Pakete

- eine public Klasse pro Datei
- Paket umfaßt alle Dateien bzw. Klassen im selben Ordner
- explizite Paketdeklaration: `package paketName;`
- Aufruf von `foo()` in der Datei `myclasses/test/AClass.java`:

  ```
  myclasses.test.AClass.foo()
  ```

- Kürzer durch Import-Deklaration (am Dateianfang)

```
import myclasses.test;          ... test.AClass.foo() ...
import myclasses.test.AClass;       ... AClass.foo() ...
import myclasses.test.*;            ... AClass.foo() ...
```

Sichtbarkeit

	`public`	`protected`	default	`private`
lokal sichtbar	ja	ja	ja	nein
global sichtbar	ja	nein	nein	nein
lokal vererbbar	ja	ja	ja	nein
global vererbbar	ja	ja	nein	nein

lokal = im selben Paket, auch außerhalb der Klasse

global = auch außerhalb des Pakets

Anwendung der Sichtbarkeitskontrolle

Public: Bei der Verwendung der Klasse und deren Instanzen benötigte Methoden, Konstanten und (selten) Variablen

Private: Methoden und Variablen, die nur innerhalb der Klasse verwendet werden sollen — vor allem, wenn Bedeutung außerhalb der Klasse nicht klar

Protected: Methoden und Variablen für Verwendung nicht benötigt, aber hilfreich bei Erweiterungen der Klasse

Default: Eng zusammenarbeitende Klassen im selben Paket sollen auf Methoden und Variablen zugreifen können, die sonst privat wären

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 !
```

Anwendungsfälle für Generizität

- gleich strukturierte Klassen oder Methoden
- erwartete Änderungen (= gleich strukturierte Versionen)
- in Spezialfällen verwendbar, wo Untertypen versagen
  (da Ersetzbarkeit für Generizität nicht nötig)
- zur Vermeidung des Overheads von dynamischem Binden
  (aber: nur selten und nur wenig bessere Effizienz und
  viele Nachteile durch Verzicht auf Ersetzbarkeit)
- Verwendung wirkt natürlich (durch latente Erfahrung)

ABER: Generizität garantiert keine Ersetzbarkeit

Gleich strukturierte Klassen, Methoden

- typisch: Containerklassen und zugreifende Methoden
  (leicht als solche zu erkennen)
- Einsatz von Generizität billig $\Rightarrow$ auf Verdacht verwenden
- Trick bei Refaktorisierung (Hinzufügen von Generizität):
  ursprüngliche Klasse erbt von neuer generischer Klasse

```
class Alt {...T...}   →   class Neu<A> {...A...}
                          class Alt extends Neu<T> {}
```

- üblicher Programmcode enthält wenige generische Klassen,
  verwendet aber viele
  (da zahlreiche typische Containerklassen vordefiniert)

Abfangen erwarteter Änderungen

- gleiche Struktur von Klassen und Methoden in unterschiedlichen Programmversionen
- auch für Klassen, die nicht offensichtlich Container sind
- bei Verdacht, dass Typen sich in neuen Versionen ändern: Typparameter 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> { }
```