# 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 !
```

# Anwendungsfälle für Generizität

- gleich strukturierte Klassen oder Methoden
- Abfangen erwarteter Änderungen
- manchmal verwendbar, wo Untertypbeziehungen versagen (Ersetzbarkeit für Generizität nicht benötigt)
- Laufzeiteffizienz sehr wichtig (kein dynamisches Binden)
- Verwendung wirkt natürlich (latente Erfahrung)

# Gleich strukturierte Klassen, Methoden

- typisch: Containerklassen und zugreifende Methoden (leicht zu erkennen)
- Einsatz von Generizität billig $\Rightarrow$ auf Verdacht verwenden (kleiner zusätzlicher Schreibaufwand)
- Schnittstellen ändern sich bei nachträglichem Einsatz (Hinweis auf teure Refaktorisierung)
- Trick bei Refaktorisierung (Hinzufügen von Generizität): ursprüngliche Klasse erbt von neuer generischer Klasse
- üblicher Programmcode enthält wenige generische Klassen (da typische Containerklassen bereits vordefiniert)

# Abfangen erwarteter Änderungen

- gleich strukturierte Klassen und Methoden auch in verschiedenen Programmversionen möglich
- auch für Klassen, die keine Container sind
- bei Verdacht, dass Parametertypen sich ändern:
  Typparameter als Parametertypen verwenden
- bei Typänderungen trotzdem Änderungen der Aufrufe
- statt direkt auf generische Klassen eher auf nichtgenerische Unterklassen davon zugreifen (nur unterste Ebene)

```
class Example<A> { ... }
class MyExample extends Example<Something> { }
```