
Arten des universellen Polymorphismus

- enthaltender Polymorphismus durch Untertypbeziehungen:
 - Ersetzbarkeit: ev. unvorhersehbare Wiederverwendung
 - kann Clients von lokalen Codeänderungen abschotten
 - nicht immer verwendbar (kovariante Probleme)
- parametrischer Polymorphismus = Generizität:
 - kaum Einschränkungen (auch für kovariante Probleme)
 - keine Ersetzbarkeit: nur vorhersehbare Wiederverwend.
 - Parameteränderung wirkt sich auf Clients aus
- Generizität und Untertypbeziehungen kombinierbar
 - Eigenschaften ergänzen einander

Generische Interfaces

```
interface Collection<A> {  
    void add (A elem);  
    Iterator<A> iterator();  
}  
  
interface Iterator<A> {  
    A next();  
    boolean hasNext();  
}
```

Verwendungsbeispiele:

- `Collection<String>` (enthält `void add (String elem)`)
- `Collection<Integer>` (enthält `void add (Integer elem)`)

Generische Klassen

```
public class List<A> implements Collection<A> {
    protected class Node {
        A elem; Node next = null;
        Node (A elem) { this.elem = elem; } }
    protected Node head = null, tail = null;
    protected class ListIter implements Iterator<A> {
        protected Node p = head;
        public boolean hasNext() { return p != null; }
        public A next() {
            if (p == null) return null;
            A elem = p.elem; p = p.next; return elem; } }
    public void add (A x) {
        if (head == null) tail = head = new Node(x);
        else tail = tail.next = new Node(x); }
    public Iterator<A> iterator() {return new ListIter();}}
```

Verwendung generischer Klassen

```
class ListTest {
    public static void main (String[] args) {
        List<Integer> xs = new List<Integer>();
        xs.add (new Integer(0));
        Integer x = xs.iterator().next();
        List<String> ys = new List<String>();
        ys.add ("zerro");
        String y = ys.iterator().next();
        List<List<Integer>> zs =
            new List<List<Integer>>();
        zs.add(xs);
        // zs.add(ys); ! Compiler meldet Fehler !
        List<Integer> z = zs.iterator().next();
    }
}
```

Generische Methoden

```
interface Comparator<A> {
    int compare (A x, A y);
}

class Collections {
    public static <A> A max (Collection<A> xs,
                           Comparator<A> c ) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (c.compare (w, x) < 0) w = x;
        }
        return w;
    }
}
```

Verwendung generischer Methoden

```
List<Integer> xs = ...;
List<String> ys = ...;
Comparator<Integer> cx = ...;
Comparator<String> cy = ...;
Integer rx = Collections.max (xs, cx);
String ry = Collections.max (ys, cy);
// ... rz = Collections.max (xs, cy); ! Fehler !
```

Gebundene Typparameter

```
interface StringHashable {  
    String hashCode();  
}
```

```
class StringHashMap<Key extends StringHashable, Value> {  
    public void put (Key k, Value v) {  
        String hash = k.hashCode();  
        ...  
    }  
    ...  
}
```

Rekursive Typparameter

```
interface Comparable<A> {
    int compareTo (A that); // res. < 0 if this < that
                          // res. == 0 if this == that
                          // res. > 0 if this > that
}

class MyInteger implements Comparable<MyInteger> {
    private int value;
    public MyInteger (int v) { value = v; }
    public int intValue() { return value; }
    public int compareTo (MyInteger that) {
        return this.value - that.value;
    }
}
```

Rekursive gebundene Typparameter

```
class Collections2 {
    public static <A extends Comparable<A>>
        A max (Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0)
                w = x;
        }
        return w;
    }
}
```

Generizität \neq Ersetzbarkeit

`X<A>` kein Untertyp von `X`
(wenn A und B ungleich)

`List<Student>` kein Untertyp von `List<Person>`

aber

`MyInteger` Untertyp von `Comparable<MyInteger>`

Wildcards als Typen

- `void drawAll (List<Polygon> p) { ... }`
Lesen und Schreiben des Inhalts von p
kein Argument vom Typ `List<Square>` oder `List<Object>`
- `void drawAll (List<? extends Polygon> p) { ... }`
nur Lesen des Inhalts von p
Aufruf mit Argument vom Typ `List<Square>` erlaubt
- `void addPolygon (List<? super Polygon> to) { ... }`
nur Schreiben des Inhalts von to
Aufruf mit Argument vom Typ `List<Object>` erlaubt

Wildcards und rekursive Typparameter

```
class ComparableList<A extends Comparable<? super A>>
    extends List<A> {
    public A max() {
        Iterator<A> xi = this.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next(); if (w.compareTo(x) < 0) w = x;
        }
        return w;
    }
}
```

Z.B.: `ComparableList<U>` möglich für Untertyp `U` von `MyInteger`