Seminararbeit

Isabel Harms
73313

Hochschule Karlsruhe
University of Applied Sciences

Bounded Generics in Java

21. 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 [9].

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 zunächst versuchen eine generische Typisierung durch Objects umzusetzten und im Vergleich 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 Kompilierung unübersehbar gemacht hätten. Doch wie kann man eine solche Typsicherheit 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 [1]. 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 1), 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. Für bestimmte Problemstellungen ist es notwendig die harte Typisierung von Java zu umgehen und Programme zuschreiben, die allgemeiner Anwendbar sind als es mit einer derart genauen Artangabe möglich wäre.

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

2.1 Generische Programme mit Object

Um dies zu veranschaulichen beschäftigen wir uns nun mit der in Java häufig verwendeten LinkedList. Diese Klasse erbt vom List Interface und verwaltet Objekte in Containern welche immer von einem Element auf das nächste Verweisen. Wie alle Listen, implementiert sie Methoden wie *add*, *remove*, oder *isEmpty*.
In den Containern steckende Objekte müssten nach unseren bisher besprochenen Möglichkeiten einen festen Typ besitzen, welche auch für die Funktionsparameter und Rückgabetypen für die Methoden der Klasse dienen. Betrachten wir eine LinkedList zunächst ausschließlich für die Anwendung auf Integers.

```
1 class intLinkedList {
2     boolean add(int e);
3     int remove(int index);
4     boolean isEmpty();
5 }
```

Hier ist nun auch sichtbar, dass die add Methode nur ints entgegen nimmt und keine anderen Typen in die Liste hinzugefügt werden können. Demnach gibt auch die *remove* Methode ein int zurück. Die *isEmpty* Methode hingegen ist unabhängig von den in er Liste enthaltenen Elementen und prüft lediglich, ob die Liste überhaupt ein Element enthält.

Hiermit haben wir eine Klasse, welche mit ints, aber auch ausschließlich diesen hantieren kann. Diese Funktionalität ist jedoch nicht nur für ints sinnvoll, sondern auch für eine Reihe anderer Datentypen. Wollten wir diese Funktion nun z.B. auch für Strings, so müsste sie komplett neu geschrieben werden, obwohl sich der Inhalt eigentlich komplett überschneidet.

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. Daher ergibt sich der Wunsch, diese Klasse weiter zu öffnen und für alle Möglichen Objekte freizugeben.

Es sollte bekannt sein, dass in Java-Methoden-Parametern auch untergeordnete Klassen, also die Subtypen miteingeschlossen sind. Beispielsweise könnte eine Methode die auf Zahlen agiert einen Parameter des Typs *Number* er-

warten, 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 [5]. Das heißt, dass alle Klassen automatisch von dieser Klasse erben. Würden wir also unsere LinkedList so definieren, dass die Elemente alle vom Typ *Object* sein müssen, so würde dies tatsächlich alle Objekte mit einbeziehen und die Methode wäre maximal verallgemeinert.

```
1 interface List {
2     boolean add(Object e);
3     Object remove(int index);
4     boolean isEmpty();
5 }
```

Hiermit haben wir die Typisierung von Java eigentlich komplett umgangen und haben nun wiederum keinerlei Kenntnisse, was genau sich in einer Liste befinden könnte. Welche Konsequenzen dies haben kann zeigt sich im folgenden.

2.1.1 Typecasting

Wir betrachten hierfür ein Codebeispiel aus den Folien von Dr. Hyunyoung Lee [7]:

```
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.

Welche Folgen hat es nun keine Informationen mehr über unsere Objekte zu haben?
Zwar hat die *Object*-Klasse einige grundlegende Funktionalitäten, aber sobald wir eine Klassen-spezifischere Aktion auf einem Objekt ausführen wollen verlangt der Java-Compiler einen Typecast.

Welche Aktionen könnten dies notwendig machen?

Zuweisung Ein häufig Fall wäre zum Beispiel eine Zuweisung an eine Variable eines festeren Typs. 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 von *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 3.1.2.

Operation Ein weiterer Grund für einen Typcast ist die anwendung bestimmter Operationen, welche nich allgemein auf Objects anwendbar sind. 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 2.

Eigenschaften der Klasse Einer der vermutlich häufigsten Gründe für einen Typecast in diesem Kontext wäre, um auf die Attribute der Klasse

zugreifen, oder ihre Methoden aufrufen zu können. Dies tritt häufiger bei komplexeren Datentypen auf, da die bisher behandelten Typen keine Attribute besitzen.
Dafür erstellen wir die simple Klasse *Contact*

```
1 class Contact {
2     String name;
3     String phoneNumber;
4     public Contact(String name, String phoneNumber) {
5         this.name = name;
6         this.phoneNumber = phoneNumber;
7     }
8 }
```

Sie stellt einen Eintrag in einem Telefonbuch dar und besitzt entsprechende Variablen. Sie soll nun in einer Liste verwaltet werden. Dafür wird eine LinkedList erstellt und ein Kontakt erzeugt und hinzugefügt.

```
1 List l = new LinkedList();
2 l.add(new Contact("Isabel", "12345"));
```

Angenommen es gäbe eine Methode, mit welcher wir eine Telefonnummer anrufen könnten, welche diese als Parameter erwartet. Von einem aus der Liste entnommenen Kontakt wollen wir nun die Telefonnummer in diese Methode geben. Aber auf dieses Attribut können wir nur mit Hilfe eines Casts zugreifen.

```
1 call(((Contact) l.iterator().next()).phoneNumber);
```

Bei einem alternativen Aufbau der Klasse, mit privaten Variablen und Getter-Methoden würde sich beim Aufruf der Methode das selbe Problem ergeben.

Natürlich gibt es auch noch andere Gründe für Typecasting, zum Beispiel um das Objekt in eine Methode als Parameter mit festerem Typ zu übergeben. 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. Aber warum ist Typecasting hier so problematisch?

2.1.2 Typverletzungen

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. Bei Ausführung kommt es daher zu einer ClassCastException.

Selbige Probleme können auch bei den anderen Typecasts auftreten. In diesen Beispielen ist es zwar für den Programmierer noch überschaubar, bei Projekten größerer Skala kann jedoch schnell der Überblick verloren gehen und Fehler dieser Art könnten vermehrt auftreten.

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.
Es ist also eine Typisierung gefragt, die ohne Einschränkung des Programms verhindert, dass derartige Fehler erst zur Laufzeit auffallen.

3 Generics

Heutzutage gibt es viele Sprachen, die parametrischen Polymorphismus unterstützen, unter anderem C sharp, Go, oder auch, wie in dieser Arbeit ausgeführt, Java ab der Version 1.5.
Warum dies eine wertvolle Ergänzung zu Java darstellt, soll nun erläutert werden. Dafür wird veranschaulicht, wie Generics die Probleme lösen, welche wir in Abschnitt 2.1.1 Typecasting und 2.1.2 Typverletzungen der „generischen“ Programmierung mit der *Object*-Klasse festgestellt haben.

3.1 Vergleich zu Object

In Java gibt es eine Vielzahl von Frameworks, welche generische Parametriesierung verwenden. Insbesondere das *java.util* Paket enthält zahlreiche generische Klassen. Eine davon haben wir im Abschnitt 2 Typisierung bereits angesprochen.

Die LinkedList haben wir ohne die Verwendung von Generics umgesetzt und festgestellt, dass sich dabei einige ausschlaggebende Probleme ergeben, die durch eine Umsetzung mit Generics nun überwunden werden müssen.

3.1.1 Typecasting

Übertragen wir die LinkedList zunächst in ihre eigentliche generische Syntax, behalten jedoch die Grundidee bei, dass der Typ nicht weiter spezifiziert ist. Es wird erneut deutlich, dass es sich um eine Liste jeglicher Objekte handelt.

```
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 Entwicklungsumgebung erkennbar. Der Folgende Versuch würde eine Fehlermeldung erzeugen, wie zu sehen in 3, da versucht wird ein Element des Typs 'char' in eine „LinkedList of Integer“ einzufügen.

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

Zuweisung 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

Operation Greifen wir aber noch ein weiteres Beispiel wieder auf: Die Addition aller Elemente in der Liste.

```
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);
```

Im Fokus steht hier Zeile 6, in welcher die += Operation nun ohne Typecast ausgeführt werden kann.

Es ist anzumerken, dass hier 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.

3.1.2 Typsicherheit

Kommen wir nun zu den problematischen Typverletzungen, welche ohne die Verwendung von parametrischem Polymorphismus aufgetreten sind. Wir erzeugen ebenfalls eine „LinkedList“ of Integer, diesmal mit festem Typparameter und versuchen erneut eine Zuweisung an einen String durchzuführen.

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

Diesmal ist jedoch bereits vor Ausführung des Programmes bekannt, dass der Rückgabetyp der Funtion *next* ein Integer sein wird. Daher wird die Entwicklungsumgebung diese Zeile frühzeitig als fehlerhaft erkennen und eine entsprechende Fehlermeldung anzeigen (4).

Betrachten wir dies noch für ein weiteres Beispiel, das Anwenden eine Operation:

```
1 int x = 0;
2 List<String> l = new LinkedList<>();
3 l.add("1"); l.add("42"); l.add("100");
4 Iterator<String> current = l.iterator();
```

Erneut ist es das Ziel, eine String-List als Int-List zu behandeln und so einen Fehler zu erzeugen. Daher haben wir den Typparameter ausgetauscht. Der add Funktion können nun nur noch Strings übergeben werden, weshalb diese Werte angepasst wurden.

```
1 while (current.hasNext()) {
2     x += current.next();
3 }
4 print(x);
```

Beim Versuch die += Operation auszuführen bemerkt der Compiler, dass dies aufgrund des Typs von x nicht möglich ist (5). Es wird angeboten, den Typ von x stattdessen auf String zu setzten.

```
1 String x = "0";
```

Nun ist diese Operation möglich, wird aber nicht mehr als normale Addition, sondern als String-Konkatenation interpretiert. Die Ausgabe ist wie zu erwarten 0142100;

3.2 Beispiel: Typsicheres Interface

In dieser Sektion wollen wir uns mit dem Comparable Interface von Java beschäftigen. Es handelt sich, wie der Name schon vermuten lässt, um eine Schnittstelle, die die Eigenschaft der Vergleichbarkeit zu anderen Objekten sicher stellt. Sie besitzt einen generischen Typ, bei dem es sich in der Regel um die Klasse handelt, welche das Interface implementiert. Die *compareTo*-Methode verwendet den generischen Typ der Klasse als Typ des Parameters.

```
1 public interface Comparable<T> {
2     public int compareTo(T o);
3 }
```

Das hat zur Folge, dass Instanzen von Klassen, welche dieses Interface implementieren nur vergleichbar sind mit anderen Objekten der selben Klasse sind.

Betrachten wir eine Implementation anhand der in 2.1.1 bereits eingeführten Klasse *Contact*.

```
1 public class Contact implements Comparable<Contact> {
2 }
```

Hier lässt sich feststellen, dass anstatt des generischen Typs T nun die eigene Klasse eingesetzt wurde.
Da sie das Interface implementiert, muss sie auch die besagte Methode implementieren. In diese Version der Funktion können ebenfalls nur noch Kontakte mitgegeben werden.

```
1     @Override
2     public int compareTo(Contact contact) {
3         return this.name.compareTo(contact.name);
4     }
```

Wir wollen anhand der in den Kontakten hinterlegten Namen vergleichen, und greifen nun innerhalb der Funktion auf dieses Attribut zu. Da es sich bei diesen Namen um Strings handelt und diese Klasse ebenfalls das Comparable Interface implementiert, können wir es uns einfach machen und die entsprechende CompareTo Methode der String Klasse verwenden.

3.3 Subtyping

In Verbindung mit der Object Klasse haben wir Subtyping bereits erwähnt. Dort haben wir besprochen, dass es sich bei allen Objekten um Subtypen von Object handelt.
Betrachten wir dieses Beispiel[3], so würde wahrscheinlich intuitiv vermutet werden, dass es sich bei einer „List of String“ auch um einen Subtyp einer List of Object handelt, da dies auch bei den Typparametern String und Object der Fall ist.

```
1 List<String> ls = new LinkedList<String>();
2 List<Object> lo = ls;
```

Dies ist jedoch nicht der Fall. Grund dafür ist, dass der Liste *ls* unter dem Alias *lo* nun auch Objekte jeder Art hinzugefügt werden könnten. Dadurch wäre der Typ nicht mehr gesichert und die durch die Verwendung von Generics erarbeiteten Vorteile wären sofort wieder zu nicht gemacht.

Der Java-Compiler wird eine solche Zuweisung daher nicht zulassen (6).

3.4 Unbounded Wildcards

Generische Typen können nicht nur mit einem bestimmten Referenztyp parametrisiert werden. Ebenfalls seit Version 1.5 kann eine Wildcard, dargestellt durch ein Fragezeichen, eingesetzt werden. Es hat die Bedeutung „Beliebiger Referenztyp“.

Dies wollen wir nun anhand einiger Beispiele veranschaulichen [6].
Man könnte also eine Liste mit Wildcard-Parameter erstellen, welche dann eine Referenz auf konkret parametrisierte LinkedList enthält.

```
1 List<?> l = new LinkedList<Integer>
2 l = new LinkedList<NaturalNumber>;
```

Welchen Nutzen hat dies? Nun, wir haben im vorherigen Kapitel (6) angesprochen, dass subtyping in generischen Klassen nicht so möglich ist, wie es in nicht generischen Klassen ist.
In der folgenden Methode wollen wir die Elemente einer Collection drucken. Diese Methode soll für alle Collections zugänglich sein.

```
1 void printCollection(Collection<Object> c) {
2     for (Object e : c) {
```

```
3           System.out.println(e);
4       }
5   }
```

Es wurde eine Collection of Object als Typparemeter gewählt, im Glauben dadurch wäre die Methode für alle Collections zugänglich. Wir haben allerdings bereits geklär, dass dem nicht der Fall ist.

Mit Wildcard-Parameter sind alle Referenztypen miteingeschlossen. Er erreicht also genau das gewünschte.

```
1  void printCollection(Collection<?> c) {
2    for (Object e : c)
3      System.out.println(e);
4  }
```

Alternativ könnte man diese Methode auch mit generischem Typ umsetzen.

```
1  <T> printCollection(Collection<T> c) {
2    for (Object e : c)
3      System.out.println(e);
4  }
```

In diesem Fall sind die Methoden gleichwertig.

Wildcard-Parameter und generische Parameter sind nicht immer austauschbar und können in spezifischeren Fällen leicht von einander abweichen. Außerdem können Wildcards nur als Referenzparameter eingesetzt werden, wie bei Collection <? >, nicht aber als direkter Typparameter, wie es ein generischer Typ könnte.
Außerdem können auf mit Wildcards parametrisierten Objekten keine Methoden aufgerufen werden, welche als Methodenparameter den Typparameter erwarten.

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

Dies hat den Grund, dass der Compiler keine genauen Informationen über den Typ der enthaltenen Elemente hat. Die Liste hat schließlich laut Definition den Typ „Beliebig“. Die Typsicherheit könnte daher beim Hinzufügen eines Objektes nicht mehr sichergestellt werden.

Für unsere Zwecke wollen wir uns aber auf die bounded Wildcards konzentrieren, welche im weiteren Verlauf dieser Arbeit noch thematisiert werden.

3.5 Verbliebene Fehlermöglichkeiten

Nun haben wir aufgezeigt, dass Generics einen großen Schritt in Richtung unserer gesuchten Vorstellung einer möglichst sicheren und flexiblen Lösung machen. Nicht nur haben wir es geschafft die Gefahr von Laufzeitfehlern zu verringern und unseren Codestyle zu verbessern, sondern haben es auch erreicht objektspezifische Interaktionen mit den Elementen aus generischen Klassen auszuführen, ohne uns dabei auf einen Typecast zu verlassen.

Einige Aspekte haben wir jedoch bisher nicht angesprochen. Zum einen waren bisherige generische Klassen auch immer für alle Objekte zugänglich und sinnvoll. Das ist jedoch nicht immer der Fall. Die folgende Klasse soll eine natürliche Zahl verkörpern [2].

```
1 public class NaturalNumber<T> {
2
3     private T n;
4
5     public NaturalNumber(T n)  { this.n = n; }
6
7     // ...
8 }
```

Neben der Variable für den Wert dieser Zahl, beinhaltet sie lediglich den Konstruktor. Dass es sich bei diesem Wert beispielweise nicht um einen String oder gar ein komplexeres Objekt handeln darf, ist offensichtlich. Etwas in der Art zu erzeugen ist aber ohne Probleme möglich.

```
1 NaturalNumber<String> myNumber = new NaturalNumber<>("test");
```

Auch beim Ausführen dieses Programmstücks kommt es zu keinem Fehler, denn bisher haben wir noch nichts mit dieser Zahl gemacht. Selbst ein float oder ähnliches sind hier fragwürdig zu verwenden, da es sich um eine Ganzzahl handeln sollte.

Nun wollen wir innerhalb dieser Klasse Funktionen hinzufügen, welche ihren eigenen Wert verwenden und davon ausgehen, dass es sich dabei um eine

Ganzzahl handelt. Ein simples Beispiel hierfür wäre eiine boolsche Methode, welche zurückgibt, ob dieser Wert gerade ist oder nicht.

```
1 public boolean isEven() {
2     return (int)n % 2 == 0;
3 }
```

Wie wir sehen ist für diese Funktionalität erneut ein Typecast von Nöten, eine Problematik die wir ja eigentlich dringend umgehen wollten. Auch hier führt er zu ähnlichen Fehlern:

```
1 myNumber.isEven();
```

Rufen wir diese Methode nun auf unserem erzeugten Objekt auf wird wiederum versucht einen String auf int zu casten. Dass dies nicht gut geht sollte mittlerweile klar sein. Die Entwicklungsumgebung weißt vorher nicht daraufhin, da es keine Möglichkeit gibt zu wissen, dass dies zu einem Fehler führen wird. Stattdessen löst diese Zeile eine *ClassCastException* aus.

Damit haben wir nun wiederum die Möglichkeit verloren sicher auf spezifische Methoden zuzugreifen und nehmen unserer Klasse einen Großteil ihrer Funktionalität weg. Innerhalb einer generischen Klasse stoßen wir also nun auf ähnliche Probleme, wie es vorher ohne Generics außerhalb der Klasse der fall war.
Doch wie könnten wir unserer Klasse mitteilen, dass ihr generischer Parameter nun nicht mehr von jedem Typ sein darf?

4 Bounded Generics

4.1 Upper Bound

Standardmäßig begrenzt man mit bounded Generics 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 der generische Typ nicht eingeschränkt. man könnte also sagen, dass bisher die obere Grenze generischer Klassen immer durch die Klasse *Object* dargestellt wurde.

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

Für Object zugängliche Methoden, wie z.B. *toString* waren bekanntlich auch schon vorher verwendbar.
Dieser Typ soll nun genauer eingeschränkt werden, um so zu ermöglichen auch Methoden von bestimmten Unterklassen zu verwenden. Für unser Beispiel bedeutet dies, den Typ auf Integer und seine Unterklassen zu verkleinern.

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

Betrachten wir die *isEven* methode, so fällt auf, dass der Typecast nun wegfällt und durch die *Number* eigene Methode *intValue* ersetzt wurde.

Der Versuch eine Instanz zu erzeugen, die nicht innerhalb dieser Grenze liegt, kann nun erkannt werden.

```
1 NaturalNumber<Double> myNumber = new NaturalNumber<Double>();
```

Der Compiler kann diesen Fehler ganz einfach erkennen, da Double keine Unterklasse von Integer ist (7).

Warum dann nicht den Typ direkt auf Number ausweiten?
Die Grenze der Klasse soll den Typ möglichst genau auf die Typen beschränken, für die sie auch sinnvoll ist. Auch wenn es dabei zu keinem Laufzeitfehler kommen würde, soll es sich bei einer NaturalNumber um eine Ganzzahl handeln. Diese Möglichkeit offen zu lassen würde viel schlimmeres erlauben: Einen logischen Fehler, der aber keinen Error erzeugt und somit unerkannt bleibt.

Um dies an einem Beispiel zu betrachten definieren wir eine Klasse für ein Viereck.

```
1 public abstract class Quadrangle extends Shape {
2     protected int height;
3     protected int width;
4
5     public Quadrangle(int height, int width) {
6         this.height = height;
7         this.width = width;
8     }
9 }
```

Zudem gibt es mehrere Klassen, welche von dieser erben, z.B. *Rectangle* als direkter Nachkommen und *Square* als indirekter.

Die folgende Methode soll den Flächeninhalt eins Vierecks berechnen.

```
1 public <T extends Quadrangle> int area(T quadrangle) {
2     return quadrangle.width * quadrangle.height;
3 }
```

Bounded Generics ermöglichen hier nicht nur die Verwendung der Attribute *height* und *width* der Viereck-Klasse. Ebenfalls zu beachten ist, dass es auch zahlreiche andere Formen geben kann, welche diese Attribute besitzen. Trotzdem dürfen diese nicht für diese Methode verwendet werden, da deren Flächeninhalt vermutlich nicht mit der selben Formel berechnet wird und daher einen fehlerhaften Wert liefern würde.

4.1.1 Mehrere Bounds

Es ist auch möglich mehrere obere Grenzen festzulegen. Eine sinnvolle Verwendung hierfür ist die folgende Methode von tutorialspoint [8]

```
<T extends Number & Comparable<T>> T max(T x, T y, T z) {
        T max = x;
        if (y.compareTo(max) > 0) {
            max = y;
        }

        if (z.compareTo(max) > 0) {
            max = z;
        }
        return max;
    }
}
```

Auch hier kommt das *Comparable*-Interface erneut zum Einsatz. Der Parameter muss nun beide angegebenen Eigenschaften besitzen, also eine Unterklasse von Number sein und die Schnittstelle Comparable implementieren.

Es ist anzumerken, dass im oben eingetroffenen Fall mehrerer Grenzen mit einer Klasse und einem Interface immer zunächst die Klasse, und dann die Interfaces aufgelistet werden müssen, da sonst eine Fehlermeldung angezeigt wird (8).

4.2 Bounded Wildcards

4.2.1 Upper Bound

Auch Wildcards können eine obere Grenze haben. Wie beim unbouded Wildcard-Typ auch, gibt es auch für den upper bounded Typ oftmals eine äquivalente generische Methode. Vergleichen wir diese beiden Methoden [6]:

```
1 <T extends Shape> void drawAll(Collection<T> shapes) {
2   for (T s : shapes)
3     s.draw();
4 }
```

```
1 void drawAll(Collection<? extends Shape> shapes) {
2   for (Shape s : shapes)
3     s.draw();
4 }
```

Sie weisen die selbe Funktionalität auf. Erwartet wird eine Collection, dessen Elemente Subtypen von Shape sind. Diese werden dann nacheinander über den Aufruf der *draw*-Methode gezeichnet.
Upperbounded Wildcards haben also ähnliche Funktionalität wie bounded Generics, allerdings können Sie nicht mehrere Grenzen verwenden.

4.2.2 Lower Bound

Im Gegensatz zu generischen Typen, können Wildcards aber auch eine untere Grenze haben. Die Verwendung dieser lower Bounds ist nicht ganz so naheliegend, wie es bei den upper Bounds der Fall war.
Betrachten wir die folgende Methode [6]. Ebenso wie die Methode aus 4.1.1 berechnet sie das Maximum. Anstatt eines dritten Werts bekommt sie als Parameter allerdings einen Comparator, nachdem die Werte verglichen werden. Dieser hat einen generischen Parameter und kann Werte dieses Typs und seiner Subtypen vergleichen.

```
1 T max(T t1, T t2, Comparator<T> cmp) {
2   if ( cmp.compare(t1,t2) < 0 )
3     return t2;
4   else
5     return t1;
6 }
```

Es gibt jedoch Comparatoren, die nicht von einem bestimmten Typ sind und diesen dennoch vergleichen können. Nämlich die Comparatoren, die als Typ eine der Elternklassen haben. Allerdings macht es auch keinen Sinn der Methode einen Comparator mitzugeben, der den Typ T dann nicht vergleichen kann. Deshalb müssen wir den Typ auf die Superklassen des Typs einschränken. Mit Hilfe von Wildcards geht dies ganz einfach:

```
1 T max(T t1, T t2, Comparator<? super T> cmp) {
2   if ( cmp.compare(t1,t2) < 0 )
3     return t2;
4   else
5     return t1;
6 }
```

Wären die beiden Variable t1 und t2 nun vom Typ Integer, so könnten auch ein Comparator of Number oder Object mitgeben werden, aber kein Comparator of String.
Es ist zu beachten, dass Wildcards nicht gleichzeitig eine untere und obere Grenze haben können.

4.3 Beispiel: Bounded Generics

Zum Abschluss wollen wir die in dieser Arbeit behandelten Themen in einem Beispiel zusammenführen.

Im Abschnitt 3.2 Typsicheres Interface haben wir uns bereits mit dem Interface Comparable und möglichen Implementierung dessen beschäftigt. Da Implementierungen dieser Klasse alle die Eigenschaft der Vergleichbarkeit aufweisen, sind sie auch automatisch sortierbar.
Unser Ziel ist es nun, eine Klasse zu entwickeln, welche Objekte, die das Interface Comparable implementieren, sortieren kann. Da diese Objekte nur vergleichbar mit anderen Objekten derselben Klasse sind, eignet es sich diese Sorter Klasse generisch festzulegen. In diesem Beispiel [4] betrachten wir einen mergeSort Algorithmus.

Die Klasse *MergeSortGeneric* besitzt einen generischen Parameter, welcher das Comparable Interface implementieren muss. Da dieses Interface wiederum generisch ist können wir nun den Referenzparameter mit Hilfe einer lower bounded Wildcard einschränken. Dies ist sinnvoll, da das Comparable ähnlich

wie im Beispiel aus Sektion nicht ausschließlich vom Typ T sein darf, sondern auch von seinen Superklassen.

```
class MergeSortGeneric<T extends Comparable<? super T>> {
}
```

Diese Klasse soll eine Methode bekommen welche ein festgelegtes Interval eines Arrays sortieren kann. Der Typ des Arrays entspricht dem generischen Typ der Klasse.

```
void mergeSort(T[] array, int start, int end) {
    //...
}
```

Diese Methode soll das Array standardmäßig in 2 Hälften teilen, diese sortieren und am Schluss wieder zusammenführen. Die Sortierung erfolgt rekursiv über erneutes Aufrufen der Methode, bis das Abbruchkriterium eintritt: Der Abschnitt enthält nur noch ein einziges Element. Dies tritt ein sobald der Start des Intervals dem Ende entspricht.

```
    if (start < end)
    {
        // find the middle point
        int middle = (start + end) / 2;

        mergeSort(array, start, middle);
        mergeSort(array, middle + 1, end);

        // merge the sorted halves
        merge(array, start, middle, end);
    }
```

Zusammengeführt wird in einer weiteren Methode namens *merge*. Auch sie muss als Array-Typ den generischen Typ der Klasse besitzen. Da sie nicht relevant für die generischen Parameter ist wird sie hier jedoch nicht weiter erläutert.

Dennoch lässt sich feststellen, dass bounded Generics in Verbindung mit dem bounded Wildcard-Parameter ein mächtiges Mittel darstellen um komplexe Problemstellungen umzusetzten.

5 Fazit

Abschließend sollen die neu erlernten Techniken zusammengefasst werden.
Es wurde festgestellt, dass flexible Typisierung notwendig ist um anpassungsfähige Programme zu entwickeln, welche dem Programmierer nicht zu viel Freiraum nehmen und Redundanz möglichst vermeiden.

Zudem lässt sich sagen, dass der parametrische Polymorphismus eine sinnvolle Ergänzung zu Java darstellt, da er es ermöglicht Typecasting zu umgehen und einen hohen Grad an Typsicherheit zu erlangen. Diese war durch die Alternative, die Verwendung von Objekten der Klasse *Object*, nicht gewährleistet. Denn diese Methode hat entweder eine starke Einschränkung der Verwendungsmöglichkeiten zur Folge, da nun alle Objekte komplett verallgemeinert werden, oder riskiert eine ClassCastException beim Versuch den Typ genauer einzugrenzen.
Dennoch können herkömmliche Generics allein keine hinreichende Sicherheit gegenüber Typverletzungen gewährleisten, da der Typparameter nicht genauer eingrenzbar ist und so jegliche Verwendungen dessen innerhalb der generischen Klasse allgemein bleiben müssen oder ebenfalls eine Exception riskieren.

Die Lösung für diese Problematik ist der f-bounded Polymorphismus. Er ermöglicht sehr genaue Einschränkungen des Typparameters und stellt so sicher, dass generische Klassen und Methoden nur mit den Typen initialisiert werden können auf die sie auch anwendbar sind.
Mit Hilfe von Wildcards ist sogar eine untere Grenze möglich, die uns in Spezialfällen zusätzliche Einschränkungen ermöglicht.

Zusammengefasst wurde festgestellt, dass Java durch bounded Generics einen hohen Grad an Flexibilität und Typsicherheit gewährleistet, der ohne deren Verwendung nicht zu ersetzen wäre. Daher handelt es sich um eine notwendige und nützliche Bereicherung für diese Programmiersprache.

Literatur

[1] Academic. *Statische Typisierung*. URL: https://de-academic.com/dic.nsf/dewiki/1325039 (besucht am 20. 11. 2022).

[2] Oracle and/or its affiliates. *Bounded Type Parameters*. URL: https://docs.oracle.com/javase/tutorial/java/generics/bounded.html (besucht am 19. 11. 2022).

[3] Oracle and/or its affiliates. *Generics and Subtyping*. URL: https://docs.oracle.com/javase/tutorial/extra/generics/subtype.html (besucht am 19. 11. 2022).

[4] Big-O. *Generic Merge Sort in Java*. URL: https://big-o.io/examples/merge-sort/java-generic/ (besucht am 19. 11. 2022).

[5] JavaTpoint. *Object class in Java*. URL: https://www.javatpoint.com/object-class (besucht am 19. 11. 2022).

[6] Angelika Langer. *Java Generics - Wildcards*. URL: http://www.angelikalanger.com/Articles/EffectiveJava/31.Wildcards/31.Wildcards.html (besucht am 21. 11. 2022).

[7] 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).

[8] tutorialspoint. *Java Generics - Multiple Bounds*. URL: https://www.tutorialspoint.com/java_generics/java_generics_multiple_bounds.htm (besucht am 20. 11. 2022).

[9] 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).

A Anhang

A.1 Bilder

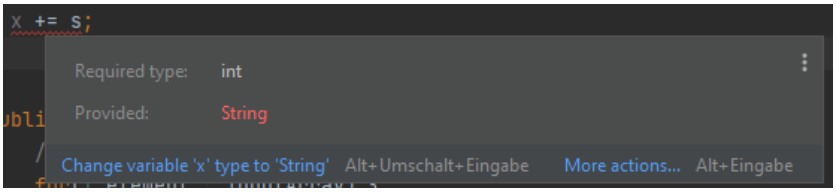

Abbildung 1:

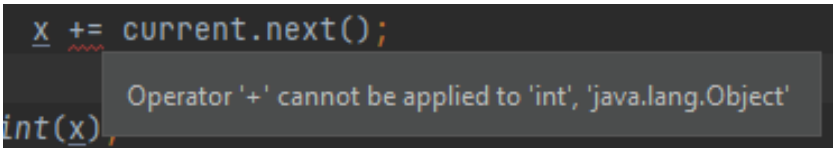

Abbildung 2:

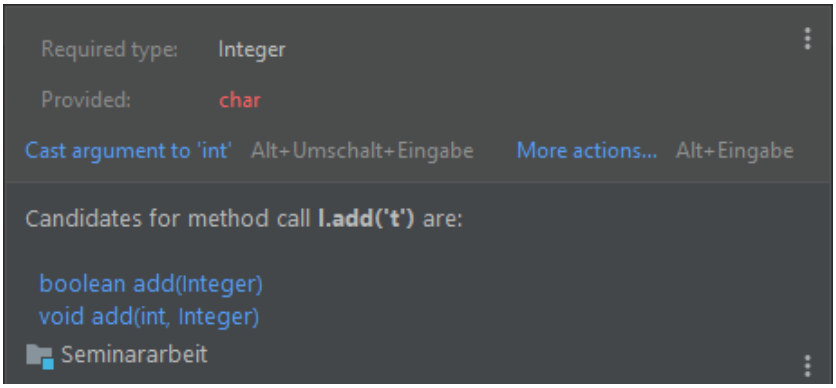

Abbildung 3:

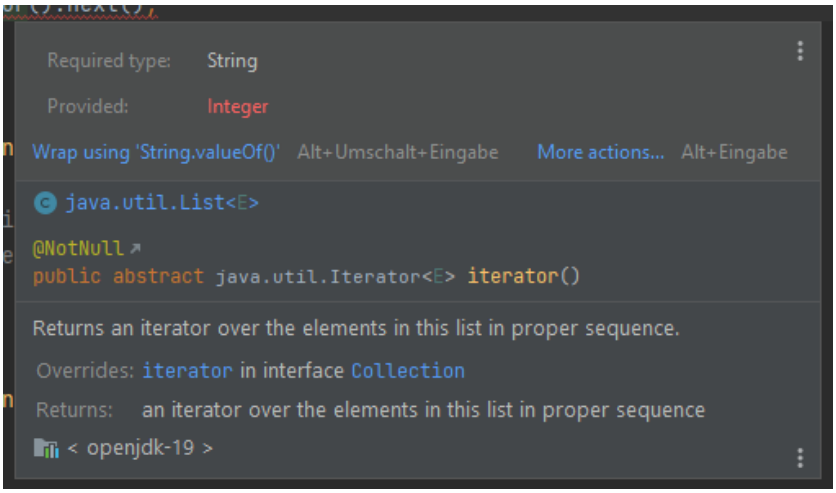

Abbildung 4: Der Versuch, den Rückgabewert der Funktion next (Integer) an einen String zuzuweisen

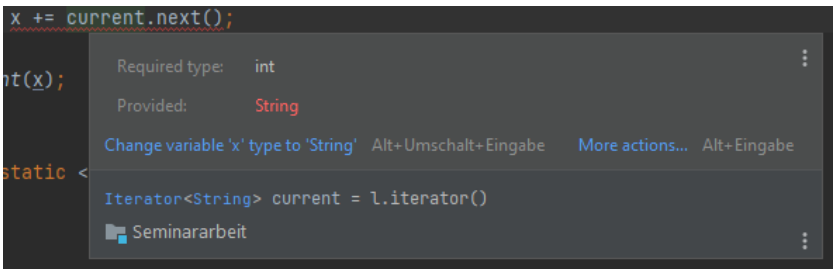

Abbildung 5: Integer += String

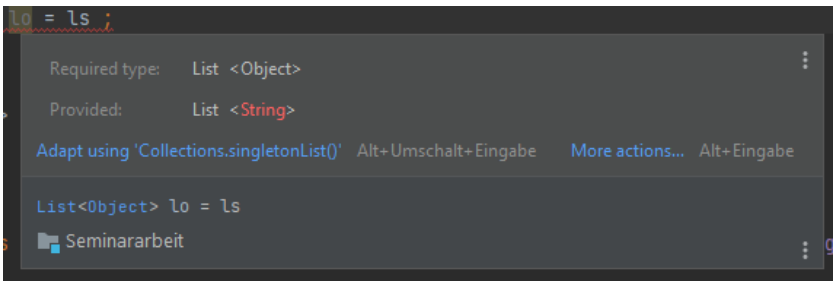

Abbildung 6: List¡String¿ ist kein Subtype von List¡Object¿

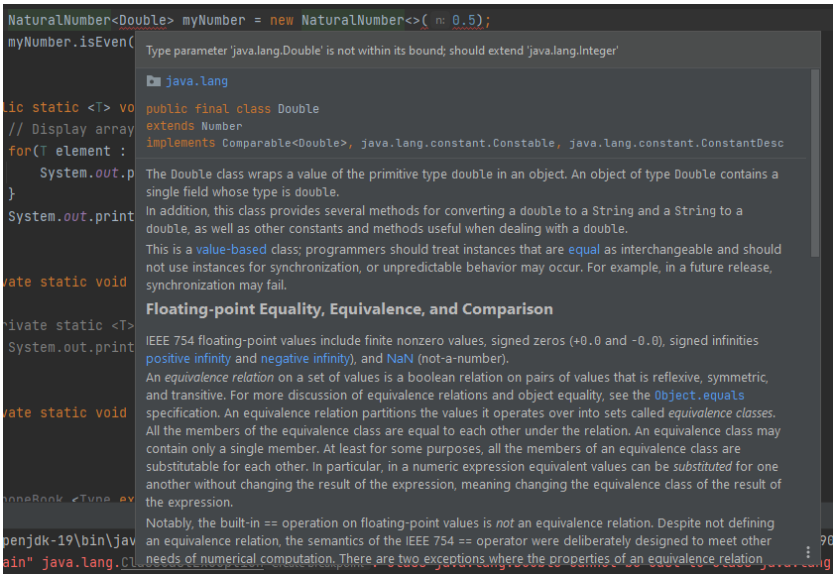

Abbildung 7: Versuch NaturalNumber mit Double zu parametrisieren

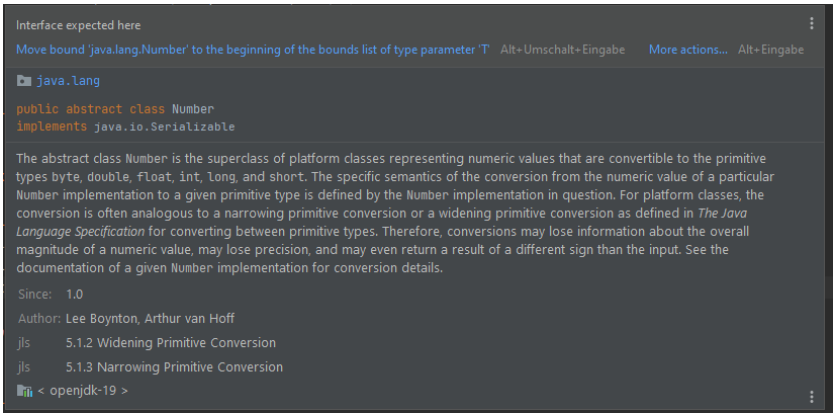

Abbildung 8: Obere Grenze mit Interface vor Klasse