Generic Java

Jan Bernhardt

15. Januar 1999

Was ist Generic Java?

- Erweiterung von Java um *Generizität*
- Generizität: Typen als Argumente
- Typ-parametrisierbare Klassen und Methoden
- Ada: *Generics*, C++: *Templates*
- GJ-Syntax vergleichbar mit C++-Templates
- Abwärtskompatibel: Java-Programme haben die gleiche Semantik unter GJ
- Autoren:
  - Philip Wadler, Bell Labs
  - Martin Odersky, University of South Australia
  - Gilad Bracha, Sun
  - David Stoutamire, Sun

Motivation für Generic Java

- Bessere Wiederverwendbarkeit des Codes
- Kürzerer Code
- Bessere Lesbarkeit
- Verbesserte Typsicherheit
- Auf- und abwärtskompatibel
- Effizienz

Generizität und Standard-Java: *Generic Idiom*

- Generische Typen können in Java simuliert werden:
  - Am Anfang der Klassenhierachie steht die Klasse `Object`, d.h. jedes Java-Objekt ist als Instanz der Klasse `Object` referenzierbar
  - "Generische" Typen als Parameter für generische Parameter Referenzen auf Objekte des allgemeinsten Typ `Object`
  - Typecasts

Ein Beispiel: Collections

- Klasse zum Aufnehmen von Objekten (Container, Behältertyp)
- Schnittstellen:
  - `add`: Methode, um Objekte hinzuzufügen
  - `iterate`: Methode, um die Objekte in dem Container aufzuzählen
- Code:

```
interface Collection {
  public void add( Object data );
  public Iterator iterate();
}
```

Iteratoren

- Klasse zum Aufzählen der Elemente einer Collection
- Schnittstellen:
  - `hasNext`: stellt fest, ob weitere Elemente in der Collection enthalten sind
  - `next`: Methode, um das nächste Element aus der Collection zu erhalten
- Code:

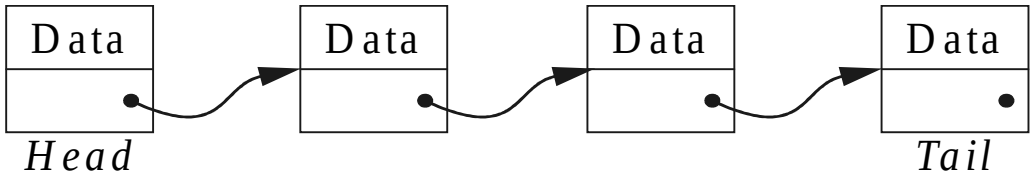
```
interface Iterator {
  public Object next();
  public boolean hasNext();
}
```

Implementierung einer verketteten Liste in Java

```
class LinkedList implements Collection {
  Node head = null;
  Node tail = null;

  public void add( Object data ) {
    // Hinzufügen eines Elements
  }

  public Iterator iterate() {
    // Rückgabe eines Iterators für die
    // verkettete Liste
  }
}
```



Implementierung

```
class LinkedList implements Collection {
  Node head = null, tail = null;
  public void add( Object data ) {
    if( head == null ) {
      head = new Node( data );
      tail = head;
    } else {
      tail.next = new Node( data );
      tail = tail.next;
    }
  }
  public Iterator iterate() {
    return new Iterator() {
      Node actual = head;
      public boolean hasNext() {
        return actual != null;
      }
      public Object next() {
        if( actual != null ) {
          Object o = actual.data;
          actual = actual.next;
          return o;
        } else {
          throw new NoSuchElementException();
        }
    };
  }
}
```

Benutzung der `LinkedList` in Java

```
LinkedList bl = new LinkedList();
bl.add( new Byte(1) );
bl.add( new Byte(2) );
Byte b = (Byte) bl.iterate().next();

LinkedList sl = new LinkedList();
sl.add( "eins" );
sl.add( "zwei" );
String s = (String) sl.iterate().next();

// Die nächste Instruktion führt zu einem
// Laufzeitfehler!
Byte anotherB =
  (Byte) sl.iterate().next();
```

Implementierung in GJ

```
interface Collection<A> {
  public void add( A data );
  public Iterator<A> iterate();
}

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

class LinkedList<A> implements Collection<A> {
  Node head = null, tail = null;

  public void add( A data ) {
    // Hinzufügen eines Elements
  }
  public Iterator<A> iterate() {
    // Rückgabe eines Iterators für die
    // verkettete Liste
  }
}
```

`LinkedList` in GJ

```
class LinkedList<A> implements Collection<A> {
  Node head = null, tail = null;
  public void add( A data ) {
    if( head == null ) {
      head = new Node( data ); tail = head;
    } else {
      tail.next = new Node( data );
      tail = tail.next;
    }
  }
  public Iterator<A> iterate() {
    return new Iterator<A> () {
      Node actual = head;
      public boolean hasNext() { ... }
      public A next() {
        ...
      } else { throw new NoSuchElement(); }
    };
  }
}
```

Benutzung der `LinkedList` in GJ

```
LinkedList<Byte> bl = new LinkedList<Byte>();
bl.add( new Byte(1) );
bl.add( new Byte(2) );
Byte b = bl.iterate().next();

LinkedList<String> sl =
  new LinkedList<String>();
sl.add( "eins" );
sl.add( "zwei" );
String s = sl.iterate().next();

// In GJ führt die nächste Instruktion zu
// einem Kompilierfehler.
Byte anotherB = sl.iterate().next();
```

Wie wird Generic Java übersetzt?

- Übersetzung von GJ in Java
- Verfahren: *erasure*
  - Beseitigen des Typparameters: `<A>`
  - Ersetzung von Typvariablen in der Regel durch Typ `Object`
  - Einfügen von passenden Typecasts um Methodenaufrufe, bei denen der Ergebnistyp eine Typvariable ist
- Der erzeugte Code entspricht dem, was man in Java nach dem *Generic Idiom* programmiert hätte
- Der GJ-Compiler garantiert, daß kein eingefügter Typecast fehlschlägt (*cast-iron-guarantee*)

Übersetzung 1

Generic Java:

```
class LinkedList<A> implements Collection<A> {
  Node head = null, tail = null;
  public void add( A data ) {
    // Hinzufügen eines Elements
  }
  public Iterator<A> iterate() {
    // Rückgabe eines Iterators für die
    // verkettete Liste
  }
}
```

Übersetzung:

```
class LinkedList implements Collection {
  Node head = null, tail = null;
  public void add( Object data ) {
    // Hinzufügen eines Elements
  }
  public Iterator iterate() {
    // Rückgabe eines Iterators für die
    // verkettete Liste
  }
}
```

Übersetzung 2

Generic Java:

```
LinkedList<Byte> bl = new LinkedList<Byte>();
bl.add( new Byte(1) );
bl.add( new Byte(2) );
Byte b = bl.iterate().next();
```

Übersetzung:

```
LinkedList bl = new LinkedList();
bl.add( new Byte(1) );
bl.add( new Byte(2) );
Byte b = (Byte) bl.iterate().next();
```

Erweiterung um Vergleichbarkeit

- Vergleiche zwischen Objekten ermöglichen: größer, kleiner, gleich
- Objekte, die verglichen werden können, implementieren das `Comparable`-Interface:

```
interface Comparable<A> {
  public int compareTo( A that );
}
```

- Ergebniswert:
  - 0 für gleich
  - > 0 für größer
  - < 0 für kleiner

Implementierung am Beispiel von `Byte`

```
interface Comparable<A> {
  public int compareTo( A that );
}

class Byte implements Comparable<Byte> {
  private byte value;

  public Byte( byte value ) {
    this.value = value;
  }

  public byte byteValue() {
    return value;
  }

  public int compareTo( Byte that ) {
    return value - that.value;
  }
}
```

Übersetzung in Java

```
interface Comparable {
  public int compareTo( Object that );
}

class Byte implements Comparable {
  private byte value;

  public Byte( byte value ) {
    this.value = value;
  }

  public byte byteValue() {
    return value;
  }

  public int compareTo( Byte that ) {
    return value - that.value;
  }

  // bridge-method
  public int compareTo( Object that ) {
    return this.compareTo( (Byte) that );
  }
}
```

Wann muß eine Bridge eingefügt werden?

Bridges müssen eingefügt werden, wenn

A Wenn eine Klasse eine Typvariable einer Superklasse oder eines Interfaces instanziiert

```
class Byte implements Comparable<Byte>
```

B Eine Methode implementiert, die als Argument eine Typvariable hat

oder

C Als Ergebnis eine Referenz auf eine instanziierte Typvariable liefert

Noch mehr Brücken

Nicht immer können Brücken bei der Übersetzung in Java einfügt werden:

```
class Interval implements
  Interator<Integer> {
  private int a;
  private int b;
  public Interval( int a, int b ) {
    this.a = a; this.b = b;
  }
  public boolean hasNext() { /* Code */ }
  public Integer next() {
    return new Integer( i++ );
  }
}
```

Bridge-Method:

```
public Object next() {
  return this.next();
}
```

Problem: Java kann Methoden einer Klasse mit gleicher Parameterliste und anderem Rückgabetyp nicht unterscheiden

Codierung in JVM-Byte-Code

- Zwei Methoden mit gleicher Argumentliste und verschiedenem Ergebnistyp sind in Java nicht zulässig
- In der Java Virtual Machine lassen sich Methoden an ihrer vollen Signatur (Argumenttypen und Ergebnistyp) unterscheiden
- GJ kompiliert Bridge-Methoden in solchen Fällen direkt in JVM-Byte-Code

Bounding

- Nicht immer ist allgemeine Generizität wünschenswert
- Typparameter lassen sich einschränken
- Einschränkung auf Klassen, die ein bestimmtes Interface implementieren:

```
class aClass<A implements B> {
}
```

- Einschränkung auf auf Klassen, die von bestimmten Klassen geerbt haben:

```
class bClass<C extends D> {
}
```

Parametrisierte Methoden

- Neben Klassen und Interfaces lassen sich auch Methoden typparametrisieren
- Typparameter werden vor der Methoden-Signatur angegeben:

```
class Collections {
  public static <A> A first(
    Collection<A> col ) {
    return col.iterate().next();
  }
}
```

- Aus den Aufruf-Parametern wird der Instanztyp des Typparameters ermittelt:

```
LinkedList<String> sl = new
  LinkedList<String>();
sl.add( "erster" );
// Aufruf mit Typ Collection<String>
// A instanziiert als String
String s = Collections.first( sl );
```

Beispiel: Bounding und parametrisierte Methoden

```
class Collections {
  public static <A implements Comparable<A>>
    A max( Collection<A> col ) {
    Iterator<A> it = col.iterate();
    A max = it.next();

    while( col.hasNext() ) {
      A item = it.next();
      if( max.compareTo( item ) < 0 ) {
        // Neues Maximum
        max = item;
      }
    }

    return max;
}
```

Übersetzung

```
class Collections {
  public static Comparable
    max( Collection col ) {
    Iterator it = col.iterate();
    Comparable max = it.next();

    while( col.hasNext() ) {
      Comparable item = it.next();
      if( max.compareTo( item ) < 0 ) {
        // Neues Maximum
        max = item;
      }
    }

    return max;
}
```

Untertypen

- Y<A> Untertyp von X<A>, dann ist Y<Aclass> Untertyp von X<Aclass>
- Beispiel:

```
// Korrekte Zuweisung
Collection<String> c = new LinkedList<String>();
```

- Aber: Aus Y Untertyp von X folgt **nicht** Aclass<Y> Untertyp von Aclass<X>
- Beispiel:

```
LinkedList<String> sl =
  new LinkedList<String>();
// Zuweisung führt zum Kompilierfehler!
LinkedList<Object> ol = sl;

ol.add( new Byte( 1 ) );
String s = sl.iterate().next();
```

Retrofitting

- GJ kann alten unparametrisierten Code parameterisiert benutzen
- Mechanismus: *Retrofitting*
- Beispiel: Kompilierter Standard-Java-Code für eine verkettete Liste liegt vor
- Angabe der Retrofit-Klassenbeschreibung für die unparametrisierte Klasse:

```
class LinkedList<A> implements
   Collection<A> {
  public LinkedList();
  public void add( A data );
  public Iterator<A> iterate();
}
```

- GJ erzeugt neuen Code, mit Informationen über Parametrisierung, die in Attributen der JVM-Class-Dateien gespeichert werden