Programmieren II

Informatik (B. Sc.)
2. Semester

Unit 07

Einführung in Generische Datentypen

Prof. Dr. Nane Kratzke

Units

Rekursion

01 Einführung in die rekursive Programmierung
02 Sequenzbasierte Rekursionen
03 Rekursive Datenstrukturen

Funktionale Programmierung

04 Einführung in die funktionale Programmierung
05 Funktionale Programmierung mit Streams
06 Thinking in Filter → Map → Reduce

Generische Datentypen <T>

07 Einführung in Gen. Datentypen (Type Erasure)
08 Bounded Types

Graphical User Interfaces

09 Einführung in Swing (Typvertreter)
10 Model - View - Controller (MVC)
11 MVC an einer Beispielanwendung

Unit 07

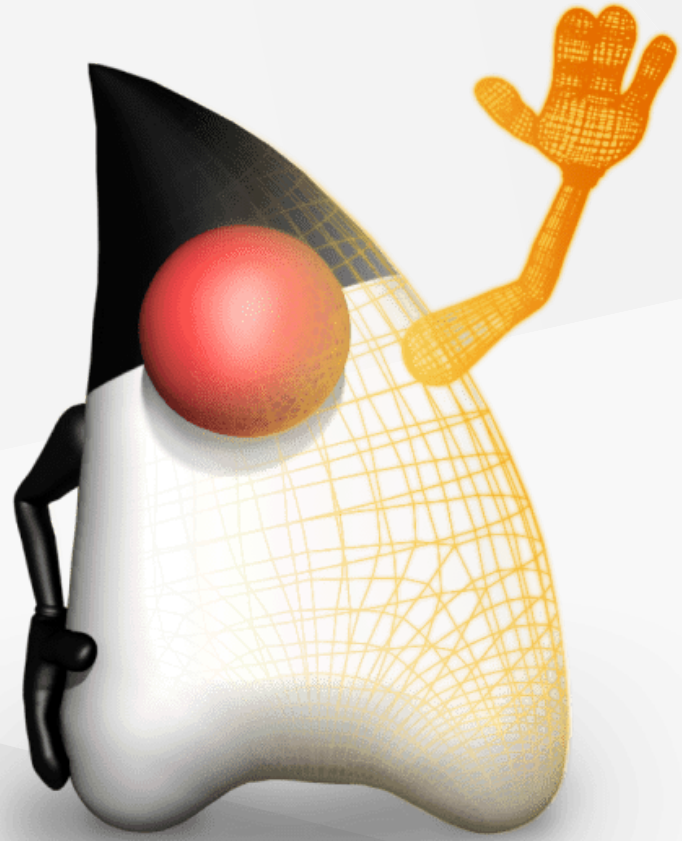
Einführung in Generische Datentypen (Type Erasure)

Inhalte:

- Historie und Einführung in die Generische Programmierung
- Type Erasure
- Generische Klassen und Interfaces
- Eigenständig generische Methoden

Übungsaufgaben

- Stichprobe bestimmen
- Faltung von Sequenzen
- Generische verkettete Liste



Historie und Einführung in die Generische Programmierung

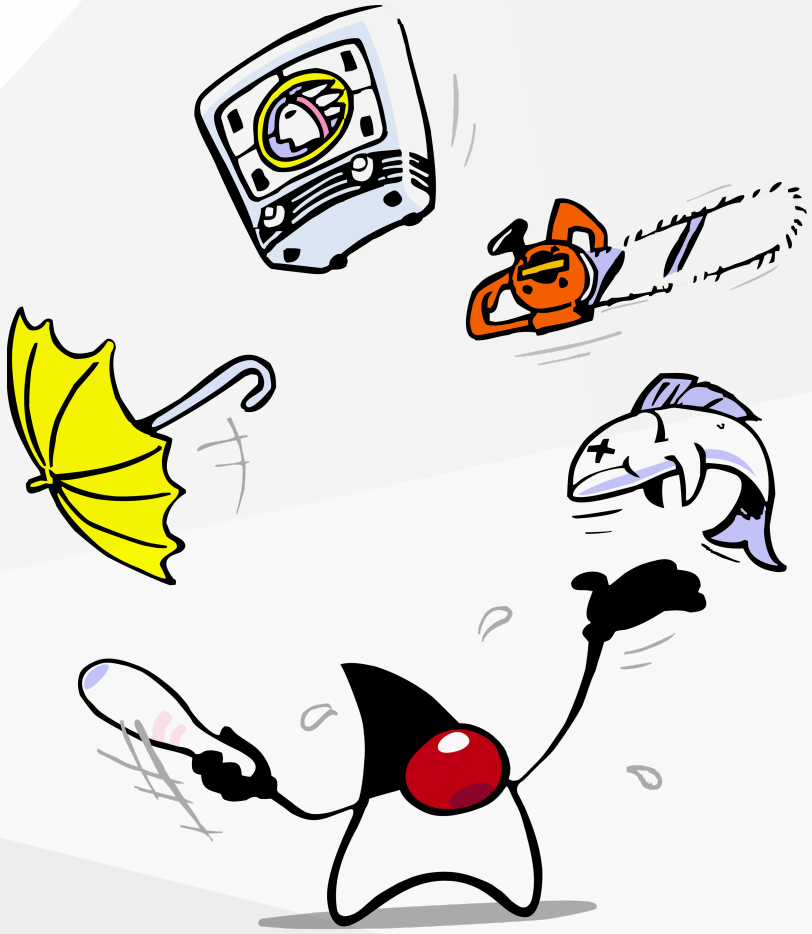
Historie

- Generische Programmierung wurde erst mit Java 5 im Jahr 2004 eingeführt.
- *Daher gibt es allerdings auch einige "Brüche", die an einigen Stellen zu seltsamen "Kapriolen" führen, die andere Sprachen so nicht kennen.*
- Dennoch war Java 5 ein bedeutender Schritt in der Evolution der Sprache, der es ermöglichte, Typensicherheit und Flexibilität von Java erheblich zu verbessern.
- Ein wichtiger Bezugspunkt ist die Sprache "Pizza", die in den 1990er Jahren entwickelt wurde und ähnliche Konzepte wie Generics verwendete.
- Pizza führte zur Entwicklung der Sprache GJ (Generics in Java), die als Grundlage für die Generics in Java 5 diente.
- Martin Odersky, einer der Autoren von Pizza, entwickelte später die Programmiersprache Scala, die ebenfalls von diesen Konzepten beeinflusst wurde.
- Die Einführung von Generics in Java war ein Teil eines größeren Trends, in dem Programmiersprachen aus den Lektionen der Vorgängerversionen lernten und neue Konzepte und Paradigmen zu integrieren.
- Dies zeigt sich auch in der Art und Weise, wie Java sich seitdem weiterentwickelt hat, um zunehmend funktionale Programmierparadigmen zu unterstützen (z.B. Lambdas und Streams).

Einleitung

Generische Programmierung (oder *Parametric Polymorphism*) in Java ermöglicht die Erstellung von Klassen, Methoden und Interfaces, die mit beliebigen Datentypen arbeiten können, ohne die Typsicherheit zu beeinträchtigen. Generics ermöglichen eine typsichere Programmierung und reduzieren die Notwendigkeit von Typumwandlungen.

- Typsicherheit: Generics bieten Compile-Time-Typüberprüfungen, die Fehler im Code vor der Ausführung verhindern.
- Wiederverwendbarkeit: Ein generischer Code kann mit unterschiedlichen Datentypen verwendet werden, was die Notwendigkeit reduziert, mehrere Versionen der gleichen Klasse oder Methode zu erstellen.
- Vermeidung von Casts: Durch die Verwendung von Generics müssen keine Typumwandlungen (Casts) mehr verwendet werden, was die Lesbarkeit und Wartbarkeit des Codes verbessert.



Beispiel

Generische Typen sind in Java Klassen oder Interfaces, die mindestens einen Typparameter enthalten. Beispielsweise können wir eine generische Klasse Box erstellen, wobei Bezeichner wie `T` als Platzhalter für den Typen dienen, den die Box speichern kann.

Generische Klassendefinition:

```java
public class Box<T> {
  private T value;

  public void setValue(T value) {
    this.value = value;
  }

  public T getValue() {
    return value;
  }
}
```

Instantiierung einer generischen Klasse:

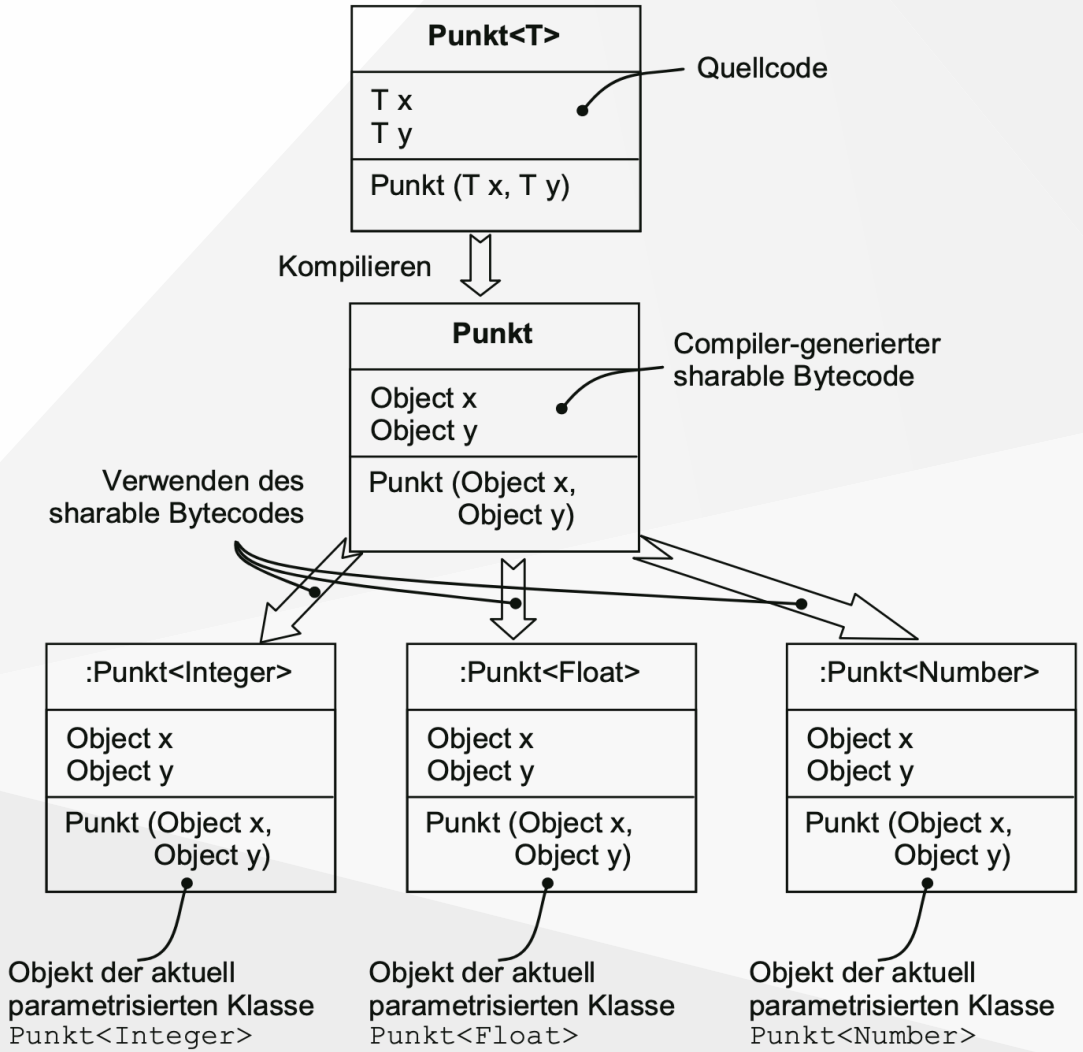
```java
Box<String> stringBox = new Box<>();
stringBox.setValue("Hello World");

Box<Integer> integerBox = new Box<>();
integerBox.setValue(123);

System.out.println(stringBox.getValue());
// => Hello World (Typ String)

System.out.println(integerBox.getValue() + 1);
// => 124 (Typ Integer)
```

Technischer Hintergrund
Type Erasure

Type Erasure

Type Erasure ist der Prozess, bei dem alle Informationen zu den Typparametern einer generischen Klasse oder Methode bei der Übersetzung des Programms entfernt werden.Generische Typen existieren daher zur Laufzeit nicht mehr als solche.

- Ersatz der Typparameter: Bei der Kompilierung wird jeder Typparameter (z. B. `T` in `Punkt<T>`) durch den allgemeinsten Referenztyp `Object` ersetzt.
- Keine Typparameter zur Laufzeit: Da generische Typen zur Laufzeit nicht verfügbar sind, kann die Laufzeitumgebung keine Typüberprüfungen auf der Basis von Typparametern zur Laufzeit durchführen. Stattdessen werden alle Typprüfungen zur Compile-Zeit erledigt.



Bildquelle: Java als erste Programmiersprache, Springer, 2016

Type Erasure: Vor- und Nachteile

(+) Rückwärtskompatibilität: Type Erasure erlaubt es, generische Typen zu der existierenden Java-API zu addieren, ohne bereits existierende Klassen zu ändern.

(+) Einfachheit: Die Laufzeitumgebung von Java bleibt einfach, da sie keine neuen Typen zur Laufzeit verwalten muss.

(-) Limitierte Unterstützung für Primitive Typen: Generics unterstützen keine primitiven Datentypen (z. B. `int`, `char` etc.), sondern es müssen Wrapper-Klassen (z. B. `Integer`, `Character`) verwendet werden.
Das führt zu Autoboxing-Erfordernissen und einigen "Sonderlichkeiten".

- Wrapper-Klassen können bspw. *null* sein, was bei primitiven Typen nicht der Fall ist.
- Wrapper-Klassen unterscheiden zwischen Identität == und Gleichheit *equals* während primitive Datentypen nur Gleichheit == kennen.

Primitiver Typ	Wrapper-Klasse	Beschreibung
`boolean`	`Boolean`	`true` und `false`.
`byte`	`Byte`	Speichert eine 8-Bit-Zahl.
`char`	`Character`	16-Bit Unicode-Zeichen.
`short`	`Short`	Speichert eine 16-Bit-Zahl.
`int`	`Integer`	Speichert eine 32-Bit-Zahl.
`long`	`Long`	Speichert eine 64-Bit-Zahl.
`float`	`Float`	Speichert eine 32-Bit Fließkommazahl.
`double`	`Double`	Speichert eine 64-Bit Fließkommazahl.

Type Erasure: Beispiel

Generischer Datentyp mit Typparameter `T`

```
class Box<T> {

  private T value;

  public void setValue(T value) {
    this.value = value;
  }

  public T getValue() {
    return value;
  }
}
```

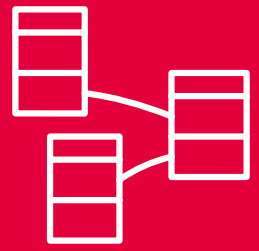
Die Informtionen zu `T` gehen durch den Type Erasure also verloren. Der Compiler spendiert bei ersetzten Rückgabetypen für alle `Object` einen entsprechenden Type Cast.

Type Erasure: Alle `T` durch `Object` ersetzt

```
class Box {

  private Object value;

  public void setValue(Object value) {
    this.value = value;
  }

  public Object getValue() {
    return value;
  }
}
```

```
Box<MyType> box = new Box<>();
MyType wert = (MyType)box.getValue();
```

Erforderlicher Cast wird automatisch durch den Compiler erzeugt, und muss nicht explizit angegeben werden.



Generische Klassen, Interfaces und eigenständig generische Methoden

Definition und Instantiierung generischer Klassen

Eine generische Klasse wird durch Angabe eines oder mehrerer Typ-Parameter in `<>`-Klammern definiert:

```java
public class Pair<T, U> {
  private T first;
  private U second;

  public Pair(T a, U b) {
    this.first = a; this.second = b;
  }

  public T getFirst() { return this.first; }

  public U getSecond() { return this.second; }

  public String toString() {
    return String.format("(%s, %s)",
      this.first, this.second
    );
  }
}
```

Objekte mit mit unterschiedlichen inneren Datentypen, können dann durch Angabe von **aktuellen Typ-Parametern** in den `<>`-Klammern angegeben werden.

```java
// Z.B. Angabe eines Key-Value Paars
Pair<String, Integer> p = new Pair<>("Age", 30);
System.out.println(p);

// Z.B. Koordinaten
Pair<Double, Double> q = new Pair<>(52.5, 13.4);
System.out.println(q);
```

Anmerkung: *Die Angabe von* `new Pair<>("Age"m 30);` *ist eine Kurznotation (Diamond-Notation) für* `new Pair<String, Integer>("Age", 30);` *Bis Java 7 musste immer die Langform angegeben werden. Verzichten Sie auf die Angabe der* `<>` *(also* `new Pair("Age", 30)`*) ist der Compiler nicht mehr in der Lage die statische Typprüfung vornehmen zu können und wird entsprechend warnen.*

Definition und Nutzung generischer Interfaces (Schnittstellen)

Auch Schnittstellen kann man generisch definieren.

```
public interface Appendable<T> { void append(T item); }
```

Wenn man diese in Klassen implementiert, kann dies entweder durch eine generische Klasse oder eine konkrete Klasse geschehen.

Generische Klasse (Durchschleifen von `T`):

```
class GrowableData<T> implements Appendable<T> {
  private List<T> items;

  public GrowableData() {
    items = new ArrayList<>();
  }

  public void append(T item) {
    items.add(item);
  }
}
```

Konkrete Klasse (Setzen von `T`):

```
class GrowableText implements Appendable<String> {
  private List<String> items;

  public GrowableText() {
    items = new ArrayList<>();
  }

  public void append(String item) {
    items.add(item);
  }
}
```

Eigenständig generische Methoden

Eigenständig generische Methoden sind Methoden, die Typ-Parameter definieren, ohne dass die umgebende Klasse ebenfalls generisch sein muss.

Man kann also innerhalb einer nicht-generischen Klasse Methoden erstellen, die dennoch generische Funktionalität bieten.

Die Einführung eines Typ-Parameters `<T>` erfolgt dabei nicht am Kopf der Klasse sondern am Methodenkopf!

```java
class RandomPick { // Nicht generische Klasse

  // Eigenständig generische Methode
  public <T> T sample(List<T> list) {
    int ri = new Random().nextInt(list.size());
    return list.get(ri);
  }

  public static void main(String[] args) {
    Integer number = sample(List.of(1, 2, 3, 4, 5));
    System.out.println("Random number: " + number);
    // => z.B. 5
    String fruit = sample(List.of(
      "Apple", "Banana", "Cherry", "Orange"
    ));
    System.out.println("Random fruit: " + fruit);
    // => z.B. Cherry
  }
}
```

Übungsaufgaben

Stichprobe bestimmen, Faltung von Sequenzen, Generische verkettete Liste

1. Aufgabe:

Entwickeln Sie **eine** Methode `sample()`, die aus einer Liste von Werten beliebigen Typs eine Stichprobe von einem definierbaren Anteil bestimmt.

Vorgabe:

```
List<Integer> ints = sample(0.2, List.of(1,2,3,4,5,6,7,8,9,0));
System.out.println(ints); // => z.B. [3, 4]

List<String> strings = sample(0.5, List.of("Welt", "lass", "Dich", "umarmen"));
System.out.println(strings); // => z.B. ["lass", "Welt"]
```

Lösung:

>>> YOUR-TURN <<<

2. Aufgabe:

Schreiben Sie nun eine generische Methode namens `fold()`, die auf einer Liste von Werten beliebigen Typs, einen zweistelligen Operator anwendet, der von links nach rechts immer das vorderste Element mit dem Resultat der letzten Operation zusammenfasst, beginnend ab einem Startwert.

Beispiele:

```
List<Integer> values = List.of(1, 2, 3, 4, 5);
Integer r = fold(0, values, (a, b) -> a + b);
System.out.println(r); // => 15

String s = fold("", List.of("Hello", "World"), (a, b) -> (a + " " + b).trim());
System.out.println(s); // => Hello World

Boolean t = fold(true, List.of(true, false, true, false), (a, b) -> a && b);
System.out.println(t); // => false
```

Veranschaulichung:

```
0  [1, 2, 3, 4, 5] mit (a, b) -> a + b
1  [2, 3, 4, 5]
3  [3, 4, 5]
6  [4, 5]
10 [5]
15 []
```

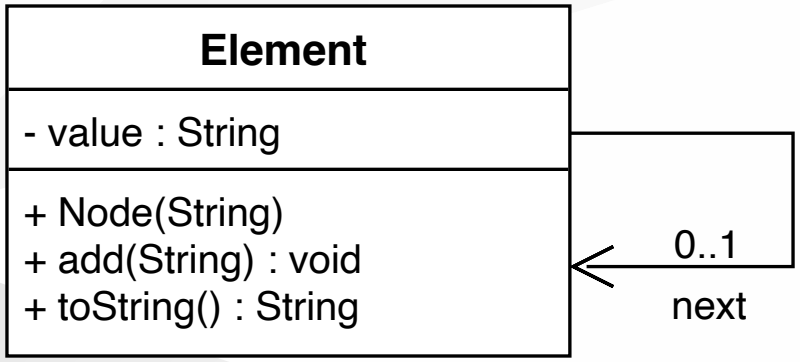
Kommt einem irgendwie bekannt vor? Oder?

Lösung:

>>> YOUR-TURN <<<

3. Aufgabe:

UML:



Anmerkungen:

Implementieren Sie diese Klasse aus Unit 03 bitte generisch. Passen Sie die Methode `add()` allerdings so an, dass diese Method-Chaining erlaubt.

Frag KIRA: Was versteht man unter Method Chaining?

```java
class Element {
  private String val;        // Inhalt des Elements
  private Element next;      // Referenz auf gleichen Typ!

  public Element(String v) { // Legt neues Element an
    this.val = v;
  }

  // Hängt am Ende der Liste ein neues Element an
  public void add(String v) {
    if (this.next == null) this.next = new Element(v); // Basisfall
    else this.next.add(v);
  }

  // Rekursiver Durchlauf
  private String serialize() {
    if (this.next == null) return this.val; // Basisfall
    return this.val + ", " + this.next.serialize(); // Kopf + Rest
  }

  public String toString() {
    return "[" + this.serialize() + "]";
  }
}
```

Zusammenfassung

- Parametric Polymorphism
  Generische Programmierung ermöglicht die Erstellung von Klassen, Methoden und Interfaces, die mit beliebigen Datentypen arbeiten können, ohne die Typsicherheit zu beeinträchtigen.
- Type Erasure
  Entfernt alle Informationen zu den Typparametern einer generischen Klasse oder Methode bei der Übersetzung des Programms. Generische Typen existieren daher zur Laufzeit nicht mehr als solche.
- Generische Klassen und Interfaces
  Generische Klassen und Interfaces ermöglichen es, Typen im Code zu parametrisieren, und so die Wiederverwendbarkeit von Code zu erhöhen, da sie mit beliebigen Referenz-Datentypen arbeiten können, ohne dass der spezifische Typ zum Programmierzeitpunkt festgelegt werden muss.
- Eigenständig generische Methoden
  Generische Methoden sind Methoden, die Typ-Parameter definieren, sodass sie mit verschiedenen Datentypen arbeiten können. Der Typ-Parameter wird dabei nur für die jeweilige Methode festgelegt, nicht für die gesamte Klasse. Dies bedeutet, dass jede generische Methode unabhängig von der Klassendeklaration unterschiedliche Typ-Parameter haben kann.

TECHNISCHE HOCHSCHULE LÜBECK

Kontakt

Prof. Dr. Nane Kratzke

Technische Hochschule Lübeck

Mail: nane.kratzke(at)th-luebeck.de

Code Strong!

Alle Icons stammen von iconify.design • Bilder von Pixabay

Lösungsvorschlag zur 1. Aufgabe

```java
import java.util.*

public class Main {
  public static <T> List<T> sample(double fraction, List<T> items) {
    int sampleSize = (int) (Math.max(0.0, Math.min(1.0, fraction)) * items.size());
    Set<T> sampleSet = new HashSet<>(); // Wir nutzen Set, um Duplikate zu vermeiden
    Random random = new Random();

    while (sampleSet.size() < sampleSize) {
        T item = items.get(random.nextInt(items.size())); // Zufallselement ziehen
        sampleSet.add(item);
    }

    return new ArrayList<>(sampleSet);
  }

  public static void main(String[] args) {
    List<Integer> ints = sample(0.2, List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 0));
    System.out.println(ints);

    List<String> strings = sample(0.5, List.of("Welt", "lass", "Dich", "umarmen"));
    System.out.println(strings);
  }
}
```

Lösungsvorschlag zur 2. Aufgabe

Eigentlich nur eine Namensvariante der `reduce()` Methode von Streams. Also wieso nicht einfach darauf abbilden?

```java
import java.util.*; import java.util.function.*;

public class Main {
    public static <T> T fold(T start, List<T> items, BinaryOperator<T> accumulator) {
        return items.stream().reduce(start, accumulator);
    }

    public static void main(String[] args) {
        Integer r = fold(0, List.of(1, 2, 3, 4, 5), (a, b) -> a + b);
        System.out.println(r); // => 15

        String s = fold("", List.of("Hello", "World"), (a, b) -> (a + " " + b).trim());
        System.out.println(s); // => Hello World

        Boolean t = fold(true, List.of(true, false, true, false), (a, b) -> a && b);
        System.out.println(t); // => false
    }
}
```

Lösungsvorschlag zur 3. Aufgabe

```java
class Element<T> {
  private T val;
  private Element<T> next;

  public Element(T v) { this.val = v; }

  public Element<T> add(T v) { // Method-Chaining !!!
    if (this.next == null) {
        this.next = new Element<>(v);
    } else this.next.add(v);
    return this; // Method-Chaining !!!
  }

  private String serialize() {
    if (this.next == null) return this.val + "";
    return this.val + ", " + this.next.serialize();
  }

  public String toString() {
    return "[" + this.serialize() + "]";
  }
}
```

```java
Element<Integer> values = new Element<>(5);
values.add(7).add(8).add(9);
System.out.println(values); // => [5, 7, 8, 9]

Element<String> texts = new Element<>("Hello");
texts.add("strange").add("World").add("!");
System.out.println(texts);
// => [Hello, strange, World, !]
```

Method Chaining ist ein Programmierkonzept, das in objektorientierten Programmiersprachen wie Java häufig verwendet wird. Es ermöglicht das Verketten von Methodenaufrufen, sodass mehrere Methoden in einer einzigen Anweisung aufgerufen werden können.

Z.B.: `texts.add("strange").add("World").add("!");`

Dies wird erreicht, indem jede Methode eine Referenz auf das aktuelle Objekt zurückgibt.