Generizität – Liste für beliebige Typen

```
public class GenList<A> {
    private ListNode<A> head = null;

    public void add (A elem) {
        head = new ListNode<A> (elem, head);
    }
    public boolean contains (A elem) {
        return head != null && head.contains(elem);
    }
    public void remove (A elem) {
        if (head != null)
            head = head.remove(elem);
    }
}
```

```
class ListNode<A> {
    private A elem;
    private ListNode<A> next;

    ListNode (A e, ListNode<A> n) { elem = e; next = n; }

    boolean contains (A e) {
        return e.equals(elem) ||
               (next!=null && next.contains(e));
    }
    ListNode<A> remove (A e) {
        if (e.equals(elem))
            return next;
        else if (next != null)
            next = next.remove(e);
        return this;
    }
}
```

Anwendungsbeispiel

```
public class Student {
    private int mnr;
    ...
    public boolean equals (Object that) {
       return this.getClass() == that.getClass()
              && mnr==((Student)that).mnr;
    }
}

    GenList<Student> list = new GenList<Student>();
    list.add (new Student());
```

Referenztypen und Autoboxing

```
GenList<Integer> ilist = new GenList<Integer>();
ilist.add(1);
ilist.add(new Integer(678));

GenList<Boolean> blist = new GenList<Boolean>();
blist.add(true);
blist.add(new Boolean(false));

if (ilist.contains(678) {
    ...
}
```

Gebundene Generizität

```
public interface HasValue { int value(); }

public class SumList<A extends HasValue> {
    private ListNode<A> head = null;
    private int sum = 0;
    public void add (A elem) {
        head = new ListNode<A> (elem, head);
        sum += elem.value();
    }
    ...
}

public class Student implements HasValue {
    private int numberOfCourses;
    public int value() { return numberOfCourses; }
}
```

Rekursive gebundene Generizität

```
public interface Comparable<T> { int compareTo (T that); }
public class GenTree<A extends Comparable<A>> {
    private TreeNode<A> root = null; ...
}
class TreeNode<A extends Comparable<A>> {
    private A elem;
    boolean contains (A e) {
        int c = e.compareTo(elem);
        if (c < 0) ... else if (c == 0) ... else ...
    }
    ...
}
public class Student implements Comparable<Student> {
    public int compareTo (Student that) { ... }
}
```

Abstraktion über Datenstrukturen

```
public interface Collection<E> {
    void add (E e);           // füge e hinzu
    boolean contains (E e);   // ist e enthalten?
    void remove (E e);        // lösche ein e
}

public class GenList<A> implements Collection<A> { ... }
public class GenTree<A extends Comparable<A>>
                        implements Collection<A> { ... }

public class Words {
    private Collection<String> ws = new GenTree<String>;
    public void add (String w) {
        if (!ws.contains(w)) ws.add(w);
    }
}
```

Iteratoren

```
public interface Iterator<E> {
    A next();             // gib nächstes Element zurück
    boolean hasNext();  // gibt es weitere Elemente?
}
public interface Iterable<E> {
    Iterator<E> iterator();  // erzeuge Iterator
}
public interface Collection<E> extends Iterable<E> { ... }

    ...
    static void printAll (Collection<String> ws) {
        Iterator<String> wi = ws.iterator();
        while (wi.hasNext())
            System.out.println(wi.next());
    }
```

Iterator über Liste

```
public class GenList<A> implements Collection<A> {
    private ListNode<A> head = null; ...
    public Iterator<A> iterator() {
        return new ListIter<A>(head);
    }
}
class ListIter<A> implements Iterator<A> {
    private ListNode<A> n;
    ListIter (ListNode<A> head) { n = head; }
    public boolean hasNext() { return n != null; }
    public A next() {
        if (n == null) return null;
        ListNode<A> r = n.elem; n = n.next; return r;
    }   // Sichtbarkeit von elem und next beachten
}
```

Stack als Liste

```
public class GenStack<A> {
    private ListNode<A> top = null;
    public void push (A elem) {
        top = new ListNode<A>(elem, top);
    }
    public A pop() {
        if (top == null)
            return null;
        A r = top.elem;  // Sichtbarkeit beachten
        top = top.next;
        return r;
    }
    public boolean isEmpty() { return top == null; }
}
```

Vorgefertigte Teile

- Collection, Iterator: Paradebeispiele für vorgefertigte Teile
- schreibt man kaum selbst, sondern verwendet sie einfach
- erprobt und gut unterstützt

```
static void printAll (Collection<String> ws) {
    for (String s: ws)
        System.out.println(s);
}
```

Vorgefertigte Teile – Hindernisse

Hinderung an der Verwendung fertiger Teile durch:

- Unkenntnis
- unterschiedliche Modelle
- mangelndes Vertrauen
- „Ich kann es besser“

Verwendung zahlt sich aus, ist aber mit Arbeit verbunden

Top Down

```
public static void main (String[] args) {
    List<Integer> nums = readNums();
    Collections.sort(nums);
    print(nums);
}
private static List<Integer> readNums () {
    Scanner sc = new Scanner(System.in);
    List<Integer> nums = new LinkedList<Integer>();
    while (sc.hasNextInt())
        nums.add(sc.nextInt());
    return nums;
}
private static void print (List<Integer> nums) {
    for (int i: nums)
        System.out.println(i);
}
```

Bottom Up

```
public class SortedNums {
    private GenTree<Integer> nums = new GenTree<Integer>();
    public void readFrom (Scanner in) {
        while (in.hasNextInt())
            nums.add(in.nextInt());
    }
    public void printTo (PrintStream out) {
        for (int i: nums)
            out.println(i);
    }
    public static void main (String[] args) {
        SortedNums nums = new SortedNums();
        nums.readFrom(new Scanner(System.in));
        nums.printTo(System.out);
    }
}
```