

---

# 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;
    }
}
```

---

# Keine impliziten Untertypbeziehungen

- Untertypbeziehung bei expliziter Vererbung

Beispiel: `class MyList<A> extends List<List<A>> { ... }`

es gilt: `MyList<Integer> ≤ List<List<Integer>>`

`MyList<String> ≤ List<List<String>>`

`MyInteger ≤ Comparable<MyInteger>`

- Untertypbeziehungen auf Typen, die Typparameter ersetzen, implizieren keine generischen Untertypbeziehungen

`MyList<Student>  $\not\leq$  MyList<Person>`

`MyList<Person>  $\not\leq$  MyList<Student>`

`MyList<MyInteger>  $\not\leq$  List<List<Comparable<MyInteger>>>`

---

# Java-Arrays sind nicht generisch

```
class NoLoophole {
    public static String loophole (Integer y) {
        List<String> xs = new List<String>();
        List<Object> ys = xs; // compile-time error
        ys.add(y);
        return xs.iterator().next();
    }
}

class Loophole {
    public static String loophole (Integer y) {
        String[] xs = new String[10];
        Object[] ys = xs; // no compile-time error
        ys[0] = y; // throws exception
        return xs[0];
    }
}
```

---

# 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`

---

# Einschränkungen von Java

Typparameter A überall erlaubt, wo in Java Typen erlaubt, aber mit Einschränkungen:

- nicht direkt zur Erzeugung neuer Objekte

```
new A()    ist illegal
```

```
new A[10]  ist illegal
```

- Typumwandlungen nur erlaubt, wenn gleiche Typparameter in deklariertem Obertyp

```
List<List<String>> x = new MyList<String>();
```

```
MyList<String> y = (MyList<String>)x; // erlaubt
```

```
Object a = new MyList<String>();
```

```
MyList<String> b = (MyList<String>)a; // Syntaxfehler!
```



---

# Übersetzung von Generizität

## **homogene Übersetzung:** (z. B. Java)

generische Klasse in eine nichtgenerische Klasse übersetzt

Typparameter durch Schranke oder Object ersetzt

Typumwandlungen für Rückgabewerte (immer erfolgreich)

## **heterogene Übersetzung:** (z. B. templates in C++)

unterschiedliche Klassen für unterschiedliche Typparameterersetzungen erzeugt („copy and paste“)

Vorteile: effizienter; int, char, ... direkt verwendbar

Nachteile: viele Klassen, schwer simulierbar

---

# Arten gebundener Generizität

- Untertypen gegebener Schranken erlaubt (Java, Eiffel)  
klar sichtbare Semantik  
aber geeignete Typhierarchien manchmal schwer festlegbar
- Eigenschaften implizit (heterogene Übersetzung, C++)  
flexibel und einfach, da kein gemeinsamer Obertyp nötig  
aber oft Fehlermeldungen von schlechter Qualität
- Eigenschaften explizit angeben (Ada, C++ concepts)  
noch flexibler, da Umbenennungen möglich  
aber längerer Programmcode nötig