Generics

Typpolymorphie

Viele Datenstrukturen funktionieren mit Elementen verschiedener Typen.

- Listen von Strings
- Listen von Integern
- Listen von Listen von Strings
- ...

Solche Datenstrukturen nennt man polymorph (griechisch: vielgestaltig).

Generics wurden 2004 mit der Version 1.5 in Java eingeführt, um den Umgang mit polymorphen Datenstrukturen zu verbessern.

Bis Java 1.4: Ohne Generics

Polymorphie in alten Java-Versionen: Benutze `Object` überall dort, wo Werte verschieder Typen stehen können.

Beispiel: Listen von `Object`-Werten.

```
// veraltet!
public class List {
  ...
  public void add(Object e) {...}
  public Object get(int index) {...}
  public Iterator iterator() {...}
  ...
}

public class Iterator {
  ...
  public Object next() {...}
}
```

Problem: Geringere Genauigkeit! Kompiler erkennt weniger Probleme. Z.B. Zahl hinzufügen zu Liste, welche nur Strings enthalten soll, liefert ohne Generics keinen Kompilerfehler.

Bis Java 1.4: Ohne Generics

Typischer Fehler:

```
// Liste nodes soll node-Ids als Strings speichern
List nodes = new List(); // veraltet
...
list.add("363179");
list.add("564164");
...
list.add(564162); // Anfuehrungszeichen vergessen

...

// veraltet. Solchen Code nicht mehr schreiben!
for (Iterator i = nodes.iterator(); i.hasNext(); ) {
  String s = (String) i.next(); // wirft ClassCastException
  ...
}
```

Der Fehler wird vom Kompiler nicht bemerkt.

Das Programm bricht bei der Ausführung mit einer `ClassCastException` ab.

Seit Java 1.5: Generics

Generics eliminieren diese Fehlerquelle.

```
public class List<E> {
  ...
  public void add(E e) {...}
  public E get(int index) {...}
  public Iterator<E> iterator() {...}
  ...
}

public class Iterator<E> {
  ...
  public E next() {...}
}
```

Dies erlaubt genauer Typen:

- `List<String>` für Listen von Strings
- `List<Number>` für Listen von Zahlen (`Double`, `Integer`,...)
- `List<Object>` für Listen von beliebigen Objekten

Seit Java 1.5: Generics

Typische Fehler werden vom Compiler erkannt:

```
List<String> nodes = new LinkedList<>();
...
list.add("363179");
list.add("564164");
...
// Der Compiler erkennt jetzt den Fehler hier:
list.add(564162);
...
for (Iterator<String> i = nodes.iterator(); i.hasNext(); ) {
  String s = i.next(); // kein Typcast notwendig
  ...
}
```

Definition generischer Klassen

Klassendefinitionen können über Typvariablen parameterisiert werden `class Klasse<Variable_1, ..., Variable_k> {...}`.

Beispiel:

```
public class Pair<S, T> {
  private S fst;
  private T snd;

  public Pair(S fst, T snd) {
    this.fst = fst;
    this.snd = snd;
  }

  public S getFst() {
    return fst;
  }

  public T getSnd() {
    return snd;
  }
}
```

Typvariablen können in der Klasse wie normale Typen benutzt werden.

Definition generischer Klassen

Der Bereich der Typvariablen kann eingeschränkt werden.

```
public class NumberPair<S extends Number, T extends Number> {
  private S fst;
  private T snd;

  public NumberPair(S fst, T snd) {
    this.fst = fst;
    this.snd = snd;
 }

  public float floatAverage() {
    return (fst.floatValue() + snd.floatValue())/2.0f;
  }
}
```

Für `S` und `T` sind jetzt nur noch Unterklassen von `Number` erlaubt.

Die Klasse `Number` ist in der Standardbibliothek als gemeinsame Oberklasse von `Integer`, `Double`, `Long` etc. definiert.

- Beispielmethode: `float floatValue()` zur Umwandlung in einen `float`-Wert.

Instantiierung generischer Klassen

Um eine generische Klasse zu benutzen, müssen konkrete Typen für die Typvariablen angegeben werden.

Instantiierung:

```
Klasse<Typausdruck_1, ..., Typausdruck_k> {...}.
```

Beispiele:

- `List<String>`
- `List<List<String>>`
- nicht möglich: `List<int>`
  Grundtypen sind als Parameter nicht erlaubt;
  benutze entsprechende Wrapper-Klassen `Integer`, `Double`, ...
- nicht möglich: `List<String, String>`
  `List` nimmt nur einen Typparameter. Möglich wäre aber
  `List<Pair<String,String>>`

Instantiierung generischer Klassen

Beispiel:

```
public static void main(String[] args) {
  Integer i = 3;
  Float f = 4.0f;

  NumberPair<Float, Integer> p = new NumberPair<Float, Integer>(f, i);
  float a = p.floatAverage();
  System.out.println(a); // druckt 3.5

  // Typfehler, da String nicht Unterklasse von Number ist.
  NumberPair<String, Integer> p = new NumberPair<String, Integer>("12", i);
}
```

Ebenso würde der Kompiler merken, wenn man versehentlich die Position von `Float` und `Integer` vertauscht:

`NumberPair<Float,Integer>` und `NumberPair<Integer,Float>` sind verschiedene Typen.

Generische Methoden

Auch einzelne Methoden können generisch definiert werden.

- Alle Typvariablen werden vor dem Methodenkopf angegeben.
- Generische Methoden werden wie bisher aufgerufen, d.h. ohne ausdrückliche Erwähnung eines Parametertyps.

Beispiel:

```
public class Zufall {

  public static <T> T zufall(T m, T n) {
    if (Math.random() > 0.5) {
      return m;
    } else {
      return n;
} } }
```

Aufruf dieser Methode:

```
String s = Zufall.zufall("Essen", "Schlafen");
Integer i = Zufall.zufall(100, 50);
Object o = Zufall.zufall("Essen",42);
```

Übersetzung von Generischen Typen

Es gibt mehrere Möglichkeiten zur Übersetzung von generischen Datentypen.

Heterogene Übersetzung

- Für jede Instantiierung (etwa `List<String>`, `List<Integer>`, `List<Point2D>`) wird individueller Code erzeugt, also drei Klassen.

Homogene Übersetzung

- Für jede parametrisierte Klasse (etwa `List<E>`) wird genau eine Klasse erzeugt, welche die generischen Typinformationen löscht („type erasure“) und durch `Object` ersetzt.
- Vorteil: Rückwärtskompatibilität mit alten Java-Programmen.

Java nutzt die homogene Übersetzung, C++ die heterogene.

Homogene Übersetzung in Java

Beispiel: `Pair<S, T>` wird in etwa in folgende Klasse übersetzt.
(In Wirklichkeit wird gleich in Bytecode übersetzt.)

```
public class Pair {
  private Object fst;
  private Object snd;

  public Pair(Object fst, Object snd) {
    this.fst = fst;
    this.snd = snd;
  }

  public Object getFst() {
    return fst;
  }

  public Object getSnd() {
    return snd;
  }
}
```

Dies entspricht dem Ansatz alter Java-Versionen.
Generics stellen die korrekte Benutzung dieser Klasse sicher.

Rückwärtskompatibilität: Raw Types

Generische Klassen können auch ohne Typparameter verwendet werden, damit sie mit „altem“ Programmcode zusammenarbeiten können.

- Ein parametrisierter Typ ohne Typangabe heißt Raw-Type.
- Der Raw-Type entspricht dem Typ, den man durch Ersetzung der Typvariablen durch `Object` erhält.
  Beispiel: `Pair` auf letzter Folie ist Raw-Type für `Pair<S, T>`.
- Raw-Types bieten die gleiche Funktionalität wie parametrisierte Typen, jedoch werden die Parametertypen nicht zur Compilezeit überprüft.

Raw Types dienen nur der Rückwärtskompatibilität und sollten in neuem Programmcode nicht verwendet werden!

Rückwärtskompatibilität: Arrays

Arrays in Java sind nicht typsicher.

- Folgendes Programm wird vom Compiler akzeptiert, obwohl ein Wert vom falschen Typ in das Array geschrieben wird.

```
Object[] array = new String[10];
array[0] = new Integer(12);
```

  $\Rightarrow$ Fehler `ArrayStoreException` erst später zur Laufzeit

- *Zum Vergleich:* Mit Generics wird der Fehler erkannt.

```
List<Object> list = new LinkedList<String>();
list.add(new Integer(12));
```

  $\Rightarrow$ Programm wird vom Kompiler abgelehnt!

Rückwärtskompatibilität: Arrays

Um die Typsicherheit von generischen Typen zu erhalten, dürfen Arrays von generischen Typen nur eingeschränkt verwendet werden.

- Arrays von generischen Typen können nicht erzeugt werden.

```
// ok:
String[] array1 = new String[100];

// nicht erlaubt:
List<String>[] array2 = new List<String>[100];
```

- `ArrayList<List<String>>` hat diese Einschränkungen nicht.
- `ArrayList<E>` ist `E[]` immer vorzuziehen.

  Ausnahme: Einfache Typen wie `double[][]` können zu lesbarerem Code führen als `ArrayList<ArrayList<Double>>`.

Vererbung

Von generischen Klassen kann in gewohnter Weise abgeleitet werden.

- Einsetzen konkreter Typausdrücke in der Oberklasse:

```
public class Point extends Pair<Double, Double> {

  public Point(double x, double y) {
    super(x, y);
  }
}
```

- Generische Unterklasse:

```
public class SamePair<T> extends Pair<T, T> {

  public SamePair(T fst, T snd) {
    super(fst, snd);
  }
}
```

Vererbung

Aber: Wenn `S` von `T` erbt, folgt daraus *nicht*, dass `List<S>` ein Untertyp von `List<T>` ist!

Beispiel:

```
List<String> listStr = new LinkedList<String>();

// List<String> ist kein Untertyp von List<Object>
List<Object> listObj = listStr; // nicht erlaubt:
// waere die Zuweisung erlaubt, dann
// koennte man einen falschen Typ einschmuggeln:
listObj.add(new Integer(12));
```

Achtung: Das Problem liegt nicht darin, `List<Object>` einen `Integer` hinzuzufügen, sondern darin `List<String>` als `List<Object>` aufzufassen!

Vererbung

Beispiel:
Angenommen man hat eine eigene Listen-Klasse geschreiben:

```
public class MyList<E> {
   ...
   public void add(E element) {...};
   public void addAll(List<E> other) {...};
}
```

Problem: `addAll` funktioniert nur für Listen mit exakt gleichem Elementtyp.

```
MyList<Number> list1 = new MyList<Number>();
list1.add(new Integer(13)); // ok
list1.add(new Double(13.0)); // ok

List<Number> list2 = new LinkedList<>();
list2.add(new Integer(13)); // ok
list1.addAll(list2); // ok

List<Integer> list3 = new LinkedList<>();
list3.add(new Integer(13)); // ok
list1.addAll(list3); // Fehler!
                     // Nicht moeglich, da list2 nicht Typ List<Number> hat.
```

Wildcards

Mit Wildcards kann man auch andere Typen in den Generics erlauben.

```
public class MyList<E> {
  ...
  public void addAll(List<? extends E> other) {...};
}
```

Mit dieser Definition funktioniert `addAll` auch für Argumente, die den Typ `List<A>` haben, falls `A` eine Unterklasse von `E` ist.

Im letzten Beispiel funktioniert auch `list1.addAll(list3)`.

Wildcards

Machmal sind auch unbeschränkte Wildcards nützlich.

Beispiel:

```
static boolean isEmpty(List<?> list) {
  return (list.size() == 0);
};
```

Der Aufruf `isEmpty(e)` funktioniert für beliebige Listen.

Zum Vergleich:

```
static boolean isEmpty1(List<Object> list) {
  return (list.size() == 0);
};
```

Der Aufruf `isEmpty1(e)` funktioniert nicht, wenn `e` den Typ `List<Integer>` hat.

Wildcards

In der Dokumentation findet man auch kompliziertere Konstruktionen, wie etwa in `Collection`:

```
static <T extends Comparable<? super T>>
    sort(List<T> list) { ... };
```

Bedeutung:

- `T` muss eine Unterklasse (Subtyp) von `Comparable<S>` sein
- `S` muss eine Oberklasse (Supertyp) von `T` sein.

D.h. die Methode `sort` kann Listen von allen Typen `T` sortieren, welche das Interface `Comparable` entweder selbst implementieren oder die Implementierung von einem Ahnen geerbt haben

```
public interface Comparable<T> {
  public int compareTo(T o);
}
```

Zusammenfassung

- Implementierung polymorpher Datenstrukturen in Java
- Generische Klassen und/oder Methoden möglich
- Homogene Übersetzung
- Vererbung: Auch wenn `A` von `B` erbt besteht zwischen `C<A>` und `C<B>` keine Relation (Stichwort: invariant).
- Wildcards erlauben verschiedene Parametertypen.
  (covariant mit `<? extends T>`, contravariant mit `<? super T>`,)
- Hinweise:
  - Raw-Types nicht in neuen Programmen verwenden
  - `ArrayList` ist normalen Arrays oft vorzuziehen
  - Verwende möglichst wenige Typumwandlungen
    (z.B. `(String) iterator.next()`), da diese vom Kompiler nicht überprüft werden können, sondern erst zur Laufzeit.