Generics

Beispiel

```
public class PersonGarageDemo {
    public static void main(String[] args) {
        Person angi = new Person();
        angi.setName("Angela Merkel");
        Auto bmw = new Auto();
        bmw.setMarke("BMW");
        bmw.setPs(100);
        Auto seat = new Auto();
        seat.setMarke("SEAT");
        seat.setPs(40);
        angi.setGarage1(new Garage(bmw));
        angi.setGarage2( new Garage(seat));

        System.out.println( angi.getName() + " hat in den Garagen " +
                            angi.getGarage1().get() + " und " + angi.getGarage2().get() );

        Auto auto1 = (Auto) angi.getGarage1().get();   // (2)
        Auto auto2 = (Auto) angi.getGarage2().get();

        System.out.println( auto1.compareTo( auto2 ) > 0 ? "Links" : "Rechts" );
    }

}
```

Zwei Probleme

- ich kann reintun, was ich will
- beim Rausnehmen muss ich mich daran erinnern, was ich reingetan hab

Problem

- Explizite Typanpassungen sollten vermieden werden,
- da es erst zur Laufzeit zu Problemen kommt.

Lösung

- für jeden Fahrzeugtyp eine Garage
  - AutoGarage, MotorradGarage, LKWGarage
- **Problem:** viel Codeduplizierung

Lösung

- **Generics!!!**

Beispiel

formaler Typparameter T

```
public class GarageG<T> {
    private T value;
    public GarageG() {}
    public GarageG(T value) {this.value = value;}
    public void set(T value) {this.value = value;}
    public T get() {return value;}
    public boolean isEmpty() {return value != null;}
    public void empty() {value = null;}
}
```

Namenskonventionen

- T steht für Typ
- E für Element
- K für Key / Schlüssel
- V für Value / Wert

Nutzung

```
GarageG<Auto> autoGarage = new GarageG<Auto>();
GarageG<LKW> lkwGarage = new GarageG<LKW>();
```

Nutzung

```
autoGarage.set( new Auto() );
Auto x = autoGarage.get();// Keine Typanpassung mehr nötig
lkwGarage.set( new LKW() );
LKW s = lkwGarage.get();
```

Begriffe

- generic Type -> *Garage<T>*
- formal type parameter -> *T*
- parameterized type -> *Garage<Auto>*
- type parameter -> *Auto*
- raw type -> *Garage*

Präzesion zur Compilerzeit

- *Garage<Auto> autoGarage* vs. *Garage autoGarage*
- *Garage<LKW> lkwGarage* vs. *Garage lkwGarage*
- *Garage<Garage<Auto>> garageOfGaragen* vs. *Garage garageOfGaragen*

Übung

- Uebung1
- Uebung2

Diamonds

```
Map<String, String> map1 = new HashMap<String, String>();
Map<String, String> map2 = new HashMap<>();
```

aber:

```
Map<> map3 = new HashMap<String, String>();
```

Compilerfehler

Diamonds

```
public class Diamonds {

    List<String> list;

    public void listInitialiser() {
        list = new ArrayList<>();
    }

}
```

erlaubt **aber** u.U. nicht angebracht!!!

Generische Schnittstellen

```
public interface Comparable<T> {

    public int compareTo(T o);

}
```

Übung

- Uebung3.txt

Generische Methoden

- Methode kann generisch sein, ohne dass die Klasse es ist
- die spitzen Klammern, die den Platzhalter für den realen Typ beinhalten vor den Returntyp stellen, also etwa auch vor void und es gibt nur diesen Platz, jeder andere Ort ist falsch.

Generische Methoden

```
public <T> String genericTypeToString(T t) {
    return t.toString();
}

public <T> void genericType(T t) {
    System.out.println(t);
}
```

```
public static <T> T random(T m, T n) {
    return Math.random() > 0.5 ? m : n;
}

public static void main(String[] args) {
    GenerischeMethoden gm = new GenerischeMethoden();
    System.out.println(gm.genericTypeToString(new Auto()));

    gm.genericType("berlin");

    System.out.println(GenerischeMethoden.random(4, 5));
}
```

Was macht der Compiler?

- zwei Möglichkeiten
  - Heterogene Variante(Erzeugung individuellen Codes)
  - Homogene Variante(anstelle des Typpar. nur Object)
- **letzteres macht Java!!!!!**

Typeerasure

- kommt bei Übersetzung von generischem Java-Quellcode durch den Compiler zur Anwendung
- bezeichnet eine Zuordnung von Typen zu anderen Typen
- nach dem Type Erasure sind keine Typparameter und parametrisierte Typen mehr vorhanden.

Gründe für Typeerasure

- Kompatibilität mit altem Code
- Einfache Umsetzung von Generics
- Keine Änderung der JVM nötig
- Ein Nachteil entsteht nur bei längeren Compilezeiten,
- die meist irrelevant ist

Probleme durch Typerasure

```
<T> void newGarageContent(T t) {
    new T();
}
```

nicht möglich

Probleme durch Typeerasure

```
if(g instanceof GarageG<Auto>) {
    System.out.println("");
}
```

Probleme beim Typeerasure

```
GarageG<Auto> aGarage = (GarageG<Auto>) new GarageG<LKW>();
```

```
GarageG aGarage = (GarageG) new GarageG();
```

Keine generischen Ausnahmen

- spezielle Regel des Compilers verhindert

```
public class MyException<T> extends Exception {

}
```

liefe auf zwei identische catch-Blöcke hinaus und das ist nicht erlaubt

```
try {

} catch(MyException<Long> e1) {

} catch(MyException<Double> e2) {

}
```

Probleme des Typeerasure

```
public static boolean isEmpty( T value ) { return value == null; }   // N
```

Übung

- Uebung4.txt

Raw Types

```
public void print() {
     GarageG g;
     g = new GarageG();
     System.out.println(g);
}
```

Zu lösen mit

```java
@SuppressWarnings("rawtypes")
public void print() {
    GarageG g;
    g = new GarageG();
    System.out.println(g);
}
```

Bounds

Sicherstellung T erweitert
Fahrzeug und implementiert Comparable<Auto>

```
public class GarageG<T extends Fahrzeug & Comparable<Auto>> {
    private T value;
    public GarageG() {}
    public GarageG(T value) {this.value = value;}
    public void set(T value) {this.value = value;}
    public T get() {return value;}
    public boolean isEmpty() {return value != null;}
    public void empty() {value = null;}
}
```

Typparameter in der throws Klausel

```
public interface CharIterable<E extends Exception>
{
  boolean hasNext() throws E;
  char    next()    throws E;
}
```

Typparameter in der throws-Klausel

```java
public class StringIterable implements CharIterable<RuntimeException>
{
  private final String string;
  private int    pos;

  public StringIterable( String string )
  {
    this.string = string;
  }

  @Override public boolean hasNext()
  {
    return pos < string.length();
  }

  @Override public char next()
  {
    return string.charAt( pos++ );
  }
}
```

Typparameter in der throws-Klausel

```
/*
 * das diese Klasse nix gescheites tut, wollen wir ignorieren
 */
public class WebIterable implements CharIterable<IOException>
{

  public WebIterable( String url ) throws IOException
  {
  }

  @Override public boolean hasNext() throws IOException
  {
      return true;
  }

  @Override public char next() throws IOException
  {
      return 'c';
  }
}
```

Generics sind kovariant

```
Set<String> set = new HashSet<String>();
```

- kein Problem; **aber**

```
Set<Object> set = new HashSet<String>();
```

Compilerfehler

Type mismatch: cannot convert from HashSet<String> to Set<Object>

Generics sind kovariant

- LKW und Auto erweitern beide Fahrzeug

```
public class ContainerUsage {
    GarageG<Auto> autoGarage;
    public void print() {
        Auto a1 = new Auto();
        autoGarage = new GarageG<LKW>(a1);
        System.out.println(autoGarage);

    }

}
```

Compiler meckert ‚Type mismatch‘

Lösung des Problems

- Wildcards mit ?

```
public boolean isGarageLeer(GarageG<?>... g) {
    for ( GarageG<?> garage : g )
        if ( garage.isEmpty() )
          return true;

      return false;
}
```

- ? steht **nicht** für Object!!!!!

Wildcard ?

- ist die Wildcard ? im Einsatz wissen wir(bzw. der Compiler) nichts über den Typ

```
GarageG<?> garage = new GarageG<Auto>();
```

- erlaubt aber über den Typparameter ist nichts bekannt

Auswirkungen der Wildcard ?

Ein Aufruf von p1.get() ist legal, denn alles, was die Methode liefern wird, ist immer ein Object, auch wenn es null ist. Die Anweisung Object g = garage.get(); ist dementsprechend korrekt.

```
Auto a1 = new Auto();

GarageG<?> garage = new GarageG<Auto>();
Object g = garage.get();
garage.set(a1);
```

nicht erlaubt, da wir über den Typ nichts wissen

Wildcards

- Nicht möglich
- Wildcards erlauben nur lesenden Zugriff
- Schreiben ist nicht möglich

Uebung Wildcard

- Uebung5.txt

Bounded Wildcards

- Rückgabetyp Object
- manchmal braucht man aber vielleicht eine gewisse Einschränkung, um bestimmet Methoden anwenden zu können.

```
public void print(AbstractList<? extends Fahrzeug> list) {
    for (Fahrzeug fahrzeug : list)
        System.out.println(fahrzeug);
}
```

- Die Methode akzeptiert alle Arten von Fahrzeuglisten d.h. auch eine Liste mit Subtyp von Fahrzeug

Bounded Wildcards

- Upper Bound durch Angabe von extends
- mit upper bound aber nur lesender Zugriff
- Lower Bound durch Angabe von super
- alle Supertypen werden akzeptiert
- Mit lower bound schreibender Zugriff

```
List<? super Fahrzeug> l1 = new ArrayList<>();
l1.add(new Auto());
```

UebungUpperBoundWildcard

- Uebung6.txt

Bounded Wildcard-Typen und Bounded Typvariablen

```
boolean areLighterThan( List<? extends Fahrzeug> list, double maxWeight ) {
    return false;
}
```

```
boolean areLighterThan( List<? extends Fahrzeug> list, double maxWeight ) {
    return false;
}
```

Immer dann, wenn der formale Typparameter (etwa T) nur in der Signatur auftaucht und es in der Methode selbst keinen Rückgriff auf den Typ T gibt, wähle die Variante mit der Wildcard.

also:

Bounded Typevariable

```
public static <T extends Fahrzeug> T lightest( Collection<T> collection )
{
  Iterator<T> iterator = collection.iterator();
  T lightest = iterator.next();

  while ( iterator.hasNext() )
  {
    T next = iterator.next();

    if ( next.getWeight() < lightest.getWeight() )
      lightest = next;
  }

  return lightest;
}
```

Auf Bounded Wildcard-Typen in Rückgaben verzichten

```java
//schlecht, da Aufrufer nix über die Rückgabe weiß
public static List<?> leftSublist1( List<? extends Fahrzeug> list )
{
  return list.subList( 0, list.size() / 2 );
}

//schon besser, da er weiß, dass es sich um Fahrzeuge handelt
public static List<? extends Fahrzeug> leftSublist2( List<? extends Fahrzeug> list ) {
    return new ArrayList<Fahrzeug>();
}

//beste Variante, da das, was er reingibt auch zurückgegeben wird
public static <T extends Fahrzeug> List<T> leftSublist3( List<T> list )
{
  return list.subList( 0, list.size() / 2 );
}
```

LESS

- Lesen = extends, Schreiben = Super (LESS)
  - oder auch
- producer-extends, consumer-super (PECS)

LESS

```
public static void copy(List<? extends Fahrzeug> src, List<? extends Fahrzeug> dest) {
    for (Fahrzeug fahrzeug : src) {
        dest.add(fahrzeug);
    }
}
```

aber:

```
public static void copy1(List<? extends Fahrzeug> src, List<? super Fahrzeug> dest) {
    for (Fahrzeug fahrzeug : src) {
        dest.add(fahrzeug);
    }
}
```

Wildcard-Capture

```java
public void reverse(List<? extends Object> list) {
    // die übergebene list soll umgedreht werden - also read/write Zugriff
    // erforderlich
    ArrayList<Object> copiedList = new ArrayList<Object>(list);
    for (int i = 0; i < list.size(); i++) {
        // COMPILEFEHLER - wir wissen Elemente welcher Klasse sich in list
        // befinden.
        // Deshalb dürfen wir dort nicht einfach "beliebige" Objects
        // reinpacken (in Wirklichkeit
        // wissen wir, daß wir dort einfach nur die gleichen Elemente
        // reinpacken wie vorher, aber
        // das weiß der Compiler nicht - der verbietet dies deshalb)
        list.set(i, copiedList.get(copiedList.size() - i - 1));
    }
}

//LÖSUNG
public void reverse2(List<? extends Object> list) {
    reverseByWildcardCapture(list);
}

private <T> void reverseByWildcardCapture(List<T> list) {
    ArrayList<T> copiedList = new ArrayList<T>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, copiedList.get(copiedList.size() - i - 1));
    }
}
```

Übung

- Uebung7.txt
- Uebung8.txt
- Uebung9.txt