Seminararbeit

Isabel Harms
73313

Hochschule Karlsruhe
University of Applied Sciences

Bounded Generics in Java

11. November 2022

Zusammenfassung

Inhaltsverzeichnis

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 [3].

Dieses und viele andere Beispiele legen dar, dass eine fehlende Typisierung katastrophale Konsequenzen haben kann. Wie derartige Fehler frühzeitig erkannt und demnach vermieden werden können, soll im folgenden thematisiert werden.

In dieser Arbeit wird die Relevanz von flexibler Typisierung erläutert und warum Generics, auch parametrischer Polymorphismus genannt, eine sinnvolle Ergänzung zu Java darstellen. Dazu wird zunächst der Begriff der allgemeinen Generics eingeführt und insbesondere werden die Konsequenzen aufgezeigt, die auftreten können wenn sie nicht verwendet werden. Dies wird anhand von Beispielen erläutert, welche die generische Typisierung durch Objects ersetzten und entsprechend deutlich machen, welche Vorteile es mit sich bringen kann die Typbestimmung zu parametrisieren.
Im zweiten Schritt werden die verbleibende Fehlermöglichkeiten herkömmlicher Generics aufgezeigt und im Zuge dessen der Begriff der bounded Generics eingefügt. Der Sinn und Zweck dieses „f-bounded polymorphism“ wird anhand von Beispielen erläutert. Hierbei müssen wir uns mit Vererbungshierachien auseinandersetzten, da sie ein wirksames Mittel sind, Programmteile auf diejenigen Objekte einzuschränken, auf die sie auch anwendbar sind.

2 Typisierung

Wenn wir uns an das eben erwähnte Beispiel der Raumsonde erinnern, lässt sich feststellen, dass Objekte der Art „Fuß-angabe“ und „Meter-Angabe“ diesen Fehler schon bei der Compilierung unübersehbar gemacht hätten. Doch wie kann man eine solche Typ-Sicherheit garantieren und derartige Fehler zuverlässig und frühzeitig erkennen?

Die aufkommende Softwareentwicklung 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.
Sprachen wie Java profitieren durch ihre statische Typisierung[1] nicht nur von einer besseren Performance, sondern auch von geringerer Fehleranfälligkeit. 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.

```
1 int x = 0;
2 String s = "text";
3 x += s;
```

Dieser Java-Code verursacht dank der festen Typisierung in genau diesem Fall eine Fehlermeldung (Wie zu sehen in 5), die rechtzeitig vermittelt, dass die Datentypen der verwendeten Variablen inkompatibel sind.

Dennoch hat diese Art von strenger Typisierung den Nachteil, dass sie viel Flexibilität nimmt und die Mehrfachverwendung von Code stark einschränkt. Es ist also eine Typisierung gefragt, die ohne Einschränkung des Programms verhindert, dass derartige Fehler erst zur Laufzeit auffallen.

[1] Datentypen von Variablen sind bereits vor Laufzeit bekannt und können nicht mehr manipuliert werden.

3 Generics

Eine naheliegende Lösung hierfür wäre die Verwendung von Generics, auch parametrischer Polymorphismus genannt. 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.
Warum dies eine wertvolle Ergänzung darstellt und welche Probleme sich in älteren Versionen ergeben haben wird nun erläutert.

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.

Hier sehen wir eine simple Methode, welche die Elemente eines String-Arrays in Konsole druckt. Sie hat eine harte Typisierung lässt somit keinen Freiraum für alternative Parameter-Typen.

```
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 }
```

Diese und folgende Methoden dieses Kapitels sind Varianten eines Codesbeispiels von TutorialsPoint [2].

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.

In Java sind Generics seit der Version 1.5 und somit erst seit 2004 implementiert. Demnach muss es eine Alternative geben, mit welcher dieses Problem auch ohne generische Typen gelöst werden kann.
Es stellt sich also die Frage, wie Methoden auch ohne den Einsatz von parametrischem Polymorphismus verallgemeinert werden können.

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. Die Klasse Object stellt in Java die Wurzel der Klassen Hierarchie dar. Das heißt, dass alle Klassen automatisch von dieser Klasse erben. 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     }
```

Nun kann dieser Funktion jedes Array mitgegeben werden.

Im Vergleich hierzu die Variante, welche Generics verwendet:

```
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    }
```

Äquivalent zum Object-Array verhindert ein Array mit variablem Typ eine mehrfache nötige Programmierung der selben Methode. In diesem Beispiel scheint diese generische Methode sogar etwas komplizierter als die vorherige Lösung mit der Object Klasse. Warum also Generics verwenden wenn beide Varianten für eine Verallgemeinerung von Methoden sorgen?

Eigentlich haben wir die Typisierung von Java mit Hilfe von Object komplett umgangen. Welche Konsequenzen dies haben kann zeigt sich in den folgenden Kapiteln.

3.2 Typecasting

In Java gibt es eine Vielzahl von Frameworks, welche generische Parametriesierung verwenden. Insbesondere das *java.util* Paket enthält zahlreiche generische Klassen. Betrachten wir z.B. eine LinkedList stellt sich die Frage wie diese vor der Einführung von Generics angewandt wurde.
Wir betrachten hierfür ein Codebeispiel aus den Folien von Dr. Hyunyoung Lee [1]:

```
1  List l = new LinkedList();
2  l.add(new Integer(0));
```

Bei Erzeugung der Instanz wird kein Typ mitgegeben, die Liste kann also Elemente jeder Klasse enthalten. Die *add* Funktion muss demnach jegliche Objekte entgegennehmen können. Dies hat leider auch zur Folge, dass nun nicht bekannt ist welche Art von Objekten sich in der Liste befinden.

3.2.1 In einer Zuweisung

In diesem Codebeispiel muss für die Zuweisung des Rückgabewerts der Methode *next* an eine Variable des Typs Integers zunächst auf dessen Typ gecasted werden.

```
1  Integer x = (Integer) l.iterator().next(); //need type cast
```

Denn obwohl diese Liste nur einen Integer enthält ist für den Compiler 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.

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.

Übertragen wir diesen Code zunächst in eine generische Schreibweise, wird erneut deutlich, dass es sich bei den Elementen um jegliche Objects handeln kann.

```
1  List<Object> l = new LinkedList<Object>();
```

```
2 l.add(0);
3 Integer x = (Integer) l.iterator().next();
```

Auch bei dieser „LinkedList of Object“ ist noch ein Typecast von Nöten. Nun kann jedoch genauer beschränkt werden auf welche Art von Objekten wir uns genau beziehen wollen und was in dieser Liste verwaltet werden soll. In unserem Fall handelt es sich dabei um Integers. Schränken wir uns also auf diesen Typ, inklusive seiner Unterklassen ein:

```
1  List<Integer> l = new LinkedList<Integer>();
2  l.add(new Integer(0));
```

In die *add* Methode können nun nur noch Integers reingereicht werden. Hält man sich an diese Vorgabe nicht, so ist dies bereits in der IDE erkennbar. Der Folgende Versuch würde eine Fehlermeldung erzeugen, wie zu sehen in 2, da versucht wird ein Element des Typs 'char' in eine „LinkedList of Integer“ einzufügen.

```
1 l.add('t');
```

Ebenfalls ist der Rückgabetyp der *next* Funktion nun als Integer bekannt.

```
1  Integer x = l.iterator().next(); // no need for type cast
```

Daher ist nun ein Typecast für diese Zuweisung überflüssig

3.2.2 Für einen Methodenaufruf/ eine Operation

Es gibt noch andere Fälle in denen ein Typecast erforderlich wird, wenn die Verwendung von Generics nicht möglich wäre. Einige Operationen oder Methodenaufrufe sind nur auf bestimmten Objekten möglich.
Wenn also beispielsweise eine „List of Object“ ausschließlich Interegers enthält und diese alle aufaddiert werden sollen, so können die Elemente nicht einfach so auf eine Variable vom Typ int addiert werden.

```
1 int x = 0;
2 List l = new LinkedList();
3 l.add(1); l.add(42); l.add(100);
4 Iterator current = l.iterator();
5 while (current.hasNext()) {
6     x += (int) current.next();
7 }
8 print(x);
```

Mit Hilfe einer while-Schleife können wir von Element zu Element wandern und wollen jedes auf eine mit 0 initialisierte Variable aufaddieren.
Die += Operation ist jedoch auf bestimmt Datentypen eingeschränkt und kann nicht allgemein auf Objects angewendet werden. Mit dem Typecast liefert die Entwicklungsumgebung die korrekte Ausgabe 143, ohne ihn würde sie aber einen Fehler anzeigen 4.

Generics erleichtern diesen Vorgang:

```
1 int x = 0;
2 List<Integer> l = new LinkedList<Integer>();
3 l.add(1); l.add(42); l.add(100);
4 Iterator<Integer> current = l.iterator();
5 while (current.hasNext()) {
6     x += current.next();
7 }
8 print(x);
```

Zu bemerken ist hier, dass nicht nur die Liste parametrisiert ist, sondern auch der Iterator. Somit hat auch die auf dem Iterator aufgerufene Methode *next* den in den spitzen Klammern angegeben Rückgabetyp Integer.
Der Iterator der generischen Klasse hat den selben Typparameter wie die Klasse selbst. Dieser wird bei Erzeugung der Klasse übernommen.

3.3 Typsicherheit

Bisher sind wir immer davon ausgegangen, dass der Programmierer alles richtig macht und genau weiß, auf welche Art von Objekt gecasted werden muss. Das ist in der Realität aber natürlich nicht immer der Fall. Eventuell inkorrektes Typecasting, wie zum Beispiel im Folgenden kann zu Laufzeitfehlern führen:

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

Eigentlich handelt es sich hier um sehr ähnlichen Code wie im Beispiel des Typecastings. Problem ist nur, dass es sich in der Zeile der Zuweisung um einen Integer handelt und daher nicht auf Sting gecasted werden kann.
Der Compiler kann hiervon jedoch nichts wissen, da nicht bekannt ist welche Objekte in der Liste enthalten sein könnten. In diesem Beispiel ist es zwar für den Programmierer noch überschaubar, aber Fehler dieser Art sind in Programmes größerer Skala keine Seltenheit.

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).

Besonders wichtig zu bedenken ist, dass dieser Fehlerfall nicht das worst-Case Szenario ist. Viel schlimmer ist es, wenn es nur in bestimmten Situationen zum Laufzeitfehler kommt und dieser beim Testen übersehen wird, oder besonders, wenn es wie bei unserer Raumsonde gar nicht erst zu einem Error kommt, sondern das Programm mit den fehlerhaften Werten weiter rechnet und es nicht einmal auffällt, dass es sich hierbei um einen falschen Wert handelt.

3.3.1 Beispiel: Typsicheres Interface

3.4 Verbliebene Fehlermöglichkeiten

4 bounded Generics

4.1 Mehrere Bounds

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

[3] 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).

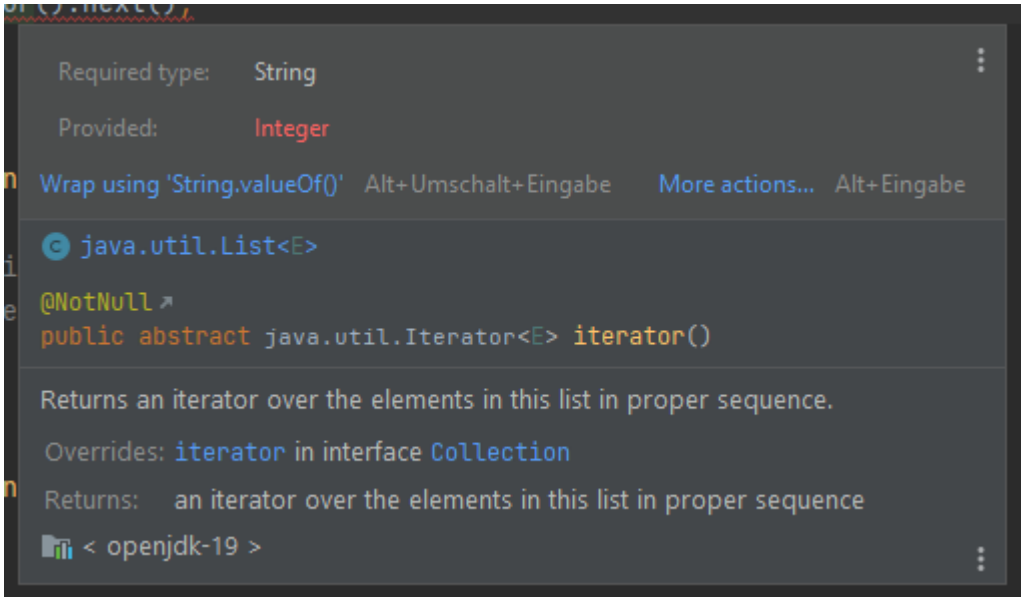

Abbildung 1:

A Anhang

A.1 Bilder

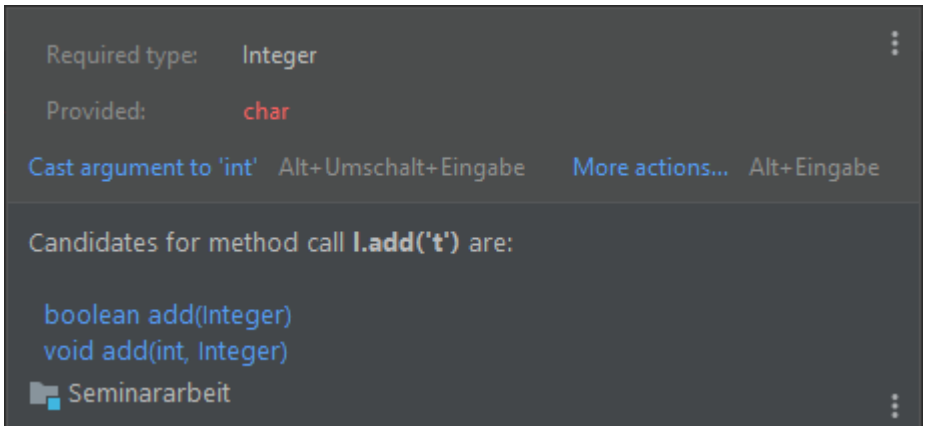

Abbildung 2:

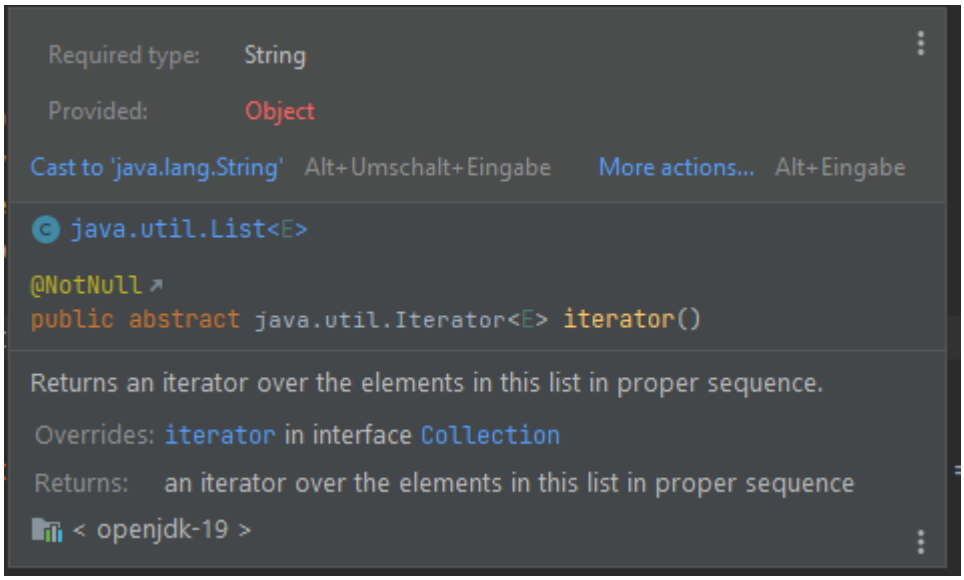

Abbildung 3: Fehlermeldung in IntelliJ

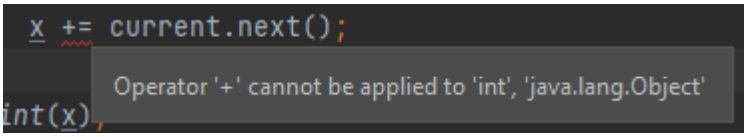

Abbildung 4:

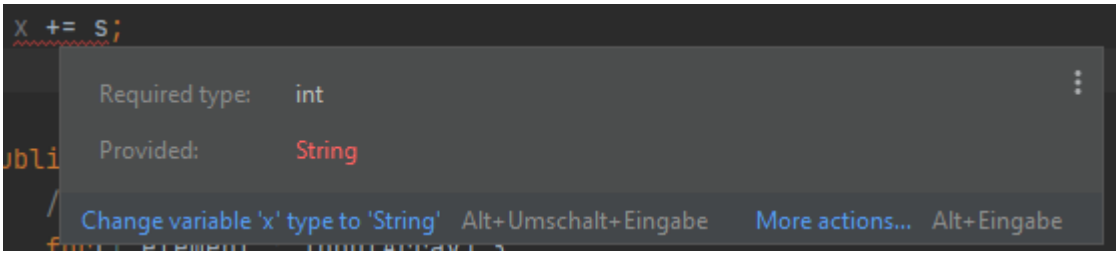
x += s;
Required type: int
Provided: String
Change variable 'x' type to 'String' Alt+Umschalt+Eingabe
More actions... Alt+Eingabe

Abbildung 5: