Seminararbeit

Isabel Harms
73313

Hochschule Karlsruhe
University of Applied Sciences

Bounded Generics in Java

7. November 2022

Zusammenfassung

Inhaltsverzeichnis

1 Einleitung **3**

2 Typisierung **5**

3 Generics **6**
- 3.1 Definition . . . 6
- 3.2 Exkurs: Polymorphismus . . . 8
- 3.3 Vorteile . . . 9
  - 3.3.1 Redundanz . . . 9
  - 3.3.2 Type Cast . . . 10
  - 3.3.3 Typsicherheit . . . 10

4 Objects **12**
- 4.1 Definition . . . 12
- 4.2 Vergleich zu Generics . . . 12

5 Bounded Generics **14**
- 5.1 Definition . . . 14
- 5.2 Beispiel Bounded Generics . . . 15
- 5.3 Mehrere Bounds . . . 17

6 Fazit **18**

Quellen **19**

A Anhang **20**
- A.1 Bilder . . . 20

1 Einleitung

Am 23. September 1999 verlor eine Raumsonde der NASA den Kontakt zur Erde, da sie zu nah an die Oberfläche des Mars gelangte und durch seine Atmosphäre verbrannte. Der Grund: Bei der Landenavigation kam es zu einer Verwechslung der metrischen Einheit Meter und der amerikanischen Einheit Fuß. Die damaligen Kosten der Mission betrugen fast 200 Millionen Dollar [4].

Objekte der Art „Fuß-angabe“ und „Meter-Angabe“ hätten diesen Fehler schon bei der Compilierung unübersehbar gemacht. Doch wie kann man eine solche Typ-Sicherheit garantieren und derartige Fehler zuverlässig und frühzeitig erkennen?
Es ist eine Typisierung gefragt, die ohne Einschränkung des Programms verhindert, dass solche Fehler erst zur Laufzeit auffallen. Der Versuch, zum Beispiel eine Zahl auf einen Text zu addieren, muss vom Compiler erkannt werden und das Programm sollte gar nicht erst auf Assemblerbefehle heruntergebrochen, geschweige denn ausgeführt werden.
Die aufkommende Software-entwicklung reagierte auf dieses und eine Vielzahl anderer Probleme durch die Entwicklung von Sprachen, welche eine Typisierung von Daten erlauben. Dabei werden die Adressen von Speicherbereichen nicht mehr durch den Programmierer vergeben, sondern selbst wiederum durch ein Programm, den Compiler.
Betrachten wir die historische Entwicklung der Programmiersprachen, dann beginnt dies mit den sog. Assemblersprachen. Sie beinhalten bekanntlich keine Typisierung und zeigen durch ihre Fehleranfälligkeit die Motivation auf, eine solche einzuführen.
Heutzutage gibt es eine Vielzahl an Sprachen, die Generics unterstützen, unter anderem C sharp, Go, oder auch, wie in dieser Arbeit ausgeführt, Java ab der Version 1.5.

In dieser Arbeit wird Sinn und Zweck der „bounded generics“anhand von Beispielen erläutert. Dazu wird zunächst der Begriff der allgemeinen Generics erläutert und insbesondere werden die Konsequenzen aufgezeigt, die eine fehlende Typisierung mit sich bringen kann.
// welche Vorteile es haben kann, die Typbestimmung zu parametrisieren.
Im zweiten Schritt wird gezeigt wird die Alternative der Objects vorgestellt und die Unterscheiden werden erläutert.
Letztendlich werden die verbliebenen Fehlermöglichkeiten erörtert und im

Zuge dessen der Begriff der bounded Generics eingefügt. Hierbei müssen wir uns mit Polymorphismus auseinandersetzten, da Vererbungshierachien ein wirksames Mittel sind, Programmteile auf diejenigen Objekte einzuschränken, auf die sie auch anwendbar sind.

2 Typisierung

3 Generics

3.1 Definition

Als Generics bezeichnet man im allgemeinen eine Eigenschaft von objektorientierten Programmiersprachen, Objekte einerseits flexibel, andererseits mit größtmöglicher Typsicherheit zu behandeln. Es handelt sich ein Programmierparadigma, welches die Verwendung von zum Zeitpunkt der Implementierung nicht spezifizierten Typen zulässt. Diese Typvariablen werden erst bei Ausführung des Programmes konkretisiert und stellen bis dahin, wie Variablen für einen Wert, nur einen Platzhalter für den Typ dar. Sie kommen zum Einsatz, wenn ein Klassen- oder Methoden Parameter nicht allgemein eingeschränkt werden soll und erst zur Laufzeit konkretisiert wird. In Java sind diese seid der Version 1.5 und somit bereits seit 2004 implementiert.
Ein bekanntes Beispiel für eine parametrisierte Klasse in Java sind zum Beispiel ArrayLists.

```
1 ArrayList<String> names = new ArrayList<>()
```

Hier wird bei der Erzeugung eines Objekts als Parameter der Typ der String mitgegeben und somit versichert, dass alle Elemente die in dieser Klasse verwaltet werden auch von diesem Typ sind. Es handelt sich um eine „ArrayList of String“.
Betrachten wir die Implemetierung der Klasse kann man sie sich vereinfacht etwa so vorstellen:

```
1  public class ArrayList<E> {
2      //...
3      public boolean add(E e) {
4          add(e, elementData, size);
5          return true;
6      }
7      public E remove(int index) {
8          Objects.checkIndex(index, size);
9          final Object[] es = elementData;
10         E oldValue = (E) es[index];
11         fastRemove(es, index);
12
13         return oldValue;
14     }
15 }
```

Im Klassenkopf wird in spitzen Klammern ein polymorpher Container als Repräsentant verschiedener Datentypen verwendet, der wie im vorherigen Beispiel beim erzeugen des Objektes konkretisiert wird. verschiedene Methoden innerhalb dieser Klasse arbeiten ebenfalls mit diesem Container und verwenden ihn für Funktionsparameter (wie z.B. in *add*), oder als Rückgabetyp (wie in *remove*).
Zudem ist anzumerken, dass E frei benennbar ist. Häufig verwendete Aliase sind unter anderem:

- T - Type
- E - Element
- K - Key
- N - Number
- V - Value

3.2 Exkurs: Polymorphismus

Generics sind eine Form von parametric polymorphism, eine Art der Programmierung, in der die Typen der variablen eher außenvor gelassen werden. Polymorhpismus (gr. Vielgestaltigkeit) tritt in Vererbungshierarchien auf und ist nützlich um die Wiederverwendbarkeit von Code zu steigern.
Generische Typen sind nur eine Möglichkeit für einen Zugriff auf polymorphe Strukturen. Ein weiteres Mittel, Quellcoderedundanz zu vermeiden und Strukturen übersichtlich zu modellieren, ist die hierarchische Anordnung von Klassen mittels Vererbung. Sie ist ein mächtiges Mittel, Objekte zu klassifizieren.

Ein bekanntes Beispiel für statische Polymorphie ist das Überladen von Funktionen. Es können mehrere Funktionen mit dem selben Namen existieren, solange sich ihre Parameterlisten unterscheiden (overloading). Dies kann durch eine andere Anzahl an Methoden-Parametern, oder auch durch unterschiedliche Typen dieser Parameter auftreten.
Dynamischer Polmorphismus hingegen, tritt auf wenn innerhalb einer Vererbungshierarchie identische Methodenköpfe vorliegen, und Kinderklassen dann die äquivalenten Methoden ihrer Eltern überschreiben (overriding). Diese Form von Polymorphismus macht es z.B. möglich, eine Elternklasse als Typ eines Funktionsparameters zu verwenden und es damit ebenfalls allen Kinderklassen zu ermöglichen in dieser Methode verwendet zu werden. Dies ist möglich, da alle Kinderklassen mindestens über die Funktionalität ihrer Eltern verfügen.
Das hat jedoch zur Folge, dass bei Verwendung einer Methode dieses Objekts nicht bekannt ist um welche Methode es sich genau handelt, da ja auch nicht bekannt ist, ob es sich um die angegebene Eltern Klasse handelt oder um eine ihrer Kinder. Je nachdem wird die entsprechende Methode verwendet und das Ergebnis kann dementsprechend variieren.
Beispielsweise könnte eine Methode einen Parameter vom Typ der abstrakten Klasse Tier entgegennehmen und die Methode *fortbewegen* darauf aufrufen. Für verschieden Tiere würde dies komplett unterschiedlich aussehen.

Diese Funktionalität wird besonders interessant wenn wir uns mit f-bounded-polymorphismus beschäftigen.

3.3 Vorteile

3.3.1 Redundanz

Die Redundanz von Code erschwert nicht nur dessen Übersicht und Wartbarkeit, sondern kann auch eine vermeidbare Fehlerquelle darstellen, wenn bei einer Anpassung eine der vielen Stellen vergessen wird. Generics können hilfreich sein um eine eventuell mehrfach nötige Implemetierung durch Verallgemeinerung zu vermeiden.

Möchte man nicht soweit gehen eine ganze Klasse zu parametrisieren, so kann dies auch schon anhand einfacher Methoden festgestellt werden.

```
1 public static void printArray( String[] inputArray ) {
2     // Display array elements
3      for(String element : inputArray) {
4         System.out.printf("%s␣", element);
5     }
6     System.out.println();
7 }
```

Ziel dieser Methode ist es, alle Elemente eines String-Arrays nacheinander in der Konsole auszudrucken. Diese Funktionalität ist jedoch nicht ausschließlich für Strings sinnvoll, sondern auch für eine Reihe anderer Datentypen. Wollten wir diese Funktion nun z.B. auch für ein Integer-Array, so müsste sie komplett neu geschrieben werden, obwohl sich der Inhalt eigentlich komplett überschneidet. Damit diese Methode nicht für jeden dieser Typen einzeln angepasst werden muss, verwenden wir statt eines festen Typs einen *generic type*.

```
1 public static < E > void printArray( E[] inputArray ) {
2       // Display array elements
3       for(E element : inputArray) {
4          System.out.printf("%s␣", element);
5       }
6       System.out.println();
7    }
```

[3] Neben des Ersetzens alles festen String-Types durch unseren Container, muss der Container hierfür zusätzlich noch im MethodenKopf in spitzen Klammern angebenen werden, da wir uns nicht in einer generischen Klasse befinden.

Nun können jegliche Arten von Arrays in die Methode übergeben und in die Konsole gedruckt werden.

3.3.2 Type Cast

In vielen Fällen haben Generics noch einen weiteren Wert, an den viele auf den ersten Blick nicht denken. Betrachten wir dieses Codebeispiel von Dr. Hyunyoung Lee [1] zunächst ohne die Verwendung von Generics.
Hier wurde eine bekanntes Anwendungsbeispiel von Klassenparametrisierung, die LinkedList so verwendet, als hätte sie genau diesen Schlüsselaspekt nicht.

```
1  List l = new LinkedList();
2  l.add(new Integer(0));
3  Integer x = (Integer) l.iterator().next(); // need type cast
```

In diesem Codebeispiel muss für die Zuweisung des Rückgabewerts der Methode *next* an eine Variable des Typs Integers zunächst auf dessen Typ gecasted werden. Denn obwohl diese Liste nur einen Integer enthält ist nicht bekannt ob das zurückgegebene Objekt *next* auch diesen Typ hat. Schließlich könnten sich auch ganz andere Objekte dort befinden. In diesem Fall geht das ganze gut, doch einen solchen Cast auf ein Objekt mit unbekanntem Typ anzuwenden kann noch andere Probleme hervorrufen, wie beschrieben in 3.3.3.

Typecasting sollte bekanntlich eher vermieden werden, da es ein Zeichen von unsauberem und schlecht wartbarem Code ist. Meist gibt es eine alternative Strukturierung des Codes:

```
1  List<Integer> l = new LinkedList<Integer>();
2  l.add(new Integer(0));
3  Integer x = l.iterator().next(); // no need for type cast
```

Es fällt auf, dass durch die Parametrisierung der Klasse als *„LinkedList of Integer“* automatisch auch der Rückgabetyp der Methode *next()* als solcher eingesetzt wurde und den eben noch notwendigen Cast überflüssig macht.

3.3.3 Typsicherheit

Wir betrachten das selbe Beispiel, doch diesmal versuchen wir ein Element aus der Liste in einer Variablen des Typs String zu speichern. Auch hier ist

wieder ein Typecast von Nöten, damit nicht bereits in der IDE ein Fehler angezeigt wird.

```
1  List l = new LinkedList();
2  l.add(new Integer(0));
3  String s = (String) l.iterator().next();
```

Problem ist nur, dass das zurück gegebene Objekt, wie in diesem Kontext auch ersichtlich, ein Integer ist und daher nicht auf Sting gecasted werden kann. Bei der Ausführung dieser Zeile kommt es zu einem Laufzeitfehler.
Versuchen wir dieselbe Aktion mit der Verwendung von Generics, so ist bereits vor Kompilierung des Programmes bekannt, dass der Rückgabetyp der Funtion *next* ein Integer sein wird.

```
1  List<Integer> l = new LinkedList<Integer>();
2  l.add(new Integer(0));
3  String s = l.iterator().next();
```

Daher wird die Entwicklungsumgebung diese Zeile frühzeitig als fehlerhaft erkennen und eine entsprechende Fehlermeldung anzeigen (Hier am Beispiel von IntelliJ 1).

4 Objects

4.1 Definition

Die Klasse Object stellt in Java die Wurzel der Klassen Hierarchie dar. Das heißt, dass alle Klassen automatisch von dieser Klasse erben. Sie bietet einige grundlegenden Funktionalitäten wie zum Beispiel die *getClass* oder die *clone* Methode, die dadurch für jede einzelne Klasse zur Verfügung stehen. Seit Java Version...
Keine Type Safety Beim erzeugen einer ArrayList kann der Typ spezifiziert werden und so ein späterer Laufzeit-Fehler vermieden werden. RunTime Exeptions

4.2 Vergleich zu Generics

Es sollte bekannt sein, dass in Java-Methoden-Parametern auch Untergeordnete Klassen miteingeschlossen sind. Beispielsweise könnte eine Methode die auf Zahlen agiert einen Parameter des Typs *Number* erwarten, und somit auch alle Untergeodneten Typen, wie z.B. int, float oder double akzeptieren. Würden wir also den Typ des Parameters als Object festlegen, so wären Objekte aller Klassen gültig und die Methode wäre maximal verallgemeinert. Für das bereits aufgeführte Beispiel der printArray Methode sähe das wie folgt aus:

```
1     public static void printArray( Object[] inputArray ) {
2         // Display array elements
3         for(Object element : inputArray) {
4             System.out.printf("%s ", element);
5         }
6         System.out.println();
7     }
```

Äquivalent könnten so auch Klassen unparametrisiert bleiben und alle Methoden mit Objekten hantieren. Warum also den scheinbar komplizierten Weg über Generics wählen?

Eigentlich haben wir das Problem mit einer solch drastischen Verallgemeinerung bereits aufgezeigt.
Erinnern wir uns außerdem an das Beispiel der LinkedList ohne generics und an die Fehlermeldung welche ohne Typecast aufgetreten ist, so stellen wir fest,

dass das eigentliche Problem ein zu allgemeiner Rückgabetyp der Funktion *next* ist. Diese gibt nämlich auf Grund der fehlenden Parametrisierung der Klasse ein Object zurück. Was genau sich dahinter wirklich verbirgt kann erst bei Ausführung des Programmes festgestellt werden.

5 Bounded Generics

5.1 Definition

Bei Bounded Generics begrenzt man einen parametrisierten Typ anhand einer oberen Grenzen.

```
1 <T extends superClassName>
```

Das Keyword *extends* bezieht sich hierbei sowohl auf Vererbung von einer super Klasse, aber auch auf Implementierungen eines Interfaces.

Bisher war es ohne diese Beschränkung nur möglich Methoden auf T aufzurufen, welche von allen Typen unterstützt werden. Standardmäßig bildet also die Klasse Object die obere Grenze des Typs.

```
1 <T extends Object>
```

Auf einem Objekt mit diesem Type kann man also alle Methoden aufrufen, welche von Object vorgegeben werden (z.B. toString), oder man kann es in jegliche Methoden als Parameter geben, die alle Klassen akzeptieren (z.B die print Methode).

Die Existenz von bounded Generics ermöglicht es nun ebenfalls Methoden spezifischerer Klassen zu verwenden.

5.2 Beispiel Bounded Generics

Betrachten wir diese Klasse [2]

```
1  public class myNumber<T extends Number> {
2
3      private final T n;
4
5      public myNumber(T n) {
6          this.n = n;
7      }
8
9      public boolean isEven() {
10         return n.intValue() % 2 == 0;
11     }
12 }
```

5.3 Mehrere Bounds

6 Fazit

Literatur

[1] Dr. Hyunyoung Lee. *Bounded Types with Generics in Java*. URL: https://people.engr.tamu.edu/hlee42/csce314/lec13-java-genericsA.pdf (besucht am 05. 10. 2022).

[2] oracle. *bounded generics*. URL: https://docs.oracle.com/javase/tutorial/java/generics/bounded.html (besucht am 24. 10. 2022).

[3] Tutorials Point. *Java - Generics*. URL: https://www.tutorialspoint.com/java/java_generics.htm (besucht am 05. 11. 2022).

[4] NASA Official: Dr. David R. Williams. *Mars Climate Orbiter*. URL: https://nssdc.gsfc.nasa.gov/nmc/spacecraft/display.action?id=1998-073A (besucht am 24. 10. 2022).

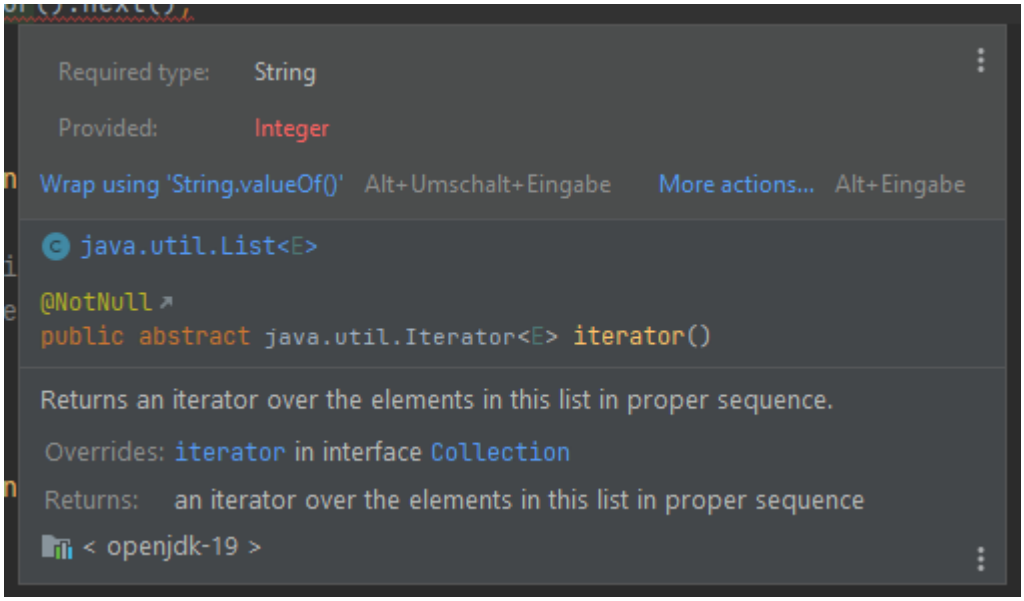

Abbildung 1: Fehlermeldung in IntelliJ

A Anhang

A.1 Bilder