



TECHNISCHE
UNIVERSITÄT
WIEN
Vienna | Austria



Institut für
Computertechnik
Institute of
Computer Technology

Generics in Java

Thomas Rathfux
Ralph Hoch

Überblick

- Vorteile von Generics
- Generische Klassen und Interfaces
- Generizität und Vererbung
- Type-Bounding und Wildcards
- Wildcards bei Collections
- Beispiele für Type-Bounding
- Typisierte statische Methoden
- Diamond Operator
- Generics und Untertypen





Vorteile von Generics

Vorteile von Generics

- Generische Klassen bieten eine Implementierung, die unabhängig vom tatsächlich verwendeten Typ ist.
- Parametrisierbare Klassen können dazu führen, dass Source-Code wiederverwendet werden kann.
- Durch explizite Typinformation kann der Compiler stärkere Typüberprüfung vornehmen (siehe List vs. List<E>)
- Typische Anwendungen:
 - Datenstrukturen: Listen, Baumstrukturen, Sets, Maps, ...
 - Algorithmen
 - Functional Interfaces
 -



Generische Klassen und Interfaces

Generics / Generizität

- Einführung zusätzliche Variablen für Typen, sogenannte Typ-Parameter.
- An Stelle expliziter Typen werden im Programm Typ-Parameter verwendet.
- Generics können bei Klassen, Abstrakte Klassen, Interfaces, Enumerations und Methoden verwendet werden.
- Typ-Parameter sind nur Namen, die bei der Objekt-Instanziierung durch konkrete Typen ersetzt werden.

Gründe für die Verwendung von Generics

- Generizität bietet statische Typsicherheit!
- Steigerung der Flexibilität von Modularisierungseinheiten.
- Sinnvoller Einsatz kann die Wartbarkeit verbessert.
- Wiederverwendbarkeit wenn die Implementierung unabhängig vom Typ der Elemente möglich ist (z.B. Bibliotheken wie Apache Commons).
Effiziente Implementierung von Datenstrukturen (Collections, Container, ...).
- Generizität kann dynamischen Typfehler beseitigen und gleichzeitig die Lesbarkeit von Programmen verbessern.

Beispiele

- Beispiele für generische Klassen und Interfaces aus den Java Klassenbibliotheken:

Raw Type	Generic Type
Collection	Collection<E>
List	List<E>
Iterable	Iterable<E>
Comparable	Comparable<T>
HashMap	HashMap<K,V>



Beispiel Source-Code

```
// Liste ohne Generics
List rawList = new ArrayList();
rawList.add( „Raw Type List" );
Object obj = rawList.get(0);
String s = (String) obj;
```

```
// Liste mit Generics
List<String> genericList = new ArrayList<String>();
genericList.add( „Generic List" );
String s = genericList.get(0);
```



Beispiel Source-Code

```
// Liste ohne Generics
List rawList = new ArrayList();

// Runtime-Exception
rawList.add( new Integer(42) );
String s = (String) rawList.get(0);
```

```
// Liste mit Generics
List<String> genericList = new ArrayList<String>();

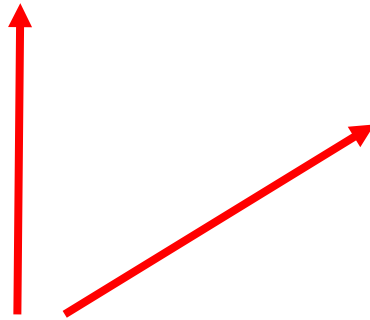
// Compilation-Error
genericList.add(new Integer(42));
String s = genericList.get(0);
```

Generische Klasse (parametrisierte Klasse)

```
public class Vehicle<T> {  
    private T engine = null;  
    private String name = null;  
  
    public Vehicle(String name) {  
        this.name = name;  
    }  
  
    // parameterized get- and set- Methods  
    public T getEngine() {  
        return engine;  
    }  
  
    public void setEngine(T engine) {  
        this.engine = engine;  
    }  
}
```

Instanziierung einer generischen Klasse

```
public class App {
    public static void main(String[] args) {
        Vehicle< IV12Engine > muscleCar = new Vehicle< IV12Engine > ();
        Vehicle< IV6Engine > car = new Vehicle< IV6Engine > ();
    }
}
```



Bei der Erzeugung des Objekts wird die Typisierung der Instanz der Klasse Vehicle festgelegt.

Alle Variablen vom Typ T werden bei der Instanziierung mit dem Typ IV6Engine ersetzt.

```
public class Vehicle<T> {
    private T engine = null;
    private String name = null;

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

    // parameterized get-
    // and set-Methods
    public T getEngine() {
        return engine;
    }

    public void setEngine(T engine) {
        this.engine = engine;
    }
}
```

Beispiele aus der Java-Bibliothek

```
public interface Iterator<T> {  
    public T next();  
    public boolean hasNext();  
}
```

```
public interface Collection<E> {  
    public boolean add(E e);  
    public boolean addAll( Collection <? extends E> c );  
    public void clear();  
    public boolean contains( Object o );  
    public boolean containsAll( Collection< ? > c );  
    public Iterator<E> iterator( );  
    // ...  
    // other abstract methods see: java.util.Collection  
}
```

Zusammenfassung

- Eine Klassen-Implementierung für mehrere Typen.
- Eine Klasse kann mehrere generische Typen haben (z.B. `HashMap<K, V>`).
- Typvariablen werden bei der Instanziierung des Objektes gesetzt.
- Ermöglicht Typenprüfung durch den Compiler.
- Generics sind ab der Java Version 1.5 elementarer Bestandteil der Sprache.



Generizität und Vererbung

Generische Klassen - Vererbung

```
public class MySubClass< T > extends MySuperClass< T >{  
    // constructors  
    // methods  
    // Type parameter T is passed along.  
}
```

```
public class MySubClass< S, V > extends MySuperClass< V >{  
    public S memberVariable = null;  
    // constructors  
    // methods  
    // Type parameter V is passed along.  
    // Type of member variable defined by type parameter S.  
}
```


Generische Klassen - Vererbung

```
public class MySubClass< T > extends MySuperClass2< T, String >{  
    // constructors  
    // methods  
    // Type parameter T is passed along.  
    // MySuperClass2 is parameterized with  
    // type String for the required second type parameter.  
}
```

```
public class MySubClass< S, V > extends MySuperClass2< S, V >{  
    // constructors  
    // methods  
    // Both type parameters are passed along.  
}
```



Type-Bounding und Wildcards

Nachteile der freien Typisierung

```
public class App {
    public static void main(String[] args) {
        Vehicle< IV12Engine > muscleCar = new Vehicle< IV12Engine > ();
        Vehicle< IV6Engine > car = new Vehicle< IV6Engine > ();
    }
}
```



Die Klasse Vehicle schränkt den Parametertyp in keiner Weise ein. Es ist daher möglich auch ein Vehicle typisiert mit z.B. Number zu erzeugen.

Das typisierte Attribut „engine“ sollte aber eine Referenz auf einen Motor halten. Mit der Instanziierung mit dem Typ Number wird dies verletzt !

```
public class Vehicle<T> {
    private T engine = null;
    private String name = null;

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

    // parameterized get-
    // and set-Methods
    public T getEngine() {
        return engine;
    }

    public void setEngine(T engine) {
        this.engine = engine;
    }
}
```

Type-Bounding

- Einschränkung der freien Typisierung mittels Type-Bounding.
- Die zur Instanziierung erlaubten Typen können mittels Type-Bounding eingeschränkt werden.

Type-Bounding	Wildcard
Beliebiger (unbekannter) Typ	< ? >
Superklasse von [Type]	< ? super [Type] >
Subklasse von [Type]	< ? extends [Type] >
Subklasse von [Type] und [IType1] und [IType2]	< ? extends [Type] & [IType1] & [IType2] >

Type-Bounding

- Wildcards haben unterschiedliche Bedeutung:
 - Klassendeklaration
 - Methodenparametern
 - Variablendeklaration
 - Besonderheiten bei Collections



Type-Bounding bei der Klassendeklaration

```
class [CLASSNAME] < T extends [TYPE] , ... > { }
```

Eingrenzung der Parametrisierung der Klasse auf den Typ [TYPE] oder einen Untertyp davon.

```
public class GClass< T extends Number >{  
    // constructors  
    // methods  
}
```

```
public class App {  
    public static void main(String[] args) {  
        GClass< Integer > demo = new GClass< Integer > ();  
    }  
}
```

Type-Bounding bei der Klassendeklaration

```
class [CLASSNAME] < T super [TYPE] , ... > { }
```

Eingrenzung der Parametrisierung der Klasse auf den Typ [TYPE] oder einen Obertyp davon.

```
public class GClass< T super Integer >{  
    // constructors  
    // methods  
}
```

```
public class App {  
    public static void main(String[] args) {  
        GClass< Number > demo = new GClass< Number > ();  
    }  
}
```

Type-Bounding bei Methodenparametern

```
public void [METHOD] (Class< ? extends [TYPE] > param, ... ) { }
```

Zusicherung, dass der übergebene Typparameter mindestens vom Typ [TYPE] ist (= obere Schranke) und dessen Verhalten unterstützt.

```
public class GClass {  
    // Kontrollfrage: Warum muss der Typ des Attributes  
    // IEngine sein ?  
    private IEngine engine = null;  
  
    public void setEngine( Vehicle<? extends IEngine > vehicle){  
        this.engine = vehicle.getEngine();  
    }  
}
```


Type-Bounding bei Methodenparametern

```
public void [METHOD] (Class< ? super [TYPE] > param, ... ) { }
```

Zusicherung, dass der übergebene Typ-Parameter maximal vom Typ [TYPE] ist (= untere Schranke). Verhalten wird nur durch die Basisklasse Object zugesichert.

```
public class CleaningFacility {  
  
    public void wash( WashBag <? super Trousers > bag) {  
        // Es kann nur ein Waschsack mit Kleidung, die in  
        // der Typhierarchie über Trousers liegt, gewaschen werden.  
        // Ein Wachsack typisiert mit einer von Trousers  
        // abgeleiteten Klasse Stoffhose (TextileTrousers) kann  
        // nicht übergeben werden.  
    }  
  
}
```

Type-Bounding bei Methodenparametern

```
public void [METHOD] (Class< ? > param, ... ) { }
```

Der übergebene Parameter hat eine nicht definierte Parametrisierung. Es ist keine Zusicherung gegeben, außer dass es sich dabei um ein *Object* handeln muss (Java spezifisch).

Verwendung wenn der Parametertyp für die Ausführung der Methode nicht relevant ist.

```
public class CleaningFacility {  
    public void print( WashBag < ? > bag) {  
        System.out.println( element.toString() );  
    }  
}
```

Type-Bounding bei der Variablendeklaration

```
Class< ? extends [TYPE] > variableName = ...;
```

Die Variable kann nur Objekte referenzieren, die mit einer Typisierung [TYPE] oder eines Subtyps davon instanziiert wurde.

```
public class GClass {  
    public void main ( String args[]){  
        Vehicle<? extends IEngine> car = new Vehicle<IV12Engine>();  
  
        // Compiler-Fehler bei der folgenden Zuweisung. Warum?  
        car = new Vehicle<Number>();  
    }  
}
```

```
public class Vehicle< T > {  
    // Keine Einschränkung bei der Typisierung von T !  
}
```

Type-Bounding bei der Variablendeklaration

```
Class< ? > variableName = ...;
```

Die Variable kann beliebig typisierte Objekte referenzieren.

```
public class GClass {  
    public void main ( String args[] ) {  
        Vehicle< ? > car = new Vehicle< IEngine > ();  
        car = new Vehicle< Number > ();  
  
        // Was ist der Unterschied zur Variablendeklaration  
        // mit Vehicle< Object > ?  
    }  
}
```

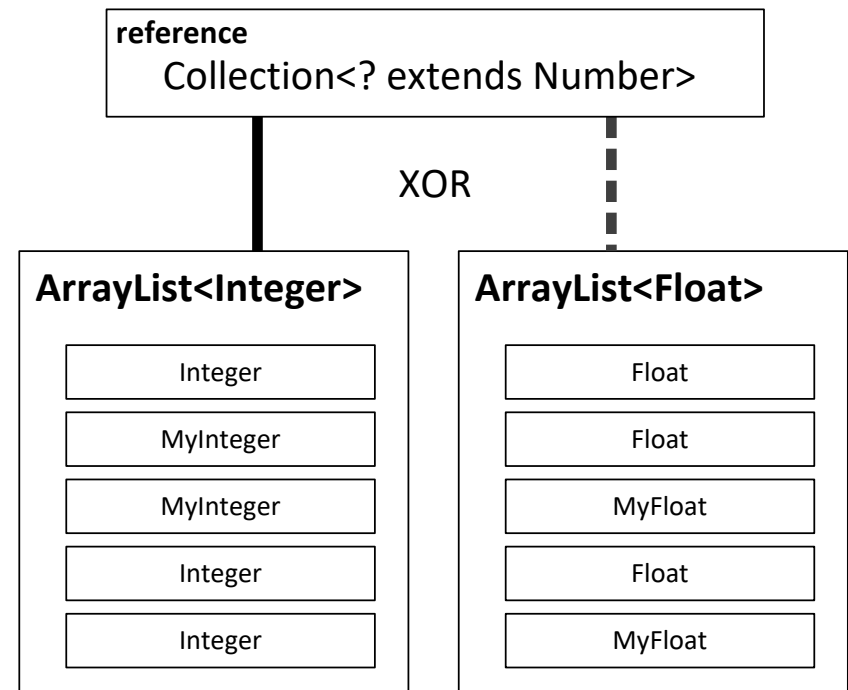
```
public class Vehicle< T > {  
    // Keine Einschränkung bei der Typisierung von T !  
}
```



Wildcards bei Collections

Wildcards bei Collections

- Wildcards schränken die erlaubten Typen in Listen ein.
- Die Listeninstanz ist immer eine homogenen Liste (z.B. `ArrayList<Integer>`).
- Die Referenz kann aber auf unterschiedliche Listen verweisen.
- Der Inhalt der Liste ist homogen bezüglich des Listen-Types (Polymorphismus).



Wildcards bei Collections

... Collection < **[TYPE]** > variableName ...

- Zusicherung eines Typen
- Erlaubt lesen und schreiben
- Zusicherung des Verhaltens vom Bounding-Type

```
public void drawAll(List<Polygon> p) {  
    Rectangle rect = new Rectangle( );  
    p.add( rect );  
    p.remove( rect );  
    // Erlaubt lesen und schreiben auf der Liste p  
    // Allerdings kann nur eine List vom  
    // Typ Polygon übergeben werden aber keine List<Square>  
}
```

Wildcards bei Collections

... Collection < ? **extends** [TYPE] > variableName ...

- Einschränkung auf den gleichen Typ oder dessen Subtypen.
- Nur lesender Zugriff.
- Zusicherung des Verhalten vom Bounding-Type, erlaubt aber Subtypen.

Wildcards bei Collections

... Collection < ? **extends** [TYPE] > variableName ...

```
public void drawAll(List<? extends Polygon> p) {  
  
    // Erlaubt NUR lesen aber nicht schreiben auf der Liste p.  
    // Es kann auch eine Liste von jedem Untertyp vom  
    // Typ Polygon übergeben werden z.B. List<Square>.  
    for (Polygon poly : p) {  
        poly.getArea();  
    }  
  
    // Compiler-Fehler: p könnte auch eine List<Rectangle> sein  
    // und Triangle in keiner passenden Untertyp Beziehung.  
    // Schreiben auf die Liste wird daher durch den Compiler  
    // unterbunden!  
    p.add(new Triangle( ));  
}
```

Wildcards bei Collections

... Collection < ? **super** [TYPE] > variableName ...

- Einschränkung auf den gleichen Typ oder einen Obertyp.
- Nur schreibender Zugriff. Aber dabei nur die Zusicherung, dass ohne Laufzeitprobleme ein [TYPE] oder ein Subtyp davon geschrieben werden kann.
- Die obere Schranke des Typs ist Object.

Wildcards bei Collections

... Collection < ? **super** [TYPE] > variableName ...

```
public void addPolygon(List<? super Polygon> toList) {  
    // Erlaubt für die Liste toList nur schreiben in die Liste.  
    // Es kann auch eine Liste vom Obertyp von Polygon  
    // übergeben werden. z.B. List<Object>  
    toList.add(new Polygon( ));  
    toList.add(new Rectangle( ));  
    // lesender Zugriff auf toList.  
    // toList könnte auch eine List<Object> sein!  
    for (Polygon poly : toList) { // Compiler-Fehler  
        poly.getArea();          // Compiler-Fehler  
    }  
    // Erlaubt ist aber  
    for (Object obj : toList) {  
        obj.toString();  
    }  
}
```

Wildcards bei Collections

... Collection < ? > variableName ...

- Definiert eine Collection von unbekanntem Typ („*Collection of unknown*“).
- Nur die Zusicherung, dass es eine homogene Liste ist.
- Verwendet wenn der Typ nicht relevant ist.
- Kein schreibender Zugriff möglich (bzw. sinnvoll).

Wildcards bei Collections

... Collection < ? > variableName ...

```
public void printAll(List<?> list) {  
    // Lesender Zugriff auf die Liste möglich.  
    // Nur Methoden von Object immer vorhanden.  
    for ( Object obj : list ) {  
        System.out.println( obj.toString() );  
        if ( obj instanceof Polygon ) {  
            ((Polygon)obj).getArea();  
        }  
    }  
  
    // Compiler-Fehler: Einfügen eines Object könnte die  
    // Homogenität der referenzierten Liste zerstören!  
    list.add( new Object() );  
}
```



Beispiele für Type-Bounding

Anwendung von Bounded Types

```
public class App {
    public static void main(String[] args) {
        Vehicle< IV12Engine > muscleCar = new Vehicle< IV12Engine > ();
        // Compiler-Fehler!
        Vehicle< Number > car = new Vehicle< Number > ();
    }
}
```

```
public class Vehicle<T extends IEngine> {
    private T engine = null;
    private String name = null;

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

    // parameterized get-
    // and set-Methods
    public T getEngine() {
        return engine;
    }

    public void setEngine(T engine) {
        this.engine = engine;
    }
}
```

Compiler wird aufgrund der statischen Typenprüfung einen Fehler erkennen!

Da die Klasse Vehicle nur Typen die das Interface IEngine implementieren zulässt, ist der Typ Number für die Instanziierung nicht zulässig.

Beispiel 1

```
public class GSample1< KEY, VALUE, X extends VALUE > {  
  
    private HashMap< KEY, VALUE > mapList = new HashMap<>();  
  
    public List< KEY > retrieveAllKeys() {  
        List< KEY > tempList = new ArrayList<>(mapList.keySet());  
        return tempList;  
    }  
  
    public List< VALUE > retrieveAllValues() {  
        List< VALUE > tempList = new ArrayList<>(mapList.values());  
        return tempList;  
    }  
    // ... more Methods on following slide  
}
```



Beispiel 1

```
public class GSample1< KEY, VALUE, X extends VALUE > {  
  
    private HashMap< KEY, VALUE > mapList = new HashMap<>();  
  
    // ... Methods defined on previous slide  
    public void addKeyValuePair(KEY keyElement,  
                                VALUE valueElement) {  
        mapList.put(keyElement, valueElement);  
    }  
  
    public void add(KEY keyValue, X specialValue) {  
        mapList.putIfAbsent(keyValue, specialValue);  
    }  
}
```

Kontrollfrage: Was passiert wenn X nicht mehr als Bounded-Type VALUE festgelegt ist ?





Typisierte statische Methoden

Typisierte statische Methoden

```
public class HelperUtil {  
  
    public static < TYP extends Comparable< TYP >>  
        void add(List< TYP > list, TYP element1, TYP element2) {  
  
        if (element1.compareTo(element2) > 0) {  
            list.add(element1);  
        } else {  
            list.add(element2);  
        }  
    }  
}
```

Mögliche Anwendung: Alle übergebenen Parameter sollen den gleichen, aber noch unbekannten, Typ haben.

Kontrollfrage: Wie und wann wird der Typ-Parameter bei der statischen Methode festgelegt ?






Diamond Operator

Diamond Operator <>

```
public class HelperUtil {  
  
    public static void main (String args[]) {  
        List<Integer> numbers = new LinkedList<Integer>();  
        List<Integer> diamond = new LinkedList<>();  
    }  
}
```



Typinferenz

Aufgrund der Typinferenz (Typableitung) kann der Typ des zu erzeugenden Objekts vom Compiler automatisch aufgelöst werden.

Im obigen Beispiel wird daher die `LinkedList` typisiert mit `Integer` erzeugt.





Generics und Untertypen

Generics und Untertypen

```
public class Polymorphism {  
  
    public static void main (String args[]) {  
        // Compiler-Fehler: keine impliziten Untertypen!  
        List<Number> numbers = new LinkedList<Integer>();  
    }  
}
```

Number ist eine Superklasse von Integer. List ist eine Superklasse von LinkedList. Man könnte annehmen, dass die Zeile oben fehlerfrei kompiliert.

ABER

Keine impliziten Untertypen. Generizität unterstützt keine impliziten Untertypbeziehungen. So besteht zwischen `List<X>` und `List<Y>` keine Untertypbeziehung wenn X und Y verschieden sind, auch dann nicht, wenn Y von X abgeleitet ist oder umgekehrt.

Generics und Untertypen

```
public class Wildcards {  
  
    public static void main (String args[]) {  
        List< ? > numbers = new LinkedList< Number >();  
        numbers.add(new Integer(112)); // Compiler-Fehler  
        numbers.add(new Long(666));   // Compiler-Fehler  
    }  
}
```

List<?> erzwingt wie List<Number> eine homogene Liste, das heißt die Liste besteht aus Elementen eines Typs bzw. dessen Subtypen. Aber der Typ der Liste *numbers* ist unbekannt. Da der Typ unbekannt ist, darf kein Objekt eines potentiell anderen Typs über add() hinzugefügt werden.

Java **verbietet** daher schreibende Operationen.

Generics und Untertypen

```
public class Wildcards {  
  
    public static void main (String args[]) {  
        List<? extends Number> numbers = new LinkedList<Number>();  
        numbers.add(new Integer(112)); // Compiler-Fehler  
        numbers.add(new Long(666));    // Compiler-Fehler  
    }  
}
```

`List<? extends Number>` erlaubt eine Zuweisung einer Liste von jedem Untertyp von `Number`. Das ursprüngliche Problem wie bei `List<Number>` ist dabei jedoch genauso vorhanden.

Auch hier gilt: **Keine impliziten Untertypen.**

Wozu Wildcards ?

```
public class Wildcards {  
  
    public static void main (String args[]) {  
        List<Number> numbersList1 = new ArrayList<Number>();  
        List<? extends Number> numbersList2 = new ArrayList<Number>();  
        List<Integer> integerList = new ArrayList<Integer>();  
        numbersList2 = integerList;  
        numbersList2 = numbersList1;  
        numbersList1 = integerList;    // Compiler-Fehler  
    }  
}
```

Wildcards ermöglichen eine Zusicherung der Variablentypen zu definieren. Im Fall von *numbersList2* ist also nur zugesichert, dass der Typ der referenzierten Liste zumindest vom Typ `Number` ist. Bei *numberList1* ist der Typ explizit auf `Number` festgelegt.

Object statt Wildcard

```
public class Wildcards {  
  
    public static void main (String args[]) {  
        List<Object> pseudoRawType = new ArrayList<Object>();  
  
        // Erlaubte Zuweisungen über gemeinsamen Obertyp  
        pseudoRawType.add(new Integer(112));  
        pseudoRawType.add(new Long(666));  
        pseudoRawType.add(new String("Hallo"));  
  
        // ...  
        // Compiler-Fehler: nicht erlaubte Zuweisung  
        pseudoRawType = new ArrayList< Number > ();  
  
        // Kontrollfrage: Unterschied List<?> und List<Object>  
  
    }  
}
```

instanceof und Type-Casting

```
public class TypeHiding {  
  
    public static void main (String args[]) {  
        List<Integer> intList = new LinkedList<Integer>();  
        if (intList instanceof List) { }  
  
        // Compiler-Fehler  
        if (intList instanceof List<Integer>) { }  
  
        // Zuweisung möglich aber eventuell treten  
        // zur Laufzeit Runtime-Exceptions auf!  
        List<Number> numbersA = (List) intList;  
  
        // Compiler-Fehler: Java unterstützt keine expliziten  
        // typisierten Casts einer generischen Klasse  
        List<Number> numbersB = (List<Number>) intList;  
    }  
}
```

Der *instanceof*-Operator kann nicht mit parametrisierten Typen verwendet werden. Es ist daher mittels **instanceof** NICHT möglich eine parametertypsichere Prüfung zu machen.

Bücher und Web-Links

Bücher:

Programmieren in Java, Reinhard Schiedermeier, 2. Auflage, ISBN 978-3-86894-031-2

Weblinks:

<https://docs.oracle.com/javase/tutorial/java/generics/types.html>

<http://www.torsten-horn.de/techdocs/java-generics.htm#Wann-Raw-Typen>

http://openbook.rheinwerk-verlag.de/javainsel/javainsel_09_001.html#dodtp1e5aa8bf-acb7-4f0f-b3cf-176e06f73845

