title: "Generics" permalink: /04-generics/ mathjax: true

Generics

In den vergangenen zwei Kapiteln haben wir die zwei Datenstrukturen Liste und Set (realisiert als Binärbaum) kennen gelernt. Dabei hatten wir die Schnittstellen so gewählt, dass sie auf konkrete Datentypen festgelegt waren:

```
interface IntList {
        int get(int i);
        void add(int v);
        int remove(int i);
        // etc.
}
```

```
interface CharSet {
        boolean add(char c);
        boolean contains(char c);
        char remove(char c);
        // etc.
}
```

Da es nun eine ganze Reihe von Datentypen gibt, und insbesondere eigene Klassen ja *ebenfalls* Datentypen sind, drängt sich die Frage auf: Kann man das nicht allgemeiner implementieren, so dass man nicht für jeden Datentyp eine spezielle Datenstruktur implementiert werden muss?

Bleiben wir zunächst bei der Liste. Offensichtlich ist die *Struktur* einer Liste unabhängig davon, welche konkreten Elemente darin abgespeichert werden. Wir erinnern uns: Bei der `IntList` hatten wir ein Containerelement verwendet, welches genau einen `int` Wert gespeichert hatte:

```
class IntListImpl implements IntList {
        class Element {
                int value;
                Element next;
        }
        // ...
}
```

Nun ist es in Java so, dass alle Objekte -- egal welcher Klasse -- immer auch `java.lang.Object` sind, die gemeinsamen *Basisklasse*. Das heisst, wenn wir nun anstatt des konkreten `int` Typs überall `Object` verwenden, so sollte die Liste für alle Datentypen funktionieren.

> Hinweis: Basisklassen sowie das dazugehörige Konzept der Vererbung behandeln wir gegen Ende des Semesters. Für dieses Kapitel soll genügen, dass jedes Objekt, egal welcher Klasse, auch ein `Object` ist.

```
interface List {
        void add(Object o);
        Object get(int i);
        // etc.
}
```

```
class ListImpl implements List {
        class Element {
                Object value;
                Element next;
                Element(Object o, Element e) {
                        value = o;
                        next = e;
                }
        }

        Element head;

        public void add(Object o) {
                if (head == null) {
                        head = new Element(o, null);
                        return;
                }

                Element it = head;
                while (it.next != null)
                        it = it.next;

                it.next = new Element(o, null));
        }

        public Object get(int i) {
                if (head == null)
                        throw new NoSuchElementException();

                Element it = head;
                while (i-- > 0)
                        it = it.next;

                return it.value;
        }

        // usw., analog zur IntListImpl
}
```

Diese Implementierung kann man nun relativ unkompliziert einsetzen:

```
List li = new ListImpl();

li.add("Hallo");  // OK: jeder String ist auch ein Object
li.add("Welt");

for (int i = 0; i < li.size(); i++)
        System.out.println(li.get(i));  // OK: jedes Object kann .toString()
```

Aber Vorsicht, folgender Ausdruck ist keine korrekte Syntax:

```
String s = li.get(0);  // Compilerfehler: Object is not String
```

Nun sieht das geschulte Auge aber sofort: Zwar wird bei `add` der `String` automatisch in den allgemeineren Datentyp `Object` umgewandelt (Stichwort *automatische Typumwandlung*), jedoch ist der Rückgabetyp von `get` ein `Object`. Will man also das Objekt wie vorher verwenden, so muss man eine *erzwungene Typumwandlung* vornehmen:

```
List li = new ListImpl();
li.add("Hallo, Welt!");
String hw = (String) li.get(0);  // OK: erzwungene Typumwandlung
```

Exkurs: Autoboxing und Unboxing von primitiven Datentypen

Wie Sie bereits aus dem letzten Semester wissen, nimmt der Datentyp `String` eine besondere Rolle ein. Zwar werden für diesen zum Teil Operatoren (z.B. `+`) unterstützt, tatsächlich aber sind Zeichenketten Objekte der *Klasse* `String`. Folglich ist auch jeder `String` ein `Object`.

Wie verhält es sich nun mit den primitiven Datentypen, sind `int` und `float` auch `Object`? Hier kommt das Konzept Autoboxing bzw. Unboxing zu tragen: Werte in Variablen eines primitiven Datentyps werden in Instanzen der zugehörigen *Wrapperklasse* gepackt (engl.: *boxing*), bzw. aus dieser wieder ausgepackt.

```
int i = 5;          // wie gehabt.

// in beide Richtungen kein expliziter Typecast erforderlich!
Integer boxed1 = 4;  // boxing von int-Literal
Integer boxed2 = i;  // boxing von int-Variable
i = boxed1;          // unboxing
```

Die Wrapperklassen heissen wie der primitive Datentyp, nur eben als Klassenname: `Integer` für `int`, `Float` für `float`, `Character` für `char`, usw.

Wir können daher unsere obige Liste auch für primitive Datentypen verwenden:

```
List li = new ListImpl();

// autoboxing nach Double
li.add(3.1415);

// Typecast Object -> Double, dann unboxing
double d = (Double) li.get(0);
```

Im obigen Beispiel ist der Typecast daher nötig, da wir ja den Rückgabewert der `get` Methode von `Object` nach `Double` unwandeln ("casten") müssen, bevor es "ausgepackt" werden kann.

Typ(un)sicherheit

Die allgemeine Liste, welche `Object` verwaltet ist Fluch und Segen in einem. Zwar kann sie für beliebige Datentypen verwendet werden, das kann jedoch Schwierigkeiten mit sich bringen. Ein Beispiel:

```
List li = new ListImpl();

li.add(1);
li.add(2);
li.add("Hans");

for (int i = 0; i < li.length(); i++) {
        int val = (Integer) li.get(i);
        System.out.println(val);
}
```

Führt man obigen Code aus, so erhält man folgende Ausnahme:

```
java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
```

Da ja das dritte Element in der Liste eben nicht vom Typ `Integer` (bzw. `int`) war, sondern eben `String`. Wie kann man nun erzwingen, dass eine Liste immer nur Elemente von einem Typ enthält?

Eine (schlechte) Methode wäre, beim Einfügen sicherzustellen, dass das neue Element den gleichen Typ wie das `head` Element hat, und entsprechend eine Ausnahme zu werfen:

```
if (v.getClass() != head.value.getClass())
        throw new IllegalArgumentException();
```

Damit ist das Problem aber nur *verschoben*: Im obigen Beispil erhalten wir statt einer `ClassCastException` nun eine `IllegalArgumentException` -- ebenfalls zur Laufzeit!

Wünschenswert wäre aber eine Möglichkeit, den in der Liste enthaltenen Datentyp *zur Compilezeit* festzusetzen, und so mögliche Fehler in der Benutzung zu unterbinden.

Generics

Seit Version 5 (2004) gibt es in Java dazu die sogenannten *Generics*: Ein syntaktisches Mittel, um *generische* Datentypen zu implementieren, welche dann mit *konkreten* Typen verwendet werden können. Die Syntax lautet wie folgt:

```
interface List<T> {
        void add(T o);
        T get(int i);
}
```

Dabei wird dem Klassen- bzw. Schnittstellennamen eine (oder mehrere, durch Komma getrennte) Typvariable(n) in spitzen Klammern nachgestell (hier: `<T>`). Diese Typvariable kann dann *innerhalb des Blockes* wie ein Datentyp verwendet werden (z.B.: `T get(int i)`). Der Name der Typvariable kann dabei frei gewählt werden, es hat sich allerdings durchgesetzt eine einzelne Typvariable `T` (wie "Typ") zu nennen, nächste Woche werden wir noch `K` und `V` für *Key* und *Value* kennen lernen.

Wir können Generics auch auf die `implements` Beziehung anwenden:

```
class ListImpl<T> implements List<T> {
        // ...
}
```

Hierbei ist allerdings zu beachten, dass das *erste* `T` (hinter `ListImpl`) die Typvariable ist, bei `List<T>` ist `T` hingegen das *Typargument*. Wir hätten also auch schreiben können:

```
class ListImpl<Z> implements List<Z> {
        // ...
}
```

Zu lesen als: die in `Z` generische Klasse `ListImpl` implementiert die generische Klasse `List<T>`, wobei `T := Z`. Folglich könnte auch eine *normale* Klasse ein *parametrisiertes* Interface implementieren:

```
class IntListImpl implements List<Integer> { ... }
```

Zu lesen als: Die Klasse `IntListImpl` implementiert die generische Klasse `List<T>`, wobei `T := Integer`.

Implementierung von Generics

Um nun eine Klasse (oder Interface) generisch zu machen, tauschen wir an sämtlichen Stellen wo wir vorher einen konkreten Typ verwendet hatten, diesen durch die Typvariable:

```
class ListImpl<T> implements List<T> {
        private class Element {
                T value;  // Datenelement, vorher Object
                Element next;
                Element(T o, Element e) {  // !
                        value = o;
                        next = e;
                }
        }

        private Element head;

        public T get(int i) {  // !
                if (head == null)
                        throw new NoSuchElementException();

                Element it = head;
                while (i-- > 0)
                        it = it.next;
                return it.value;
        }

        public void add(T v) {  // !
                if (head == null) {
                        head = new Element(v, null);
                        return;
                }

                Element it = head;
                while (it.next != null)
                        it = it.next;
                it.next = new Element(v, null);
        }
        // etc.
}
```

Wenn wir diese generische Klasse nun verwenden wollen, so müssen wir für die Typvariablen entsprechend konkrete Typen angeben, und damit die Klasse *parametrisieren*. Als "Gegenleistung" dafür garantiert uns der Compiler Typsicherheit, was sich darin niederschlägt, dass nun zum einen kein Typcast mehr erforderlich ist, und zum anderen die Verwendung mit falschen Datentypen zur Compilezeit erkannt werden kann:

```
List<Integer> li = new ListImpl<Integer>();
li.add(1);
int a = li.get(0);  // kein Cast mehr notwendig

li.add("Hans");  // Compilerfehler!
```

Raw-Type

Interessanter weise ist folgendes korrekte Syntax, auch wenn der Compiler eine Warnung dazu gibt:

```
List li = new ListImpl();  // Compilerwarnung!
// ...
```

Wird eine generische Klasse *ohne* Typargument verwendet, so spricht man vom "raw type", welcher dann im Endeffekt die selbe Funktionsweise hat, wie unsere originale Implementierung mit `Object`, und das wiederum ist in etwa `List<Object>`. Der raw type ist nötig, damit die neue JVM (5 und neuer) abwärtskompatibel zu den älteren Versionen ist, in welchen Generics noch nicht verfügbar waren.

Es wird aber wärmstens empfohlen, Datenstrukturen und Algorithmen wo immer möglich generisch zu implementieren.

Einschränkungen für Typparameter

Letzte Woche hatten wir als weitere grundlegende Datenstruktur das Set (realisiert durch einen Baum) kennen gelernt. Dieses können wir nun ebenso generisch formulieren:

```
interface Set<T> {
        void add(T c);
        boolean contains(T c);
        int size();
}
```

Zur Implementierung ändern wir analog zur Liste unsere Containerklasse:

```
class SetImpl<T> implements Set<T> {
        class Element {
                T value;
                Element left, right;
                // ...
        }
}
```

Bei der `contains` Methode allerdings stoßen wir auf Widerstand, man beachte die kommentierten Stellen 1 und 2.

```
public boolean contains(T t) {
        if (root == null)
                return false;

        Element it = root;
```

```
        while (it != null) {
                // 1 (Semantikfehler)
                if (t == it.value)
                        return true;
                // 2 (Compilerfehler)
                else if (t < it.value) {
                        it = it.left;
                } else {
                        it = it.right;
                }
        }

        // nicht gefunden!
        return false;
}
```

Den Semantikfehler an Stelle 1, den Test auf Gleichheit (nicht Identität `==`), können wir mit `equals` realisieren, welche jedes Objekt hat.

Der Compilerfehler an Stelle 2, der Vergleich der Objekte, ob links oder rechts abgestiegen werden soll, ist in dieser Form aber syntaktisch nicht möglich.

Objektgleichheit mit equals

In Java können zwei Objekte mit `equals` auf *inhaltliche Gleichheit* verglichen werden:

```
String s1 = "Hans";
String s2 = "Dampf";

System.out.println(s1.equals(s2));  // "false"
```

Verwendet man `equals` um Objekte selbst entwickelter Klassen zu vergleichen, so muss die Methode wie bei Interfaces *überschrieben* werden. Dies geschieht immer nach dem selben Schema:

```
class MeineKlasse {
        int attribut;
        public boolean equals(Object o) {
                // 1. Das_selbe_ Objekt?
                if (o == this)
                        return true;

                // 2. Wenn die Klasse nicht passt...
                if (!(o instanceof MeineKlasse))
                        return false;

                // 3. erzwungene Typumwandlung
                MeineKlasse other = (MeineKlasse) o;

                // 4. Attribute vergleichen
```

```
                if (this.attribut != other.attribut)
                        return false;

                return true;
        }
}
```

Objektvergleich mit compareTo

In der Übung 2 haben Sie bereits das `StringSet` implementiert -- und dabei festgestellt, dass die Klasse `String` eine Methode `int compareTo(String other)` bereitstellt. Ein Blick in die Dokumentation zeigt gleich oben, unter "All Implemented Interfaces": `Comparable<String>`.

In der Tat handelt es sich bei `Comparable` um ein generisches Interface (Dokumentation) welches recht einfach gestrickt ist:

```
public interface Comparable<T> {
        /**
         * @return 0 bei Gleichheit, negativ wenn `o` groesser, positiv wenn `o`
kleier ist.
         */
        int compareTo(T o);
}
```

Da die Klasse `String` das *parametrisierte* Interface `Comparable<String>` implementiert, verfügt sie über eine entsprechende Methode `compareTo(String o)`, welche zum Vergleich verwendet werden kann. Diese soll `0` zurückgeben, wenn beide Objekte (inhalts-)gleich sind, einen negativen Wert wenn `o` groesser bzw. einen positiven Wert, wenn `o` kleiner ist.

Bounds

Übertragen auf unser generisches Set heisst das doch: Das Set soll zwar generisch sein, aber bitte nur für solche Datentypen `T`, welche das Interface `Comparable<T>` (also den Vergleich mit Elementen gleichen Typs) implementieren.

Um das auszudrücken gibt es in den Java Generics sogennante *Bounds*, also Rahmenbedingungen, welche die Typparameter erfüllen müssen. Bounds werden *innerhalb* der spitzen Klammern, und mit dem Schlüsselwort `extends` spezifiziert:

```
interface Set<T extends Comparable<T>> {
        void add(T c);
        boolean contains(T c);
        int size();
}
```

```
class SetImpl<T extends Comparable<T>> implements Set<T> {
        // ...
}
```

Das Interface `Set` soll also generisch in `T` sein, aber nur für solche `T`, welche das Interface `Comparable<T>` implementieren. Sollten mehrere solcher Einschränkungen nötig sein, so können Sie diese mit `&` (kein Komma!) aneinanderreihen, z.B. `<T extends Comparable<T> & Serializable>`.

Die Spezifikation der Bounds erlaubt uns nun bei Variablen vom Typparameter die Methoden des Interfaces aufzurufen:

```
// in der SetImpl.contains(T t) Methode
while (it != null) {
        // erlaubt, da T extends Comparable<T>
        int c = t.compareTo(it.value);

        if (c == 0)  // oder t.equals(it.value)
                return true;
        else if (c < 0) {
                it = it.left;
        } else {
                it = it.right;
        }
}
```

Generische Methoden

Wir haben nun gesehen, wie Schnittstellen und Klassen generisch definiert und implementiert werden können. In Java kann man allerdings auch *Methoden* generisch implementieren. Hier am Beispiel von Sortieren durch Auswählen:

```
class Sortieren {
        public static <T> void sort(T[] a) {
                // ...
        }
}
```

Dabei dabei wird die Typvariable *nach* den Sichtbarkeiten aber *vor* dem Rückgabedatentyp eingefügt. Wie vergleicht man nun einzelne Objekte? Eine Möglichekeit sind wieder Bounds:

```
class Sortieren {
        public static <T extends Comparable<T>> void sort(T[] a) {
                for (int i = 0; i < a.length; i++) {
                        for (int j = i + 1; j < a.length; j++) {
                                if (a[j].compareTo(a[i]) < 0) {
```

```
                                        T h = a[i];
                                        a[i] = a[j];
                                        a[j] = h;
                                }
                        }
                }
        }
}
```

Das hat nun aber den Nachteil, dass alle Objekte die Sie vergleichen möchten das Interface `Comparable` implementieren. Eine andere Möglichkeit ist es, der `sort` Methode einen `Comparator<T>` mitzugeben. Dieses Interface ist ebenfalls generisch, und bietet ist vereinfacht so definiert:

```
interface Comparator<T> {
        /**
         * @return 0 bei Gleichheit, negativ wenn `a` kleiner `b`, positiv wenn
`a` groesser `b`
         */
        int compare(T a, T b);
}
```

Ein `Comparator<T>` kann also Objekte vom Typ `T` vergleichen. Damit können wir die `sort` Methode generisch, aber *ohne* Bounds realisieren:

```
class Sortieren {
        public static <T> void sort(T[] a, Comparator<T> comp) {
                for (int i = 0; i < a.length; i++) {
                        for (int j = i + 1; j < a.length; j++) {
                                int c = comp.compare(a[j], a[i]);
                                if (c < 0) {
                                        T h = a[i];
                                        a[i] = a[j];
                                        a[j] = h;
                                }
                        }
                }
        }
}
```

Ein `Comparator` wird übrigens sehr oft als *anonyme Klasse* realisiert:

```
// Syntax: new Interface () { /* implementierung */ }
Comparator<Integer> c = new Comparator<Integer>() {
        public int compare(Integer o1, Integer o2) {
                return Integer.compare(o1, o2);
        }
};
```

```
Sortieren.sort(a, c);
```

Gültigkeit von Typvariablen

Wir haben Typvariablen für Attribute in Klassen bzw. für innere Klassen verwendet. Zur Gültigkeit ist zu beachten:

- Typvariablen von nicht-statischen Klassen können nur in *Instanzvariablen* und *-methoden* verwendet werden.
- Typvariablen von statischen Klassen können nur für statische Methoden verwendet werden.

```
// Klasse Aeussere generisch in T
class Aeussere<T> {
        T variable;  // ok

        // nicht moeglich:
        // static T gehtnicht;

        // innere Klasse; gibts nur "in" Instanz von `Aeussere`
        class Innere {
                T hilfe;  // ok

                // ebenso nicht moeglich:
                // static T gehtnicht;
        }

        // statische Methode; ohne Instanz von Aeussere benutzbar
        static <E> void someStaticMethod(E arg) {
                // statische Methode, generisch in E

                // nicht moeglich:
                // T gehtnicht;
        }

        // statische innere Klasse; ebenso ohne Inst. von Auessere benutzbar
        static class StatischeInnere<Z> {
                Z variable;  // ok

                // geht nicht:
                // T gehtnicht;

                // geht auch nicht, da E nur bei someStaticMethod
                // E gehtauchnicht;
        }
}
```

Zusammenfassung

- **Generics** sind ein elegantes syntaktisches Mittel, um Datenstrukturen und Algorithmen für beliebige (aber feste) Datentypen zu implementieren.
- Generics geben hierbei **Typsicherheit zur Compilezeit:** etwaige Typfehler werden vom Compiler erkannt, wodurch Laufzeitfehler vermieden werden können.
- Bei **generischen Klassen** wird die **Typvariable** *nach dem Namen* platziert, bei **generischen Methoden** allerdings *vor dem Rückgabedatentyp*.
- **Bounds** können verwendet werden um Anforderungen an die **Typargumente** zu definieren.
- Das Interface `Comparable` erzwingt die `compareTo` Methode, um ein Objekt mit einem anderen (typischerweise des selben Typs) zu vergleichen
- Ein `Comparator` ist ein Objekt, welches zwei Objekte gleichen aber festen Typs miteinander vergleicht.
- `Comparable` kann ein nützlicher Bound sein, wenn Elemente verglichen werden sollen; will oder kann man die Typvariable nicht einschränken, so kann alternativ ein entsprechender `Comparator` verwendet werden.
- Typvariablen von *nicht-statischen Klassen* können nur im *nicht-statischen Kontext* verwendet werden; sowohl statische als auch nicht-statische Methoden oder innere Klassen können generisch sein.

Begrifflichkeit im Überblick

- `interface List<T>` ist ein *generischer Typ*, mit *Typvariable* (oder *Typparameter*) `T`.
- `List<String>` ist das *parametrisierte* Interface, `String` das *Typargument*
- Ist das Interface bzw. die Klasse zwar generisch implementiert, verwendet man den Datentyp aber *ohne Typargument*, so spricht man vom *raw type*.

Nur Für Interessierte

Wir haben hier nur einen kleinen Teil von Generics besprochen, welcher aber bereits für sehr viele Anwendungen ausreicht. Insbesondere mit Vererbung können Generics allerdings noch deutlich komplizierter werden, und je nach Anwendung sogenannte Wildcards (`?`) und *upper* (`extends`) bzw. *lower* (`super`) Bounds erfordern. Dieses geht aber weit über diese Veranstaltung hinaus, darum darf ich Sie für genauere Informationen auf die offizielle Dokumentation verweisen.

■