WBS TRAINING

GENERICS

OP2 - Die Säulen der Objektorientierten Programmierung

Java

WBS TRAINING AG
Lorenzweg 5
D-12099 Berlin
Amtsgericht Berlin HRB 68531
Sitz der Gesellschaft: Berlin

Vorstand:
Heinrich Kronbichler,
Joachim Giese
Aufsichtsrat (Vorsitz): Dr. Daniel Stadler
USt-IdNr.: DE 209 768 248

GLS Gemeinschaftsbank eG
IBAN: DE18 4306 0967 1146 1814 00
BIC: GENODEM1GLS

DQS-zertifiziert nach
DIN EN ISO 9001:2015 Reg.-Nr. 015344 QM15
Zulassung nach AZAV Reg.-Nr. 015344 AZAV

WBS TRAINING

Inhalt

1 Generics 2

1.1 Begriff 2

1.2 Normale Klasse mit generischer Methode 2

1.3 Generische Klasse 3

1.4 Abgeleitete generische Klasse mit Typ-Einschränkung 4

1.5 Generics mit mehrfacher Typbeschränkung (mit &) 5

1.6 Generisches Interface und konkrete Implementierung 6

1.7 Wildcards mit ?, <? extends T>, <? super T> 7

1.7.1 <?> — Unbekannter Typ 7

1.7.2 <? extends T> — Obere Schranke (Upper Bound) 7

1.7.3 <? super T> — Untere Schranke (Lower Bound) 8

1.7.4 Vergleichstabelle 8

1.7.5 Beispiel 9

1.8 Häufige Anwendungen 10

1.8.1 Generische Collections (Listen, Sets, Maps, etc.) 10

1.8.2 Comparatoren und Sortierungen 10

1.8.3 Streams und Functional Interfaces (Java 8+) 10

1.8.4 Generics in Frameworks und Bibliotheken 10

1.9 UML 11

1 Generics

1.1 Begriff

Generics (seit Java 5) ermöglichen es, **Klassen, Interfaces und Methoden** so zu schreiben, dass sie mit **beliebigen Datentypen** arbeiten können – ohne auf **Casting** oder **unsichere Typen** zurückzugreifen. Vorteile: **Typsicherheit**, **Wiederverwendbarkeit** und **weniger Code-Duplikate.**

```
<T>: Typ-Parameter (z. B. T, E, K, V sind Konventionen)
List<String>: „Liste von Strings“ → nicht dasselbe wie List<Object>
```

1.2 Normale Klasse mit generischer Methode

<T> vor dem Rückgabetyp zeigt: Das ist eine **generische** Methode. Eine generische Methode funktioniert mit jedem Typ **T**. Die Klasse selbst ist nicht generisch. Der Typ-Parameter wird in spitzen Klammern vor den Rückgabetyp der Methode geschrieben. Der Typ-Parameter kann dann als Rückgabetyp oder Parametertyp verwendet werden.

Type Inference (Typableitung) bedeutet, dass der Java-Compiler automatisch den Datentyp eines generischen Parameters aus dem Kontext erkennt – du musst ihn nicht explizit angeben.

```
public class Dienstleister {
   public <T> void ausgeben(T wert) {
      T lokal = wert; // Variable mit dem Typ-Parameter
      System.out.println("Wert: " + lokal);
   }

   public static void main(String[] args) {
      Dienstleister d = new Dienstleister();
      d.<String>ausgeben("Hallo"); //explizit
      d.ausgeben("Hallo"); // Type Inference
      d.ausgeben(123); // Type Inference
   }
}
```

1.3 Generische Klasse

Syntax: `NameDerKlasse<TypVariable>`
Der Bezeichner der Typ-Variable folgt den Regeln der Variablenbezeichnung. Die Klasse ist mit <T> deklariert. Der konkrete Typ (String, Integer, …) wird beim Instanziieren angegeben. Generische und Nicht-Generische Member können in generischer Klasse auch gemischt werden. Generische Typen sind immer Referenztypen, daher kann *null* zugewiesen werden (`daten = null`).

```
class Speicher<T> {
   protected T daten; //generisch
   public int counter; /nicht generisch

   //Konstruktor

   public Speicher ()
   {
      daten = null;
   }

   public Speicher(T daten) {

      this.daten = daten;

   }

   public void speichern(T daten) {

      this.daten = daten;

   }

   public T laden() {

      return daten;

   }
}


public class Demo {

   public static void main(String[] args) {

      Speicher<String> zs = new Speicher<>();

      zs.speichern("text");

   }
}
```

1.4 Abgeleitete generische Klasse mit Typ-Einschränkung

Wird von generischen Klassen geerbt, sollte die erbende Klasse ebenfalls generisch sein, um den Typ-Parameter (der generische Typ) zur Verfügung zu stellen. **Einschränkung** des Typs in <>: `<T extends Number>`. Nun dürfen nur *Number* und Subklassen davon als Typ verwendet werden. `Number` ist Oberklasse von `Integer, Double, ...)`.

```
class ZahlenSpeicher<T extends Number> extends Speicher<T> {
   public double quadrat() {
      return daten.doubleValue() * daten.doubleValue();
   }
}

public class Demo {
   public static void main(String[] args) {
      ZahlenSpeicher<Integer> zs = new ZahlenSpeicher<>();
      zs.speichern(7);
      System.out.println("Quadrat: " + zs.quadrat());
   }
}
```

1.5 Generics mit mehrfacher Typbeschränkung (mit &)

Vorteil: Du kannst in der Methode sicher *compareTo* aufrufen, ohne Casting. Es ist sichergestellt, dass die Typen vergleichbar und zahlbasiert sind.
`X` muss eine Zahl sein und mit `T` vergleichbar (z.B. `T` = Integer, dann `X` = Integer oder vergleichbarer Typ): `<X extends Number & Comparable<T>>`

```
class ZahlenSpeicher <T extends Number> {
   private T value;
   public ZahlenSpeicher (T value) {
      this. value = value;
   }
   public <X extends Number & Comparable<T>> void istGroesser (X wert) {
      if (wert.compareTo(value) > 0) {
         System.out.println(wert + " ist > " + value);
      } else {
         System.out.println(wert + " ist < " + value);
      }
   }
}

public class Test {
   public static void main(String[] args) {
      // T = Integer
      ZahlenSpeicher <Integer> speicher = new ZahlenSpeicher <>(10);
      // X = Integer (passt: Integer ist Number und Comparable<Integer>)
      speicher.istGroesser (15); // 15 ist größer als 10
      // X = Double (passt NICHT: Double ist Comparable<Double>,
      // nicht Comparable<Integer> → Compilerfehler!)
      // speicher.istGroesser (12.5); // Compilerfehler
   }
}
```

1.6 Generisches Interface und konkrete Implementierung

Interface hat zwei generische Typen `F` (From) und `T` (To). Implementierung legt beide Typen konkret fest: `String → Integer`.

```
// Generisches Interface
interface Konverter<F, T> {
    T konvertiere(F eingabe);
}

// Implementierung: String → Integer
class StringZuIntegerKonverter implements Konverter<String, Integer> {
    @Override
    public Integer konvertiere(String eingabe) {
        return Integer.parseInt(eingabe);
    }
}

public class Test {
    public static void main(String[] args) {
        Konverter<String, Integer> konv = new StringZuIntegerKonverter();
        System.out.println(konv.konvertiere("123"));
    }
}
```

1.7 Wildcards mit ?, <? extends T>, <? super T>

Wildcards (das `?`) können wir beim Erstellen von Variablen generischer Klassen verwenden, um Typ-Einschränkungen vorzunehmen. Hier haben wir neben `extends` auch noch `super` als Auswahl. `extends` erlaubt den angegebenen Typen und alle Subklassen, `super` erlaubt nur den angegebenen Typen und dessen Superklassen.

1.7.1 <?> — Unbekannter Typ

Bedeutet: Liste mit einem unbekannten Typ. Du kannst lesen, aber nichts hinzufügen außer null. Praktisch z. B. zum Durchlaufen beliebiger Listen:

```
List<?> irgendwas = List.of("Hallo", "Welt");
public void druckeAlle(List<?> liste) {
   for (Object o : liste) {
      System.out.println(o);
   }
}
```

1.7.2 <? extends T> — Obere Schranke (Upper Bound)

Bedeutet: Liste von `T` oder Untertypen davon (z. B. `Integer`, `Double` etc.). Du kannst lesen, weil alle Elemente mindestens `T` sind. Nicht schreibbar (außer null) – denn du weißt nicht, was der genaue Typ ist!

```
List<? extends Number> liste = List.of(1, 2.5, 3f);
```

Warum kein Schreiben? Weil liste theoretisch auch `List<Double>` sein könnte – und 5.5 wäre dann `Double`, aber du weißt nicht, ob das zur Liste passt.

```
List<? extends Number> liste = new ArrayList<Integer>();
liste.add(5.5); // Fehler!
```

1.7.3 <? super T> — Untere Schranke (Lower Bound)

Bedeutet: Liste von T oder Obertypen davon (z. B. Number, Object). Du kannst schreiben, weil alle Oberklassen von Integer auch mit Integer umgehen können. Beim Lesen erhältst du nur Object, weil Java nicht weiß, was genau drinsteckt.

```
List<? super Integer> liste = new ArrayList<Number>();
liste.add(42); // geht!
```

1.7.4 Vergleichstabelle

Schreibweise	Bedeutung	Lesen	Schreiben
`<?>`	unbekannter Typ	☑ `Object`	✗ (nur `null`)
`<? extends T>`	`T` oder Unterklassen	☑ `T`	✗
`<? super T>`	`T` oder Oberklassen	✗ `Object`	☑ `T`

1.7.5 Beispiel

```
public class WildcardBeispiel {
   public void zeigeZahlen(List<? extends Number> liste) { // Nur lesbar
      for (Number n : liste) {
         System.out.println("Zahl: " + n);
      }
   }

   public void fuegeEin(List<? super Integer> liste) { // Nur schreibbar
      liste.add(42);
   }

   public void druckeUnbekannt(List<?> liste) { // Nur durchlaufen
      for (Object o : liste) {
         System.out.println("Objekt: " + o);
      }
   }

   public static void main(String[] args) {
      WildcardBeispiel w = new WildcardBeispiel();
      List<Double> doubles = List.of(1.1, 2.2);
      List<Number> zahlen = new ArrayList<>();
      List<String> texte = List.of("A", "B");
      w.zeigeZahlen(doubles); // OK: Double ist Number
      w.fuegeEin(zahlen); // OK: Number ist super von Integer
      w.druckeUnbekannt(texte); // OK: unbekannter Typ
   }
}
```

1.8 Häufige Anwendungen

Nur eine minimale Auswahl.

1.8.1 Generische Collections (Listen, Sets, Maps, etc.)

```
List<String> namen = new ArrayList<>();
Map<String, Integer> noten = new HashMap<>();
```

1.8.2 Comparatoren und Sortierungen

```
Comparator<String> cmp = (a, b) -> a.compareToIgnoreCase(b);
Collections.sort(liste, cmp);
```

1.8.3 Streams und Functional Interfaces (Java 8+)

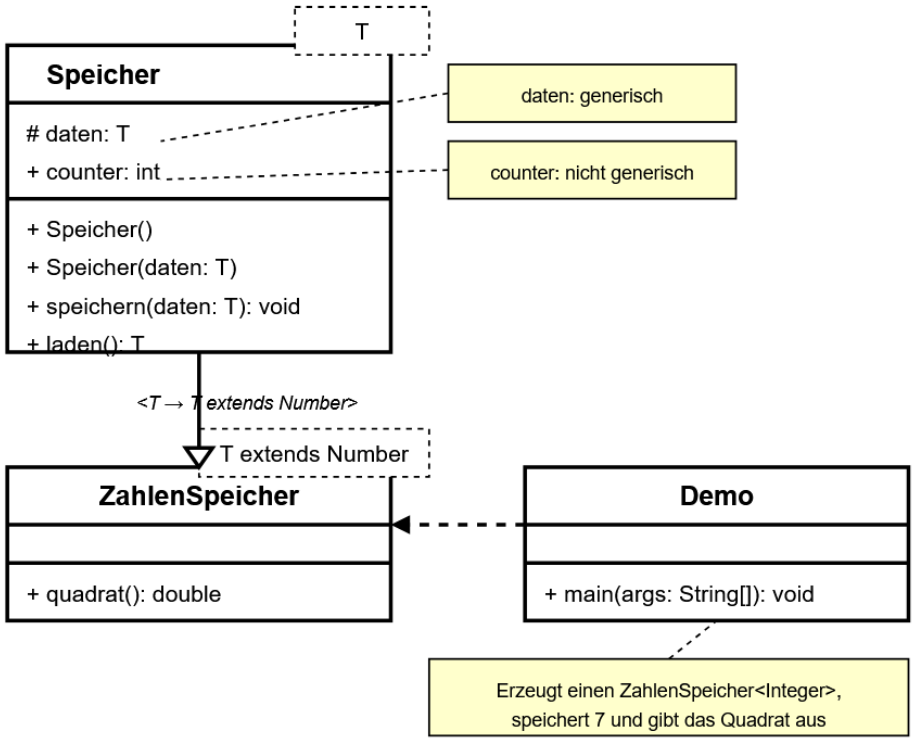
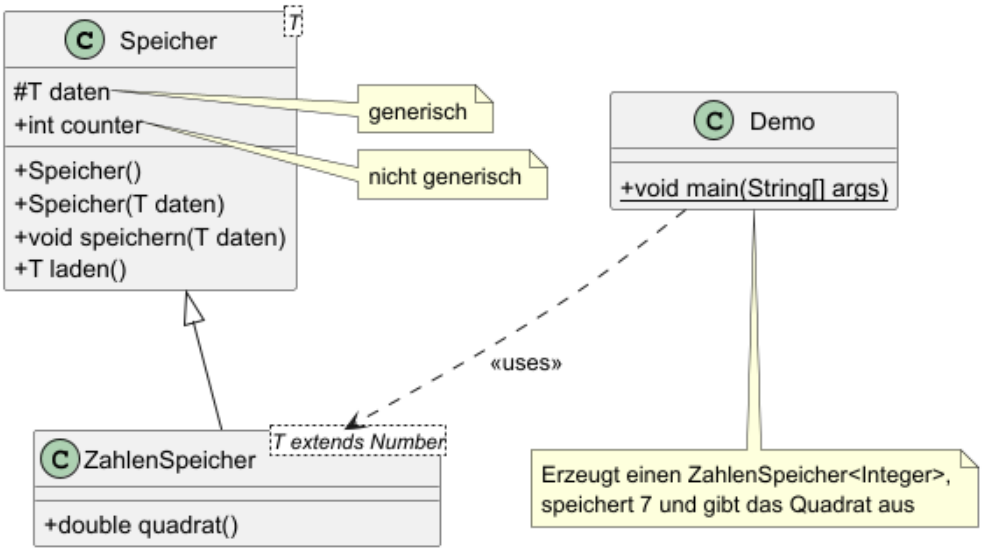
```
Stream<String> namen = List.of("Anna", "Bernd").stream();
namen.filter(s -> s.startsWith("A")).forEach(System.out::println);
```

1.8.4 Generics in Frameworks und Bibliotheken

- *JUnit* (`Assert<T>`)
- *Spring* (`JpaRepository<T, ID>`)
- *JavaFX, Swing*, etc.

1.9 UML

In UML-Notation werden Templates mit einer gestrichelten Box in der oberen rechten Ecke der Klasse dargestellt.



Plant-UML