
Dynamische Typabfragen

Abfrage der Klasse eines Objects:

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

dynamische Untertypabfrage:

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

Explizite Typumwandlung

Typumwandlungen zur Nutzung dynamischer Typinformation:

```
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!! — verbesserte Versionen von `equal` folgen später)

Dynamische Typabfragen vermeiden

Typabfragen sparsam einsetzen!

Grund: dienen oft zur Umgehung statischer Typsicherheit

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

- übersetzt eine generische in *eine* nicht-generische Klasse
- wird in Java verwendet
- Schritte:
 - spitze Klammern (mit Inhalt) weglassen
 - Typparameter durch Schranke oder `Object` ersetzen
 - Typumwandlungen für Rückgabewerte (wenn Ergebnistyp Typparameter ist)

Typumwandlungen und Generizität (1)

```
interface Collection {
    void add (Object elem);
    Iterator iterator();
}

interface Iterator {
    Object next();
    boolean hasNext();
}

class List implements Collection {
    protected class Node {
        Object elem;  Node next = null;
        Node (Object elem) { this.elem = elem; }
    }
    ...
}
```

Typumwandlungen und Generizität (2)

```
List xs = new List();  
xs.add (new Integer(0));  
Integer x = (Integer)xs.iterator().next();  
List ys = new List();  
ys.add ("zerro");  
String y = (String)ys.iterator().next();
```

Sichere Typumwandlungen

- Umwandlung in Obertyp (up-cast)
- nach dynamischer Typabfrage (down-cast)
?alternativer Programmzweig? (fehleranfällig)
- wie bei Generizität, aber nur händisch überprüft
 - gleichförmige Ersetzung der Typparameter
 - keine impliziten Untertypbeziehungen

Achtung: `List<String>` $\not\leq$ `List<Integer>`
impliziert `List` $\not\leq$ `List` (nach Ersetzung)

Intuition oft irreführend

gen. Typvergleiche und -umwandlungen

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

Verwendung übersetzter gen. Klassen

```
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

- erzeugt durch „copy and paste“ unterschiedliche Klassen für unterschiedliche Typparameterersetzungen
- erzeugt so viele nicht-generische Klassen wie nötig
- z. B. templates in C++
- Vorteile: effizienter; `int`, `char`, ... direkt verwendbar
- Nachteile: viele Klassen, Klassenvariablen kopiert

Arten gebundener Generizität

- Untertypen gegebener Schranken erlaubt (Java, Eiffel)
klar sichtbare Semantik
aber geeignete Typhierarchien manchmal schwer festlegbar
- Eigenschaften implizit (heterogene Übersetzung, C++)
flexibel und einfach, da kein gemeinsamer Obertyp nötig
aber oft Fehlermeldungen von schlechter Qualität
- Eigenschaften explizit angegeben (Ada, bald in C++)
noch flexibler, da Umbenennungen möglich
aber längerer Programmcode nötig