

Freie Universität Berlin  
Institut für Informatik

Seminararbeit  
Objektorientierte Programmiersprachen  
betreut durch Prof. Dr.-Ing. Klaus-Peter Lühr

# **Generizität**

## **in Java und C#**

Riad Djemili  
([djemili@inf.fu-berlin.de](mailto:djemili@inf.fu-berlin.de))

WiSe 2003/04

# Inhaltsverzeichnis

<b>1 Überblick</b>	<b>3</b>
<b>2 Motivation</b>	<b>3</b>
2.1 Homogene Implementation . . . . .	4
2.2 Heterogene Implementation . . . . .	5
<b>3 Generizität in Java und C#</b>	<b>5</b>
3.1 Klassen-Generizität . . . . .	6
3.2 Methoden-Generizität . . . . .	7
3.3 Typeinschränkung . . . . .	8
<b>4 Implementation</b>	<b>8</b>
4.1 Implementation in Java . . . . .	9
4.2 Implementation in C# . . . . .	10
<b>5 Übergang von ungenerischem zu generischem Code</b>	<b>10</b>
5.1 Übergang in Java . . . . .	11
5.2 Übergang in C# . . . . .	11
<b>6 Sonstiges</b>	<b>12</b>
6.1 Varianz . . . . .	12
6.2 Vererbung . . . . .	12
6.3 Ausnahmen . . . . .	13
6.4 Typinstanziierung . . . . .	13
6.5 Typkenntnis . . . . .	14
<b>7 Konklusion</b>	<b>14</b>
<b>Literatur</b>	<b>15</b>

Die Programmiersprachen Java und C# gelten als die verbreitetsten modernen objektorientierten Programmiersprachen. Sie beschreiben eine Post-C++ Generation, die sich durch automatische Speicherverwaltung, hohe Typsicherheit und Klarheit auszeichnet.

Im Jahr 2004 sollen beide Sprache mit einem elementaren Sprachkonstrukt erweitert werden: Der parametrischen Polymorphität. Diese Arbeit erläutert die Motivation dieser Entwicklung, beschreibt die neuen Konstrukte und vergleicht die Stärken und Schwächen der beiden sehr unterschiedlichen Implementationen.

## 1 Überblick

Java wird seit 1995 von Sun Microsystems entwickelt und ist aktuell eine Sprache ohne spezielle Generizitätskonstrukte. Mit der neuen Versionsnummer 1.5 (Codename Tiger) sollen im Sommer 2004 die größten Sprachaktualisierungen seit Bestehen einhergehen. Neben Aspekten, wie Autoboxing, Enumerationen und anderen, soll die parametrische Polymorphie unter dem Begriff *Generics* eingeführt werden [1].

C# wird seit 2001 von Microsoft entwickelt und ist die Hauptsprache der .NET Initiative von Microsoft. C# soll in der Version 2.0, welche Mitte bis Ende 2004 (mit .NET 2.0) erscheinen wird, ebenfalls parametrische Polymorphität erhalten [4].

In Abschnitt 2 gehe ich zunächst auf die Schwächen aktueller Entwurfstechniken in Java und C# ein. Darauf folgt in Abschnitt 3 ein genereller Überblick über die neuen Sprachkonstrukte. In Abschnitt 4 stelle ich die beiden sehr unterschiedlichen technischen Umsetzungen genauer vor, um dann in Abschnitt 5 die unterschiedlichen Strategien von Java und C# beim Wechsel in das typparametrische Paradigma darzulegen. In Abschnitt 6 folgen Ausblicke auf verschiedene speziellere Aspekte, wie Typenvarianzen, Vererbung oder Ausnahmen. In Abschnitt 7 schließe ich mit der Konklusion ab.

## 2 Motivation

Polymorphie ist das Grundprinzip moderner objektorientierter Programmiersprachen. Sie bezeichnet die Eigenschaft von Variablen, Objekte unterschiedlichen Typs speichern zu können. Variablen können *kovariant* Werte zugewiesen werden, die vom gleichen Typen oder von einem abgeleiteten Typen sind, wie die Deklaration der Variable.

Man unterscheidet dabei zwischen den statischen und dynamischen Typen. Statische Typen, sind die Typen, die bei der Deklaration einer Variable angegeben werden. Sie werden zur Übersetzungszeit festgelegt und können nicht mehr verändert werden. Dynamische Typen dagegen sind die Typen, die erst durch die Zuweisung bestimmt werden.

```
| List list = new LinkedList();
```

Listing 1: *Eine polymorphe Zuweisung mit List, als statischem Typ und LinkedList, als dynamischem Typ.*

Diese einfache Typunterscheidung, erlaubt die Entwicklung wiederverwendbareren Codes, indem sie ermöglicht eine gemeinsame Schnittstelle mit unterschiedlichsten Untertypen zu nutzen. Allerdings geht hiermit oft auch ein Verlust der Typsicherheit

und Ausdrucksstärke einher. Anhand der beiden klassischen und gegensätzlichen Entwurfsstrategien, der Nutzung des niedrigsten gemeinsamen Typen der Klassenhierarchie (homogener Ansatz) und der spezialisierten Implementationen mit unterschiedlichen Typen (heterogener Ansatz), möchte ich motivierend auf Schwächen der aktuellen Sprachversionen von Java und C# hinweisen.

## 2.1 Homogene Implementation

Bei einer homogenen Implementation wird der selbe Programmteil (z.B. Methodenrumpf) für verschiedene Typen verwendet. Hatten die Klassen in C++ noch keine einheitliche Hierarchie, so bieten modernere Programmiersprachen, wie Java und C# eine komplexe Klassenhierarchie, in der alle Typen eingebettet sind. Im speziellen erben hier alle Klassen von einer gemeinsam Oberklasse. Durch sichere Typwandlungen von einem untergeordneten Typen zur Oberklasse und unsichere Typwandlungen zurück, können Programmteile so für unterschiedliche Typen verwendet werden.

```
1 | class MyList {
2 |     boolean add(Object o) {...}
3 |     Object get(int index) {...}
4 | }
5 |
6 | class ListUser {
7 |     public ListUser() {
8 |         MyList ints = new MyList(); //associates Integers
9 |         ints.add(new Integer(4));
10 |         Integer foo = (String)ints.get(0); //runtimeerror
11 |     }
12 | }
```

Listing 2: Die Definition und Nutzung einer einfachen Liste in Java.

Das Listing 2 zeigt eine vereinfachte Darstellung einer Listenimplementation in Java. Anhand dieses Programmfragments möchte ich im folgenden Nachteile dieses Ansatzes identifizieren.

- Da die Deklaration der Behälterklasse unabhängig vom Typen ist, die sie tatsächlich halten können soll, dokumentiert sich der Code nur unbefriedigend selbst. In vielen Softwareprojekten wird daher behelfsmäßig, zusammen mit der Listendeklaration, der Verwendungszweck als Kommentar beschrieben.

```
8 | MyList ints = new MyList(); //associates Integers
```

Besser wäre, statt einer impliziten Konvention, eine explizite Sprachunterstützung.

- Typsicherheit zur Übersetzungszeit zu garantieren, ist eines der wesentlichen Ziele systemnaher Programmiersprachen. Die homogene Implementation erfordert jedoch, vor allem an den Ausgaben der allgemeinen Programmteile, potentiell fehlerhafte Typwandlungen. Da der Rückgabetypp von Methoden sich in einem homogenen Ansatz nur auf die Oberklasse bezieht, muss der Programmierer selbstständig explizite und gefährliche Typwandlungen zu den Unterklassen vollführen, falls er deren spezielle Funktionalität nutzen will. Besonders kritisch ist dabei, dass sich die Typfehler aus dynamischen Typwechseln erst zur Laufzeit bemerkbar machen.

```

9 | ints.add(new Integer(4));
10 | Integer foo = (String)ints.get(0); //runtimeerror

```

Wünschenswert wäre ein Sprachkonstrukt, das erlauben würde auf kritische Typwandlungen zur Laufzeit zu verzichten und dennoch die geforderte Flexibilität leistet.

- Typwandlungen sind nicht nur unsichere, sondern auch teure Vorgänge. Gerade in zeitkritischen Abschnitten, wäre der Verzicht auf sie, auch aus Leistungsgründen, wünschenswert.

## 2.2 Heterogene Implementation

Konzeptionell entgegengesetzt zur homogenen Implementation, ist die Entwicklung spezialisierter Teile für unterschiedliche Typen. Insbesondere in Verbindung mit Methodenüberladung (gleiche Methodennamen mit unterschiedlichen Signaturen) können verschiedene Implementationen entsprechend des Typs angeboten werden.

```

| public static double abs(double a) {return a>0 ? a:-a;}
| public static float  abs(float a)  {return a>0 ? a:-a;}
| public static int    abs(int a)     {return a>0 ? a:-a;}

```

Listing 3: *Mathematische Klassenmethoden in Java. Die Methode abs hat mehrere Überladungen, obwohl sich ihr Rumpf nicht unterscheidet.*

Vorteilhaft ist, dass der Code typsicher und effizient ist, da auf gefährliche und unnötige Typwandlungen verzichtet werden kann. Eine allgemeine Nutzung dieses Konzepts ist jedoch nicht sinnvoll, da dieser Ansatz zu redundantem und unübersichtlichen Code führt. Zudem kann oftmals bei der Entwicklung einer Klasse, noch keine genaue Aussage über ihre spätere spezielle Verwendung gemacht werden. So müsste in diesem Ansatz beispielsweise, für jeden in einer Listenklasse zu speichernen Typen, eine speziell angepasste Klasse angelegt werden. Die dazu nötige Kenntnis, über die genaue Implementation der Klasse, widerspricht der Idee vom abstrakten Datentypen.

## 3 Generizität in Java und C#

Formal wird die *parametrische Polymorphie* (im folgenden kurz *Generizität*) als Teilbereich der so genannten *universellen Polymorphie* (vgl. Abbildung 1) betrachtet. Sie ist ein spezielles Sprachkonstrukt, das die Vorteile der homogenen und heterogenen Ansätze vereint, indem sie erlaubt den statischen Typ typsicher zu variieren.

Ihre Ziele sind genau die Beseitigung, der oben erläuterten Schwächen der ungenerischen Polymorphie (vgl. Abschnitt 2). Zusammenfassend also

- Höhere Typsicherheit.
- Wieder verwendbarer Code.
- Bessere Lesbarkeit und Ausdrucksstärke.
- Höhere Effizienz.

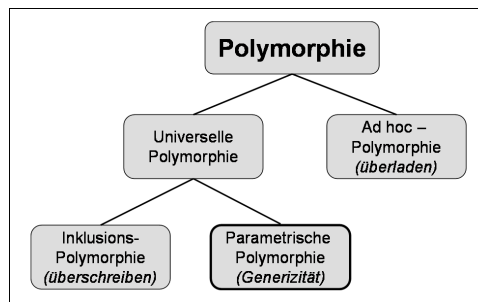


Abbildung 1: *Parametrische Polymorphie wird der universellen Polymorphie zugeordnet.*

Es folgt nun ein genereller Überblick über die Kerneigenschaften der neuen Generizität in Java und C#.

### 3.1 Klassen-Generizität

Klassen-Generizität ist die verbreitetste Art von parametrisierter Polymorphie. Die konkreten Datentypen im Klassenrumpf werden durch generische, formale Typparameter ersetzt. Diese werden in Java und C# (wie in C++) zuerst durch eine koma-separierte Liste in spitzen Klammern im Klassenkopf deklariert und dann im Rumpf, wie normale Typen eingesetzt.

```

class Stack<E> {
    public E push(E item) {...}
    public E pop() {...}
    public E peek() {...}
    public boolean empty() {...}
    public int search(Object o) {...}
}
  
```

Listing 4: *Ein generischer Stack in Java mit E als formalem Typparameter (in C# analog).*

Der tatsächliche Datentyp wird erst mit der Deklaration der generischen Klasse als Typargument angegeben. Der parametrisierte Datentyp ist dann äquivalent zu einem Datentyp, in dem alle formalen Typen durch die tatsächlichen Typen ersetzt wurden. Beide Sprachen unterstützen dabei Klassen als Parametertypen.

```

Stack<String> foo = new Stack<String>();
  
```

Listing 5: *Eine Deklaration und Instanziierung einer generischen Klasse in Java (in C# analog).*

Da in Java jedoch ein einheitliches Typsystem für primitive Typen fehlt, werden diese nicht als Typargumente unterstützt. Sie müssen behelfsmäßig und uneffizient in ihre *Wrapper*-Klassen gekapselt werden. Diese sind spezielle Adapterklassen, die primitive Typen in die allgemeine Klassenhierarchie eingliedern (z.B. `Integer` für `int` oder `Boolean` für `boolean`).

Mit Java 1.5 wird ebenfalls das so genannte *autoboxing* eingeführt wird (existiert bereits in C#). Primitive Typen werden dann bei Bedarf automatisch in ihre passende

Adapterklasse gehüllt. Ebenso werden dann Adapterklassen automatisch zu primitiven Typen gewandelt. So wird das explizite Typändern zwar dem Programmierer abgenommen, findet jedoch intern immer noch statt.

Die Adapterklassen erlauben beispielsweise `int`-Werte in einer generischer Containnerklasse zu speichern. Allerdings erscheint es aus Effizienzgründen wahrscheinlich, dass auch zukünftig, hochfrequentierte typspezifische Berechnungen (wie in Listing 3) durch spezialisierte Implementationen realisiert werden.

In `C#` sind auch primitive Typen (hier Werttypen genannt) als Typargumente erlaubt, da sie in die allgemeine Typhierarchie eingebettet sind und uniform behandelt werden können. Außerdem sind in `C#` auch Structs und Delegate als generische Typen möglich.

Die Angabe von mehreren Typparametern geschieht in beiden Sprachen einfach als koma-separierte Liste.

```
class Pair<U,T> {  
    public U first;  
    public T second;  
}
```

Listing 6: *Ein generisches Paar von verschiedenen Typen in Java oder C#.*

## 3.2 Methoden-Generizität

Analog zur Klassen-Generizität können auch Methoden mit formalen Typen definiert werden. Die Typargumente stellen quasi neben den Operationsargumenten eine eigene Argumentliste dar.

```
static <Elem>void swap(Elem[] array, int x, int y) {  
    Elem temp = array[x]; array[x] = array[y]; array[y] = temp;  
}
```

Listing 7: *Ein generische Vertausche-Klassenmethode in Java.*

```
static void swap<Elem>(Elem[] array, int x, int y) {  
    Elem temp = array[x]; array[x] = array[y]; array[y] = temp;  
}
```

Listing 8: *Ein generische Vertausche-Klassenmethode in C#.*

Bemerkenswert ist hier, dass in Java, im Gegensatz zu `C#`, die Liste der Parametertypen dem Rückgabetypen vorhergeht. Dies widerspricht etwas der natürlichen Intuition, die Liste analog zur Klassen-Generizität, nach dem Namen anzugeben. Im Spezifikationsentwurf [1] wird diese Notation zugunsten einer leichteren Parsbarkeit begründet.

Methodenaufrufe können in beiden Sprachen mit und ohne explizite Typangabe erfolgen, da der aktuelle Typparameter sich meist aus den Argumenttypen schließen lässt.

```
<Integer>swap(ints, 1, 3);  
<String>sort(strings);  
  
swap(ints, 1, 3);
```

```
| sort(strings);
```

Listing 9: Aufruf der *swap*-Methode mit und ohne explizite Typargumente in Java und C#.

### 3.3 Typeinschränkung

Klassische objektorientierte Programmiersprachen, erlauben Anforderungen an Typen mittels Vererbungsbeziehungen auszudrücken. Dies erlaubt eine Typsicherheit zur Übersetzungszeit.

```
| boolean pos(Comparable foo) {  
|     return foo.compareTo(x);  
| }
```

Listing 10: Eine einfache Methode in Java. Sie darf nur mit Typen aufgerufen werden, die vom Typen *Comparable* sind oder von ihm ableiten.

Durch die bisher erläuterten Generizitätskonstrukte ist diese Anforderung jedoch nicht ausdrückbar. Sie verschwindet quasi zunächst mit der Ersetzung durch formale Typen.

Eiffel ermöglicht deshalb, unter dem Begriff *constrained genericity*, Anforderungen auch an Typparameter explizit zu formulieren. Die erlaubten tatsächlichen Typparameter können dabei auf eine Unterklasse einer Klasse eingeschränkt werden. Dazu wird die geforderte Oberklasse in der Deklarationsliste der formalen Typen ausgewiesen und ermöglicht so die Typsicherheit zur Übersetzungszeit zu erhalten.

Entsprechend der Typhierarchie in Java und C# (Verzicht auf Mehrfachvererbung) werden die formalen Typparameter hier an maximal eine Oberklasse und eine beliebige Anzahl von Schnittstellen gebunden.

```
| class SortedList<T extends Entry implements Comparable>
```

Listing 11: Der Kopf einer sortierten Liste in Java. Ihr Typargument muss Unterklasse von *Entry* sein und die Schnittstelle *Comparable* implementieren.

In C# erfolgt die Notation mittels dem nachgelagerten *where*-Schlüsselwort und erscheint etwas umständlicher als in Java.

```
| class SortedList<T> where T : Entry, IComparable<T>
```

Listing 12: Die Kopf der sortierten Liste (vgl. Listing 11) in C#.

## 4 Implementation

Man unterscheidet bei der Art der Implementation von Generizität zwischen heterogenen und homogenen Übersetzungen. Diese sind ähnlich dem manuellen Vorgehen des Programmierers, wenn diese Konstrukte fehlen (vgl. Abschnitt 2).

In homogenen Übersetzungen wird, für jede generische Klasse, maximal eine passende Klasse generiert, die Code bereitstellt, der von allen Typparametrisierungen

gemeinsam genutzt wird. In heterogenen Übersetzungen wird für jede Typinstanziierung spezialisierter Code erzeugt, der sich nur in den konkreten Typen unterscheidet. Dieser Ansatz findet sich auch in C++.

Java und C# wählen jeweils unterschiedliche Implementationsstrategien. Da die Art der Java Implementation viele Implikationen für ihre allgemeine Funktionalität hat, gehe ich auf sie etwas ausführlicher ein.

## 4.1 Implementation in Java

Java-Code wird nicht, wie in älteren Sprachen direkt auf maschinennahen Code abgebildet, sondern auf eine Zwischensprache, dem so genannten Bytecode. Der Bytecode entspricht dem Maschinencode eines Pseudoprozessors, der durch die *Java Virtual Machine* (JVM) [3] simuliert wird. Dadurch, dass Java-Entwickler nur für die JVM programmieren und diese für unterschiedliche Systeme implementiert werden kann, ist Java selber plattformunabhängig.

Die Implementation in Java ist aus der Arbeit an *Generic Java* [2], einer ersten experimentellen Generizitätsimplemenation für Java hervorgegangen. Bei der Umsetzung der Generics gilt als oberster Grundsatz: Die JVM wird nicht verändert. Statt das Java Typsystem auf elementarer Ebene im Kern zu erweitern, wird der generische Code in aktuellen ungenerischen JVM-Code übersetzt. Java verfolgt damit eine homogene Übersetzung mit den folgenden Schritten [1].

1. Die formalen Datentypen werden durch ihre obere Grenze ersetzt (so genannte *Erasures*). Die obere Grenze ist der Typ, der durch das Einschränken des Typarguments mindestens vorausgesetzt wird. Ist der Typ nicht gebunden, so ist dies die oberste Klasse der Klassenhierarchie, also die Klasse `java.lang.Object`.
2. Da an den Klassenein- und ausgängen nur obere Grenze-Objekte ausgetauscht werden, müssen diese noch korrekt gewandelt werden. Die Implementation setzt deshalb bei Methodenrückgaben und Attributzugriffen bei Bedarf, Typwandlungen ein, um den Typen von der Oberklasse in den tatsächlichen Typen zu wandeln.
3. Schließlich erfolgt die eventuelle Einsetzung von Brückenmethoden zur Realisierung von kovarianter Vererbung (vgl. Abschnitt 6.1).

Im Gegensatz zum bisherigen manuellen Vorgehen des Programmierers, birgt diese Automatisierung natürlich die Vorteile der besseren Lesbarkeit und garantierten Fehlerfreiheit. Nachteilhafterweise bleiben bei dieser Technik allerdings intern auch die Typwandlungen erhalten. Der Aspekt einer höheren Effizienz durch Generizität wurde hier also nicht realisiert.

```
class Cell<A> {
    A value;
    A getValue();
}
..
String f(Cell<String> cell){
    return cell.value;
}
```

*wird übersetzt zu*

```

class Cell {
    Object value;
    Object getValue();
}
..
String f(Cell cell){
    return (String)cell.value;
}

```

Listing 13: Aus der generischen `Cell<A>` Klasse wird intern eine ungenerische `Cell` Klasse, bei der alle formalen Datentypen ersetzt wurden. Bei dem Attributzugriff muss der Typ gewandelt werden.

## 4.2 Implementation in C#

Bei der Implementierung von Generizität in C#, kann Microsoft von anderen Grundvoraussetzungen, als Java ausgehen. Da C# noch nicht gleichfalls etabliert und gefestigt ist wie Java, kann daher auch eine kompromisslosere Implementation verwirklicht werden. C#'s Typsystem (bzw. die unterliegende *Intermediate Language*) wird auf grundlegender Ebene, um den generischen Typbegriff erweitert. Insofern erfolgt hier eine saubere und vollständige Implementation der Generizität [4, 5].

Die Übersetzung der generischen Typen erfolgt typabhängig. Für Werttypen (primitive Typen) wird bei jeder ersten Benutzung spezieller Code generiert. Dies entspricht einer heterogenen Übersetzung.

Für Referenztypen (Klassen) wird einmal generierter gemeinsamer Code wieder verwendet. Dies entspricht einer homogenen Übersetzung. Aufgrund spezieller Speichertechniken können die verschiedenen Typparametrisierungen einer Klasse gemeinsamen Code verwenden und dennoch unterschiedliche Typen (ohne Typwandlung) nutzen.

Insgesamt wirkt die C# Implementation nicht nur eleganter, sondern ist auch deutlich performanter, als eine äquivalente manuelle Programmierung in C#, da sowohl auf autoboxing, als auch auf (implizite oder explizite) Typwandlung verzichtet werden kann.

## 5 Übergang von ungenerischem zu generischem Code

Mit der Einführung eines solch elementaren Sprachkonstrukts, wie der Generizität, vollzieht sich in dem Code ein problematischer Generationenwechsel. In nahezu allen Fällen entstehen Softwareprojekte unter Nutzung vielfältiger Bibliotheken, vor allem unter Nutzung der sehr umfangreichen Hausbibliotheken: der Java API bzw. dem .NET Rahmenwerk. Gerade zur Einführung des Generizitätskonstrukts hat man es dabei meist mit ungenerischem Code zu tun. Es kommt also (bestenfalls nur zeitweise) zu einer Vermischung der Paradigmen.

Im folgenden betrachte ich die Strategien mittels derer Java und C# dieser Problematik begegnen und welche Konsequenzen sich für die Hausbibliotheken ergeben.

## 5.1 Übergang in Java

Die API der Javasprache wird komplett typparametrisiert, d.h. alle Klassen der Bibliothek (insbesondere das Collection-Paket) werden umgestaltet (*retrofitted*). Dadurch wird die Generizität zwar konsequent und signalwirksam eingeführt, allerdings auch die Rückwärtskompatibilität gefährdet, da älterer Code nun mit einer veränderten API operieren muss.

Java macht deshalb die Typparameterangabe optional, um generischen Code weiterhin als regulären ungenerischen Code betrachten zu können. Fehlen die Typargumente, so werden automatisch, die oberen Typschranken (vgl. Abschnitt 4.1) als Typargumente angenommen. Auf diese Art instanziierte generische Objekte, werden als *Raw Classes* bezeichnet.

```
Stack s1 = new Stack();  
Stack<Object> s2 = new Stack<Object>();
```

Listing 14: *Zwei äquivalente Instanzierungen in Java. s1 und s2 haben den gleichen Typen.*

Listing 15 zeigt zwei Mischformen von Zuweisungen, die bei dem gemeinsamen Einsatz von typparametrisierten Klassen und Raw Classes leicht auftreten können. In der ersten Zuweisung werden hier implizit Integer-Objekte zu Object-Objekten gewandelt. Dies ist erlaubt, da alle Klassen Untertyp der Object-Klasse sind. Zur Erinnerung werden, bei Methodenaufrufen mit veränderten Argumenttypen oder Attributzugriffen mit veränderten Typen, Übersetzerwarnungen ausgegeben.

```
Stack s = new Stack<Integer>();  
Stack<Integer> s = new Stack();
```

Listing 15: *Erlaubte Mischformen in Java.*

Die zweite und umgekehrte Zuweisung in Listing 15 ist dagegen offensichtlicherweise nicht typsicher. Dennoch ist sie zugunsten eines leichteren Übergangs zwischen ungenerischem und generischem Code erlaubt und erzeugt nur eine *deprecation*-Warnung bei der Übersetzung.

Bemerkenswert ist noch, dass neben der Rückwärtskompatibilität, die Generics-Spezifikation [1] auch Vorwärtskompatibilität impliziert, da der generierte Code normaler aktueller JVM Code ist. Inwiefern diese Eigenschaft in der endgültigen Implementation erhalten bleibt, ist allerdings noch abzuwarten.

## 5.2 Übergang in C#

Ergänzend zum aktuellen .NET Rahmenwerk, erfolgt eine Einführung neuer generischer Typen. Diese beschränken sich dabei nahezu nur auf neue Container-Klassen (vgl. Tabelle 5.2). Die Rückwärtskompatibilität bleibt durch Beibehaltung des alten Rahmenwerks garantiert, allerdings wirkt die Einführung eines zweiten Container-Namenraums unübersichtlich. Zudem wird die Mischung von generischem und ungenerischem Code erschwert. C# stellt hierfür keine speziellen Konstrukte bereit.

Ingesamt wirkt der Übergang in C# uneleganter als in Java. Zudem ist fraglich inwiefern sich die Generizität auf das restliche Rahmenwerk auswirken wird und ob es bei der problematischen Vermischung der verschiedenen Paradigmen bleibt.

System.Collections	System.Collections.Generic
Comparer	Comparer<T>
HashTable	Dictionary<K,T>
ArrayList	List<T>
Queue	Queue<T>
SortedList	SortedDictionary<K,T>
Stack	Stack<T>
ICollection	ICollection<T>
System.Comparable	IComparable<T>
IComparer	IComparer<T>
IDictionary	IDictionary<K,T>
IEnumerable	IEnumerable<T>
IEnumerator	IEnumerator<T>
IKeyComparer	IKeyComparer<T>
IList	IList<T>

Tabelle 1: *In einem neuen Namensraum werden für das .NET Rahmenwerk generische Versionen der älteren ungenerischen Container-Klassen eingeführt.*

## 6 Sonstiges

### 6.1 Varianz

Java und C# erzwingen in ihren aktuellen Versionen invariante Rückgabetypen bei der Überschreibung von Methoden, d.h. der Rückgabetypp einer überschreibenden Methode muss identisch sein, mit dem Rückgabetypp der überschriebenen Methode.

In Java wird, mit dem Einzug der Generizität, die Spezifikation der Javasprache [3] derart geändert, dass nur noch kovariante Rückgabetypen bei der Überschreibung erfordert werden. Damit sind als Rückgabetypp überschreibender Methoden, auch alle Untertypen des Rückgabetypp der überschriebenen Methode, erlaubt.

Tatsächlich ist diese Änderung sogar ohne Anpassung der JVM möglich, da diese intern Methoden ohnehin anhand der vollständigen Signatur unterscheidet. Die Beschränkung auf invariante Rückgabetypen wird momentan nur durch den Compiler erzwungen. Da die JVM allerdings überschreibende Methoden mit anderem Rückgabetypp, als zusätzliche Methoden behandelt und nicht als Überschreibende, müssen dann bei der Übersetzung Brückenmethoden eingesetzt werden. Diese Überschreiben die alte Methode, mittels invariantem Rückgabetypp, und leiten jeden Aufruf an die richtige überschreibende Methode mit kovariantem Rückgabetypp weiter.

### 6.2 Vererbung

Es ist außerdem wichtig, sich die Typbeziehung zwischen generischen Typen klar zu machen. Es existiert weiterhin eine Typbeziehung zwischen einer Klasse und ihrer generischen Unterklasse. Es existiert jedoch keine Typbeziehung zwischen unterschiedlichen Typparametrisierungen einer generischen Klassen. Es gilt also:

- `LinkedList<String>` ist Untertyp von `List<String>`.
- `List<String>` ist nicht Untertyp von `List<Object>`.

Um dennoch eine Typbeziehung zwischen unterschiedlichen Typparametrisierungen zu nutzen, können durch die Kombination von Klassentypparameter und Methodentypparametern, auch andere Typvarianzen, wie die Kovarianz, nachgebildet werden.

```
interface Collection<E> {
    <T extends E> boolean addAll(Collection<T> c);
    <T> boolean containsAll(Collection<T> c);
}
```

Listing 16: *Eine Schnittstelle in Java bei der die Methoden, verschieden parametrisierte Collection-Objekte akzeptieren.*

Vorangetrieben durch *Variant Generic Java* [6], erlaubt die *Wildcard*-Syntax in Java eine zusätzlich kürzere Notation, durch anonyme Typparameter. Sie erlauben Typparameter, die nicht weiter referenziert werden müssen, ähnlich den anonymen Klassen, verkürzend anzugeben. An der Stelle des Namen wird dazu ein Fragezeichen notiert.

```
interface Collection<E> {
    boolean addAll(Collection<? extends E> c);
    boolean containsAll(Collection<?> c);
}
```

Listing 17: *Die Schnittstelle aus Listing 16 mit anonymen formalen Typen.*

Zusätzlich erlauben Wildcards die so genannte Kontravarianz zu forcieren, mit der nur Oberklassen einer Klasse akzeptiert werden.

### 6.3 Ausnahmen

Generische Ausnahmen werden in Java nicht vollständig unterstützt. So dürfen generische Klassen nicht von `java.lang.Throwable` ableiten. Typvariablen sind erlaubt in der `throws`-Anweisung, aber nicht in der `catch`-Anweisung.

C# unterstützt dagegen Ausnahmen, auch im generischen Kontext, vollständig. Generische Klassen dürfen von `System.Exception` ableiten und sind sowohl in der `throws`-Anweisung, als auch in der `catch`-Anweisung erlaubt.

### 6.4 Typinstanziierung

Ein typisches Szenario für generische Typen, ist die Instanziierung der Typparameter. Nachteilhafterweise ist dies in Java, aufgrund der Implementation mittels Erasures nicht möglich.

In C# können Parametertypen dagegen unter Umständen instanziiert werden. Dazu müssen sie einen öffentlichen Konstruktor ohne Argumente anbieten. Die Existenz muss als explizite zusätzliche Bedingung mit `new()` angegeben werden. Da man jede Klasse mit Konstruktorargumenten in eine Klasse mit argumentlosem Konstruktor und zusätzlichen Setzmethoden umwandeln kann, ist diese Funktionalität ausreichend.

```
class Singleton<T> where T : new() {
    private T instance;

    public T getInstance() {
```

```

        if (instance == null)
            instance = new T();

        return instance;
    }
}

```

Listing 18: *Eine generische Singletonklasse in C#. Durch die Eingrenzung new() wird signalisiert, dass der tatsächliche Typparameter für T einen parameterlosen Konstruktor haben muss.*

## 6.5 Typkenntnis

In Java herrscht zur Laufzeit kaum Kenntnis über die aktuellen Typparameter eines generischen Typen. Da die JVM nur die bereits bereinigten Klassen sieht, kann sie nur allgemeine Angaben wie z.B. über die Anzahl der Typparameter machen.

C# kann auf die volle Unterstützung der unteren Sprachtypen zugreifen. Es herrscht eine totale Typkenntnis zur Laufzeit.

Die *Type*-Klasse, welche in C# für *Reflection*-Zwecke eingesetzt wird, wird mit speziellen Eigenschaften von generischen Typen erweitert und erlaubt sowohl Kenntnis über die generischen Typen, als auch über ihre aktuellen Typparameter.

```

int         GenericParameterPosition{virtual get;}
bool        HasGenericParameters{get;}
bool        HasUnboundGenericParameters{virtual get;}
bool        IsGenericParameter{virtual get;}
bool        IsGenericTypeDefinition{virtual get;}
virtual Type BindGenericParameters(Type [] typeArgs);
virtual Type [] GetGenericParameters ();
virtual Type GetGenericTypeDefinition ();

```

Listing 19: *Eine Auswahl der neuen Eigenschaften der Type-Klasse in C#.*

## 7 Konklusion

Die nachträgliche Integration eines solch grundlegenden Sprachkonstrukts, wie der Generizität, muss immer mit Problemen verbunden sein. Zur Bewertung der neuen parametrischen Polymorphie in Java und C# müssen zwei Aspekte betrachtet werden: zum einem die Implementation selber und zum anderen die Integration in die bestehende Sprachwelt.

Die Implementation in Java ist in ihrer Umsetzung kompliziert und ihre Restriktionen teilweise unintuitiv ersichtlich. Hier bleibt dem Java-Entwickler nur anzuraten sich ein tieferes Verständnis über die Technik der Erasures anzueignen, um die Limitationen der Implementation richtig einschätzen zu können.

Dagegen ist die C# Implementation mächtiger und vor allem auch intuitiver. Zudem ist die Umsetzung auch performanter als eine äquivalente händische Umsetzung.

Was den Methodikwechsel in eine generische Sprache angeht, hat Java diesen Schritt allerdings deutlich offensiver beschritten. Die Änderung der API, bei Wahrung der Rückwärtskompatibilität ist eine sehr attraktive Eigenschaft und signalisiert wirksam auch die Bekenntnis zu der neuen Polymorphität.

Hier erscheinen die zukünftigen Auswirkungen für das .NET Rahmenwerk wesentlich nebulöser. Die Einführung eines neuen Namenraums bewahrt zwar auf einfache Art die Rückwärtskompatibilität, allerdings wirkt der Übergang etwas unmotiviert. Sicherlich ist für Microsoft hier auch zu bedenken, dass Änderungen an dem .NET Rahmenwerk sich potentiell in allen CLR-Sprachen auswirken. Auch sie müssen um Generizitätskonstrukte erweitert werden. Insofern ist die genaue Entwicklung der Generizität im .NET Rahmenwerk noch abzuwarten. Der endgültige Umstieg könnte sich in der jetzigen Form wohlmöglich erst spät oder sogar nie vollziehen.

## Literatur

- [1] Gilad Bracha, Norman Cohen, Christian Kemper, Martin Odersky, Kresten Stoutamire, David abd Thorup, and Philip Wadler. *Adding Generics to the Java Programming Language. Public Draft Specification, Version 2.0*, June 2003.
- [2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications.*, Vancouver, Canada, October 1998. Am 18.2.2004 unter <http://www.cis.unisa.edu.au/pizza/gj/Documents/gj-oopsla.pdf>.
- [3] James Gosling, Bill Job, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, Inc. Am 18.2.2004 unter [http://java.sun.com/docs/books/jls/second\\_edition/html/j.title.doc.html](http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html).
- [4] Andrew Kennedy and Don Syme. Design and implementation of Generics for the .NET Common Language Runtime. In *PLDI-01 (Proceedings of the ACM SIGPLAN '01 Conference on Programming Languages Design and Implementation)*. Microsoft, June 2001. Am 18.2.2004 unter <http://research.microsoft.com/projects/clrgen/generics.pdf>.
- [5] Juval Lowy. MSDN article: Introduction to C# Generics, August 2003. Am 18.2.2004 unter [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv\\_vstechart/html/vbconcpogramminglanguagefuturefeatures.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vbconcpogramminglanguagefuturefeatures.asp).
- [6] Sun Microsystems, Aarhus University and the Alexandra Institute. *Variance in the Java Programming Language. A Whitepaper.*, May 2003. Am 18.2.2004 unter <http://www.daimi.au.dk/plesner/variance/docs/variance-whitepaper.pdf>.