Generizität (Umgang mit Typparametern)

Dynamische Typinformation

Homogene Übersetzung der Generizität

Generizität und Typumwandlungen

Universeller Polymorphismus

enthaltender Polymorphismus durch *Untertypbeziehungen:*

- Ersetzbarkeit: unvorhersehbare Wiederverwendung,
  kann Clients von lokalen Codeänderungen abschotten,
  nicht immer verwendbar (kovariante Probleme)

parametrischer Polymorphismus = *Generizität:*

- kaum Einschränkungen (auch für kovariante Probleme),
  keine Ersetzbarkeit,
  Parameteränderung wirkt sich auf Clients aus

Generizität und Untertypbeziehungen ergänzen einander

Generische Interfaces

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

Verwendungsbeispiele:

Typ eingesetzt für Typparameter

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

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

Generische Klasse (mit inneren Klassen)

```
public class List<A> implements Collection<A> {
    private class Node {
        private A elem; private Node next = null;
        private Node(A elem) { this.elem = elem; }
    }
    private Node head = null, tail = null;
    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 Iterator<A>() {
            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; }
        };
    }
}
```

Generische Klasse (ohne innere Klassen)

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

Verwendung generischer Klasse

```
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 Methode

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

```
List<Integer> xs = ...;
List<String>  ys = ...;

Comparator<Integer> cx = ...;
Comparator<String>  cy = ...;

Integer rx = CollectionOps.max(xs, cx);
String  ry = CollectionOps.max(ys, cy);
// ...  rz = CollectionOps.max(xs, cy); ! Fehler !
```

Gebundene Typparameter

```
public interface Scalable {
    void scale(double factor);
}

public class Scene<T extends Scalable> implements Iterable<T> {
    public void addSceneElement(T e) { ... }
    public Iterator<T> iterator() { ... }
    public void scaleAll(double factor) {
        for (T e : this)
            e.scale(factor);
    }
    ...
}
```

Rekursive Typparameter

```
public 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 CollectionOps2 {
    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 $\not\rightarrow$ Ersetzbarkeit

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

daher `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`
aber kein Argument vom Typ `List<Square>` oder `List<Object>`

```
void drawAll(List<? extends Polygon> p) { ... }
```

Aufruf mit Argument vom Typ `List<Square>` erlaubt
aber nur Lesen des Inhalts von `p` (kein Schreiben) — nur wo Typen kovariant

```
void addPolygon(List<? super Polygon> to) { ... }
```

Aufruf mit Argument vom Typ `List<Object>` erlaubt
aber nur Schreiben des Inhalts von `to` (kein Lesen) — nur wo Typen kontravariant

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`

Dynamische Typabfragen

Abfrage der Klasse eines Objects:

```
Class y = x.getClass();
```

dynamische Untertypabfrage:

```
public int calculateTicketPrice(Person p) {
    if (p.age < 15 || p instanceof Student)
        return standardPrice / 2;
    return standardPrice;
}
```

Explizite Typumwandlung

Typumwandlungen zur Nutzung dynamischer Typinformation:

```
public class Point3D extends Point2D {
    private int z;

    public boolean equal(Point2D p) {
        if (p instanceof Point3D)
            return super.equal(p) && ((Point3D)p).z == z;
        return false;
    }
}
```

(FALSCH! Bessere Versionen von `equal` folgt gleich)

equal – besserer Ansatz

```
public abstract class Point {
    public final boolean equal(Point that)
        { return this.getClass()==that.getClass() && uncheckedEq(that); }
    protected abstract boolean uncheckedEq(Point p);
}
public class Point2D extends Point {
    private int x, y;
    protected boolean uncheckedEq(Point p)
        { return x==((Point2D)p).x && y==((Point2D)p).y; }
}
public class Point3D extends Point {
    private int x, y, z;
    protected boolean uncheckedEq(Point p)
        { Point3D that = (Point3D)p;
          return x==that.x && y==that.y && z==that.z; }
}
```

Dynamische Typabfragen vermeiden

Typabfragen sparsam einsetzen da oft zur Umgehung statischer Typsicherheit eingesetzt

Beispiel für Vermeidung:

```
if (x instanceof T1)
    doSomethingOfTypeT1((T1)x);
else if (x instanceof T2)
    doSomethingOfTypeT2((T2)x);
...
else
    doSomethingOfAnyType(x);
```

ersetzen durch `x.doSomething();` (Methoden in `T1`, `T2`, ...)

Generizität: Homogene Übersetzung

wird in Java verwendet,
übersetzt eine generische in *eine* nicht-generische Klasse

Schritte:

1. spitze Klammern samt Inhalten weglassen
2. Typparameter durch (erste) Schranke oder `Object` ersetzen
3. Typumwandlungen bei Aufrufen einfügen
   wenn Ergebnis- oder Parametertyp Typparameter ist

Beispiel 1

```
public interface Collection {  // <A> weggelassen
    void add(Object elem);     // A durch Object ersetzt
    Iterator iterator();       // <A> weggelassen
}
public interface Iterator {    // <A> weggelassen
    Object next();             // A durch Object ersetzt
    boolean hasNext();         // unveraendert
}
public class List implements Collection {  // 2x <A> weg
    private class Node {
        private Object elem; private Node next = null;
        private Node(Object elem) { this.elem = elem; }
    }
    ... // ueberall <A> weg, A durch Object ersetzt
}
```

Beispiel 2

```
List xs = new List();                          // <Integer> weg
xs.add((Integer)(new Integer(0)));             // Typumwandlung
Integer x = (Integer)xs.iterator().next(); // Typumwandlung

List ys = new List();                          // <String> weg
ys.add((String)"zerro");                       // Typumwandlung
String y = (String)ys.iterator().next();   // Typumwandlung
```

Sichere Typumwandlungen

Up-Cast = Umwandlung in Obertyp

Down-Cast nach *dynamischer Typabfrage*
aber alternativer Programmzweig nötig und fehleranfällig

Down-Cast *wie bei Generizität*, aber nur händisch überprüft:

gleichförmige Ersetzung der Typparameter und
keine impliziten Untertypbeziehungen
wobei Intuition oft irreführend:

`List<String>` $\not\leq$ `List<Integer>`
impliziert `List` $\not\leq$ `List` nach Ersetzung

Java-Arrays und Generizität

`new A(...)` oder `new A[...]`: `A` darf kein Typparameter sein

```
public class NoLoophole {
    public static String loophole(Integer x) {
        List<Object> xs = new List<String>();        // error
        xs.add(x);
    }
}
public class Loophole {
    public static String loophole(Integer x) {
        Object[] xs = new String[10];             // no error
        xs[0] = x;                    // exception at runtime
        return xs[0];
    }
}
```

Generische Typvergleiche und Typumwandlungen

```
<A> Collection<A> up(List<A> xs) {
    return (Collection<A>)xs;
}

<A> List<A> down(Collection<A> xs) {
    if (xs instanceof List<A>)
        return (List<A>)xs;
    else { ... }                        // was tun?
}

List<String> bad(Object o) {
    if (o instanceof List<String>)      // error
        return (List<String>)o;         // error
    else { ... }
}
```

Verwendung von Raw-Types

```
class List<A> implements Collection<A> {
    ...
    public boolean equals(Object that) {
        if (!(that instanceof List)) return false;
        Iterator<A> xi = this.iterator();
        Iterator yi = ((List)that).iterator();
        while (xi.hasNext() && yi.hasNext()) {
            A x = xi.next();
            Object y = yi.next();
            if (!(x == null ? y == null : x.equals(y)))
                return false;
        }
        return !(xi.hasNext() || yi.hasNext());
    }
}
```

Generizität: Heterogene Übersetzung

durch Copy-and-Paste eigene Klasse (Methode) pro Typparametererersetzung

erzeugt so viele nicht-generische Klassen (Methoden) wie nötig

Vorteile: effizienter da keine Typumwandlung und Code optimierbar
`int`, `char`, ... direkt verwendbar
Typparameter uneingeschränkt verwendbar

Nachteile: oft viele Klassen und große Programme
Klassenvariablen kopiert (= unklare Semantik)
keine Raw-Types verwendbar
generischer und nichtgenerischer Code inkompatibel