Generische Einheiten in C++ und Java, ein Vergleich

Viele Programmiersprachen sind *stark getypt.* Das bedeutet, dass jede Variable und jeder Ausdruck eines Programms zu einem bestimmten Typ gehört, der schon bei der Übergabe des Programms ("zur Compilezeit") festliegt und vom Ausführer überprüft werden kann. Solche starken Typensysteme dienen dazu, das Auffinden bestimmter Programmierfehler ("Typfehler") zu automatisieren und damit zu verbilligen. Von einem *schwachen Typensystem* spricht man, wenn wichtige Typprüfungen erst während der Ausführung eines Programms durchgeführt werden können oder wenn es möglich ist, den Typ einer Variablen während der Programmausführung zu verändern. Zwischen starken und schwachen Typsystemen gibt es keine scharfe Trennlinie, eher einen fließenden Übergang.

Starke Typen sind sehr nützlich, verhindern manchmal aber auch wünschenswerte *Abstraktionen.* Z. B. sind fast alle numerischen Algorithmen (etwa zur Berechnung von Funktionen wie Wurzel, Logarithmus, Sinus etc.) weitgehend unabhängig vom genauen Typ ihrer Parameter (`float`, `double`, oder `long double` etc.), und die Befehle zur Verwaltung einer verketteten Liste oder einer Hashtabelle sind weitgehend unabhäng vom Typ der Komponenten, die in der Liste bzw. Tabelle gesammelt werden.

Um der Abstraktions-verhindernden Wirkung von starken Typen entgegenzuwirken, hat man in zahlreichen Sprachen *generische Einheiten* eingeführt. Das sind Programmteile, in denen bestimmte Typen nicht konkret festgelegt, sondern durch *Typparameter* dargestellt werden. Indem man diese Typparameter durch konkrete Typen ersetzt, erhält man aus *einer* generischen Einheit mit wenig Programmieraufwand *zahlreiche* nicht-generische Einheiten, z. B. aus einer generischen Sinus-Funktion drei konkrete Sinus-Funktionen, die man auf Werte der Typen `float`, `double` bzw. `long double` etc. anwenden kann.

Die genauen Regeln, nach denen man generische Einheiten in einem Programm vereinbaren und benutzen kann, unterscheiden sich von Sprache zu Sprache erheblich. Hier sollen generische Einheiten in C++ und Java miteinander verglichen und die wichtigsten Unterschiede herausgestellt werden.

Im folgenden Beispiel wird in C++ eine Funktionsschablone namens `print` vereinbart. Mehrere Instanzen dieser Schablone (`print`-Funktionen) werden aufgerufen:

Beispiel-01: Eine Funktionsschablone `print` in C++

```
1   template<class T>
2   void print(T t) {
3      cout << t << " " << t << endl;
4   } // print
5      ...
6      string str01("Hallo!");
7      int    int01(123456);
8      float  flt01(12.345);
9      ...
10     print(str01);
11     print(int01);
12     print(flt01);
13     ...
14  // Ausgabe der print-Befehle:
15  Hallo! Hallo!
16  123456 123456
17  12.345 12.345
```

In den Zeilen 1 bis 4 wird eine Funktionsschablone namens `print` vereinbart. In den Zeilen 10 bis 12 werden drei verschiedene Instanzen dieser Schablone (`print`-Funktionen) aufgerufen.

In C++ werden die *Instanzen* einer generischen Einheit in aller Regel als *Kopien* der Einheit realisiert. In diesen Kopien werden die generischen Parameter durch entsprechende aktuelle Parameter ersetzt.

Beim Übersetzen von Beispiel-01 erzeugen die meisten C++-Compiler drei Kopien der generischen Einheit `print` (genauer: der Zeilen 2 bis 4) und ersetzen darin den Typparameter `T` durch die konkreten Typen `string` bzw. `int` bzw. `float`. Diese Kopien der Schablone sind Funktionen, und werden in den Zeilen 10 bis 12 genau wie andere Funktionen auch aufgerufen.

Dieses *Kopieren generischer Einheiten* kann zu einer erheblichen *Vergrößerung* eines Programms führen. Im Englischen wird diese Erscheinung sehr negativ als *code bloat* ("Code Aufblähung") bezeichnet.

Beispiel-02: Eine generische Methode `print` in Java

```
18  class XYZ {
19     ...
20     static <T> void print(T t) {
21        System.out.println("t + " " + t);
22     } // print
23     ...
24     static public void main(String[] _) {
25        ...
26        String        str01 = new String       ("Hallo!");
27        StringBuilder stb01 = new StringBuilder("Wie geht's?");
28        Integer       int01 = new Integer      (123456);
29        ...
30        print(str01);
31        print(stb01);
32        print(int01);
33        ...
34     } // main
35  } // class XYZ
36  // Ausgabe der print-Befehle:
37  Hallo! Hallo!
38  Wie geht's? Wie geht's?
39  123456 123456
```

Generische Einheiten (wie die Funktion `print` in diesem Beispiel) werden von einem Java-Compiler *nicht kopiert.* In diesem einfachen Beispiel braucht der Compiler nur den Typparameter `T` durch den konkreten Typ `Object` zu ersetzen.

Diese Ersetzung hätte auch der Programmierer vornehmen können: Statt eine generische Methode `print` mit einem Typparamter `T` hätte er eine normale Methode mit einem `Object`-Parameter schreiben können. Dieses Ersetzen einer generischen Einheit durch eine "gleichwertige nicht-generische Einheit" ist aber nur in diesem simplen Beispiel so einfach.

Im folgenden Beispiel muss der Java-Compiler etwas mehr leisten als im vorigen:

Beispiel-03: Eine generische Klasse `Paar<K>` und Objekte unterschiedlicher `Paar`-Typen

```
40  class Paar<K> {
41     K k1;
42     K k2;
43
44     public Paar(K k1, K k2) {
45        this.k1 = k1;
46        this.k2 = k2;
47     } // Konstruktor Paar
48
49     public K getK1() {return k1;}
50     public K getK2() {return k2;}
51     ...
```

```
52     static public void main(String[] _) {
53        Paar<String> ps = new Paar<String>("ABC", "DEF");
54        Paar<Double> pd = new Paar<Double>(12.34, 56.78);
55
56        Paar<Long>   pl = new Paar<Long>  ("ABC", new Long(123));
57
58        String       s1 =          ps.getK1();
59        Double       d1 =          pd.getK1();
60
61        Double       d2 =          ps.getK2();
62        ...
63     } // main
64  } // class Paar<K>
```

Auch hierl wird der Typparameter `K` in der Klasse `Paar` durch den konkreten Typ `Object` ersetzt. Und da, wo `Paar`-Funktionen aufgerufen werden, fügt der Compiler geeignete *Cast-Befehle* ein, etwa so:

Der Funktionsaufruf `ps.getK1()` in Zeile 58 liefert jetzt formal ein Ergebnis vom Typ `Object`. Aber der Compiler kann sich vergewissern, dass dieses Ergebnis immer vom genaueren Typ `String` sein wird, da `ps` vom Typ `Paar<String>` ist. Also kann er in Zeile 58 einen Cast-Befehl `(String)` einfügen und dafür garantieren, dass dieser Cast immer "richtig funktionieren" und keine Ausnahme werfen wird. In Zeile 59 kann er ganz entsprechend einen Cast-Befehl `(Double)` einfügen, und dafür die Verantwortung übernehmen, etwa so:

```
58        String       s1 = (String) ps.getK1();
59        Double       d1 = (Double) pd.getK1();
```

Das ist der Kern der Technik, mit der generische Einheiten in Java realisiert werden: Statt *zahlreiche Kopien* anzufertigen, ersetzt der Ausführer im Original der Einheit Typparameter durch geeignete konkrete Typen (häufig, aber nicht immer, durch den Typ `Object`). In Aufrufen von generischen Methoden werden Cast-Befehle eingesetzt, für deren Korrektheit der Ausführer garantieren kann.

Außerdem führt der Ausführer eine Reihe von Typprüfungen durch. Im Beispiel-03 findet er zwei Typfehler:

Fehler 1: In Zeile 56 wird ein Konstruktor mit einem `String`- und einem `Long`-Parameter aufgerufen. Zum Typ `Paar<Long>` gehört aber nur ein Konstruktor mit zwei `Long`-Parametern.

Fehler 2: In Zeile 61 liefert der Funktionsaufruf `ps.getK2()` garantiert ein `String`-Objekt (weil `ps` vom Typ `Paar<String>` ist). Das passt aber nicht zum Typ `Double` der Variablen `d2`.

Hier wurden nur die Lösungen der einfachsten Probleme angedeutet. In einigen Fällen muss der Java-Compiler kompliziertere Aktionen durchführen (z. B. den kleinsten gemeinsamen Obertyp zweier Typen berechnen oder einen ganz neuen "Brückentyp" zwischen zwei Typen vereinbaren etc.). Das Ergebnis dieser Aktionen kann aber so zusammengefasst werden:

1. Jede generische Klasse (bzw. Methode) wird in nur *eine* entsprechende nicht-generische Klasse (bzw. Methode) übersetzt. Darin stehen anstelle der Typparameter konkrete Typen (z. B. `Object`).

2. An bestimmte Stellen des Programms werden Cast-Befehle eingebaut, die garantiert immer richtig funktionieren werden.

Aus diesem Implementierungsmodell für generische Einheiten ergeben sich einige Einschränkungen für den Programmierer, vor allem die folgenden (K ist dabei ein Typparamter einer generischen Einheit):

1. Ein *Cast*-Befehl wie `(K)` wäre ähnlich wirkungslos wie der Cast-Befehl `(Object)`. Der Compiler warnt einen deshalb vor solch wenig sinnvollen Casts ("unchecked conversion").

2. Man darf keine *Reihung* mit Komponenten des Typs `K` erzeugen lassen (d. h. ein Befehl wie etwa `new K[17]` ist verboten, Fehlermeldung "generic array creation").

3. Einen Typparameter wie `K` darf man nur durch *Referenztypen* ersetzen, aber nicht durch primitive Typen wie `int`, `float`, `boolean` etc. (was in C++ grundsätzlich erlaubt ist).

Diese Einschränkungen sind der Preis dafür, dass generische Einheiten in Java nicht kopiert werden und keine Programm-Aufblähung bewirken, auch wenn man sie mit zahlreichen Typen parametriesiert verwendet.

Tip: Anstelle einer Reihung wie

```
65  K[] otto = new K[17]; // Verboten, wenn K ein Typparameter ist
```

kann man innerhalb einer generischen Einheit ein Objekt des Typs `ArrayList<K>` vereinbaren und initialisieren, etwa so:

```
66  ArrayList<K> emil = new ArrayList<K>(17); // Kapazität 17, size  0
67  for (int i=0; i<17; i++) otto.add(null);  // Kapazität 17, size 17
```

Heute übliche Java-Ausführer realisieren `ArrayList`-Objekte intern als *Reihungen*. Solange man auf das Objekt `emil` keine *längenverändernden Methoden* wie `insert`, `append` oder `delete` etc. anwendet, verhält sich `emil` zur Laufzeit weitgehen so, wie die (verbotene) Reihung `otto` es tun würde (d. h. `emil` ist "genau so schnell" wie `otto`).

Hinweis: In der Programmiersprache C# (deutsch kurz Cis, englisch, etwas länger: c sharp) gibt es auch generische Einheiten, die Ähnlichkeit mit denen von Java haben. Allerdings darf man dort Typparameter auch durch *Wertetypen* wie `int`, `float`, `boolean` etc. ersetzen. *Wertetypen* in C# entsprechen ungefähr den *primitiven Typen* in Java, aber mit folgendem wichtigen Unterschied: In Java gibt es genau 8 primitive Typen. In C# gibt es 13 vorvereinbarte Wertetypen, aber der Programmierer kann (im Prinzip beliebig viele) weitere Wertetypen (z. B. Aufzählungstypen) vereinbaren. Von einer generischen Einheit mit *einem* Typparameter, die man mit 15 verschiedenen Wertetypen und 10 verschiedenen Referenztypen parametrisiert, werden zur Laufzeit 16 Kopien erzeugt: Für jeden Wertetyp je eine Kopie und für alle Referenztypen zusammen eine Kopie. Von einer generischen Einheiten mit zwei Typparametern wird für jedes Paar von Wertetypen, mit denen man sie parametriesiert, je eine Kopie erzeugt und für alle Paare von Referenztypen zusammen eine weitere Kopie. Diese Implementierung von generischen Einheiten in C# ist ein interessanter Kompromiß zwischen der "immer-kopieren-Strategie" von C++ und der "nie-kopieren-Strategie" von Java.