Technische Hochschule Rosenheim
Technical University of Applied Sciences

Objektorientierte Programmierung
Kapitel 3 – Generics

Prof. Dr. Kai Höfig

Motivation

Technische Hochschule Rosenheim
Technical University of Applied Sciences

*"**Reusability** is one of the great promises of **object-oriented** technology. Unfortunately, it's a promise that often goes unrealized. The problem is that reuse isn't free; it isn't something you get simply because you're using object-oriented development tools. Instead, it's something **you must work hard** at if you want to be successful."*
(Scott Ambler)

- **Generischer, wiederverwertbarer Code** durch *Schnittstellen, Spezialisierung, Polymorphie*, etc.
  - Konflikt: Flexibilität vs. Typsicherheit.
  - Typüberprüfung teils erst zur Laufzeit.
- Welche generischen Datenstrukturen und Algorithmen bietet die **Java Standard Library** bereits?

Ohne Generics ("Raw Types")

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Ziel:**
  - Generische Klasse `Bag`, die beliebiges Objekt als Inhalt aufnehmen kann.
- **Versuch:**

```
public class Bag {
 private Object content;
 public Bag(Object content) {
      this.content = content;}
 public Object getContent() {
      return content; }
 public void setContent(Object c){
      this.content = c;}
}
```

Verwendung der Klasse `Bag`:

```
Long bigNumber = 1111111111L;
Bag b1 = new Bag(bigNumber);
Bag b2 = new Bag("Hallo");

// later on
Long val =  (Long) b1.getContent();
String s =  (String)b2.getContent();
```

Kein Compiler-error, wenn b2 ein Long enthält!

- **Mögliche Verbesserungen**
  - Teile Compiler beim Initialisieren mit für welchen Inhaltstyp die Instanz von Bag verwendet werden soll.
  - Der Compiler kann dann überwachen, dass wirklich nur der gewünschte Inhaltstyp hinzugefügt wird.
  - Beim Entnehmen kann man sich sicher sein, dass der gewünschte Datentyp in der Bag liegt.

Generische Klassen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Deklaration** eines *generischen Typs T* für eine Klasse
  - "Parametrisierung eines Datentyps"
  - Ersetze `Object` stets durch `T`

```
public class Bag<T> {
  private T content;
  public Bag(T content) {
        this.content = content;
  };
  public T getContent() {
        return content;
  }
  public void setContent(T c) {
        this.content = c;}
}
```

- **Verwenden** eines generischen Datentyps
  - Es entstehen 2 *parametrisierte Typen* mit den *Typparametern* Long und String.
  - Kein Typecast notwendig!

```
Long bigNumber = 1111111111L;
Bag<Long> b1 = new Bag<Long>(bigNumber);
Bag<String> b2 = new Bag<String>("Hallo");

// later on
Long val = b1.getContent();
String s = b2.getContent();
```

Hinweis für Compiler, hier wird quasi der Platzhalter T mit einem Typ belegt, der ab dann fest ist.

Motivation für Generics

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- 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:

```
public interface IntList {
    // entsprechend dem []-Operator:
    int get(int i);
    void put(int i, int v);

    // die Listenlänge betreffend
    void add(int v);
    int remove(int i);

    int length();
}
```

Kann man das nicht allgemeiner implementieren, so dass man nicht für jeden Datentyp eine spezielle Datenstruktur implementiert werden muss?

```
public interface CharSet {
    void add(char c);           // Element hinzufügen
    boolean contains(char c);   // prüfen ob bereits enthalten
    char remove(char c);        // Element entfernen
    int size();                 // nicht "length", da keine Sequenz
}
```

Beispiel Liste

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- 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:

```
public class IntElement {
    int value;
    IntElement next;
    IntElement(int v, IntElement e) {
        value = v;
        next = e;
    }
}
```

Die Klassen die dasselbe mit `char`, `double`, `String`, und allen anderen Klassen machen, sähen ja genauso aus. Geht das nicht kürzer und besser?

- 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.

Versuch einer “generischen” Implementierung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wir könnten ja nun einfach mit `Object` statt einem konkreten Typen arbeiten.
- Warum ist das keine optimale Implementierung?

```
public interface GenericList {
    void add(Object o);
    Object get(Object o);
    int remove(Object o);
    int length();
}
```

```
public class GenericListImpl implements GenericList {
    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);
    }
…
}
```

Probleme bei der Implementierung mit `Object`: Verlust der Typsicherheit

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
public class startupGenerics {
    public static void main(String[] args) {
        GenericList li = new GenericListImpl();

        li.add("Hallo");  // OK: jeder String ist auch ein Object
        li.add("Welt");
        li.add(4); // geht auch. Will man das? (Typsicherheit)
        li.add(4+""); // gut, jetzt ist das zumindest wieder ein String :/

        for (int i = 0; i < li.length(); i++)
            System.out.println(li.get(i));  // OK: jedes Object kann .toString()
        //String s = li.get(0);  // Compilerfehler: Object is not String

        String hw = (String) li.get(0);  // OK: erzwungene Typumwandlung
        int i=(int) li.get(2);  // geht auch, kann aber auch leicht schief
                                // gehen, wenn dort ein String steht.
    }
}
```

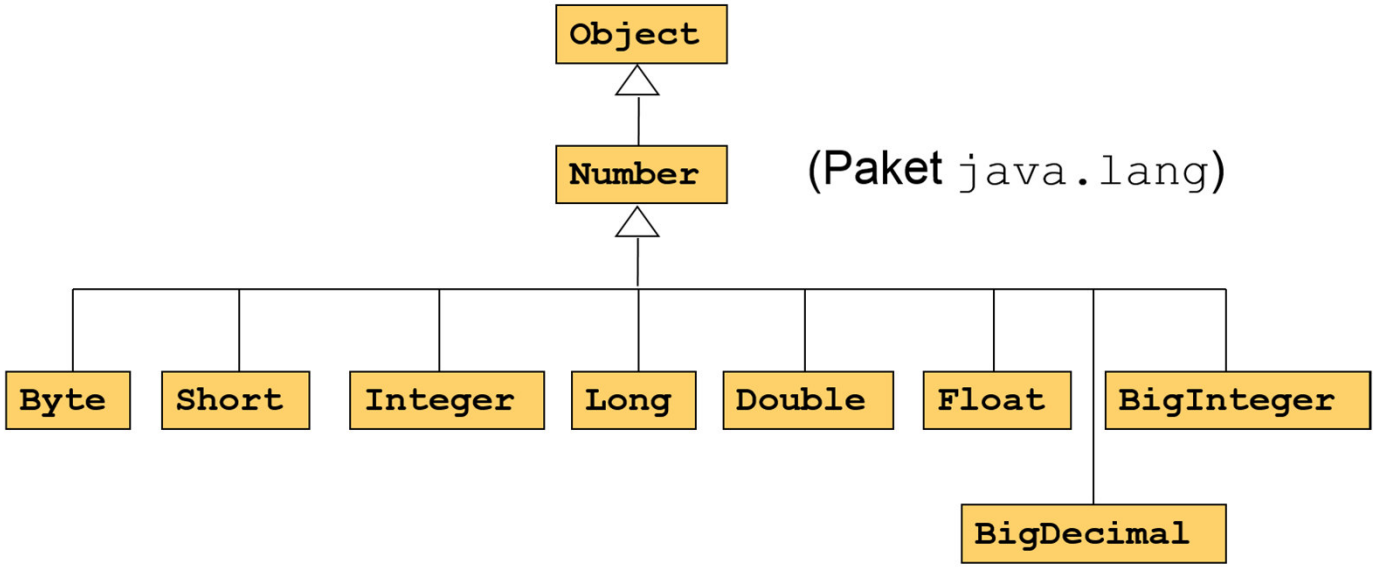
- … das geht sogar mit primitive Datentypen, wenn man die **Wrapper-Klassen** verwendet

Wrapper-Klassen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Für jeden primitiven Datentyp gibt es eine ***Wrapper-Klasse***
  - Kapseln primitive Datentypen in einem Objekt.

- Warum Wrapper-Klassen?
  - Bieten statische Methoden für Konvertierung in Strings und zurück.
  - Notwendig für Datenstrukturen der Klassenbibliothek (Collections), die nur Objekte speichern können.
  - Generics (siehe später) gibt es nur mit Wrapper-Klassen.



Wrapper-Klassen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Erzeugen von Wrapper-Objekten
  - mit Konstruktoren
  - statische ***valueOf***-Methoden
  - mittels ***Boxing***
- Alle Wrapper-Klassen überschrieben *equals()*
- Wrapper-Klassen sind **immutable**!
- **Autoboxing**
  - Automatisches Umwandeln zwischen primitiven Datentypen und Wrapper-Objekten
- Operationen ohne Wrapper-Klasse sind teilweise performanter!

```
// Erzeugung durch Konstruktoren
Boolean   b = new Boolean(true);
Character c = new Character('X');
Byte      y = new Byte(1);
Short     s = new Short(2);
Integer   i = new Integer(3);
Long      l = new Long(4);
Float     f = new Float(3.14f);
Double    d = new Double(3.14);

// Erzeugung mit valueOf
Long l1 = Long.valueOf(1000L)

// Erzeugung mit Boxing
Integer i1 = 42;

int i2 = 4711;
// Boxing -> j = Integer.valueOf(i)
Integer j = i2;
// Unboxing -> k = j.intValue()
int k = j;
```

Klasse `java.math.BigInteger`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Vorteile**
  - Darstellung beliebig großer Zahlen
  - Zahlreiche Zusatzmethoden wie z.B. modulare Arithmetik

- Objekte der Klasse sind **immutable!**
  - `public BigInteger(String val)`
  - `public BigInteger(String val, int radix)`
  - `static BigInteger valueOf(long val)`

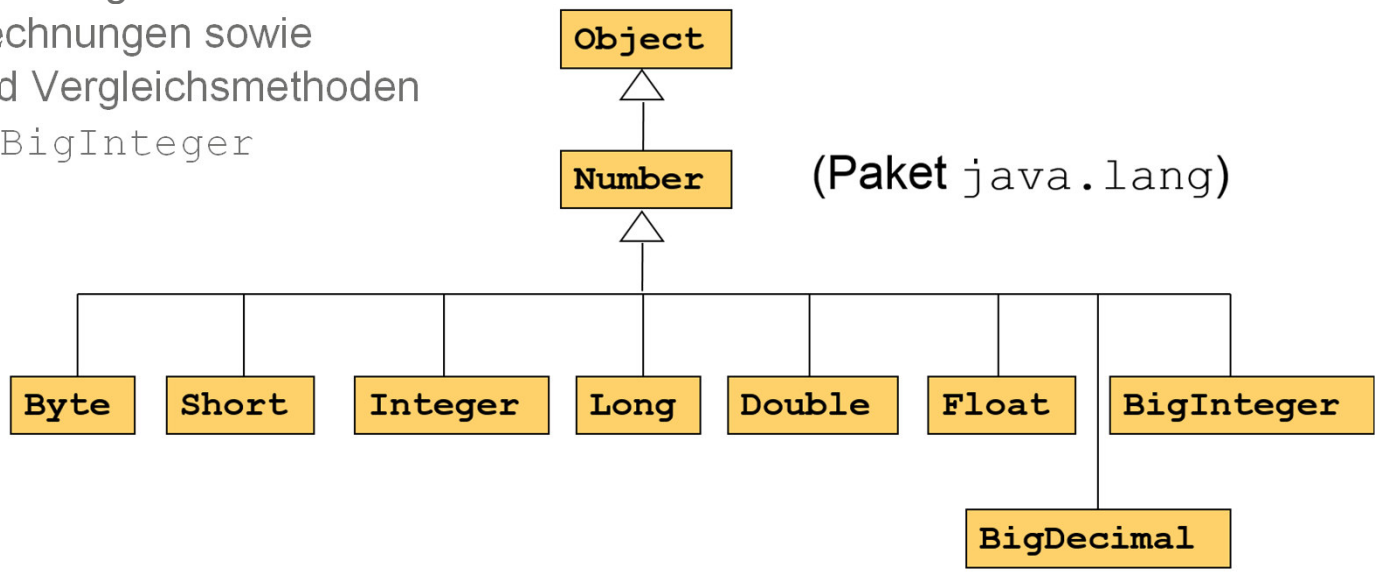
- ***Klassenspezifische Konstanten***
  - `BigInteger.ZERO`
  - `BigInteger.ONE`
  - `BigInteger.TEN`

- Zahlreiche Methoden für arithmetische Berechnungen, Vergleiche, Umwandlung in primitive Datentypen
  - https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html

Klasse `java.lang.BigDecimal`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Darstellung **beliebig genauer Fließkommazahlen**
  - https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html
- Bestehen aus
  - einer Ziffernfolge (Objekt vom Typ `BigInteger`) und
  - einer Skalierung (Anzahl der Nachkommastellen)
- Objekte der Klasse sind **immutable.**
- Methoden zur Durchführung
  - arithmetischer Berechnungen sowie
  - Konvertierungs- und Vergleichsmethoden
  - Ähnlich der Klasse `BigInteger`



Was ist hier falsch?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
BigInteger  big   = new BigInteger("12345678901234567890123");
BigInteger  small = BigInteger.valueOf(25000);

String  s  = small.toString();  // "25000"
String  t  = small.toString(7); // zur Basis 7: "132613"

big = big.add (small);          // addiert small „zu“ big

BigDecimal bd1 = new BigDecimal(big);
BigDecimal bd2 = new BigDecimal(3.14);
BigDecimal bd3 = new BigDecimal("3.14");

bd2.add(bd3);
```

Was ist hier falsch?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

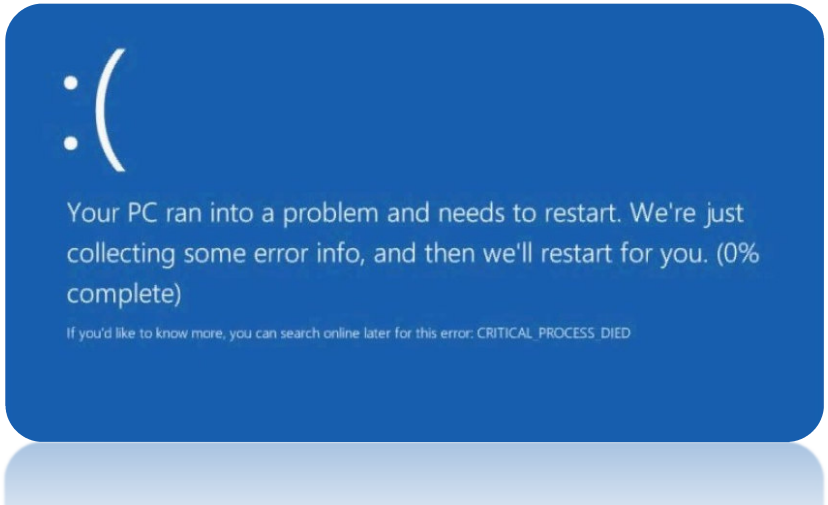
```
BigInteger  big   = new BigInteger("12345678901234567890123");
BigInteger  small = BigInteger.valueOf(25000);

String  s  = small.toString();  // "25000"
String  t  = small.toString(7); // zur Basis 7: "132613"

big = big.add (small);          // addiert small „zu“ big

BigDecimal bd1 = new BigDecimal(big);
BigDecimal bd2 = new BigDecimal(3.14);
BigDecimal bd3 = new BigDecimal("3.14");

bd2.add(bd3);
```

bd2 bleibt durch Addition unverändert. Es wird ein neues Ergebnisobjekt erzeugt, das zugewiesen werden muss.
Korrekt wäre z.B.
bd2 = bd2.add(bd3)

Fehler zur Laufzeit und Fehler zur Übersetzungszeit

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wir haben gesehen, dass eine Implementierung von generischen Klassen und Methoden duch die Verwendung von `Object` möglich ist.
  - Alle Klassen leiten von `Object` ab
  - Für primitive Datentypen gibt es ***Wrapper***
  - **Eine generische Implementierung macht unsere Methoden und Klassen (noch) wiederverwendbarer**
- **Aber**: die Typsicherheit ist verletzt, wodurch es zu **Laufzeitfehlern** kommen kann.
- **Laufzeitfehler sind besonders kritisch**, da sie während der Laufzeit, also beim Kunden, auftreten. Wir wollen also um jeden Preis Laufzeitfehler vermeiden
- Durch die Verwendung von ***Java Generics*** werden wir Laufzeitfehler los und merken schon bei der Kompilierung eines Programms, ob es Fehler bei der Typisierung gibt.



Generics

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Statt mit `Object` zu arbeiten, fügen wir nun einen Typparameter ein (einen ***Generic***)
- Wir haben ja beispielsweise eine Parameterliste einer Methode, hier der Parameter, oder Platzhalter i:

```
public interface IntList {
    int get(int i);
..}
```

- Mit ***Java Generics*** können auch Klassen nun mit einem oder mehreren Parametern versehen werden, häufig `T` für Typ, muss aber nicht:

```
public class meineKlasse<Param1,Param2> {
    private Param1 element1;
    private Param2 element2;
}
```

- Diese Parameter werden dann bei der Erzeugung von Objekten mit übergeben, ähnlich zu den Parametern eines Konstruktors. Aber diese Parameter sind im Gegensatz dazu Typen:

```
public class startupMeineKlasse {
    public static void main(String[] args) {
        MeineKlasse<Integer,String> meinObjekt = new MeineKlasse<Integer, String>();
    }
}
```

- Dabei müssen die Generics vom Typ `Object` sein (Wrapper Klassen!)

Was bringt das? Beispiel `List`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wir können nur annehmen, dass `T` vom Typ `Object` ist, trotzdem kann man damit schon viel machen.

- Werden nun andere Typen als `Integer` benutzt, z.B. `String`, kommt es zu einem Kompiler-Fehler
  → **Typsicherheit**
  → **Trotzdem hoher Grad an Wiederverwendung**

- **Nachteil**
  In der genrischen Klasse, also ListImpl<T>, kennt man den späteren Typen nicht und kann nicht viel machen

```
public class startupGenerics {
    public static void main(String[] args) {
        List<Integer> li = new ListImpl<Integer>();
        li.add(1);
        int a = li.get(0);  // kein Cast mehr notwendig

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

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

    private Element head;

    public T get(int i) { … }

    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);
    }
}
```

Einschränkungen ja, aber was ist denn möglich?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Da man mit Generics (vorerst) bei der Entwicklung nur davon ausgehen kann, dass es sich um ein `Object` handelt, ist man doch recht eingeschränkt auf die Methoden von `Object` https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

- Ein Auszug der Methoden von `Object`:

- `clone()`
- `equals(Object obj)`
- `getClass()`
- `hashCode()`
- `toString()`

- Jetzt betrachten wir `toString`, `equals` und zusätzlich

- `Comparable` https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
- `Comparator` https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Methode `toString()` der Klasse `Object`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **`public String toString()`**
  - Erzeugt eine String-Repräsentation des Objekts.
  - Liefert Zeichenkette, die Objekt in lesbarer Form beschreibt

- **Implementierung** in Klasse `Object`:
  - `getClass().getName() + '@' + Integer.toHexString(hashCode())`
  - <Klassenname>@<Hashwert/ID des Objekts>

- **Empfehlung:**
  - Für eigene Klassen: Methode überschreiben!

```
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    @Override
    public String toString(){
        return name;
    }
}
```

Was bringt das? Beispiel `List<T>`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Jetzt können wir eine generische Ausgabe erzeugen.
- Hat unser Typ mit dem wir die List erzeugen eine sinnvolle Implementierung von toString, ergibt das eine gute Ausgabe.
- Sonst sehen wir nur Klassennamen und IDs

```
class ListImpl<T> implements List<T> {
..
    @Override
    public String toString(){
        if (head == null) {
            return "";
        }

        Element it = head;
        String output=it.value.toString();
        while (it.next != null){
            it = it.next;
            output += " "+it.value.toString();
        }
        return output;
    }
}
```

```
public class startupGenerics {
    public static void main(String[] args) {
        List<Integer> li = new ListImpl<Integer>();
        li.add(1);
        li.add(2);
        li.add(3);
        System.out.println(li); // 1 2 3
    }
}
```

Methode `equals(.)` der Klasse `Object`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Überprüfen der **Identität**: "==" bzw. "!="
  - Bei *primitiven Datentypen* gleichbedeutend mit Übereinstimmung der Integer-Werte.
  - Bei *Referenzdatentypen* (Objekte) bedeutet das nur, dass Referenzen (= Adressen im Speicher) übereinstimmen.

- **Gleichheit** der Objekte, auf die Referenztypen verweisen: `equals`

- Standardimplementierung in Klasse `Object`:

```
@Override
public boolean equals(Object obj) {
    return (this == obj);
}
```

```
public class startupPerson {
    public static void main(String[] args) {
        Person p1= new Person("Klaus");
        Person p2= new Person("Maria");
        Person p3 = new Person("Klaus");

        System.out.println(p1.equals(p2)); // false
        System.out.println(p1.equals(p3)); // true
        System.out.println(p1==p3);        // false
    }
}
```

```
public class Person {
    private String name;
..
    @Override
    public boolean equals(Object obj){
        // (1) Vergleich mit 0
        if (obj == null) return false;
        // (2) Prüfe auf Identität
        if (obj == this) return true;
        // (3) Teste ob gleicher Datentyp
        if (!this.getClass().equals(obj.getClass())) {
            return false;
        }
        // Typecast und Vergleich
        Person p = (Person) obj;
        return this.name==p.name;
    }
}
```

Was bringt das? Beispiel `List<T>`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
class ListImpl<T> implements List<T> {
..
    public boolean checkPure(){
        if (head == null) {
            return true;
        }

        Element it = head;
        while (it.next != null) {
            if(!(it.value.equals(it.next.value))){
                return false;
            }
            it = it.next;
        }
        return true;
    }
}
```

- Jetzt können wir in generischen Klassen vergleichen (vorausgesetzt `equals` ist korrekt implementiert).

```
public class startupGenerics {
    public static void main(String[] args) {
        List<Integer> li = new ListImpl<Integer>();
        li.add(1);
        li.add(2);
        li.add(3);
        System.out.println(li.checkPure()); // false

        List<Integer> li2 = new ListImpl<Integer>();
        li2.add(1);
        li2.add(1);
        System.out.println(li2.checkPure()); //true
    }
}
```

Interfaces: `Comparable` und `Comparator`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Natürliche Ordnung: `interface Comparable<T>`**
  - **`public int compareTo(T o)`**
  - Implementiert Klasse diese Schnittstelle, so sind Objekte der Klasse vergleichbar.
  - Programmierer der Klasse legt jedoch fest, "wie" verglichen wird.
  - Beispiel: Ein Raum wird mit einem anderen Raum bzgl. Anzahl Sitzplätze verglichen.

- **Weitere Ordnung: `Interface Comparator<T>`**
  - **`public int compare(T o1, T o2)`**
  - Nötig, falls es ***mehrere verschiedene Vergleichskriterien*** für Objekte gibt.
  - Beispiel: "Räume" sollen einmal nach Anzahl Sitzplätze und einmal nach Quadratmetergröße sortiert werden.

- **Ergebnis** jeweils:
  - <0, wenn aktuelles bzw. linkes Objekt kleiner ist.
  - >0, wenn aktuelles bzw. linkes Objekt größer ist.
  - 0 bei "Gleichheit".

Objekte liegen also in der richtigen Reihenfolge vor

```
"A".compareTo("B") = -1
```

- Generischer Code
  - Beispiel: Sortieralgorithmen funktionieren auf allen Klassen, die Schnittstelle `Comparable` umsetzen.

Beispiel Person

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
public class Person implements Comparable<Person>{
..
    @Override
    public int compareTo(Person o) {
        return this.name.compareTo(o.name);
    }
}
```

Vergleich ohne Generics:

```
public class PersonOG implements Comparable{
    String name;
    @Override
    public int compareTo(Object o) {
        PersonOG other = (PersonOG)o; // Type safe?
        return this.name.compareTo(other.name);
    }
}
```

```
public class startupPerson {
    public static void main(String[] args) {
        Person p1= new Person("Klaus");
        Person p2= new Person("Maria");
        System.out.println(p1.compareTo(p2)); // -2
    }
}
```

- *Hier kann jetzt eine eigene Reihenfolge festgelegt werden. In dem Fall lexikographisch.*

Generic Bounds

…denn wir können ja nicht wissen, ob der Typ das Interface `comparable` implementiert

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wenn wir jetzt verlangen wollen, dass unsere generische Klasse nur mit Typen aufgerufen werden darf, die das Interface `Comparable` implementieren, verwenden wir einen ***Bound*** in der Definition der generischen Klasse.

- Beispiel `List`

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

- 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>`.

Beispiel Comparator `Person`

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
public class PhonebookPersonComparator implements Comparator<Person> {
    @Override
    public int compare(Person o1, Person o2) {
        return o1.toString().toLowerCase().compareTo(o2.toString().toLowerCase());
    }
}
```

```
public class startupPerson {
    public static void main(String[] args) {
        Person p1= new Person("Klaus");
        Person p4 =new Person("klaus");

        System.out.println(p4.compareTo(p1)); // 32

        PhonebookPersonComparator ppc = new PhonebookPersonComparator();
        System.out.println(ppc.compare(p1,p4)); //0

    }
}
```

Wieso ist `compare()` nicht `static`?
Kurze Antwort: weil Comparator alt ist.

https://stackoverflow.com/questions/21817/why-cant-i-declare-static-methods-in-an-interface?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

- Mit `Comparator` können also verschiedene Vergleichsoperationen implementiert werden.
- Mit `Comparable` wird eine natürliche Ordnung implementiert, und mit `Comparator` weitere Sonderfälle

Was bringts? Generische Algorithmen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
class Sortiere {
    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;
                }
            }
        }
    }

    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;
                }
            }
        }
    }
}
```

- Beispiel Sortieren: Die Klasse `Sortiere` kann Arrays von beliebigen Typen sortieren, solange entwerder ein passender `Comparator` mitgegeben wird, oder die Elemente des Arrays selber `Comparable` sind. Super, oder?

```
public class startupPerson {
    public static void main(String[] args) {
        Person[] pa = {p1,p2,p3,p4};
        Sortiere.sort(pa); // Sortierung natürlich
        Sortiere.sort(pa,ppc); // wie im Telefonbuch
    }
}
```

Zusammenfassung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Generics geben uns Typsicherheit bei der Wiederverwendung von Code für unterschiedliche Typen
- Dabei ist man entweder auf die Methoden von `Object` eingeschränkt, oder man schränkt die generischen Implementierungen für bestimmte Typen ein mittels Bounds. Gesehen am Beispiel `Comparator` und `Comparable`
- Generics ermöglichen uns generische Algorithmen.