Tagesprogramm

Dynamische Typinformation

Homogene Übersetzung der Generizität

Generizität und Typumwandlungen

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

Aufgabe: Typumwandlungen und Typabfragen

Such Sie in Gruppen zu zwei bis drei Personen Antworten auf folgende Fragen:

1. Warum soll man nicht nur Typumwandlungen sondern auch dynamische Typabfragen vermeiden?
2. In welchen Fällen sind dynamische Typabfragen und Typumwandlungen kaum durch dynamisches Binden ersetzbar?
3. Gibt es neben dynamischem Binden andere Möglichkeiten, dynamische Typabfragen und Typumwandlungen zu vermeiden?

Zeit: 2 Minuten

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];
    }
}
```

gen. 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 Typparameterersetzung

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