

In [ ]:
```
class Foo<T> {
    T t = new T();
}
```

- Keine Typinformation zur Laufzeit verfügbar.
  - Wie viel Speicher soll reserviert werden?
  - Welcher Konstruktor soll verwendet werden?

Erzeugen von generischen Arrays

- Folgendes funktioniert nicht:

In [ ]:
```
class Foo<T> {
   T[] t = new T[100];
}
```

- Funktionierender Hack!

In [ ]:
```
class Foo<T> {
   T[] arrayOfTs = (T[]) new Object[100];
}
```

Wildcards

Wildcards

Gemeinsamer Basistyp für Generische Objekte

- *Unabhängig von Parameter*

```
List<?> l = new List<Integer>();
```

- Erlaubt Zuweisung von Instanzen von `List<T>` für beliebige `T`
- Wird dann verwendet, wenn uns der Typ `T` nicht interessiert.

Anwendungsbeispiel

Wenn wir nur die Länge einer Liste ausgeben wollen, interessiert uns der konkrete Typ nicht.

In [ ]:
```
LinkedList<Integer> intList = new LinkedList<>();
intList.add(5);

LinkedList<String> stringList = new LinkedList<>();
stringList.push("abc");
stringList.push("def");

void lengthOfList(LinkedList<?> list) {
    System.out.println(list.size());
}
lengthOfList(intList);
lengthOfList(stringList);
```

Achtung: Über Typparameter ist nichts bekannt. Wir können diesen nicht verwenden.