Generische Typen in Java 1.5

Die Erweiterung der Java Language Specification

Seminarvortrag von Heiko Minning, mi3795
bei Prof. Dr. Uwe Schmidt, FH-Wedel
Wintersemester 2002/2003

Definition: Generizität

=> parametrisierung mit Typen
=> (Referenz-)Typen als Parameter

=> zusätzliche Abstraktionsebene (kein OO Konzept!)

=> allgemeingültiger Code (Wiederverwendung)
=> Typsicherheit

Problem

```
LinkedList list = new LinkedList();
list.add(new Byte((byte)0));
Byte x = (Byte) list.get(0);

LinkedList listlist = new LinkedList();
listlist.add(list);
Byte z = (Byte) ((LinkedList)listlist.get(0)).get(0);

String s = (String) list.get(0); // ClassCastException
```

Lösung?

- Wrapper-Klassen („StringList“) „schlecht“
- Generizität „gut!“

Mit Java 1.5 (generische Klassen)

```
    LinkedList<Byte> list = new LinkedList<Byte>();
    list.add(new Byte((byte)0));
    Byte x = list.get(0);

    LinkedList<LinkedList<Byte>> listlist = new LinkedList<LinkedList<Byte>>();
    listlist.add(list);
    Byte z = listlist.get(0).get(0);

    String s = list.get(0); // Fehler zur Compile-Zeit!


/* Ausgabe des Compilers:
 * incompatible types
 * found   : java.lang.Byte
 * required: java.lang.String
 *  String s = list.get(0)
 *                  ^
 */
```

Die „andere Seite“

```
interface Collection {
  public void add(Object x);
  public Object get(int index);
}

class LinkedList implements Collection {
  private Object head;
  private LinkedList next;

  public void add(Object x){ /*...*/ }
  public Object get(int index){ /*...*/ return null; }

}
```

in Java 1.5

```
interface Collection<A> {
  public void add(A x);
  public A get(int index);
}

class LinkedList<A> implements Collection<A> {
  private A head;
  private LinkedList<A> next;

  public void add(A x){ /*...*/ }
  public A get(int index){ /*...*/ return null; }

}
```

Generische Methoden: (zunächst 1.4.1 Code)

```
interface Comparator {
  public int compare(Object x, Object y);
}

class ByteComparator implements Comparator {
  public int compare(Object x, Object y){
    return ((Byte)x).byteValue() - ((Byte)y).byteValue();
  }
}

class CollectionOp {
  public static Object max(Collection xs, Comparator c){
    Iterator xi = xs.iterator();
    Object w = xi.next();
    while(xi.hasNext()){
      Object x = xi.next();
      if(c.compare(w,x) < 0) w = x;
    }
    return w;
  }
}
```

Das dazugehörige Testprogramm

```
class Test {
  public static void main(String[] args){
    LinkedList xs = new LinkedList();
    xs.add(new Byte((byte)0));
    xs.add(new Byte((byte)1));
    Byte x = (Byte) CollectionOp.max(xs, new ByteComparator());

    LinkedList ys = new LinkedList();
    ys.add("zero");
    ys.add("one");
    String y = (String) CollectionOp.max(ys, new ByteComparator()); // ClassCastException
  }
}
```

Generische Methoden in Java 1.5

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

class ByteComparator implements Comparator<Byte> {
  public int compare(Byte x, Byte y){
    return x.byteValue() - y.byteValue();
  }
}

class CollectionOp {
  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;
  }
}
```

Das dazugehörige Testprogramm

```
class Test {
  public static void main(String[] args){
    LinkedList<Byte> xs = new LinkedList<Byte>();
    xs.add(new Byte((byte)0));
    xs.add(new Byte((byte)1));
    Byte x = CollectionOp.max(xs, new ByteComparator());

    LinkedList<String> ys = new LinkedList<String>();
    ys.add("zero");
    ys.add("one");
    String y = CollectionOp.max(ys, new ByteComparator()); // compile-Fehler!
  }
}


/* Ausgabe des Compilers:
 * max<A>(java.util.Collection<A>,Comparator<A>) in CollectionOp cannot be applied to
 * (java.util.LinkedList<java.lang.String>,ByteComparator)
 */
```

Gut, aber:

Wie wird <A> in CollectionOp.max(...) bestimmt? => Typ-Inferenz

Bestimmung der Typen

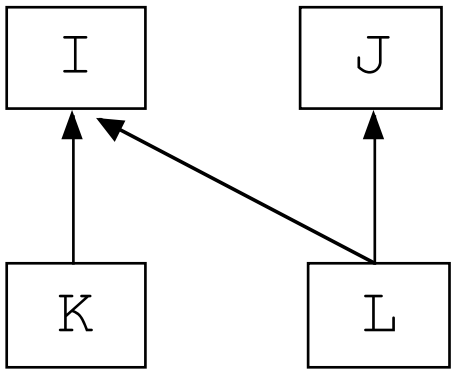
```
interface I {}
interface J {}
interface K extends I {}
interface L extends I, J {}

class X {

  public static void test(K k, L l){
    I i = choose(k,l);
  }

  public static <A> A choose(A x, A y){
    return (x.hash() < y.hash()) ? x : y;
  }

}
```



Und wenn es keine Parameter gibt?

```
public static <A> A get(){...}
public static <A> LinkedList<A> get2(){...}
```

=> Null-Typ (*)

=> Typ der Konstanten „null“ (*)
=> Subtyp aller Referenz-Typen

Beispiele:

- * ist Subtyp von Integer
- LinkedList<*> ist Subtyp von LinkedList<String>
- Pair<Byte,*> ist Subtyp von Pair<Byte, Byte>

Beachte: Linearitätsbeschränkung!
Sonst: Umgehung des Typsystems

Beispiel für mögliche Typverletzung

```
class Cell<A> {
  public A value;
  public Cell (A v) { value = v; }
  public static <B> Cell<B> allocate (B x) { return new Cell<B>(x); }
}

class Pair<A,B> {
  public A fst;
  public B snd;
  public Pair (A x, B y) { fst = x; snd = y; }
  public static <C> Pair<C,C> duplicate (C x) { return new Pair<C,C>(x,x); }
}

class Loophole {
  public static String loophole (Byte y) {
    Pair<Cell<String>,Cell<Byte>> p =
       Pair.duplicate(Cell.allocate(null)); // compile-time error
    p.snd.value = y; return p.fst.value;
  }

  public static String permitted (String x) {
    Pair<Cell<String>,Cell<String>> p =
       Pair.duplicate(Cell.allocate((String)null));
    p.fst.value = x; return p.snd.value;
  }
}
```

Beschränkte Typparameter

```
class CollectionOp {
  public static Comparable max(Collection xs){
    Iterator xi = xs.iterator();
    Comparable w = (Comparable) xi.next();
    while(xi.hasNext()){
      Comparable x = (Comparable) xi.next();
      if(w.compareTo(x) < 0) w = x;
    }
    return w;
  }
}
```

=> als generischer Code?

Umsetzung mit Generizität

```
class CollectionOp {
  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;
  }
}
```

weitere Beschränkungen möglich:

```
<A extends Number & Comparable<A> & Clonable>
```

die Übersetzung

=> Typlöschung (homogen)
=> „Vector<A>“ wird zu „Vector“
=> „A extends Comparable<A>“ wird zu „Comparable“
=> „A“ wird zu „Object“
=> einfügen von sicheren Casts

=> Code häufig identisch zu „Object“-Programmierung

=> keine Typinformation zur Laufzeit
=> gleiche Performance
=> keine Änderung an ByteCode oder JVM
=> vollständige Kompatibilität

RawType

parametrisierte Typen ohne Parameter
„Vector“ ist der RawType von „Vector<String>“

Zuweisung an RawType immer möglich
=> bei Benutzung evtl. unchecked warnung
Umgekehrte Zuweisung: unchecked warnung

```
class Loophole {
  public static String loophole (Byte y) {
    LinkedList<String> xs = new LinkedList<String>();
    LinkedList ys = xs;
    ys.add(y);              // unchecked warnung
    return xs.get(0);       // ClassCastException
  }
}
```

Zwei Fälle von unchecked warnung

```
class Testit<A>{
  public A v;
  public void set(A v){ this.v=v; }
}

public class Test{
  public static void main(String[] args){
    Testit t = new Testit(); // oder auch  = new Testit<Boolean>()
    t.v = Boolean.TRUE;
    t.set(Boolean.TRUE);
  }
}

/* Ausgabe des Compilers:
 * Test.java:9: warning: unchecked assignment to variable v of raw type
 * class Testit
 *     t.v = Boolean.TRUE;
 *      ^
 * Test.java:10: warning: unchecked call to set(A) as a member of the raw
 * type Testit
 *     t.set(Boolean.TRUE);
 *      ^
 * 2 warnings
 */
```

Das Problem mit der Vererbung

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

class ByteComparator implements Comparator<Byte> {
  public int compare(Byte x, Byte y){
    return x.byteValue() - y.byteValue();
  }
}
```

Übersetzt zu

```
class ByteComparator implements Comparator {
  public int compare(Byte x, Byte y){
    return x.byteValue() - y.byteValue();
  }

  public int compare(Object x, Object y){
    return this.compare((Byte) x, (Byte) y);
  }
}
```

Brücken mit gleicher Signatur

```
interface Iterator<A> {
  public boolean hasNext();
  public A next();
}

class Interval implements Iterator<Integer> {
  private int i;
  private int n;
  public Interval(int l, int u) { i = l; n = u; }
  public boolean hasNext() { return (i <= n); }
  public Integer next() { return new Integer(i++); }
}
```

Übersetzt zu

```
...

class Interval implements Iterator {
  private int i;
  private int n;
  public Interval(int l, int u) { i = l; n = u; }
  public boolean hasNext() { return (i <= n); }
  public /*1*/ Integer next() { return new Integer(i++); }
  // bridge
  public /*2*/ Object next() { return /*1*/ next(); }
}
```

Covariante Rückgabe bei Methoden (optional)

```
class C implements Cloneable {
  public C copy() { return (C)this.clone(); }
}

class D extends C {
  public D copy() { return (D)this.clone(); } // in Java 1.4.1 illegal!
}
```

Die Übersetzung:

```
class D extends C {
  public /*1*/ D copy() { return (D)this.clone(); }
  // bridge
  public /*2*/ C copy() { return /*1*/ this.copy(); }
}
```

Arrays

```
class BadArray {
  public static <A> A[] singleton (A x) {
    return new A[]{ x }; // unchecked warning
  }
  public static void main (String[] args) {
    String[] a = singleton("zero"); // run-time exception
  }
}
```

Ableitung parametrischer Typen ist invariant

Beispiel

`Vector<String>`

ist kein Subtyp von

`Vector<Object>`

Covariante Ableitung würde Typsystem umgehen

```
class Loophole {
  public static String loophole (Byte y) {
    LinkedList<String> xs = new LinkedList<String>();
    LinkedList<Object> ys = xs; // compile-time error „incompatible types“
    ys.add(y);                  // mögliche Typverletzung
    return xs.get(0);
  }
}
```

Aber

```
Collection<String> c = new LinkedList<String>(); // ist ok!
```

Casts von parametrischen Typen (optional)

Legal:

```
class Convert {
  public static <A> Collection<A> up (LinkedList<A> xs) {
    return (Collection<A>)xs;
  }
  public static <A> LinkedList<A> down (Collection<A> xs) {
    if (xs instanceof LinkedList<A>)
      return (LinkedList<A>)xs;
    else throw new ConvertException();
  }
}
```

Illegal:

```
class BadConvert {
  public static Object up (LinkedList<String> xs) {
    return (LinkedList<Object>) xs;      // compile-time error
  }
  public static LinkedList<String> down (Object o) {
    if (o instanceof LinkedList<String>) // compile-time error
      return (LinkedList<String>)o;      // compile-time error
    else throw new ConvertException();
  }
}
```

Zusammenfassung

- generische, optional beschränkte Typen
- wechselseitige Rekusion möglich
- polymorphe Klassen und Methoden
- covariante Methodenüberschreibung für Rückgabeparameter
- invariante Ableitung von parametrisierten Typen
- homogene Übersetzung mit Brückenmethoden, und Typlöschung
- keine generische Laufzeitinformation
- rückwartskompatibel und vorwärtskompatibel
- keine Änderung an ByteCode oder JVM
- es bleiben viele Unklarheiten offen (Spezifikation ist noch nicht fertig!)

Quellen:

- Bracha, Gilad et al.:
  Adding Generics to the Java Programming Language: Participant Draft Specification April 2001:
  http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html

- Bracha, Gilad et al.:
  Making the future safe for the past: Adding Genericity to the java programming language, OOPSLA 1998:
  http://www.research.avayalabs.com/user/wadler/pizza/gj/Documents/gj-oopsla.pdf

- Bracha, Gilad et al.:
  GJ: Extending the Java Programming language with type parameters,
  http://www.research.avayalabs.com/user/wadler/pizza/gj/Documents/gj-tutorial.pdf

weitere Quellen: siehe Ausarbeitung