

# Proseminar Programmiersprachen

## Java 5 - Generics

Michaela Hien

Betreuer: Florian Haftmann

Technische Universität München

18. Oktober 2006

# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>3</b>
<b>2</b>	<b>Die Programmiersprache Java</b>	<b>3</b>
2.1	Die Geschichte von Java . . . . .	3
2.2	Grundprinzipien von Java: Objektorientierung und Vererbung . . . . .	4
2.3	Polymorphie in Java . . . . .	4
<b>3</b>	<b>Simple Generics</b>	<b>5</b>
3.1	Warum Generics . . . . .	5
3.2	Ein Beispiel . . . . .	6
3.3	Das Konzept von Generics . . . . .	6
3.4	Typeerasure . . . . .	7
<b>4</b>	<b>Das Beispielprogramm</b>	<b>8</b>
<b>5</b>	<b>Tieferer Einstieg in Generics</b>	<b>8</b>
5.1	Subtyping . . . . .	8
5.2	Wildcards . . . . .	9
5.3	Bounded Wildcards . . . . .	10
5.4	Generische Methoden . . . . .	11
5.5	Generische Methoden oder Wildcards ? . . . . .	12
5.6	Generische Klassen . . . . .	13
<b>6</b>	<b>Arrays</b>	<b>15</b>
6.1	Vergleich Arrays und generische Collections . . . . .	15
6.2	Arrays von parametrisierten Typen . . . . .	16
<b>7</b>	<b>Kritische Betrachtung von Generics</b>	<b>17</b>
7.1	Sinnvoller Einsatz von Generics . . . . .	17
7.2	Fazit . . . . .	18
<b>A</b>	<b>Literaturverzeichnis</b>	<b>19</b>

# 1 Einführung

Java gehört mittlerweile zu den am meisten eingesetzten Programmiersprachen. Dabei wird es ständig weiterentwickelt. Zuletzt ist die Version Java 5.0 im September 2004 erschienen. Dabei wurden u.a. Generics in Java eingeführt.

Diese Seminararbeit beschäftigt sich im Rahmen eines Proseminars mit der Programmiersprache Java. Der Schwerpunkt liegt dabei auf den neu eingeführten Generics. Die Arbeit stellt die Beweggründe für die Einführung, das Konzept und die Verwendung von Generics dar. Ein Beispielprogramm soll diese Punkte veranschaulichen.

## 2 Die Programmiersprache Java

### 2.1 Die Geschichte von Java

1991 sollte James Gosling in Auftrag von Sun mit 12 weiteren Entwicklern The Green Project durchführen. Eine Software zur Steuerung von im Netzwerk verteilten Geräten. Das Projekt sollte in C++ durchgeführt werden, doch James Gosling wollte damit nicht mehr arbeiten und entwickelte in nur 18 Monaten im Rahmen des Projekts die Programmiersprache Java.

Die neue Programmiersprache sollte mit den Nachteilen von C++ aufräumen. The Green Project erforderte das die Programmiersprache netzwerkorientiert, klein, sicher und portabel ist.

The Green Project konnte sich aber nicht durchsetzen und so sah es am Anfang erstmal schlecht für Java aus.

Doch durch die Ausbreitung des Internets wurde eine Sprache gebraucht, an die die gleichen Anforderungen gestellt wurden wie für The Green Project. Es wurde der Browser HotJava entwickelt, der Java-Applets ausführen konnte, und so begann die Erfolgsgeschichte von Java.

## 2.2 Grundprinzipen von Java: Objektorientierung und Vererbung

Java ist eine objektorientierte Programmiersprache. Dies bedeutet, dass im Gegenteil zu prozessorientierten Sprachen, bei denen der Ablauf eines Programms im Mittelpunkt steht, die Struktur von Objekten im Mittelpunkt steht.

Das Programm besteht aus einer Vielzahl von Objekten, die untereinander Nachrichten austauschen. Die Kapselung von Programmteilen in Objekten hat die Vorteile, dass das Programm übersichtlicher, wieder verwendbarer und sicherer ist.

Java ist aber streng betrachtet keine reine objektorientierte Sprache, da es primitive Datentypen besitzt und eine reine objektorientierte Sprache eigentlich dadurch definiert ist, dass die Programme nur aus Objekten bestehen.

Eines der wesentlichen Eigenschaften der Objektorientierung ist die Vererbung. Dabei werden Konstruktoren, Attribute und Methoden an die abgeleitete Klasse weitergegeben werden. Die Subklasse kann diese überschreiben oder auch neue hinzufügen.

Java unterstützt keine Mehrfachvererbung. Diese kann aber in Java durch die Implementierung mehrere Interfaces nachempfunden werden.

Jedes Objekt hat eine mehr oder weniger lange Vererbungshierarchie, d.h. eine Kette von Klassen die jeweils von einander erben. Am Anfang steht dabei immer die Klasse Object, von der jede Klasse erbt. Für die Klasse Integer sieht die Vererbungshierarchie z.B. so aus:

- java.lang.Object
  - java.lang.Number
    - java.lang.Integer

## 2.3 Polymorphie in Java

Polymorphie bedeutet die gleiche Variable oder Routine kann mit verschiedenen Typen belegt werden. In Java gibt es verschieden Formen von Polymorphie.

Enthaltende Polymorphie bedeutet, dass anstelle eines bestimmten Typs auch ein Subtyp verwendet werden kann – auch Subtyping genannt.

Zur enthaltenden Polymorphie gehören auch Casts und überladene Methoden. Ist der Rückgabotyp einer Methode nicht der erwartete, sondern z.B. ein Supertyp, so kann gecastet werden.

Methoden können außerdem von Subklassen überladen werden. Die Entscheidung, welche der überladenen Methoden verwendet wird, wird erst zur Laufzeit getroffen.

Eine neue Form der Polymorphie in Java ist die parametrische Polymorphie, d.h. Klassen und Methoden stehen in Abhängigkeit von Typparametern. Dies wird Generizität (= Generics) genannt und in den folgenden Kapiteln erklärt.

## 3 Simple Generics

### 3.1 Warum Generics

Ein typisches Einsatzgebiet von Generics ist die Collection API. Collections verwalten Daten des gleichen Typs. Bis Java 1.4 konnten Collections nur Daten vom Typ Object verwalten. Wollte man z.B. ein Element aus einer Liste von Integer herausnehmen, so musste man den Typ casten, da die Liste ja nur Objects enthielt.

```
ArrayList intList = new ArrayList();
intList.add(new Integer(1));
Integer num = (Integer) intList.get(0);
```

Beispiel 3.1.1 Casts

Diese Casts haben die Nachteile, dass sie ClassCastExceptions verursachen können, der Compiler keine Typsicherheit garantieren kann und der Code unübersichtlich ist.

Der Entwickler hatte die Verantwortung, dass kein Fehler wie dieser entsteht.

```
ArrayList intList = new ArrayList();
intList.add(new Integer(1));
String text = (String) intList.get(0);
```

Beispiel 3.1.2. wirft `ClassCastException`

### 3.2 Ein Beispiel

Durch Generics ist es möglich den Typ anzugeben, den ein Element in der Collection hat.

```
ArrayList<Integer> intList = new ArrayList<Integer>();
intList.add(new Integer(1));
Integer num = intList.get(0);
```

Beispiel 3.2. Simple Generics

Der Einsatz von Generics hat also die Vorteile, dass keine Casts nötig sind und somit potenzielle Fehlerquellen umgangen werden und dass der Compiler das Einstellen und Herausnehmen von Elemente überprüfen kann. Er meldet einen Fehler, wenn ein Element eines falschen Typs eingestellt werden soll, und er erkennt automatisch den Typ des Elements beim Herausnehmen.

### 3.3 Das Konzept von Generics

Generizität bedeutet die Abstraktion durch die Verwendung von Typparametern.

```
public interface List<E> {
    void add (E x);
    ...
}
```

Beispiel 3.3. Interface List in der Collection API

Eine Klasse oder Methode heißt generisch, wenn sie in Abhängigkeit eines Typparameters ist. Begriffe:

- `<E>` formal type parameter: Parameter in der generischen Klasse/Methode
- `<Integer>` actual type parameter: eingesetzter Parameter
- `List<Integer>` parameterized typ: der in Abhängigkeit eines Parameters stehende Typ

### 3.4 Typeerasure

Eine Möglichkeit Generics umzusetzen wäre, dass der Code kopiert wird, wenn der Typ eingesetzt wird und der Code somit mehrfach vorliegt.

In Java ist das aber nichts so - stattdessen wird Code Sharing durchgeführt d.h. der Code existiert nur ein einziges Mal.

Die gesamte Umsetzung von Generics geschieht im Compiler. Die Java Virtual Maschine unterstützt eigentlich keine Generics. Wenn der generische Code wieder dekompiert wird, sieht man, dass die Anweisungen Casts enthalten. Die Typinformation steht nur während der Kompilierung zur Verfügung. Der Compiler entfernt alle generischen Typinformationen. Dies nennt man Typeerasure. Der Bytecode ist dann gleich wie der Code ohne Generics d.h. es wird Typ Object verwendet und in Subtyps gecastet. Dies erklärt z.B., dass die Prüfung von `ArrayList<Integer> == ArrayList<String>` zur Laufzeit `true` zurückgibt.

Generics sind durch Typeerasure in Java realisiert worden, damit alter Code, der vor der Einführung von Generics entwickelt wurde, mit neuem kompatibel ist. Diese Form der Umsetzung von Generizität ist relativ einfach und es ist keine Änderung der Java Virtual Machine nötig geworden. Außerdem entstehen praktisch keine Nachteile, wenn Generics in dieser Form eingesetzt werden, da allenfalls die Compilezeit länger wird und dies meist irrelevant ist.

## 4 Das Beispielprogramm

Das nachfolgende Kapitel soll anhand des Beispielprogramms, das eine Liste von Zeichenelementen verwaltet, verdeutlicht werden. Deshalb wird hier eine kurze Beschreibung des Programms vorangestellt.

Das Programm soll eine Liste von Zeichenelementen

- mit einem Printbefehl ausgeben,
- zeichnen,
- um Elemente erweitern und
- ein Element aus der Liste löschen und zurückgeben, wenn es einen bestimmten Punkt enthält.

In der Bibliothek gibt es bereits das Interface Shape für Zeichenelemente, sowie diverse Implementierungen davon z.B. als Ellipse, die verwendet werden sollen. Die Methoden sollen polymorph für Listen mit verschiedenen Typen von Zeichenelementen verwendet werden könne. Es soll gleichzeitig möglich sein sicherzustellen, dass nur ein Typ von Zeichenelement in der Liste ist z.B. nur Rechtecke. Als Listentyp soll ArrayList verwendet werden.

## 5 Tieferer Einstieg in Generics

### 5.1 Subtyping

Um sicherzustellen, dass nur eine Art von Zeichenelement in einer Liste ist, gibt man dessen Typ als Parameter bei ArrayList an z.B. ArrayList<Ellipse>. Die Frage ist, ob man diese Ellipsenliste einsetzen kann, wo eine Liste, die allgemein für Zeichenelemente definiert ist, gefordert ist. Ellipse ist ja ein Subtyp von Shape.

```
ArrayList<Ellipse> ellipseList = new ArrayList<Ellipse>();
ArrayList<Shape> shapeList = ellipseList;
shapeList.add(new Rectangle());
Ellipse ellipse = ellipseList.get(0);
```

Beispiel 5.1 Fehler: type mismatch

In der letzten Zeile wird einer Ellipse ein Rechteck zugewiesen – die Typsicherheit ist nicht mehr gegeben. Deshalb ist `ArrayList<Ellipse>` kein Subtyp von `ArrayList<Shape>`. Zeile 2 ergibt bereits einen Kompilierungsfehler. Allgemein: Eine typparametrisierte Collection ist nicht der Subtyp einer Collection, deren Typparameter ein Supertyp des Parameters ist.

## 5.2 Wildcards

Es soll eine Methode geben die alle Elemente einer Liste mit einem Printbefehl ausgibt. Bei Collections ohne Generics wäre diese Methode einfach und polymorph definiert, da die Methode `println()` Typ `Object` entgegennimmt.

```
public void printAll(AbstractList<Object> list){
    for (Object object : list)
        System.out.println(object);
}
```

Beispiel 5.2.1 nicht polymorph

Wie aber vorher festgestellt wurde ist Subtyping bei parametrisierten Collections nicht möglich. Also kann die gerade definierte Methode z.B. keine `ArrayList<Ellipse>` entgegennehmen und ist somit nicht wie gewünscht polymorph. Für diesen Fall gibt es Wildcards.

```
public void printAll(AbstractList<?> list) {
    for (Object object : list)
        System.out.println(object);
}
```

Beispiel 5.2.2 Wildcards aus dem Beispielprogramm

Diese Methode kann jede `ArrayList` entgegennehmen und ist somit polymorph verwendbar. Das `?` ist ein Platzhalter und kann in Methoden und Attributen eingesetzt werden. Zur Laufzeit ist der Typ durch `TypeErasure` `Object` also kann aus einer Liste, die mit einem Wildcard instanziiert wurde, ein Element vom Typ `Object` herausgenommen werden. Allerdings ist das Hinzufügen eines Elements nicht erlaubt, da dies aus den gleichen Gründen wie beim Subtyping nicht typsicher ist.

```
AbstractList<?> list = new ArrayList<Ellipse>();  
list.add (new Object());
```

Beispiel 5.2.3 Fehler: Es können keine Elemente hinzugefügt werden.

### 5.3 Bounded Wildcards

Für das Beispielprogramm muss eine Methode erstellt werden, die alle Zeichenelemente aus einer Liste zeichnet und für jede Art von List, die Zeichenelemente enthält, verwendbar sein z.B. sowohl für `ArrayList<Rectangle>` als auch für `ArrayList<Ellipse>`.

Das Interface `Shape` stellt eine Methode zur Verfügung, mit der ein Zeichenelement gezeichnet werden kann - `draw(Shape)`. Einfache Wildcards sind hier nicht verwendbar, da aus der Collection nur der Typ `Object` entnommen werden kann und das Beispielprogramm ohne Casts sein soll. Um den Elementtyp einer Collection zu spezifizieren, können bounded Wildcards verwendet werden.

```
public void drawAll(AbstractList<? extends Shape> list) {  
    for (Shape shape : list)  
        draw(shape);  
}
```

Beispiel 5.3.1 extends-bounded Wildcard aus dem Beispielprogramm

Die Methode akzeptiert alle Arten von Listen, die als Typparameter ein Subtyp von `Shape` besitzen. Die Spezifizierung durch `extends` wird auch `upper bound` genannt. Für durch `extends` gebundene Wildcards gilt aber das gleiche

wie für Wildcards. Nur lesender Zugriff ist möglich - es können keine Elemente hinzugefügt werden.

Sollen Elemente in die Liste eingefügt werden, so muss der Wildcard durch `super` gebunden werden. Dies wird `lower bound` genannt.

```
ArrayList<? super Ellipse> list;  
list.add(new Ellipse());
```

Beispiel 5.3.2 super-bounded Wildcard

Da der tatsächliche Typparameter der `ArrayList` nur ein Supertypen von `Ellipse` sein darf, ist die Typsicherheit gegeben, wenn eine `Ellipse` eingefügt wird.

## 5.4 Generische Methoden

Eine Methode soll aus der Liste mit Zeichenelementen ein Element löschen und zurückgeben, wenn es einen bestimmten Punkt enthält. Der Rückgabetypp soll der gleiche Typ sein wie der Typ der Listenelemente.

Eine Methode kann in Abhängigkeit von Typparametern definiert werden. Dabei darf auch der Rückgabetypp der Typparameter sein.

```
public <E extends Shape> E remove(AbstractList<E> list,  
Point point){  
    for (E shape : list) {  
        if (shape.contains(point)) {  
            list.remove(shape);  
            return shape;  
        }  
    }  
    return null;  
}
```

Beispiel 5.4.1 generische Methode aus dem Beispielprogramm

Zwischen Zugriffsmodifier und Rückgabetyt werden die Typparameter angegeben, von denen die Methode abhängig ist. Die Typparameter sind einzelne Großbuchstaben typischerweise E. Sie können genauso wie Wildcards mit extends oder super spezifiziert werden. Zur Laufzeit muss aber ein spezifischer Typ zugewiesen worden sein. Der Compiler leitet aus der der Methode übergebenen Liste den spezifischen Typ ab. Dies wird auch Typinferenz genannt. In generischen Methoden kann in eine Collection auch ein Element eingefügt werden.

```
public <E> void addElem(AbstractList<E> list, E elem) {  
    list.add(elem);  
}
```

Beispiel 5.4.2 add in einer generische Methode

## 5.5 Generische Methoden oder Wildcards ?

Grundsätzlich kann mit Typparamteren alles ausgedrückt werden, was auch mit Wildcards möglich ist, da Wildcards nur Platzhalter für Typparameter sind. Bestimmte Methoden können deshalb als generische Methode oder mit Wildcards implementiert werden.

```
public void drawAll(AbstractList<? extends Shape> list) {  
    for (Shape shape : list)  
        draw(shape);  
}  
  
public <T> void drawAll (AbstractList<T extends Shape> list){  
    for(T shape: list)  
        draw(shape);  
}
```

Beispiel 5.5 Vergleich Wildcard – generische Methode

Um den Code möglichst übersichtlich zu gestalten gilt der Grundsatz: Wenn ein Typparameter nur einmal verwendet wird, sollten Wildcards genutzt werden, ansonsten generische Methoden.

## 5.6 Generische Klassen

Genauso wie Methoden können auch Klassen typparametrisiert sein. Die Collection API besteht aus generischen Klassen.

```
public interface List<E> {  
    void add (E x);  
    ...  
}
```

Beispiel 3.3. Interface List in der Collection API

Generische Klassen können auch selbst erstellt werden mit den gleichen Regeln wie für generische Methoden.

```
public class ShapeList<E extends Shape> extends ArrayList<E> {  
  
    public void drawAll( ) {  
        for (E shape : this)  
            draw(shape);  
    }  
  
    public E remove(Point point) {  
        for (E shape : this) {  
            if (shape.contains(point)) {  
                this.remove(shape);  
                return shape;  
            }  
        }  
        return null;  
    }  
}
```

Beispiel 5.6.1 generische Klasse – modifiziertes Beispielprogramm

Ein Typparameter kann durch extends oder super spezifiziert werden und es können mehrer Typparameter gleichzeitig verwendet werden.

```
public interface Map<K,V>{
    ...
}
```

Beispiel 5.6.2 Interface Map in der Collection API

Für die Verwendung von Typparametern gibt es aber sowohl in generischen Klassen wie auch in Methoden einige Einschränkungen. Es können keine neuen Elemente des Typparameters erzeugt werden.

```
public <E> void addNewElem(AbstractList<E> list) {
    E elem = new E( );
    list.add(elem);
}
```

Beispiel 5.6.3 Fehler: neue Elemente können nicht erzeugt werden

Dies funktioniert nicht, da zur Runtime die Typinformation durch Type-erasure fehlt und E() zur Runtime somit Object() wäre. Ein Typparameter darf kein primitiver Datentyp sein. Außerdem können generische Typen nicht in statischen Elementen verwendet werden.

```
public class myList<E> {
    static E firstElem;
    ...
}
```

Beispiel 5.6.4 Fehler: kein Typparameter in statischen Elementen

Dies ist nicht möglich, da ein statisches Element nur einmal für alle Instanzen existiert und somit nur einen festen Typ haben kann. Außerdem darf ein Typparameter nicht hinter extends oder implements stehen.

```
public class myList<E> extends E{...}
public class myList<E> implements E{...}
```

Beispiel 5.6.5 Fehler: kein Typparameter hinter extends/implements

Da eine instanceof Prüfung erst zur Laufzeit geschieht, darf wegen Typerasure auch hier der Typparameter nicht verwendet werden.

## 6 Arrays

### 6.1 Vergleich Arrays und generische Collections

Kovarianz bedeutet, dass ein Typ spezialisiert werden kann, d.h. wenn ein Typ erwartet wird kann auch ein Subtyp eingesetzt werden. In Kapitel 4.1. ist festgestellt worden, dass der Typparameter einer generischen Collection nicht durch einen Subtyp ersetzt werden kann. Damit sind Typparameter nicht kovariant.

Arrays haben dagegen die Eigenschaft, dass ihre Typen kovariant sind. `Object[]` lässt sich durch `String[]` ersetzen d.h. der dynamische Typ kann vom statischen abweichen.

Durch die Kovarianz von Arraytypen können aber auch Fehler entstehen.

```
Object[ ] objArr;  
String[ ] strArr = new String[10];  
objArr = strArr;  
  
objArr[0] = new Integer(5);  
String string = strArr[0];
```

Beispiel 6.1. wirft `ArrayStoreException`

Die letzte Zeile würde einem `String` einen `Integer` zuweisen. Die vorletzte Zeile wirft bereits eine `ArrayStoreException`. Diese Exception tritt aber erst zur Laufzeit auf.

## 6.2 Arrays von parametrisierten Typen

Ein ähnliches Beispiel nur, dass das Array einen parametrisierten Typ hat:

```
ArrayList<Integer>[ ] intListArr = new ArrayList[10] ;  
Object [ ] objArr = intListArr;  
objArr[0] = new ArrayList<String>();
```

Beispiel 6.2.1 Fehler – wirft in diesen Zeilen aber keine Exception

Zur Runtime wird aber hier in der letzten Zeile keine Exception geworfen, da zur Laufzeit `ArrayList<Integer>` und `ArrayList<String>` wegen Typersure zur `ArrayList` werden und somit vom gleichen Typ sind. Die Folge ist, dass der Fehler noch später auftaucht. Es kann z.B. eine `ClassCastException` bei der Entnahme eines Elements geworfen werden. Damit ist der eigentliche Fehler beim Debuggen des Codes schwer zu lokalisieren.

```
ArrayList<Integer>[ ] intListArr = new ArrayList[10] ;
```

Beispiel 6.2.2 wirft unchecked Warnung

Allerdings gibt der Compiler bei der Zeile bereits eine unchecked Warnung, da das Array nicht generisch initialisiert wurde.

```
ArrayList<Integer>[ ] intListArr = new ArrayList<Integer>[10] ;
```

Beispiel 6.2.3 Fehler: generisches Array kann nicht erzeugt werden

Wird allerdings der Typparameter bei der Initialisierung angegeben, dann wirft der Compiler den Fehler, dass er kein generisches Array erstellen kann. Die gleiche Fehlermeldung tritt auf, wenn Arrays von einem Typparameter innerhalb einer generischen Klasse bzw. Methode initialisiert werden sollen.

```

public <T> T[ ] fromListToArray (AbstractList<T> list){
    T[ ] array = new T[10];
    ...
}

```

Beispiel 6.2.4 Fehler: generisches Array kann nicht erzeugt werden

Um Arrays von parametrisierten Typen zu initialisieren ohne eine unchecked Warnung zu bekommen, können auch Wildcards eingesetzt werden.

```

List<?>[ ] array = new List<?>[10] ;
List<Integer>intList = new ArrayList<Integer>( ) ;
intList.add(new Integer(2));
array[1] = intList;
Integer i = (Integer) array[1].get(0);

```

Beispiel 6.2.5 Array mit Wildcard

Der Code ist aber wieder mit Casts und es liegt in der Verantwortung des Programmierers, dass in der letzten Zeile keine ClassCastException entsteht. Dies bedeutet, dass Arrays nicht typsicher mit generischen Typen kombinierbar sind. Hier besteht noch Bedarf, Arrays an das Konzept von Generics anzupassen, sodass sie typsicher kombinierbar sind.

## 7 Kritische Betrachtung von Generics

### 7.1 Sinnvoller Einsatz von Generics

Grundsätzlich gilt, dass Generics eine Form von Polymorphie sind und somit immer dann sinnvoll sind, wenn der Code für unterschiedliche Typen wieder verwendbar sein soll. Allerdings könnte überall dort wo Generics eingesetzt werden auch Polymorphie durch Subtyping bzw. Casts verwendet werden. Generics haben aber den Vorteil, dass sie Typsicherheit bieten und Fehler früher und damit auch besser erkannt werden können. Bei dem Einsatz von Generics taucht ein eventueller Fehler dort auf, wo er auch entstanden ist - bei Casts

ist dem nicht so.

Die Polymorphie durch Typparameter hat aber ihre Grenzen, so können z.B. keine neuen Elemente instanziiert werden ohne den konkreten Typ zu wissen.

Im Vorfeld der Einführung von Java 5 und Generics gab es viele Diskussionen, ob Generizität Java nicht zu komplex und damit unübersichtlich macht. Durch Typerasure ist aber eine einfache Umsetzung gewählt worden und somit hat Generizität in Java nicht die gleiche Komplexität wie z.B. C++ Templates. Da Collections auch ohne Angabe des Typparameters initialisiert werden können, ist der Entwickler nicht gezwungen Generics einzusetzen. Damit aber nicht unabsichtlich die Typsicherheit umgangen wird, weisen unchecked Warnungen darauf hin.

## **7.2 Fazit**

Zusammenfassend kann man sagen:

- Generics in Java bringen praktisch keine Nachteile
- Das Wegfallen von Casts macht den Code übersichtlicher
- Bedingungen lassen sich durch extends und super besser und einfacher ausdrücken
- Fehler treten an den Stellen auf, wo sie verursacht werden
- In Bezug auf Arrays ist das Konzept der Typsicherheit noch nicht ausgereift

## A Literaturverzeichnis

### Literatur

- [1] Bracha, Gilda: Generics Tutorial - Generics in the Java Programming Language, 2004  
[http:// java.sun.com/ j2se/ 1.5/ pdf/ generics-tutorial.pdf](http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf)
- [2] Mahmoud, Qusay H.: Using and Programming Generics in J2SE 5.0, 2004  
[http:// java.sun.com/ developer/ technicalArticles/ J2SE/ generics/ index.html](http://java.sun.com/developer/technicalArticles/J2SE/generics/index.html)
- [3] Langer, Angelika: Arrays in Java Generics, 2003  
[http:// www.angelikalanger.com/ Articles/ Papers/ JavaGenerics/ ArraysInJavaGenerics.htm](http://www.angelikalanger.com/Articles/Papers/JavaGenerics/ArraysInJavaGenerics.htm)
- [4] Puntigam, Franz: Skriptum zu Objektorientierte Programmierung, 2004
- [5] Samaschke, Karsten: Java (zu J2SE 5) - Einstieg für Anspruchsvolle, 2005