TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 06: Generics

Java Oracle Tutorial: Generics

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Wrapper-Klassen

Bevor wir zum eigentlichen Thema dieses Kapitels kommen, Generizität, brauchen wir noch eine kleine Vorbereitung unter dem Titel Wrapper-Klassen. Leider können – im Gegensatz zu anderen Sprachen wie etwa C++ – primitive Datentypen in Java nicht in das Konzept Generizität eingebracht werden, es gehen nur Referenztypen. Daher braucht man für jeden primitiven Datentyp eine stellvertretende Klasse, und das sind die Wrapper-Klassen.

Wrapper-Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Boolean     bo = new Boolean   ( true );
Character   ch = new Character ( ´?´  );
Byte        by = new Byte      ( 123 )
Short       sh = new Short     ( -12345 );
Integer     in = new Integer   ( 12345678 );
Long        lo = new Long      ( 45678901234 );
Float       fl = new Float     ( 3.14 );
Double      do = new Double    ( 2.71E17 );
```

Zu jedem primitiven Datentyp gibt es im Package java.lang eine separate Wrapper-Klasse, die im Prinzip genau so heißt, wie der primitive Datentyp selbst, nur mit großem Anfangsbuchstaben.

Wrapper-Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Boolean	bo = new Boolean (true);
Character	ch = new Character (´?´);
Byte	by = new Byte (123)
Short	sh = new Short (-12345);
Integer	in = new Integer (12345678);
Long	lo = new Long (45678901234);
Float	fl = new Float (3.14);
Double	do = new Double (2.71E17);

Nur bei char und int hat man sich dafür entschieden, den Namen der Wrapper-Klasse etwas großzügiger zu gestalten.

Wrapper-Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Boolean     bo = new Boolean   ( true );
Character   ch = new Character ( ´?´  );
Byte        by = new Byte      ( 123 )
Short       sh = new Short     ( -12345 );
Integer     in = new Integer   ( 12345678 );
Long        lo = new Long      ( 45678901234 );
Float       fl = new Float     ( 3.14 );
Double      do = new Double    ( 2.71E17 );
```

Jede dieser Wrapper-Klassen hat einen Konstruktor mit einem einzelnen Parameter seines zugehörigen primitiven Datentyps, den er speichert.

Genau das ist die Funktion der Wrapper-Klassen: Ein Objekt einer Wrapper-Klasse speichert einen einzelnen Wert von seinem Datentyp, daher auch der Name Wrapper für englisch Hülle oder Umschlag. Der gespeicherte Wert wird sozusagen „umhüllt“ von dem Wrapper-Objekt.

Wrapper-Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.print ( bo.booleanValue() );     // true
System.out.print ( ch.charValue() );        // ?
System.out.print ( sh.shortValue() );       // 123
System.out.print ( in.intValue() );         // -12345
System.out.print ( lo.longValue() );        // 45678901234
System.out.print ( fl.floatValue() );       // 3.14
System.out.print ( do.doubleValue() );      // 2.71E17
```

Natürlich hat jede Wrapper-Klasse auch eine Methode zum Zugriff auf den sozusagen „umhüllten“ Wert. Das Bildungsprinzip für die Methodennamen dürfte offensichtlich sein.

***Nebenbemerkung*: Die Wrapper-Klassen haben noch weitere Zugriffsmethoden für die umhüllten Werte, die denselben Wert in Form von anderen primitiven Datentypen zurückliefern.**

Wrapper-Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Double.MAX_VALUE

Double.MIN_VALUE

Double.POSITIVE_INFINITY

Double.NEGATIVE_INFINITY

Wir hatten beispielhaft sogar schon eine der Wrapper-Klassen gesehen, nämlich in Kapitel 01b, Abschnitt „Allgemein: Primitive Datentypen“. Tatsächlich bieten die Wrapper-Klassen – über ihre eigentliche Funktion hinaus – eine ganze Reihe nützlicher Klassenkonstanten und Klassenmethoden, die Sie in der Doku zu diesen Klassen finden und mit allem bisher Gesagten auch leicht verstehen werden.

Boxing / Unboxing

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Integer i = 123;

System.out.print ( i );     //  123
```

Die Wrapper-Klassen bieten eine sehr wünschenswerte Bequemlichkeit, genannt Boxing und Unboxing. Wir sehen uns das nur am Beispiel Integer an; bei den anderen Wrapper-Klassen ist alles analog. Grob gesprochen, kann man sagen, ein primitiver Datentyp und seine Wrapper-Klasse sind weitgehend austauschbar; ein Wert von einem primitiven Datentyp wird bei Bedarf implizit in seine Wrapper-Klasse konvertiert und umgekehrt.

Boxing / Unboxing

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Integer i = 123;

System.out.print ( i );     //  123
```

Dies ist ein Beispiel für Boxing, das heißt, wenn ein int-Wert an einer Stelle steht, wo ein Integer erwartet wird, dann wird aus dem int-Wert implizit ein Integer-Objekt konstruiert. Genauer gesagt, wird der schon gesehene Konstruktor von Integer aufgerufen, der einen einzelnen Parameter vom Typ int hat.

Dasselbe wie hier bei der Initialisierung einer Variablen gilt beispielsweise auch bei Parameterwerten: Ist der formale Parametertyp Integer und der aktuale Parametertyp int, dann passt das dank Boxing zusammen: Der beim Aufruf der Methode übergebene int-Wert initialisiert den formalen Parameter vom Typ Integer.

Genauso funktioniert es bei Rückgaben von Methoden: Ist der Rückgabetyp einer Methode Integer und wird mit return ein int-Wert zurückgeliefert, dann kommt der int-Wert außerhalb der Methode als Integer an, der mit diesem int-Wert initialisiert ist.

Boxing / Unboxing

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Integer i = 123;

System.out.print ( i );    //  123
```

Und dasselbe auch umgekehrt, zum Beispiel bei Parameterübergabe: Die Methode erwartet ein int, bekommt aber ein Integer. Das passt, implizit wird die Methode intValue aufgerufen. Genau das ist Unboxing.

TECHNISCHE UNIVERSITÄT DARMSTADT

Generische Klassen

Wir kommen jetzt zum ersten Großthema dieses Kapitels, Generizität, und beginnen mit Klassen, die als Ganzes generisch sind. Danach werden wir einzelne generische Methoden betrachten

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
       return attribute1;
    }
   public void setAttribute1 ( T1 a1 ) {
      attribute1 = a1;
   }
   public Pair ( T1 a1, T2 a2 ) {
      attribute1 = a1;
      attribute2 = a2;
   }
   .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
   = new Pair < Integer, Double > ( i, d );
```

Diese Klasse Pair ist ein erstes einfaches Beispiel. Hier betrachen wir neben dem Konstruktor nur die get- und die set-Methode für das erste Attribut. Die get- und die set-Methode für das zweite Attribut sind bis auf den Typ T2 statt T1 identisch, daher betrachten wir letztere hier nicht.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
        return attribute1;
    }
    public void setAttribute1 ( T1 a1 ) {
        attribute1 = a1;
    }
    public Pair ( T1 a1, T2 a2 ) {
        attribute1 = a1;
        attribute2 = a2;
    }
    .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
    = new Pair < Integer, Double > ( i, d );
```

Bis hierhin ist alles noch normal wie immer: eine public-Klasse mit Namen Pair.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
        return attribute1;
    }
    public void setAttribute1 ( T1 a1 ) {
        attribute1 = a1;
    }
    public Pair ( T1 a1, T2 a2 ) {
        attribute1 = a1;
        attribute2 = a2;
    }
    .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
    = new Pair < Integer, Double > ( i, d );
```

Dies jetzt ist neu. T1 und T2 sind Platzhalter für zwei Referenztypen. Wir sagen, Klasse Pair ist *generisch* oder auch, Klasse Pair ist mit T1 und T2 *parametrisiert*. Anders herum sagen wir: T1 und T2 sind die *Typparameter* von Klasse Pair. Typparameter kommen wie hier direkt nach dem Klassennamen in spitzen Klammern – also kleiner-/größer-Zeichen – und untereinander durch Kommas getrennt.

Erst bei der Einrichtung von Variablen und Objekten der Klasse Pair werden T1 und T2 festgelegt, das werden wir gleich sehen, wenn wir den Codeausschnitt unten rechts diskutieren.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
       return attribute1;
    }
   public void setAttribute1 ( T1 a1 ) {
      attribute1 = a1;
   }
   public Pair ( T1 a1, T2 a2 ) {
      attribute1 = a1;
      attribute2 = a2;
   }
   .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
   = new Pair < Integer, Double > ( i, d );
```

Das erste Attribut der Klasse Pair hat als Typ den Typparameter T1, kann also in verschiedenen Objekten der Klasse Pair von unterschiedlichem Typ sein, wie wir gleich sehen werden.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
        return attribute1;
    }
    public void setAttribute1 ( T1 a1 ) {
        attribute1 = a1;
    }
    public Pair ( T1 a1, T2 a2 ) {
        attribute1 = a1;
        attribute2 = a2;
    }
    .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
    = new Pair < Integer, Double > ( i, d );
```

Dasselbe gilt für das zweite Attribut.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
       return attribute1;
    }
   public void setAttribute1 ( T1 a1 ) {
      attribute1 = a1;
   }
   public Pair ( T1 a1, T2 a2 ) {
      attribute1 = a1;
      attribute2 = a2;
   }
   .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
   = new Pair < Integer, Double > ( i, d );
```

Der Rückgabetyp der get-Methode ist ja immer der Typ des Attributs. Da der Typ des Attributs nicht festgelegt, sondern durch den Typparameter T1 repräsentiert wird, steht auch hier der Typparameter anstelle des konkreten Typs. Zurückgeliefert wird also ein Objekt von dem Typ, mit dem T1 instanziiert wird. Wie üblich, kann es statt dessen auch ein Objekt von einem Subtyp davon sein.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
       return attribute1;
    }
   public void setAttribute1 ( T1 a1 ) {
      attribute1 = a1;
   }
   public Pair ( T1 a1, T2 a2 ) {
      attribute1 = a1;
      attribute2 = a2;
   }
   .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
   = new Pair < Integer, Double > ( i, d );
```

Analog der Typ des Parameters der set-Methode: Ist ebenfalls eigentlich der Typ des Attributs, der ist aber noch nicht festgelegt, sondern wird auch hier repräsentiert durch den Typparameter T1. Als aktualer Parameter wird also ein Objekt erwartet mit T1 als dynamischem Typ oder einem Subtyp von T1.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
       return attribute1;
    }
   public void setAttribute1 ( T1 a1 ) {
      attribute1 = a1;
   }
```

```
   public Pair ( T1 a1, T2 a2 ) {
      attribute1 = a1;
      attribute2 = a2;
   }
   .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
   = new Pair < Integer, Double > ( i, d );
```

Jetzt ein Konstruktor. Dieser Konstruktor soll einfach die beiden Attribute initialisieren, mehr nicht. Die beiden initialisierenden Werte erhält er als Parameter. Wie bei der set-Methode eben, sind auch hier die Typen der beiden Parameter nicht festgelegt, sondern durch die beiden Typparameter, also T1 und T2, repräsentiert.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
        return attribute1;
    }
    public void setAttribute1 ( T1 a1 ) {
        attribute1 = a1;
    }
    public Pair ( T1 a1, T2 a2 ) {
        attribute1 = a1;
        attribute2 = a2;
    }
    .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
    = new Pair < Integer, Double > ( i, d );
```

Jetzt richten wir zum ersten Mal eine Variable und ein Objekt der generischen Klasse Pair ein. Das ist jetzt der Punkt, wo wir die konkreten Typen festlegen dürfen und müssen.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
       return attribute1;
    }
   public void setAttribute1 ( T1 a1 ) {
      attribute1 = a1;
   }
   public Pair ( T1 a1, T2 a2 ) {
      attribute1 = a1;
      attribute2 = a2;
   }
   .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
   = new Pair < Integer, Double > ( i, d );
```

Der erste Typ, also der T1 instanziierende, wird Klasse Integer sein. Wir richten eine Variable und ein Objekt der Klasse Integer ein, die wir dann zur Initialisierung des Pair-Objektes nutzen werden. Um Verwirrung zu vermeiden, verwenden wir hier kein Boxing und Unboxing, sondern schreiben alles aus.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
       return attribute1;
    }
   public void setAttribute1 ( T1 a1 ) {
      attribute1 = a1;
   }
   public Pair ( T1 a1, T2 a2 ) {
      attribute1 = a1;
      attribute2 = a2;
   }
   .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
   = new Pair < Integer, Double > ( i, d );
```

Der zweite Typ, also der T2 instanziierende, wird Klasse Double aus Package java.lang sein.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
        return attribute1;
    }
    public void setAttribute1 ( T1 a1 ) {
        attribute1 = a1;
    }
    public Pair ( T1 a1, T2 a2 ) {
        attribute1 = a1;
        attribute2 = a2;
    }
    .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
    = new Pair < Integer, Double > ( i, d );
```

Hier wird die Variable von Klasse pair eingerichtet. In denselben spitzen Klammern, also kleiner-/größer-Zeichen, wo bei der Definition der Klasse die Typparameter standen, stehen jetzt die konkreten Typen, Integer und Double.

Die Variable pair hat also als statischen Typ sozusagen den Typ „Paar von Integer und Double“. Wir sagen, Pair ist hier mit Double und Integer *instanziiert*.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
        return attribute1;
    }
    public void setAttribute1 ( T1 a1 ) {
        attribute1 = a1;
    }
    public Pair ( T1 a1, T2 a2 ) {
        attribute1 = a1;
        attribute2 = a2;
    }
    .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
    = new Pair < Integer, Double > ( i, d );
```

Hier wird das Objekt erzeugt. Der dynamische Typ ist in diesem Beispiel derselbe wie der statische. Auch hier schreiben wir wieder Integer und Double in spitzen Klammern hinter den Klassennamen.

Allgemein wird der vollständige, wir sagen der *instanziierte* Typ immer so geschrieben: erst der Name der eigentlichen Klasse, dann, in spitzen Klammern und durch Kommas getrennt, die *Instanziierungen* der einzelnen Typparameter.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
       return attribute1;
    }
   public void setAttribute1 ( T1 a1 ) {
      attribute1 = a1;
   }
```

```
   public Pair ( T1 a1, T2 a2 ) {
      attribute1 = a1;
      attribute2 = a2;
   }
   .....
}
```

```
Integer i = new Integer ( 123 );
Double d = new Double ( 3.14 );
Pair < Integer, Double >  pair
   = new Pair < Integer, Double > ( i, d );
```

Die Parameter des Konstruktors waren ja vom formalen Typ T1 beziehungsweise T2. Beim Aufruf verwenden wir also aktuale Parameter von den beiden instanziierenden Typen, in diesem Fall Integer und Double.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
        return attribute1;
    }
    public void setAttribute1 ( T1 a1 ) {
        attribute1 = a1;
    }
    public Pair ( T1 a1, T2 a2 ) {
        attribute1 = a1;
        attribute2 = a2;
    }
    .....
}
```

```
Integer i1 = pair.getAttribute1 ();
System.out.print ( i1.intValue() );   //  123
Integer i2 = new Integer ( 234 );
pair.setAttribute1 ( i2 );
```

Als nächstes schauen wir uns die Verwendung der get- und set-Methoden an, wobei wir auch hier zur besseren Nachvollziehbarkeit das implizite Boxing und Unboxing vermeiden. Wir betrachten weiterhin nur das erste Attribut; das zweite Attribut ist wieder völlig analog.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
        return attribute1;
    }
    public void setAttribute1 ( T1 a1 ) {
        attribute1 = a1;
    }
    public Pair ( T1 a1, T2 a2 ) {
        attribute1 = a1;
        attribute2 = a2;
    }
    .....
}
```

```
Integer i1 = pair.getAttribute1 ();
System.out.print ( i1.intValue() );   //  123
Integer i2 = new Integer ( 234 );
pair.setAttribute1 ( i2 );
```

Als erstes ein Aufruf der get-Methode. Der Wert des Attributs wird zurückgeliefert. Da der Attributtyp eine Klasse ist, wird also ein Verweis auf das Objekt zurückgeliefert, auf das das erste Attribut verweist. Dieser Verweis wird in der Variablen i1 gespeichert.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
        return attribute1;
    }
    public void setAttribute1 ( T1 a1 ) {
        attribute1 = a1;
    }
    public Pair ( T1 a1, T2 a2 ) {
        attribute1 = a1;
        attribute2 = a2;
    }
    .....
}
```

```
Integer i1 = pair.getAttribute1 ();
System.out.print ( i1.intValue() );   //  123
Integer i2 = new Integer ( 234 );
pair.setAttribute1 ( i2 );
```

Wir wie gesehen haben, hat die Klasse Integer eine Methode intValue, die den int-Wert zurückliefert, der im Integer-Objekt gespeichert ist. Bei der Einrichtung des Objektes hatten wir den Wert 123 gespeichert, genau der kommt natürlich hier wieder heraus.

***Nebenbemerkung*: Wie wir ebenfalls schon gesehen haben, könnte man wegen Unboxing auch einfach nur i1 hier hinschreiben, der explizite Aufruf von intValue ist dann nicht nötig, sondern ist dann implizit, also vom Compiler automatisch eingesetzt.**

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
       return attribute1;
    }
   public void setAttribute1 ( T1 a1 ) {
      attribute1 = a1;
   }
```

```
   public Pair ( T1 a1, T2 a2 ) {
      attribute1 = a1;
      attribute2 = a2;
   }
   .....
}
```

```
Integer i1 = pair.getAttribute1 ();
System.out.print ( i1.intValue() );   //  123
Integer i2 = new Integer ( 234 );
pair.setAttribute1 ( i2 );
```

Jetzt richten wir ein neues Objekt ein, das den int-Wert 234 intern speichert, und lassen die Variable i2 auf dieses Objekt verweisen.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Pair < T1, T2 > {
    private T1 attribute1;
    private T2 attribute2;
    public T1 getAttribute1 () {
       return attribute1;
    }
   public void setAttribute1 ( T1 a1 ) {
      attribute1 = a1;
   }
```

```
   public Pair ( T1 a1, T2 a2 ) {
      attribute1 = a1;
      attribute2 = a2;
   }
   .....
}
```

```
Integer i1 = pair.getAttribute1 ();
System.out.print ( i1.intValue() );   //  123
Integer i2 = new Integer ( 234 );
pair.setAttribute1 ( i2 );
```

Das machen wir natürlich nur, um jetzt den Wert des ersten Attributs mittels set-Methode zu überschreiben. Ab jetzt verweist das erste Attribut des Objektes von Klasse Pair auf das neue Integer-Objekt, welches nun einmal 234 speichert.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

Pair <Integer, Integer>

Pair <Double, Double>

Beim ersten Beispiel wurden die beiden Typparameter mit unterschiedlichen Referenztypen instanziiert. Der Vollständigkeit halber halten wir fest, dass beide Typparameter natürlich genauso gut auch mit demselben Typ instanziiert werden können.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T1,T2> {
    public static void m ( int n )  { ..... }
}


X.m ( 123 );
```

In Klassenmethoden können generische Typparameter nicht verwendet werden. Der Aufruf einer Klassenmethode einer generischen Klasse enthält die Typparameter daher nicht und sieht somit aus wie der Aufruf einer Klassenmethode einer nichtgenerischen Klasse.

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntPredicate {
  boolean test ( int n );
  ..........
}

public interface Predicate <T> {
  boolean test ( T t );
  ..........
}
```

Zum Abschluss des Abschnitts über generische Klassen noch ein Blick zurück:

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntPredicate {
   boolean test ( int n );
   ..........
}
```

```
public interface Predicate <T> {
   boolean test ( T t );
   ..........
}
```

***Erinnerung*: Im Kapitel 04c, Abschnitt über Functional Interfaces und Lambda-Ausdrücke, haben wir das Interface IntPredicate kennen gelernt, so wie es oben noch einmal abgedruckt ist.**

Generische Klassen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface IntPredicate {
  boolean test ( int n );
  ..........
}
```

```
public interface Predicate <T> {
  boolean test ( T t );
  ..........
}
```

Im selben Package, java.util.function, finden Sie auch eine generische Version mit derselben Funktionalität.

Dies ist nur ein einzelnes Beispiel für die vielen Interfaces in java.util.function. Das Prinzip ist immer dasselbe: Es gibt ein generisches Interface und mehrere nichtgenerische mit derselben Funktionalität für mehrere primitive Datentypen. Den Sinn dieser vielen kleinen Interfaces hatten wir schon bei der Einführung von Functional Interfaces im Kapitel 04c gesehen: Das sind viele kleine Bausteine für Funktionen, Prädikate und so weiter, die sich durch default-Methoden schrittweise zu größeren, komplexeren Funktionen, Prädikaten und so weiter zusammensetzen lassen.

TECHNISCHE UNIVERSITÄT DARMSTADT

Generische Methoden

Nicht nur ganze Klassen, sondern auch einzelne Methoden können generisch sein.

Generische Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public <T1,T2> Pair<T1,T2> makePair ( T1 t1, T2 t2 ) {
        return new Pair<T1,T2> ( t1, t2 );
    }
}
```

Dazu eine einfache Beispielklasse X, die *nicht* generisch ist.

Generische Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public <T1,T2> Pair<T1,T2> makePair ( T1 t1, T2 t2 ) {
        return new Pair<T1,T2> ( t1, t2 );
    }
}
```

Wenn wie hier eine einzelne Methode parametrisiert wird, dann sieht das genauso aus wie bei parametrisierten Klassen: spitze Klammern drum herum und die einzelnen Typparameter durch Kommas getrennt. Die Parametrisierung einer Methode steht vor ihrem Rückgabetyp beziehungsweise void.

Generische Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public <T1,T2> Pair<T1,T2> makePair ( T1 t1, T2 t2 ) {
        return new Pair<T1,T2> ( t1, t2 );
    }
}
```

Und dadurch, dass die Parametrisierung vor dem Rückgabetyp steht, kann der Rückgabetyp schon parametrisiert sein, denn die Typparameter sind an dieser Stelle ja schon eingeführt. Zurückgeliefert wird in diesem Beispiel also ein Paar, dessen zwei Attribute die beiden Typparameter der Methode als Typen haben, T1 und T2.

Generische Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public <T1,T2> Pair<T1,T2> makePair ( T1 t1, T2 t2 ) {
        return new Pair<T1,T2> ( t1, t2 );
    }
}
```

Der Sinn der Methode makePair ist, die beiden Parameter der Methode zu einem Paar zusammenzufassen.

Generische Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public <T1,T2> Pair<T1,T2> makePair ( T1 t1, T2 t2 ) {
        return new Pair<T1,T2> ( t1, t2 );
    }
}
```

Durch den Konstruktor wird diese Methode ein Einzeiler.

***Nebenbemerkung*: In der Praxis lohnt sich eine solche Methode makePair sicher nicht, denn man könnte anstelle des Aufrufs von makePair ja auch gleich direkt das Objekt mit diesem Konstruktoraufruf einrichten. Aber hier kommt es ja nur darauf an, dass makePair ein schönes kleines, illustratives Beispiel für generische Methoden allgemein ist.**

Generische Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X();

Pair<A,B> pair1 = x.makePair ( new A(), new B() );

Pair<C,D> pair2 = x.makePair ( new C(), new D() );
```

Wir sehen uns eine illustrative Anwendung dieser Methode makePair an.

Generische Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X();
Pair<A,B> pair1 = x.makePair ( new A(), new B() );
Pair<C,D> pair2 = x.makePair ( new C(), new D() );
```

Wie Sie soeben gesehen haben, ist Klasse X *nicht* generisch.

Generische Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X();
Pair<A,B> pair1 = x.makePair ( new A(), new B() );
Pair<C,D> pair2 = x.makePair ( new C(), new D() );
```

Die vier Klassen A, B, C und D sind beliebig, sie dienen hier nur als Spielmaterial und interessieren uns hier nicht weiter.

Generische Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X();
Pair<A,B> pair1 = x.makePair ( new A(), new B() );
Pair<C,D> pair2 = x.makePair ( new C(), new D() );
```

Der Compiler erkennt selbst aus dem Kontext, dass die beiden Typen genau diese sein müssen, wir müssen die generischen Parameter einer Methode beim Aufruf dieser Methode daher nicht hinschreiben.

Generische Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T1> {

    public <T2> Pair<T1,T2> makePair ( T1 t1, T2 t2 ) {
        return new Pair<T1,T2> ( t1, t2 );
    }
}


X<A> x = new X<A>();
Pair<A,B> pair1 = x.makePair ( new A(), new B() );
```

Noch eine kleine Variation des letzten Beispiels: Einer der beiden Typparameter parametrisiert die ganze Klasse X, der andere nur die Methode makePair. Der Typparameter T1 ist also überall in Klasse X verwendbar – wenn X denn weitere Attribute und Methoden hätte –, der Typparameter T2 nur in der Methode makePair.

Generische Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T1> {

    public <T2> Pair<T1,T2> makePair ( T1 t1, T2 t2 ) {
        return new Pair<T1,T2> ( t1, t2 );
    }
}


X<A> x = new X<A>();
Pair<A,B> pair1 = x.makePair ( new A(), new B() );
```

Entsprechend dem, was wir bisher zur Parametrisierung von ganzen Klassen und einzelnen Methoden gesagt hatten, ist also nur bei der Einrichtung einer Referenz und eines Objektes die Instanziierung für den Typparameter der Klasse anzugeben, beim Methodenaufruf weiterhin nichts.

TECHNISCHE UNIVERSITÄT DARMSTADT

Typparameter

Nach diesen einführenden Beispielen betrachten wir als nächstes die Typparameter etwas systematischer ...

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X ();

Integer i = new Integer ( 345 );
Double d = new Double ( 3.14 );
Character c = new Character ( ´a´ );
Byte b = new Byte ( 1 );

Pair<Integer,Double> pair1 = x.makePair ( i, d );
Pair<Character,Byte> pair2 = x.makePair ( c, b );
```

... und verwenden als Beispiel dafür wieder Klasse Pair sowie die nichtgenerische Variante der Klasse X mit Methode makePair, wobei makePair zwei Typparameter hat.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X ();
Integer i = new Integer ( 345 );
Double d = new Double ( 3.14 );
Character c = new Character ( ´a´ );
Byte b = new Byte ( 1 );
Pair<Integer,Double> pair1 = x.makePair ( i, d );
Pair<Character,Byte> pair2 = x.makePair ( c, b );
```

Dass Integer und Double als Typparameter gehen, haben wir schon gesehen.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X ();

Integer i = new Integer ( 345 );
Double d = new Double ( 3.14 );
Character c = new Character ( ´a´ );
Byte b = new Byte ( 1 );

Pair<Integer,Double> pair1 = x.makePair ( i, d );
Pair<Character,Byte> pair2 = x.makePair ( c, b );
```

Genauso sind beispielsweise auch diese beiden Typparameter möglich.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X ();

Integer i = new Integer ( 345 );
Double d = new Double ( 3.14 );
Character c = new Character ( ´a´ );
Byte b = new Byte ( 1 );

Pair<Integer,Double> pair1 = x.makePair ( i, d );
Pair<Character,Byte> pair2 = x.makePair ( c, b );
```

In jedem möglichen Fall können wir dasselbe machen wie vorher mit Integer und Double, zum Beispiel makePair aufrufen.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X ();

String s = new String ( “Hello World“ );
Calendar c = Calendar.getInstance();
int [ ] a = new int [ 3 ];
String [ ] b = new String [ 5 ];
Pair<String,Calendar> pair3 = x.makePair ( s, c );
Pair<int [ ],String [ ]> pair4 = x.makePair ( a, b );
```

Wir lösen uns jetzt von den Wrapper-Klassen und schauen uns ein paar andere Beispiele für mögliche Instanziierungen der Typparameter an.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X ();
String s = new String ( “Hello World“ );
Calendar c = Calendar.getInstance();
int [ ] a = new int [ 3 ];
String [ ] b = new String [ 5 ];
Pair<String,Calendar> pair3 = x.makePair ( s, c );
Pair<int [ ],String [ ]> pair4 = x.makePair ( a, b );
```

Zum Beispiel diese beiden Klassen, die wir schon mehrfach gesehen haben. Anstelle von vordefinierten Klassen kann man natürlich auch selbstdefinierte Klassen hernehmen.

***Vorgriff*: Interfaces sind genauso möglich wie Klassen, aber Interfaces sehen wir uns hier noch nicht am Beispiel an, erst später, am Ende des Abschnitts zu *eingeschränkten* Typparametern.**

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X ();
String s = new String ( “Hello World“ );
Calendar c = Calendar.getInstance();
int [ ] a = new int [ 3 ];
String [ ] b = new String [ 5 ];
Pair<String,Calendar> pair3 = x.makePair ( s, c );
Pair<int [ ],String [ ]> pair4 = x.makePair ( a, b );
```

Und auch Arrays sind als Instanziierungen von Typparametern möglich, und zwar sowohl mit einem Referenztyp als auch mit einem primitiven Datentyp als Komponententyp.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X x = new X ();

Pair<Integer,Double> pair5 = new Pair<Integer,Double> ( i, d );
Pair<Double,Integer> pair6 = new Pair<Double,Integer> ( d, i );

Pair<String,Pair<Integer,Double>> pair7
    = new Pair<String,Pair<Integer,Double>> ( s, pair5 );

Pair<Pair<Double,Integer>,String> pair8
    = new Pair<Pair<Double,Integer>,String> ( pair6, s );
```

Auch parametrisierte Klassen und Interfaces sind als Instanziierungen von Typparametern möglich. Hier zum Beispiel sind pair7 und pair8 selbst Instanziierungen von einem Typparameter von Typ Pair. Die Referenzen pair7 und pair8 verweisen also auf Paare, die ihrerseits jeweils ein Paar als eines der beiden Attribute haben. Natürlich könnten auch beide Attribute Paare sein und diese wiederum Paare als Attribute enthalten und so weiter; das ist hier nicht gezeigt.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

Pair < String, Object >

Pair < String, Pair < Integer, Double > >

Pair < int, Double >

Pair < Integer, double >

Was also geht, sind beispielsweise die beiden Fälle oben. Was definitiv *nicht* geht, sind primitive Datentypen wie in den beiden Beispielen unten. Genau dafür haben wir aber die Wrapper-Klassen mit Boxing und Unboxing, nämlich um die Unbequemlichkeit möglichst klein zu halten, indem wir primitive Datentypen durch Wrapper-Klassen hintenherum hineinschmuggeln.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }

public class Y extends X  { ..... }

public class Z extends Y  { ..... }

public class A {
   public void m ( Pair<X,Y> paar ) { ..... }
}
```

```
A a = new A();
a.m ( new Pair<X,Y>(x,y) );
a.m ( new Pair<X,Z>(x,z) );
a.m ( new Pair<Y,Y>(y,y) );
```

Ein Punkt ist zu beachten: Auch wenn hinter jeder Variable einer Basisklasse natürlich weiterhin ein Objekt einer abgeleiteten Klasse stecken kann – auf Typebene kann die Basisklasse *nicht* durch eine abgeleitete Klasse ersetzt werden.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }

public class Y extends X  { ..... }

public class Z extends Y  { ..... }

public class A {
   public void m ( Pair<X,Y> paar ) { ..... }
}
```

```
A a = new A();
a.m ( new Pair<X,Y>(x,y) );
a.m ( new Pair<X,Z>(x,z) );
a.m ( new Pair<Y,Y>(y,y) );
```

Um zu verstehen, was damit gemeint ist, betrachten wir eine Klasse X, eine von Klasse X direkt abgeleitete Klasse Y und eine von X indirekt über Y abgeleitete Klasse Z.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }

public class Y extends X  { ..... }

public class Z extends Y  { ..... }

public class A {
   public void m ( Pair<X,Y> paar ) { ..... }
}
```

```
A a = new A();
a.m ( new Pair<X,Y>(x,y) );
a.m ( new Pair<X,Z>(x,z) );
a.m ( new Pair<Y,Y>(y,y) );
```

Wie diese drei Klassen genau aussehen, interessiert uns in diesem Zusammenhang wieder einmal nicht.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }

public class Y extends X  { ..... }

public class Z extends Y  { ..... }

public class A {
   public void m ( Pair<X,Y> paar ) { ..... }
}
```

```
A a = new A();
a.m ( new Pair<X,Y>(x,y) );
a.m ( new Pair<X,Z>(x,z) );
a.m ( new Pair<Y,Y>(y,y) );
```

Jetzt nehmen wir noch eine Klasse A mit einer Methode m hinzu, die ein Pair als Parameter bekommt, das wiederum durch X und Y instanziiert ist.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }

public class Y extends X  { ..... }

public class Z extends Y  { ..... }

public class A {
   public void m ( Pair<X,Y> paar ) { ..... }
}
```

```
A a = new A();
a.m ( new Pair<X,Y>(x,y) );
a.m ( new Pair<X,Z>(x,z) );
a.m ( new Pair<Y,Y>(y,y) );
```

Diese Methode m ist *nicht* generisch, denn die beiden Typparameter der Klasse Paar sind ja mit konkreten Typen belegt, nämlich X und Y, und somit hat auch der Parameter der Methode m einen festen Typ, nämlich „Paar von X und Y“.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }

public class Y extends X  { ..... }

public class Z extends Y  { ..... }

public class A {
   public void m ( Pair<X,Y> paar ) { ..... }
}
```

```
A a = new A();
a.m ( new Pair<X,Y>(x,y) );
a.m ( new Pair<X,Z>(x,z) );
a.m ( new Pair<Y,Y>(y,y) );
```

Wir richten eine Variable und ein Objekt der Klasse A ein, um damit versuchsweise Methode m aufzurufen.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }

public class Y extends X  { ..... }

public class Z extends Y  { ..... }

public class A {
   public void m ( Pair<X,Y> paar ) { ..... }
}
```

```
A a = new A();
a.m ( new Pair<X,Y>(x,y) );
a.m ( new Pair<X,Z>(x,z) );
a.m ( new Pair<Y,Y>(y,y) );
```

In diesem Aufruf passen die beiden Typparameter exakt zusammen, also kein Problem. Die beiden Referenzen x und y müssen dafür natürlich von Klasse X und Y beziehungsweise von Subtypen davon sein.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }

public class Y extends X  { ..... }

public class Z extends Y  { ..... }

public class A {
   public void m ( Pair<X,Y> paar ) { ..... }
}
```

```
A a = new A();
a.m ( new Pair<X,Y>(x,y) );
a.m ( new Pair<X,Z>(x,z) );
a.m ( new Pair<Y,Y>(y,y) );
```

Aber es ist *nicht* erlaubt, dass anstelle des zweiten Typparameters Y beziehungsweise anstelle des ersten Typparameters X jetzt andere, direkt oder indirekt davon abgeleitete Klassen stehen.

Das ist mit der Nichtübertragbarkeit von Vererbung auf Typparameter gemeint.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Pair<String,Integer> pair;
pair = new Pair<> ( “Hello“, 123 );
```

Zum Abschluss dieses Abschnitts eine verkürzte Schreibweise. Sie ist recht bequem, wir werden sie aber im Folgenden zur Vermeidung von Verwirrung nicht anwenden.

Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Pair<String,Integer> pair;
pair = new Pair<> ( “Hello“, 123 );
```

An dieser Stelle kann der Compiler durch die beiden Parameter des Konstruktors *selbst* herausfinden, dass die beiden Typparameter von Pair mit String und Integer instanziiert sind. In solchen Fällen, wo dies eindeutig ist, kann man den Inhalt der spitzen Klammern weglassen.

Aus offensichtlichen graphischen Gründen wird dieses Konstrukt oft der Diamond-Operator genannt, wobei Diamond hier mit Raute oder Karo zu übersetzen ist. Das ist so nicht ganz korrekt, denn es handelt sich nicht um einen Operator, aber diese Formulierung hat sich eben eingebürgert.

TECHNISCHE UNIVERSITÄT DARMSTADT

Eingeschränkte Typparameter

Sehr häufig wird von einem Typparameter erwartet, dass er bestimmte Methoden mit bestimmten Methodenköpfen besitzt, denn diese Methoden sollen dann in den Methoden der generischen Klasse aufgerufen werden. Verschiedene Programmiersprachen bieten dafür sehr unterschiedliche Konzepte. Wir schauen uns im Folgenden natürlich das Konzept von Java an.

***Vorgriff*: Auf die Konzepte anderer Programmiersprachen wird im Kapitel zu Polymorphie kurz eingegangen.**

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class X {
    public void m1 ()  { ..... }
    abstract public int m2 ();
}
public class Y extends X  { ..... }
public class Z extends Y  { ..... }
```

Zum Konzept von Java erst einmal eine illustrative Basisklasse X. Von Klasse X ist Klasse Y abgeleitet, und davon wiederum Klasse Z.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class X {
    public void m1 ()  { ..... }
    abstract public int m2 ();
}
public class Y extends X  { ..... }
public class Z extends Y  { ..... }
```

Wie so häufig, interessieren wir uns auch hier nicht wirklich für die Methodenimplementationen, sondern in diesem Fall nur für die Köpfe der beiden Methoden m1 und m2 in X.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
abstract public class X {
    public void m1 ()  { ..... }
    abstract public int m2 ();
}
public class Y extends X  { ..... }
public class Z extends Y  { ..... }
```

Insbesondere interessieren wir uns nicht dafür, ob die Methoden in Klasse X überhaupt implementiert oder nicht vielleicht sogar abstrakt sind, daher zu Demonstrationszwecken eine abstrakte Methode in X, so dass Klasse X selbst wie immer ebenfalls abstrakt sein muss.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A  < T extends X > {
    public void m ( T t )  {
        t.m1();
        System.out.print ( t.m2( ) );
    }
}
```

```
A<X>  a1 = new A<X>();

A<Y>  a2 = new A<Y>();

A<Z>  a3 = new A<Z>();

A<String>  a4 = new A<String>();
```

Wir brauchen noch eine generische Beispielklasse A, um Einschränkung des Typparameters auf X und die von X abgeleiteten Klassen zu demonstrieren.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A < T extends X > {
    public void m ( T t ) {
        t.m1();
        System.out.print ( t.m2( ) );
    }
}
```

```
A<X>  a1 = new A<X>();

A<Y>  a2 = new A<Y>();

A<Z>  a3 = new A<Z>();


A<String>  a4 = new A<String>();
```

Das geht so wie hier: Der Typparameter T wird durch das Schlüsselwort extends dahingehend eingeschränkt, dass T entweder gleich X oder direkt oder indirekt von X abgeleitet sein darf. Wir sagen, Typparameter T ist durch Klasse X *beschränkt*. Anstelle einer Klasse wie hier kann auch ein Interface einen Typparameter beschränken.

***Nebenbemerkung*: Gezeigt ist hier nur der Fall eines einzigen Typparameters. Selbstverständlich funktioniert das Konzept der eingeschränkten Typparameter auch bei Klassen, die mit mehr als einem Typparameter parametrisiert sind, und auch mit Methoden, die mit einem oder mehreren Typparametern parametrisiert sind. Alle oder nur einzelne Typparameter können dann unabhängig voneinander eingeschränkt sein, durch dieselbe Klasse beziehungsweise durch dasselbe Interface oder durch unterschiedliche – wie man es eben für die jeweilige konkrete Situation braucht.**

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A  < T extends X > {
    public void m ( T t ) {
        t.m1();
        System.out.print ( t.m2( ) );
    }
}
```

```
A<X>  a1 = new A<X>();

A<Y>  a2 = new A<Y>();

A<Z>  a3 = new A<Z>();

A<String>  a4 = new A<String>();
```

Wie bisher auch, kann T jetzt verwendet werden als Typ eines Parameters wie hier oder auch als Rückgabetyp.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A  < T extends X > {
    public void m ( T t )  {
       t.m1();
       System.out.print ( t.m2( ) );
    }
}
```

```
A<X>  a1 = new A<X>();

A<Y>  a2 = new A<Y>();

A<Z>  a3 = new A<Z>();


A<String>  a4 = new A<String>();
```

Der entscheidende Punkt des Beispiels wird *jetzt* farblich hervorgehoben: In Klasse X sind ja die Methoden m1 und m2 definiert. Zwar ist m2 in X nicht *implementiert*, aber das ist egal, Hauptsache *definiert*. Hinter einem Parameter von Klasse X verbirgt sich dann eben ein Objekt von einer abgeleiteten Klasse, in der m2 implementiert ist. So wie wir es immer bei abstrakten Klassen gesehen haben.

Genau das ist die Motivation dafür, dass Typparameter in Java so eingeschränkt werden können: Denn wenn T garantiert entweder gleich X oder abgeleitet von X ist, dann ist absolut sicher, dass T nur Typen annehmen kann, in denen die Methoden m1 und m2 so definiert sind, wie sie hier aufgerufen werden. Es passt garantiert alles zusammen.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A  < T extends X > {
    public void m ( T t )  {
        t.m1();
        System.out.print ( t.m2( ) );
    }
}
```

```
A<X>  a1 = new A<X>();
A<Y>  a2 = new A<Y>();
A<Z>  a3 = new A<Z>();

A<String>  a4 = new A<String>();
```

Jetzt müssen wir noch sehen, wie das mit Referenzen und Objekten dieser Beispielklasse A *systematisch* aussieht.

Wie gesagt, kann natürlich Klasse X selbst der Typparameter sein.

Da die Klasse X abstrakt ist, können wir zwar keine Objekte von X verwenden, aber durchaus Objekte von Y und Z. Ansonsten gingen natürlich auch Objekte von X.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A  < T extends X > {
    public void m ( T t )  {
        t.m1();
        System.out.print ( t.m2( ) );
    }
}
```

```
A<X>  a1 = new A<X>();
A<Y>  a2 = new A<Y>();
A<Z>  a3 = new A<Z>();

A<String>  a4 = new A<String>();
```

Genauso jede *direkt* von X abgeleitete Klasse wie Y.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A  < T extends X > {
    public void m ( T t )  {
        t.m1();
        System.out.print ( t.m2( ) );
    }
}
```

```
A<X>  a1 = new A<X>();
A<Y>  a2 = new A<Y>();
A<Z>  a3 = new A<Z>();

A<String>  a4 = new A<String>();
```

Aber auch jede *in*direkt von X abgeleitete Klasse wie Z.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A  < T extends X > {
    public void m ( T t )  {
        t.m1();
        System.out.print ( t.m2( ) );
    }
}
```

```
A<X>  a1 = new A<X>();
A<Y>  a2 = new A<Y>();
A<Z>  a3 = new A<Z>();

A<String>  a4 = new A<String>();
```

Aber Klassen ungleich X, die auch *nicht* direkt oder indirekt von X abgeleitet sind, dürfen *nicht* Typparameter der Klasse A sein. Die letzte Zeile führt zu einer Fehlermeldung des Compilers nebst Abbruch der Übersetzung. Das ist auch gut so, denn wenn eine Klasse wie hier Klasse String *nicht* von X abgeleitet ist, dann sind die Methoden m1 und m2 in der Regel auch nicht vorhanden, und wenn sie doch vorhanden sind, dann nicht unbedingt mit derselben Parameterliste und demselben Rückgabetyp wie in Klasse X, also nicht kompatibel zur Verwendung in Klasse A.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T extends Number> {
   public double m ( T t ) {
      return t.doubleValue();
   }
}
```

Das Beispiel eben mit Klassen X, Y, Z und A war rein schematisch. Als nächstes ein sehr kleines, aber durchaus schon realistisches Anwendungsbeispiel.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T extends Number> {
   public double m ( T t ) {
      return t.doubleValue();
   }
}
```

Von der abstrakten Klasse Number in Package java.lang sind die Wrapper-Klassen abgeleitet, die wir zu Beginn dieses Kapitels eingeführt hatten, also Integer, Double und so weiter. Nur Klasse Character ist nicht von Number, sondern direkt von Object abgeleitet, was durchaus Sinn macht, da der primitive Datentyp char trotz aller Gemeinsamkeiten mit ganzzahligen Datentypen eben doch etwas anderes ist.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T extends Number> {
   public double m ( T t ) {
      return t.doubleValue();
   }
}
```

In Klasse Number ist unter anderem schon die abstrakte Methode doubleValue definiert in der Form, wie wir sie für Klasse Double gesehen haben. Diese Methode ist also auch in allen Wrapper-Klassen definiert − dort dann natürlich auch jeweils implementiert, passend zum jeweiligen primitiven Datentyp.

Durch die Typeinschränkung auf Number ist gewährleistet, dass die Methode doubleValue so für T existiert, wie sie hier verwendet wird.

Eingeschränkte Typparameter


```
public class X { ..... }

public interface Int1 { ..... }

public interface Int2 { ..... }

public interface Int3 { ..... }

public class A <T extends X & Int1 & Int2 & Int3>
```

Zum Abschluss des Abschnitts zu eingeschränkten Typparametern noch der Fall, dass ein Typparameter *mehrfach* eingeschränkt ist.

***Erinnerung*: Wir haben schon des Öfteren gesehen, dass eine Klasse nur von *einer* anderen Klasse abgeleitet sein, aber beliebig viele Interfaces implementieren kann.**

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }

public interface Int1 { ..... }

public interface Int2 { ..... }

public interface Int3 { ..... }

public class A <T extends X & Int1 & Int2 & Int3>
```

Mit dieser Schreibweise lässt sich erzwingen, dass der Typparameter T sowohl auf Klasse X als auch auf die drei hier aufgezähltes Interfaces eingeschränkt ist. Das heißt, erlaubt als Instanziierungen von T sind nur Klassen, die jedes der drei Interfaces direkt oder indirekt implementieren und entweder gleich X oder von X direkt oder indirekt abgeleitet sind. Die einzelnen Elemente der Aufzählung werden durch Et-Zeichen (englisch Ampersand) voneinander getrennt.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X { ..... }

public interface Int1 { ..... }

public interface Int2 { ..... }

public interface Int3 { ..... }

public class A <T extends X & Int1 & Int2 & Int3>
```

Ist tatsächlich eine Klasse in der Liste dabei, muss sie als erstes kommen.

Eingeschränkte Typparameter

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// public class X { ..... }

public interface Int1 { ..... }

public interface Int2 { ..... }

public interface Int3 { ..... }

public class A <T extends Int1 & Int2 & Int3>
```

In dem Fall, dass *keine* Basisklasse aufgezählt wird, also wenn T nur durch Interfaces beschränkt ist, kann T anstelle einer implementierenden Klasse selbstverständlich auch ein erweiterndes Interface sein. Das heißt, T müsste alle drei Interfaces erweitern.

TECHNISCHE UNIVERSITÄT DARMSTADT

Wildcards

Eben haben wir Einschränkungen von Typparametern bei der Definition einer generischen Klasse oder Methode betrachtet.

Auch bei der *Instanziierung* von Typparametern kann man Einschränkungen definieren. Der Suchbegriff, mit dem Sie Informationen dazu finden, lautet *Wildcard*, also englisch Joker oder Platzhalter. Wie so häufig, hat sich keine einheitliche deutsche Übersetzung des englischen Fachbegriffs etabliert, also verwenden wir im Weiteren den englischen Begriff.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T> {
    public T attribute;
}

public class A {
    public double m ( X<Number> n ) {
        return n.attribute.doubleValue();
}
```

Wir betrachten wieder ein kleines, illustratives Beispiel.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T> {
   public T attribute;
}

public class A {
   public double m ( X<Number> n ) {
      return n.attribute.doubleValue();
}
```

Eine Klasse A hat eine Methode m mit einem Parameter n der instanziierten Klasse X. Da X instanziiert ist, ist A keine generische Klasse und m keine generische Methode.

***Erinnerung*: im Abschnitt zu Typparametern weiter vorne in diesem Kapitel haben wir gesehen, dass der aktuale Parameter tatsächlich nur Klasse X sein darf und instanziiert mit Klasse Number sein muss. Klasse X instanziiert mit einer der Wrapper-Klassen, die von Number abgeleitet sind, geht *nicht* als aktualer Parameter.**

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T> {
   public T attribute;
}

public class A {
   public double m ( X<? extends Number> n ) {
      return n.attribute.doubleValue();
}
```

Das ist jetzt neu: Mit Fragezeichen und extends vor dem Typ – hier: Number – wird festgelegt, dass der Typ des Parameters n entweder Number oder eine von Number direkt oder indirekt abgeleitete Klasse ist. Dadurch ist es möglich, auch aktuale Parameter zu verwenden, die anstelle von Number eine der von Number abgeleiteten Wrapper-Klassen als Instanziierung von X haben.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

Unterscheidung:

public <T extends Number> double m (X<T> n, T m) { ..... }

→ Die *ganze Methode* ist generisch:
Number und alle Subtypen von Number

public double m (X<? extends Number> n, Integer m) { ..... }

→ Die Methode ist *nicht* generisch

→ Der *einzelne Parameter* ist generisch:
Number und alle Subtypen von Number

Es ist wichtig, sich den fundamentalen Unterschied klarzumachen zwischen diesen beiden soeben gesehenen Konzepten, die sich auf den ersten Blick ja doch sehr ähnlich sehen.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T> { ..... }

public class A {
    public String m ( X<?> obj )
        return obj.toString();
}
```

Auch das Fragezeichen allein ist ein Wildcard, das allgemeinst mögliche.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T> { ..... }

public class A {
    public String m ( X<?> obj )
        return obj.toString();
}
```

In diesem Fall, also wenn auf das Fragezeichen *nicht* das Schlüsselwort extends und ein Typname folgen, dann ist das implizit zu verstehen als „extends Object“. Es darf dann nur Funktionalität verwendet werden, die schon in Object – und damit in allen Klassen in Java – definiert sind.

***Nebenbemerkung*: Von der Schreibweise her ist das natürlich perfekt konform dazu, dass eine Klasse implizit von Object abgeleitet ist, wenn sie nicht explizit mit extends von einer anderen Klasse abgeleitet wird.**

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T> {
   public void m1 ( T t ) { ..... }
}

public class A {
    public void m2 ( X<? super Double> x )
        x.m1 ( new Double(3.14) );
}
```

Mit extends kann der Typparameter nach *oben* in der Vererbungshierarchie beschränkt werden, im Englischen spricht man auch von Upper Bounded Wildcards. Es gibt auch die Beschränkung nach *unten*, also Lower Bounded Wildcards.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T> {
  public void m1 ( T t ) { ..... }
}

public class A {
   public void m2 ( X<? super Double> x )
       x.m1 ( new Double(3.14) );
}
```

Dazu wird einfach nur das Schlüsselwort extends durch das Schlüsselwort super ersetzt. In diesem Beispiel kann Methode m2 von Klasse A mit verschiedenen Instanziierungen von X als aktualen Parametern aufgerufen werden, nämlich mit der angegebenen Klasse – hier: Double –, der direkten Basisklasse von Double, allen indirekten Basisklassen von Double sowie allen von Double implementierten Interfaces.

Da Double direkt von Number und Number direkt von Object abgeleitet ist, kommen im Falle von Double nur diese drei Klassen in Frage. Hinzu kommen zwei Interfaces, die von Double implementiert werden, die uns hier aber nicht interessieren, siehe die Spezifikation der Klasse Double.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T> {
   public void m1 ( T t ) { ..... }
}

public class A {
    public void m2 ( X<? super Double> x )
        x.m1 ( new Double(3.14) );
}
```

Diese Verwendung von Klasse Double ist konform dazu, dass neben Double auch alle direkten und indirekten Basisklassen sowie alle direkt oder indirekt implementierten Interfaces möglich sind, denn bei allen diesen Typen T ist es ja zulässig und problemlos möglich, ein Objekt der Klasse Double als aktualen Parameter einem formalen Parameter von Datentyp T zuzuweisen.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X <T> {
   public void m1 ( T t ) { ..... }
}

public class A {
    public void m2 ( X<? super Number> x )
        x.m1 ( new Double(3.14) );
}
```

Nur noch eine kleine Variation dieses Beispiels: Wenn der Typ des aktualen Parameters von Methode m1 nicht exakt gleich dem instanziierenden Typ, sondern direkt oder indirekt davon abgeleitet ist, dann funktioniert das natürlich, denn der instanziierende Typ ist ja der Typ des formalen Parameters bei m, und ein Double-Objekt kann eben einer Number zugewiesen werden.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

Empfehlungen (Oracle)

- In-Parameter → extends
- Out-Parameter → super
- In/Out-Parameter → weder noch
- Rückgabe → weder noch

Wir müssen noch kurz diskutieren, wann wir sinnvollerweise extends, wann super und wann keines von beiden verwenden sollten.

Das hier Gesagte orientiert sich an den „Guidelines for Wildcard Use“ in der Oracle-Dokumentation des Sprache Java. Die Begriffe In-Parameter und Out-Parameter sind an die dort verwendeten Begriffe angelehnt.

***Vorgriff*: Im Kapitel 07 greifen wir dieses Thema nochmals auf Basis des dort dann vorhandenen Beispielmaterials auf.**

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

Empfehlungen (Oracle)

- In-Parameter → extends
- Out-Parameter → super
- In/Out-Parameter → weder noch
- Rückgabe → weder noch

In-Parameter einer Methode sind solche, deren Werte in der Methode nur gelesen, aber nicht überschrieben werden.

In unserem Beispiel für extends war der Parameter n tatsächlich ein In-Parameter: Der Wert von n.attribute wurde nur gelesen, nicht gesetzt. Auch bei dem Beispiel, in dem das Fragezeichen ohne ein extends verwendet wurde, war der Zugriff rein lesend: durch die Methode toString. Unsere beiden Beispiele erfüllen also die Empfehlung von Oracle, und wir haben die Sinnhaftigkeit dieser Empfehlung im Grunde schon anhand der beiden Beispiele diskutiert.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

Empfehlungen (Oracle)

- **In-Parameter → extends**
- **Out-Parameter → super**
- **In/Out-Parameter → weder noch**
- **Rückgabe → weder noch**

Beim Beispiel für super anstelle von extends wurde nichts gelesen, sondern nur geschrieben, der Parameter x war also ein Out-Parameter. Auch hier hatten wir die Empfehlung von Oracle anhand des Beispiels im Grunde schon diskutiert.

.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

Empfehlungen (Oracle)

- **In-Parameter → extends**
- **Out-Parameter → super**
- **In/Out-Parameter → weder noch**
- **Rückgabe → weder noch**

Wenn ein Typparameter bei lesendem Zugriff nach oben und bei schreibendem Zugriff nach unten beschränkt sein sollte, dann folgt logisch zwingend, dass der Typparameter sowohl nach oben als auch nach unten beschränkt werden sollte in Methoden, in denen sowohl lesend als auch schreibend auf Objekte des Typparameters zugegriffen wird. Mit anderen Worten: Nur die Instanziierung mit einem einzigen Typ bleibt übrig, so wie wir es vor dem Abschnitt zu Wildcards als einzige Möglichkeit gesehen hatten.

Wildcards

TECHNISCHE UNIVERSITÄT DARMSTADT

Empfehlungen (Oracle)

- **In-Parameter → extends**
- **Out-Parameter → super**
- **In/Out-Parameter → weder noch**
- **Rückgabe → weder noch**

In den Beispielen hatten wir nur Parameter von Methoden betrachtet. Wildcards funktionieren im Prinzip auch bei Rückgaben. Oracle empfiehlt allerdings, bei Rückgaben darauf zu verzichten, da die Verwendung einer solchen Methode schnell kompliziert werden kann. Darauf gehen wir hier nicht näher ein.

TECHNISCHE UNIVERSITÄT DARMSTADT

Comparator

Als nächstes ein Interface, das für sich genommen absolut kennenswert ist, sich aber auch als Fallbeispiel für das Verwenden von Generics anbietet.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class MyNumberComparator
  < T extends Number > implements Comparator<T> {
     public int compare ( T t1, T t2 ) {
          ..........
     }
}
```

Im Package java.util findet sich ein Functional Interface namens Comparator, also auf Deutsch soviel wie „Vergleicher“. Dieses Interface wird sehr häufig verwendet, vor allem im Zusammenhang mit dem Thema von Kapitel 07, Collections. Interface Comparator ist generisch und hat genau einen Typparameter. Dieses Interface lassen wir nun durch eine generische Klasse namens MyNumberComparator implementieren, um den Zweck von Comparator zu verstehen.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class MyNumberComparator
  < T extends Number > implements Comparator<T> {
    public int compare ( T t1, T t2 ) {
        ..........
    }
}
```

Den Typparameter schränken wir aber ein durch die uns bekannte Klasse Number aus Package java.lang.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class MyNumberComparator
  < T extends Number > implements Comparator<T> {
    public int compare ( T t1, T t2 ) {
          ..........
    }
}
```

Interface Comparator hat eine Methode compare, die zwei Parameter vom generischen Typparameter bekommt und ein int zurückliefert. Genauer gesagt, soll 0 zurückgeliefert werden, wenn die beiden Objekte als äquivalent anzusehen sind; eine negative Zahl, wenn der Wert des ersten Objektes als dem des zweiten Objektes vorangehend anzusehen ist; und eine positive Zahl, wenn der Wert des ersten Objektes als dem Wert des zweiten Objektes nachfolgend anzusehen ist. Wir werden bald sehen, warum dieser Satz so merkwürdig kompliziert formuliert ist, also warum wir nicht einfach von „kleiner“, „größer“ und „gleich“ sprechen.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int compare ( T t1, T t2 ) {
    double d1 = t1.doubleValue();
    double d2 = t2.doubleValue();
    if ( d1 < d2 )
        return -1;
    if ( d1 > d2 )
        return +1;
    return 0;
}
```

Der Inhalt der Methode compare passte nicht auch noch auf die Folie, daher nun auf einer eigenen Folie im Detail.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int compare ( T t1, T t2 ) {
   double d1 = t1.doubleValue();
   double d2 = t2.doubleValue();
   if ( d1 < d2 )
      return -1;
   if ( d1 > d2 )
      return +1;
   return 0;
}
```

Wir haben schon gesehen, dass Klasse Number – und somit alle von Number abgeleiteten Klassen wie Integer und Double – eine Methode doubleValue hat. Die Rückgabe ist von der Logik her immer dieselbe: Der Wert, der im Objekt gespeichert ist, wird intern nach double konvertiert und zurückgeliefert (bei Klasse Double entfällt natürlich die Konversion).

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int compare ( T t1, T t2 ) {
   double d1 = t1.doubleValue();
   double d2 = t2.doubleValue();
   if ( d1 < d2 )
      return -1;
   if ( d1 > d2 )
      return +1;
   return 0;
}
```

Wie gesagt, soll in allen Implementationen von Comparator eine negative Zahl zurückgeliefert werden, falls der erste Parameter als dem zweiten vorangehend anzusehen ist. Wenn wir wie hier den gewöhnlichen numerischen Größenvergleich realisieren wollen, heißt das: falls der erste Parameter *kleiner* als der zweite ist.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int compare ( T t1, T t2 ) {
   double d1 = t1.doubleValue();
   double d2 = t2.doubleValue();
   if ( d1 < d2 )
      return -1;
   if ( d1 > d2 )
      return +1;
   return 0;
}
```

Hingegen soll ein positiver Wert herauskomme, falls der erste Parameter als dem zweiten Parameter nachfolgend anzusehen ist, beim gewöhnlichen numerischen Größenvergleich also: falls der erste Parameter *größer* als der zweite ist.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int compare ( T t1, T t2 ) {
   double d1 = t1.doubleValue();
   double d2 = t2.doubleValue();
   if ( d1 < d2 )
      return -1;
   if ( d1 > d2 )
      return +1;
   return 0;
}
```

Und schließlich soll 0 zurückgeliefert werden im dritten, verbliebenen Fall, also wenn beide Parameter als äquivalent anzusehen sind, was beim gewöhnlichen numerischen Größenvergleich eben Wertgleichheit von d1 und d2 bedeutet.

***Erinnerung*: Im Kapitel 03b hatten wir die Unterscheidung zwischen Objektidentität und Wertgleichheit im gleichnamigen Abschnitt herausgearbeitet.**

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
MyNumberComparator<Number> cmp1
    = new MyNumberComparator<Number> ();
MyNumberComparator<Float> cmp2
    = new MyNumberComparator<Float> ();
MyNumberComparator<Byte> cmp3
    = new MyNumberComparator<Byte> ();
MyNumberComparator<Integer> cmp4
    = new MyNumberComparator<Integer> ();
```

Wir können nun Variablen und Objekte dieser Klasse MyNumberComparator mit verschiedenen festen Referenztypen als Instanziierungen einrichten. Wir wissen aber, abgesehen von Number selbst müssen alle diese Typen direkt oder indirekt von Number abgeleitet sein.

Den vierten Fall, also Klasse Integer, nehmen wir auf der folgenden Folie als Beispiel für die weiteren Betrachtungen.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Integer n1 = new Integer ( 123 );
Integer n2 = new Integer ( 321 );
if ( cmp4.compare(n1,n2) < 0 )
   System.out.print ( “Wie erwartet :-)“ );
else
   System.out.print ( “Unerwartet!“ );
```

Dazu richten wir also zwei Variable und Objekte von Klasse Integer ein. Der Wert von n1 ist kleiner als der von n2.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Integer n1 = new Integer ( 123 );
Integer n2 = new Integer ( 321 );

if ( cmp4.compare(n1,n2) < 0 )
   System.out.print ( “Wie erwartet :-)“ );
else
   System.out.print ( “Unerwartet!“ );
```

Hier vergleichen wir n1 und n2 mit Hilfe des passenden Comparators von der letzten Folie. Dieser Vergleich wird natürlich true zurückliefern, so haben wir Methode compare von MyNumberComparator ja implementiert.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class MyReverseNumberComparator
  < T extends Number > implements Comparator<T> {
    private MyNumberComparator mnc = new MyNumberComparator();
    public int compare ( T t1, T t2 ) {
        return mnc.compare ( t2, t1 );
    }
}
```

Wir kommen nun erstmals dazu, warum die Formulierungen, was Methode compare zurückliefern soll, so merkwürdig kompliziert waren.

Aber zunächst einmal eine kleine Frage: Klassen wie Number und davon abgeleitet implementieren ein Interface Comparable und haben dadurch eine Methode compareTo, die genau den Größenvergleich leistet, den auch MyNumberComparator bietet. Eigentlich alle Klassen in der Standardbibliothek, bei denen solche Vergleiche Sinn machen, implementieren Comparable. Warum also überhaupt daneben noch ein separates Interface Comparator?

Die Antwort hat mit diesen merkwürdigen Formulierungen zu tun: Interface Comparator erlaubt es, den Vergleich der Elemente eines Referenztyps auf verschiedene Art zu definieren, so dass in jedem Kontext die passende Definition der Reihenfolge in Form einer spezifischen Comparator implementierenden Klasse verwendet werden kann.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class MyReverseNumberComparator
  < T extends Number > implements Comparator<T> {
    private MyNumberComparator mnc = new MyNumberComparator();
    public int compare ( T t1, T t2 ) {
        return mnc.compare ( t2, t1 );
    }
}
```

***Vorgriff*: Im Kapitel 07 zu Collections werden wir zum Beispiel Sortieren von Listen betrachten. Die entsprechende Methode hat einen Comparator als Parameter, der angibt, nach welcher Logik zwei Elemente zu vergleichen sind, woraus sich ergibt, nach welcher Logik die Liste sortiert wird.**

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class MyReverseNumberComparator
  < T extends Number > implements Comparator<T> {
    private MyNumberComparator mnc = new MyNumberComparator();
    public int compare ( T t1, T t2 ) {
        return mnc.compare ( t2, t1 );
    }
}
```

Um den Gedanken hinter Interface Comparator zu illustrieren, implementieren wir Comparator noch ein zweites Mal, aber diesmal mit umgekehrter Logik: Die größere von zwei Zahlen soll der anderen vorangehen. Steckt man ein Objekt von MyReverseNumberComparator in die angekündigte, später besprochene Sortiermethode, dann wird die Liste *absteigend* sortiert; mit der soeben implementierten Klasse MyNumberComparator wird sie *aufsteigend* sortiert.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class MyReverseNumberComparator
  < T extends Number > implements Comparator<T> {
    private MyNumberComparator mnc = new MyNumberComparator();
    public int compare ( T t1, T t2 ) {
        return mnc.compare ( t2, t1 );
    }
}
```

Diese Klasse lässt sich jetzt sehr schnell schreiben: einfach ein MyNumberComparator als privates Attribut und alle Aufrufe von compare an dieses Attribut delegieren, ...

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class MyReverseNumberComparator
  < T extends Number > implements Comparator<T> {
    private MyNumberComparator mnc = new MyNumberComparator();
    public int compare ( T t1, T t2 ) {
        return mnc.compare ( t2, t1 );
    }
}
```

... aber mit umgedrehter Parameterliste, t2 vor t1, um den Größenvergleich *umzukehren*.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class MyReverseComparator <T> implements Comparator<T> {
    private Comparator<T> cmp;
    public MyReverseComparator ( Comparator<T> cmp ) {
        this.cmp = cmp;
    }
    public int compare ( T t1, T t2 ) {
        return cmp.compare ( t2, t1 );
    }
}
```

Wir können sogar noch einen Schritt weitergehen und den Comparator, dessen Wirken wir umdrehen wollen, offenhalten. Dann spricht auch nichts dagegen, den generischen Typ ganz offen zu halten, also nicht mehr wie eben bei MyReverseNumberComparator auf Number und die Subtypen von Number zu beschränken.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Student {
    public String lastName;
    public String firstName;
    public int enrollmentNumber;
}
```

Wir lösen uns jetzt von numerischen Größenvergleichen und betrachten Anwendungsmöglichkeiten von Comparator, die ganz anders geartet sind, um einen Eindruck von der wahren Ausdrucksstärke von Comparator zu bekommen. Damit wird dann hoffentlich auch endgültig klar werden, warum die Formulierungen zur Spezifikation der Rückgabe von Methode compare so gewunden sind.

Als Beispiel richten wir dazu eine Klasse ein, die die Stammdaten für einen Studenten oder eine Studentin enthält. Hier in diesem kleinen Beispiel natürlich nur eine kleine Auswahl aller Stammdaten.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StudentNameComparator
implements Comparator <Student> {
   public int compare ( Student s1, Student s2 ) {
      if ( s1.lastName.compareTo(s2.lastName) == 0 )
         return s1.firstName.compareTo(s2.firstName);
      return s1.lastName.compareTo(s2.lastName);
   }
}
```

Für diese Klasse implementieren wir nun *zwei* Comparator-Klassen, die Studierende nach völlig unterschiedlichen Kriterien ordnen werden. Hier die erste der beiden Comparator-Klassen.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StudentNameComparator
implements Comparator <Student> {
   public int compare ( Student s1, Student s2 ) {
      if ( s1.lastName.compareTo(s2.lastName) == 0 )
         return s1.firstName.compareTo(s2.firstName);
      return s1.lastName.compareTo(s2.lastName);
   }
}
```

Mit Comparator-Klasse ist natürlich wie bisher gemeint: Diese Klasse implementiert Interface Comparator. Genauer gesagt, in diesem Fall Comparator vom Typ Student.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StudentNameComparator
implements Comparator <Student> {
   public int compare ( Student s1, Student s2 ) {
       if ( s1.lastName.compareTo(s2.lastName) == 0 )
          return s1.firstName.compareTo(s2.firstName);
       return s1.lastName.compareTo(s2.lastName);
   }
}
```

Wir müssen also eine compare-Methode mit zwei Student-Parametern implementieren. Wie der Name StudentNameComparator schon andeutet, sollen die Namen der beiden Studierenden verglichen werden, und zwar in der üblichen Telefonbuchordnung. Das heißt: Erst werden die Nachnamen lexikographisch verglichen, und nur wenn die Nachnamen gleich sind, werden die Vornamen lexikographisch verglichen. Sind Vorname *und* Nachname identisch, kann *dieser* Comparator die beiden Studierenden nicht mehr unterscheiden.

Für viele Zwecke ist es auch nicht notwendig, zwei Studierende voneinander zu unterscheiden, die denselben Vor- und Nachnamen haben, zum Beispiel wenn eine Liste von Studierenden nach ihren Namen in Telefonbuchordnung zu sortieren sind und es beim Ergebnis nicht darauf ankommt, welcher von zwei Studierenden desselben Namens als erstes in der Sortierung kommt.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StudentNameComparator
implements Comparator <Student> {
   public int compare ( Student s1, Student s2 ) {
      if ( s1.lastName.compareTo(s2.lastName) == 0 )
         return s1.firstName.compareTo(s2.firstName);
      return s1.lastName.compareTo(s2.lastName);
   }
}
```

Als erstes vergleichen wir also die Nachnamen der beiden Studierenden. Dazu nutzen wir aus, dass die Klasse String eine Methode compareTo mit einem Parameter vom Typ String hat. Diese Methode liefert nach derselben Logik wie bei Comparator eine negative Zahl zurück, falls s1 lexikographisch *vor* s2 kommt, eine positive Zahl, falls s1 lexikographisch *nach* s2 kommt, und 0, falls beide Strings identisch sind im Sinne von Wertgleichheit, das heißt, dieselben Zeichen in derselben Reihenfolge, aber in der Regel unterschiedliche String-Objekte.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StudentNameComparator
implements Comparator <Student> {
   public int compare ( Student s1, Student s2 ) {
      if ( s1.lastName.compareTo(s2.lastName) == 0 )
         return s1.firstName.compareTo(s2.firstName);
      return s1.lastName.compareTo(s2.lastName);
   }
}
```

Falls also die beiden Nachnamen gleich sind, werden auf dieselbe Art die beiden Vornamen verglichen, und die lexikographische Reihenfolge der beiden Vornamen entscheidet die Reihung der beiden Gesamtnamen.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StudentNameComparator
implements Comparator <Student> {
   public int compare ( Student s1, Student s2 ) {
      if ( s1.lastName.compareTo(s2.lastName) == 0 )
         return s1.firstName.compareTo(s2.firstName);
      return s1.lastName.compareTo(s2.lastName);
   }
}
```

Und falls die beiden Nachnamen *nicht* gleich sind, entscheidet eben die lexikographische Reihenfolge der beiden *Nach*namen.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class EnrollmentNumberComparator
implements Comparator <Student> {
    public int compare ( Student s1, Student s2 ) {
        if ( s1.enrollmentNumber < s2.enrollmentNumber )
            return −1;
        if ( s1.enrollmentNumber > s2.enrollmentNumber )
            return +1;
        return 0;
    }
}
```

Jetzt die angekündigte zweite Comparator-Klasse für Klasse Student. Diese vergleicht jetzt nicht die *Namen*, sondern die *Matrikelnummern*.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class EnrollmentNumberComparator
implements Comparator <Student> {
    public int compare ( Student s1, Student s2 ) {
        if ( s1.enrollmentNumber < s2.enrollmentNumber )
            return −1;
        if ( s1.enrollmentNumber > s2.enrollmentNumber )
            return +1;
        return 0;
    }
}
```

Wieder Methode compare mit zwei Parametern vom Typ Student.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class EnrollmentNumberComparator
implements Comparator <Student> {
    public int compare ( Student s1, Student s2 ) {
        if ( s1.enrollmentNumber < s2.enrollmentNumber )
            return −1;
        if ( s1.enrollmentNumber > s2.enrollmentNumber )
            return +1;
        return 0;
    }
}
```

Die Matrikelnummer ist vom primitiven Datentyp int. Daher können die Matrikelnummern der beiden Studierenden ohne Umstände miteinander verglichen werden. Falls die Matrikelnummer des ersten Studierenden kleiner als die des zweiten ist, wird also –1 zurückgeliefert ...

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class EnrollmentNumberComparator
implements Comparator <Student> {
    public int compare ( Student s1, Student s2 ) {
        if ( s1.enrollmentNumber < s2.enrollmentNumber )
            return −1;
        if ( s1.enrollmentNumber > s2.enrollmentNumber )
            return +1;
        return 0;
    }
}
```

... und +1, falls die Matrikelnummer des ersten Studierenden größer als die des zweiten ist.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class EnrollmentNumberComparator
implements Comparator <Student> {
    public int compare ( Student s1, Student s2 ) {
        if ( s1.enrollmentNumber < s2.enrollmentNumber )
            return −1;
        if ( s1.enrollmentNumber > s2.enrollmentNumber )
            return +1;
        return 0;
    }
}
```

Schließlich 0, falls beide Matrikelnummern identisch sind. Sollte jede Matrikelnummer wirklich nur einmal vergeben worden sein, wie es sich gehört, dann kann dieser Fall also nur eintreten, wenn ein Studierender mit sich selbst verglichen wird. Trotzdem wird der Compiler verlangen, dass auch im Fall, dass keine der beiden if-Abfragen true liefert, eine return-Anweisung ausgeführt wird – selbst in Situationen, in denen wir uns sicher sind, dass dieser Fall nie eintreten wird.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class AbsoluteErrorComparator implements Comparator <Double> {
    private double epsilon;
    public AbsoluteErrorComparator ( double epsilon ) {
        this.epsilon = epsilon;
    }
    public int compare ( Double x1, Double x2 ) {
        if ( x1 + epsilon < x2 )
          return −1;
        if ( x2 + epsilon < x1 )
          return +1;
        return 0;
    }
}
```

Zum Abschluss der Besprechung von Comparator noch ein Beispiel, das berücksichtigt, dass double-Werte nicht unbedingt exakt dargestellt werden können, so dass zwei beinahe gleich große Werte als gleich anzusehen sind.

***Erinnerung*: Im Kapitel04a hatten wir diese Problematik schon einmal kurz aufgegriffen, Stichwort check-within.**

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class AbsoluteErrorComparator implements Comparator <Double> {
    private double epsilon;
    public AbsoluteErrorComparator ( double epsilon ) {
        this.epsilon = epsilon;
    }
    public int compare ( Double x1, Double x2 ) {
        if ( x1 + epsilon < x2 )
          return −1;
        if ( x2 + epsilon < x1 )
          return +1;
        return 0;
    }
}
```

In der numerischen Mathematik unterscheidet man den absoluten und den relativen Fehler bei der Prüfung, ob zwei reelle Zahlen als gleich anzusehen sind, auch wenn sie sich in der soundsovielten Nachkommastelle unterscheiden. Wir betrachten hier nur beispielhaft den absoluten Fehler; ein Comparator für den relativen Fehler wäre praktisch identisch, nur etwas komplizierter.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class AbsoluteErrorComparator implements Comparator <Double> {
    private double epsilon;
    public AbsoluteErrorComparator ( double epsilon ) {
        this.epsilon = epsilon;
    }
    public int compare ( Double x1, Double x2 ) {
        if ( x1 + epsilon < x2 )
          return −1;
        if ( x2 + epsilon < x1 )
          return +1;
        return 0;
    }
}
```

Traditionell steht in der Mathematik der kleine griechische Buchstabe epsilon für Fehlerschranken.

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class AbsoluteErrorComparator implements Comparator <Double> {
    private double epsilon;
    public AbsoluteErrorComparator ( double epsilon ) {
        this.epsilon = epsilon;
    }
    public int compare ( Double x1, Double x2 ) {
        if ( x1 + epsilon < x2 )
          return −1;
        if ( x2 + epsilon < x1 )
          return +1;
        return 0;
    }
}
```

***Erinnerung*: Im Abschnitt zu Wrapper-Klassen am Beginn dieses Kapitels haben wir Unboxing gesehen: Wo ein Wert des primitiven Datentyps double erwartet wird, kann auch eine Referenz von Klasse Double stehen.**

Comparator

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class AbsoluteErrorComparator implements Comparator <Double> {
    private double epsilon;
    public AbsoluteErrorComparator ( double epsilon ) {
        this.epsilon = epsilon;
    }
    public int compare ( Double x1, Double x2 ) {
        if ( x1 + epsilon < x2 )
          return −1;
        if ( x2 + epsilon < x1 )
          return +1;
        return 0;
    }
}
```

Das ist jetzt der entscheidende Punkt: Gemäß absoluter Fehlermessung unterscheiden sich zwei reelle Zahlen nur dann, wenn sie sich um mindestens epsilon unterscheiden.

Damit ist das Thema Comparator erst einmal beendet. Wir werden Comparator in Kapitel 07 wiedersehen.

TECHNISCHE UNIVERSITÄT DARMSTADT

Einschränkungen bei Generics in Java

Oracle Java documentation: Restrictions on Generics

Generics in Java sind leider an einigen Stellen nicht so uneingeschränkt nutzbar wie die entsprechenden Sprachkonstrukte in anderen Programmiersprachen wie beispielsweise C++.

***Nebenbemerkung*: Der Grund ist, dass der Java Bytecode keine Typinformationen enthält, diese aber in manchen Situationen gebraucht wird. Solche Situationen gehen dann eben in Java nicht. Generics sind relativ spät in Java hineingekommen, als eine Änderung der Spezifikation des Java Bytecode überhaupt nicht mehr zur Diskussion stand.**

Einschränkungen bei Generics

TECHNISCHE UNIVERSITÄT DARMSTADT

- Keine primitiven Datentypen als Instanziierungen von Typparametern
- Keine Erzeugung von Objekten / Arrays von Typparametern mit Operator new
- Keine Klassenattribute von Typparametern
- Kein Downcast oder instanceof mit Typparametern
- Kein throw-catch mit Typparametern
- Keine Methodenüberladung auf Typparametern

Diese Aufzählung lehnt sich an den Artikel „Restrictions on Generics“ in der Oracle-Dokumentation von Java an.

Einschränkungen bei Generics

TECHNISCHE UNIVERSITÄT DARMSTADT

- Keine primitiven Datentypen als Instanziierungen von Typparametern
- Keine Erzeugung von Objekten / Arrays von Typparametern mit Operator new
- Keine Klassenattribute von Typparametern
- Kein Downcast oder instanceof mit Typparametern
- Kein throw-catch mit Typparametern
- Keine Methodenüberladung auf Typparametern

Dieser Einschränkung waren wir schon zu Beginn dieses Kapitels begegnet: Nur Referenztypen, also Klassen, Interfaces und Arraytypen, können Typparameter instanziieren.

Einschränkungen bei Generics

TECHNISCHE UNIVERSITÄT DARMSTADT

- Keine primitiven Datentypen als Instanziierungen von Typparametern
- Keine Erzeugung von Objekten / Arrays von Typparametern mit Operator new
- Keine Klassenattribute von Typparametern
- Kein Downcast oder instanceof mit Typparametern
- Kein throw-catch mit Typparametern
- Keine Methodenüberladung auf Typparametern

Operator new ist auf Typparameter nicht anwendbar. Das betrifft einerseits einzelne Objekte, andererseits ganze Arrays.

Einschränkungen bei Generics

TECHNISCHE UNIVERSITÄT DARMSTADT

- Keine primitiven Datentypen als Instanziierungen von Typparametern
- Keine Erzeugung von Objekten / Arrays von Typparametern mit Operator new
- Keine Klassenattribute von Typparametern
- Kein Downcast oder instanceof mit Typparametern
- Kein throw-catch mit Typparametern
- Keine Methodenüberladung auf Typparametern

Weder static-Variable noch -konstante von einem Typparameter sind möglich.

Einschränkungen bei Generics

- Keine primitiven Datentypen als Instanziierungen von Typparametern
- Keine Erzeugung von Objekten / Arrays von Typparametern mit Operator new
- Keine Klassenattribute von Typparametern
- Kein Downcast oder instanceof mit Typparametern
- Kein throw-catch mit Typparametern
- Keine Methodenüberladung auf Typparametern

Auch nicht Downcasts auf Typparameter oder vorherige Typabfragen mit instanceof.

Einschränkungen bei Generics

TECHNISCHE UNIVERSITÄT DARMSTADT

- Keine primitiven Datentypen als Instanziierungen von Typparametern
- Keine Erzeugung von Objekten / Arrays von Typparametern mit Operator new
- Keine Klassenattribute von Typparametern
- Kein Downcast oder instanceof mit Typparametern
- Kein throw-catch mit Typparametern
- Keine Methodenüberladung auf Typparametern

Prinzipiell lassen sich Typparameter auch mit Exception-Klassen instanziieren. Aber sie können dann nicht auf die übliche Weise als Exceptions verwendet werden mit throw und catch.

Einschränkungen bei Generics

TECHNISCHE UNIVERSITÄT DARMSTADT

- Keine primitiven Datentypen als Instanziierungen von Typparametern
- Keine Erzeugung von Objekten / Arrays von Typparametern mit Operator new
- Keine Klassenattribute von Typparametern
- Kein Downcast oder instanceof mit Typparametern
- Kein throw-catch mit Typparametern
- Keine Methodenüberladung auf Typparametern

Und schließlich: Typparameter reichen nicht zur Unterscheidung zweier Methoden desselben Namens mit ansonsten gleicher Parameterliste.

Damit ist dieser Abschnitt und das ganze Kapitel beendet.