CARL VON OSSIETZKY universität OLDENBURG

Sommersemester 2020

Objektorientierte Modellierung und Programmierung

Dr. Christian Schönberg

CARL VON OSSIETZKY universität OLDENBURG

Generische Klassen und Methoden

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Klassen mit Typparametern
- Methoden mit Typparametern

CARL VON OSSIETZKY universität OLDENBURG

Generische Klassen und Methoden

CARL VON OSSIETZKY universität OLDENBURG

Motivation

- Implementierung von gleicher Funktionalität für verschiedene Datentypen
  - **`LinkedStringList`**, **`LinkedIntegerList`**, **`LinkedDoubleList`**
  - **`List`** (mit **`Object`**)
- Die Funktionalität (Datenstruktur, Operationen) ist generisch, d.h. für alle gleich
- Lösung: Parametrisierung der Datenstruktur und Operationen
  - **`LinkedList<String>`**, **`LinkedList<Integer>`**, **`LinkedList<Double>`**, **…**

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Generische Klassen

```
LinkedList<String> stringList = new LinkedList<String>();
stringList.add("1");
stringList.add("2");
stringList.add(3);
String s = stringList.get(1);

LinkedList<Integer> intList = new LinkedList<Integer>();
intList.add(1);
intList.add(2);
intList.add("3");
int i = intList.get(1);
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Generische Klassen

```
LinkedList<String> stringList = new LinkedList<String>();
stringList.add("1");
stringList.add("2");
stringList.add(3);
```

The method **add(String)** in the type **LinkedList<String>** is not applicable for the arguments (**int**)

```
String s = stringList.get(1);

LinkedList<Integer> intList = new LinkedList<Integer>();
intList.add(1);
intList.add(2);
intList.add("3");
```

The method **add(Integer)** in the type **LinkedList<Integer>** is not applicable for the arguments (**String**)

```
int i = intList.get(1);
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Generische Klassen

```
LinkedList<String> stringList = new LinkedList<String>();
stringList.add("1");
stringList.add("2");
String s = stringList.get(1);

LinkedList<Integer> intList = new LinkedList<Integer>();
intList.add(1);
intList.add(2);
int i = intList.get(1);
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Generische Klassen

```
LinkedList<String> stringList = new LinkedList<>();
stringList.add("1");
stringList.add("2");
String s = stringList.get(1);

LinkedList<Integer> intList = new LinkedList<>();
intList.add(1);
intList.add(2);
int i = intList.get(1);
```

CARL VON OSSIETZKY universität OLDENBURG

Beispiel: Generische Klassen

```
List<String> stringList = new LinkedList<>();
stringList.add("1");
stringList.add("2");
String s = stringList.get(1);

List<Integer> intList = new LinkedList<>();
intList.add(1);
intList.add(2);
int i = intList.get(1);
```

Beispiel: Generische Klassen (2)

```
LinkedList<Object> objectList = new LinkedList<>();
objectList.add("1");
objectList.add(2);

String s = (String) objectList.get(0);
int i = (int) objectList.get(1);
```

Wenn der gewünschte Typ spezieller ist als der vorhandene → Typecast

Problem: Sicherstellen, dass die richtigen Instanzen auf die richtigen Typen gecasted werden → vermeiden

CARL VON OSSIETZKY universität OLDENBURG

Erinnerung: Autoboxing

```
int i1 = 1;
Integer i2 = 2;
Integer i3 = i1;

LinkedList<Integer> intList = new LinkedList<>();
intList.add(1);
intList.add(new Integer(1));
int i = intList.get(0);
```

Java ersetzt automatisch Werte vom Typ **`int`** durch gleichwertige Instanzen vom Typ **`Integer`** und umgekehrt, wo dies benötigt wird. Analog für **`Character`**/**`char`**, **`Double`**/**`double`**, **`Float`**/**`float`**, **`Boolean`**/**`boolean`**, …

CARL VON OSSIETZKY universität OLDENBURG

Implementierung von Generischen Klassen/Interfaces

```
public class LinkedList<T> {

    private LinkedListItem<T> head;

    public void add(T data) { … }

    public T get(int index) { … }

}
```

CARL VON OSSIETZKY universität OLDENBURG

Implementierung von Generischen Klassen/Interfaces

```
class LinkedListItem<T> {

    private T data;
    private LinkedListItem<T> next;

    public T getData() {
        return data;
    }
    public void setData(T data) {
        this.data = data;
    }
    public LinkedListItem<T> getNext() {
        return next;
    }
    public void setNext(LinkedListItem<T> next) {
        this.next = next;
    }

}
```

CARL VON OSSIETZKY universität OLDENBURG

Implementierung von Generischen Klassen/Interfaces

```
public void add(T data) {
    LinkedListItem<T> item = new LinkedListItem<>();
    item.setData(data);
    if (head == null) {
        head = item;
    } else {
        LinkedListItem<T> current = head;
        while (current.getNext() != null) {
            current = current.getNext();
        }
        current.setNext(item);
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Implementierung von Generischen Klassen/Iterfaces

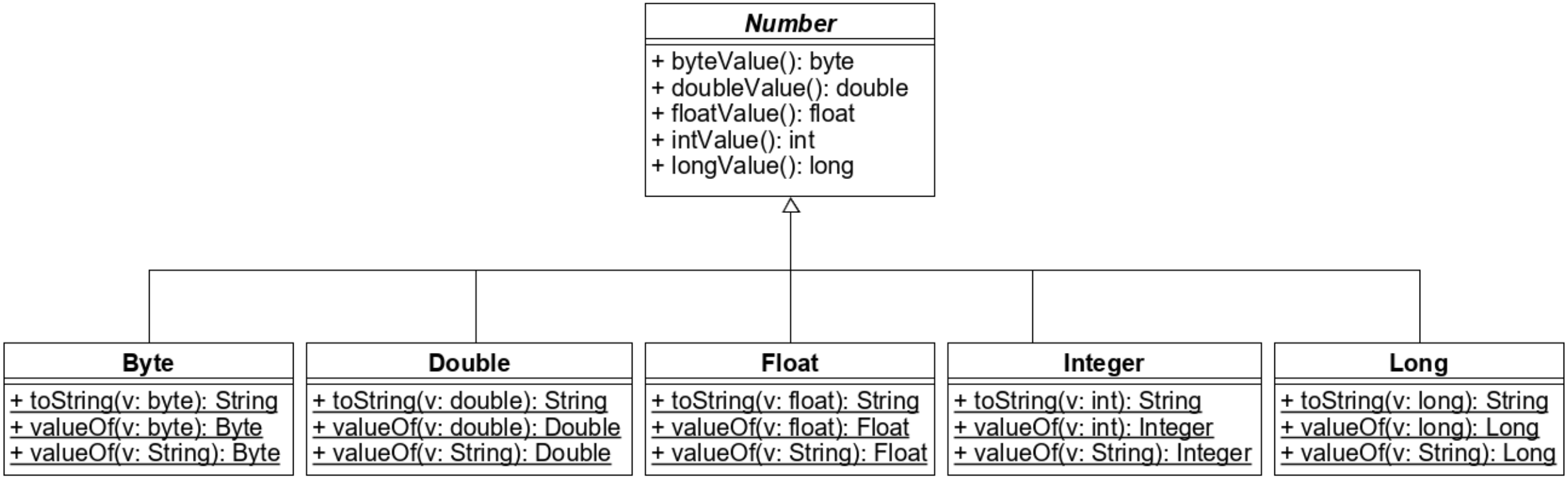
```
public T get(int index) {
    if (index < 0 || head == null) {
        return null;
    } else {
        LinkedListItem<T> current = head;
        for (int i = 0; i < index; i++) {
            if (current.getNext() == null) {
                return null;
            }
            current = current.getNext();
        }
        return current.getData();
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Generische Klassen/Interfaces - mehrere Typ-Parameter

```
public interface Map<K, V> {
  void put(K key, V value);
  V get(K key);
}

public class ArrayMap<K, V> implements Map<K, V> {
  private K[] keys;
  private V[] values;

  public ArrayMap(int size) {…}
  public void put(K key, V value) {…}
  public V get(K key) {}

  public static void main(String[] args) {
    ArrayMap<String, Integer> ageMap = new ArrayMap<>(5);
    ageMap.put("maria", 24);
    System.out.println("age of maria = " + ageMap.get("maria"));
  }
}
```

Die Numbers-Hierarchie



CARL VON OSSIETZKY universität OLDENBURG

Einschränkung der Generizität

```
public class LinkedNumberList<T extends Number> {

    // ...

}
```

```
LinkedNumberList<Integer> intList = new LinkedNumberList<>();
intList.add(1);

LinkedNumberList<Double> doubleList = new LinkedNumberList<>();
doubleList.add(3.14D);

LinkedNumberList<String> stringList = new LinkedNumberList<>();
```

Bound mismatch: The type **String** is not a valid substitute for the bounded parameter **<T extends Number>** of the type **LinkedNumberList<T>**

T muss eine Unterklasse von **Number** sein.

CARL VON OSSIETZKY universität OLDENBURG

Einschränkung der Generizität

```
public class LinkedNumberList<T extends Number> {

    // ...

}
```

```
LinkedNumberList<Integer> intList = new LinkedNumberList<>();
intList.add(1);

LinkedNumberList<Double> doubleList = new LinkedNumberList<>();
doubleList.add(3.14D);
```

Warum ist eine Klasse mit beschränktem Parameter (bounded parameter)
sinnvoller als eine Instanz von **`List<Number>`**?

CARL VON OSSIETZKY universität OLDENBURG

Mehrfache Einschränkung der Generizität

```
interface I { void mI(); }
interface J { void mJ(); }

class A<T extends Random & I & J> {
    void test(T obj) {
        obj.nextInt();
        obj.mI();
        obj.mJ();
    }
}

class P1 extends Random implements J {
    public void mJ() { }
}

class P2 extends Random implements I, J {
    public void mI() { }
    public void mJ() { }
}

A<P2> a1 = new A<>();
A<P1> a2 = new A<>();
```

Bound mismatch: The type **P1** is not a valid substitute for the bounded parameter **<T extends Random & I & J>** of the type **A<T>**

CARL VON OSSIETZKY universität OLDENBURG

Polymorphie und Generiziät

```
Number number = new Integer(5);
LinkedList<Number> numberList = new LinkedList<Integer>();
```

Type mismatch: cannot convert from **LinkedList<Integer>** to **LinkedList<Number>**

Obwohl **Number** eine Oberklasse von **Integer** ist,
ist **LinkedList<Number>** **keine** Oberklasse von **LinkedList<Integer>**!

```
numberList.add(3.14D);
```

LinkedList<Number> würde Operationen erlauben, die **LinkedList<Integer>** nicht unterstützt!

CARL VON OSSIETZKY universität OLDENBURG

Implementierung von Generischen Methoden

```
public static <T extends Number> void quickSort(T[] data) {
    quickSort(data, 0, data.length - 1);
}
```

CARL VON OSSIETZKY universität OLDENBURG

Implementierung von Generischen Methoden

```
private static <T extends Number> void quickSort(T[] data, int leftIndex, int rightIndex) {
    if (rightIndex > leftIndex) {
        T pivotValue = data[rightIndex];
        int i = leftIndex;
        int j = rightIndex - 1;
        while (true) {
            while (i < rightIndex && data[i].doubleValue() <= pivotValue.doubleValue())
                i = i + 1;
            while (j > i && data[j].doubleValue() >= pivotValue.doubleValue())
                j = j - 1;
            if (i >= j)
                break;
            swap(data, i, j);
        }
        swap(data, i, rightIndex);
        quickSort(data, leftIndex, i - 1);
        quickSort(data, i + 1, rightIndex);
    }
}
```

CARL VON OSSIETZKY universität OLDENBURG

Implementierung von Generischen Methoden

```
private static <T> void swap(T[] data, int index1, int index2) {
    T tmp = data[index1];
    data[index1] = data[index2];
    data[index2] = tmp;
}
```

CARL VON OSSIETZKY universität OLDENBURG

Implementierung von Generischen Methoden

```
Integer[] intData = new Integer[] { 1, 3, 2, 5 };
quickSort(intData);
// [1, 2, 3, 5]

Double[] doubleData = new Double[] { 4.3, 2.1 };
quickSort(doubleData);
// [2.1, 4.3]
```

Generische Methoden - Supertypen

```
public static <T> T selectRandom(T v1, T v2) {
    if (Math.random() < 0.5) {
        return v1;
    } else {
        return v2;
    }
}
```

```
int v1 = selectRandom(43, 56);
String v2 = selectRandom("hallo", "welt");
Object v3 = selectRandom(43, "hallo");
```

Type-Erasure

Compile-Zeit

```
public static <T extends Number> boolean compare(T a, T b) { … }
```

```
public static <T> boolean contains(List<T> list, T data) { … }
```

Laufzeit

```
public static boolean compare(Number a, Number b) { … }
```

```
public static boolean contains(List list, Object data) { … }
```

Der Java-Compiler ersetzt zur Laufzeit

- **T** durch seine Beschränkung (oder **`Object`** bei unbeschränkten Parametern) und
- parametrisierte Typen durch Basistypen

CARL VON OSSIETZKY universität OLDENBURG

Überladen und Generizität

```
public static <T extends Number> boolean compare(T a, T b) { … }

public static boolean compare(Number a, Number b) { … }

public static <T> boolean contains(LinkedList<T> list, T data) { … }

public static boolean contains(LinkedList list, Object data) { … }
```

Erasure of method **compare()** is the same as another method
Erasure of method **contains()** is the same as another method

```
public static void sort(LinkedList<Integer> data) { … }

public static void sort(LinkedList<String> data) { … }
```

Erasure of method **sort()** is the same as another method

Überschreiben und Generizität

```java
public interface SortLibrary {

    public void sort(LinkedList<String> data);

}
```

```java
public class QuickSortLibrary implements SortLibrary {

    @Override
    public void sort(LinkedList<String> data) { … }
}
```

```java
public class HeapSortLibrary implements SortLibrary {

    @Override
    public void sort(LinkedList<Integer> data) { … }
}
```

Name clash: The method **sort(LinkedList<Integer>)** of type **HeapSortLibrary** has the same erasure as **sort(LinkedList<String>)** of type **SortLibrary** but does not override it

Zur Laufzeit nicht mehr zu unterscheiden (gleiche Type-Erasure), aber zur Compile-Zeit unterschiedlich
→ kein Überschreiben möglich

Wildcards

```
LinkedList<?> list1 = new LinkedList<String>();
LinkedList<?> list2 = new LinkedList<Integer>();
LinkedList<?> list3 = new LinkedList<?>();
```

Cannot instantiate the type **LinkedList<?>**

```
list1.add("a");
```

The method **add(?)** in the type **LinkedList<?>** is not applicable for the arguments **(String)**

Der Wildcard (dt. Joker) kann als Platzhalter für einen beliebigen Referenztyp verwendet werden, aber nur auf der Deklarationsseite, nicht bei der Instanziierung.
Bei der Instanziierung muss schließlich bekannt sein, was für ein Typ verwendet wird.

Da der Typ nicht bekannt ist, kann in diesem Beispiel auch nicht geprüft werden, ob der Parameter zu ihm passt oder nicht.

Erweiterte Wildcards

```
LinkedList<? extends Number> list1 = new LinkedList<Integer>();
LinkedList<? extends Number> list2 = new LinkedList<Double>();
LinkedList<Number> list3 = new LinkedList<Integer>();
```

Type mismatch: cannot convert from **LinkedList<Integer>** to **LinkedList<Number>**

```
list1.add(1);
```

The method **add(? extends Number)** in the type **LinkedList<? extends Number>** is not applicable for the arguments **(int)**

Wie bereits gesehen ist **LinkedList<Number>** **keine** Oberklasse von **LinkedList<Integer>**.
Aber **LinkedList<? extends Number>** ist eine Oberklasse von **LinkedList<Integer>**.

1 ist zwar eine Instanz von **Integer**, aber wir wissen ja nicht, dass **?** hier **Integer** repräsentiert!

CARL VON OSSIETZKY universität OLDENBURG

Verwendung von Wildcards

```
public double sum(LinkedList<? extends Number> list) {
    double result = 0.0;
    for (Number n : list) {
        result += n.doubleValue();
    }
    return result;
}
```

```
LinkedList<Integer> intList = new LinkedList<>();
LinkedList<Double> doubleList = new LinkedList<>();
double sum1 = sum(intList);
double sum2 = sum(doubleList);
```

Warum kann die Methode **`sum`** keinen Parameter vom Typ **`LinkedList<Number>`** definieren?

? als Ober- statt Unterklasse

```
public int intsum(LinkedList<? super Integer> list) {
    int result = 0;
    for (Object value : list) {
        if (value instanceof Integer) {
            int n = (Integer) value;
            result += n;
        }
    }
    return result;
}
```

list muss ein Obertyp von **Integer** sein

```
LinkedList<Integer> intList = new LinkedList<>();
LinkedList<Number> numList = new LinkedList<>();
LinkedList<Object> objList = new LinkedList<>();
LinkedList<Double> doubleList = new LinkedList<>();
int isum1 = intsum(intList);
int isum2 = intsum(numList);
int isum3 = intsum(objList);
int isum4 = intsum(doubleList);
```

The method **intsum(LinkedList<? super Integer>)** is not applicable for the arguments **(LinkedList<Double>)**

Grenzen von Generics

```
public <T> void genericsTest(T value) {
    T[] values = new T[1];
```

Cannot create a generic array of **T**

```
    values[0] = new T();
```

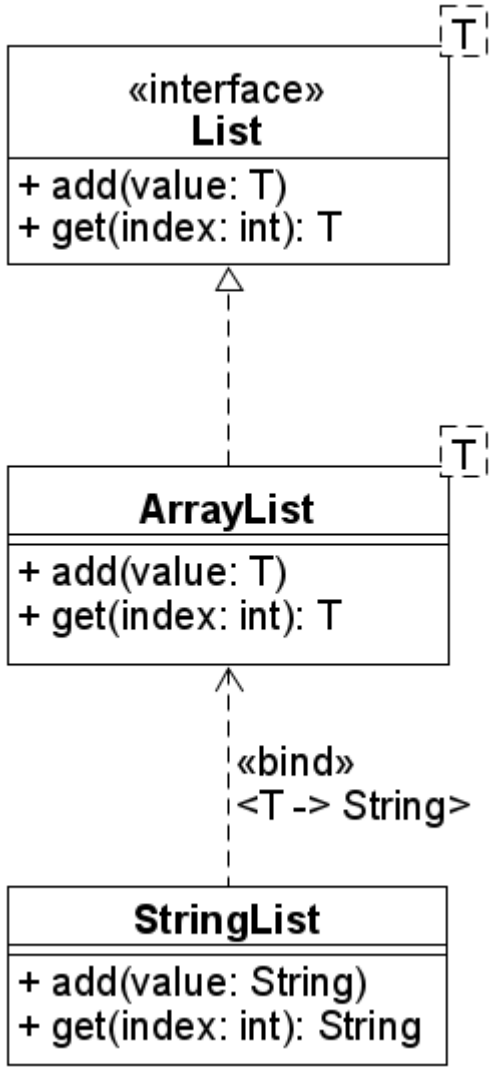
Cannot instantiate the type **T**

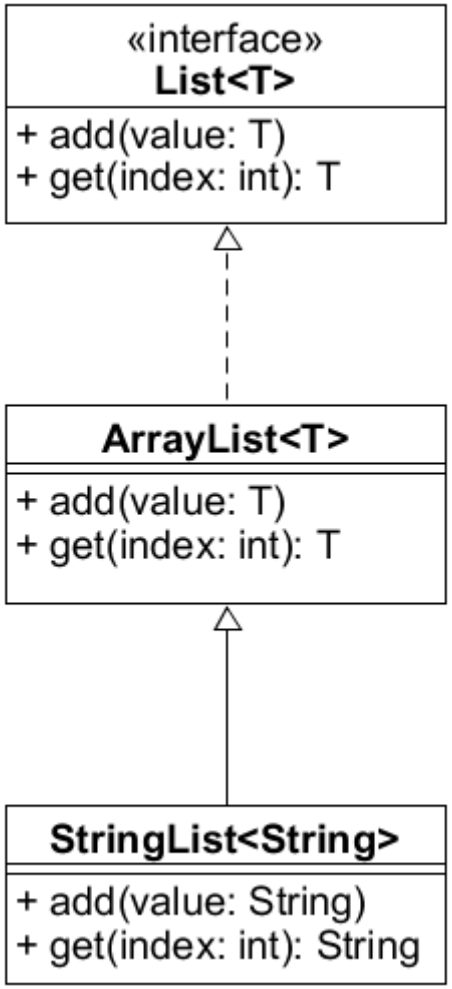
```
}
static T attr;
```

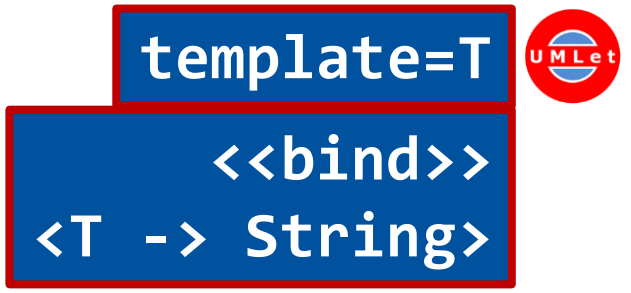
Cannot make a static reference to the non-static type **T**

```
@SuppressWarnings("unchecked")
public static <T extends Number> void genericsTest2(T value) {
    T[] values = (T[]) new Number[1];
    values[0] = (T) Integer.valueOf(5);
    System.out.println(values[0]);
}
```

Generics in der UML







CARL VON OSSIETZKY universität OLDENBURG

Generics Beispiel

angelehnt an:

Johannes Nowak: *Fortgeschrittene Programmierung mit Java 5*. dpunkt.verlag, 2004.

CARL VON OSSIETZKY universität OLDENBURG

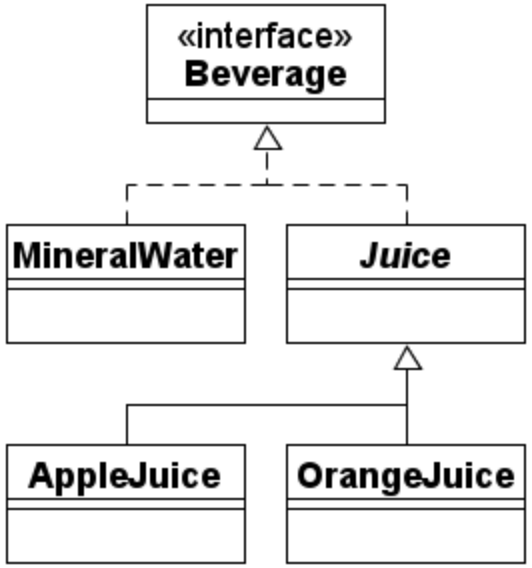
Domäne

- Getränke, Flaschen, Kisten
- Idee: Typisierte Flaschen, die nur mit einem bestimmten Getränk gefüllt werden können

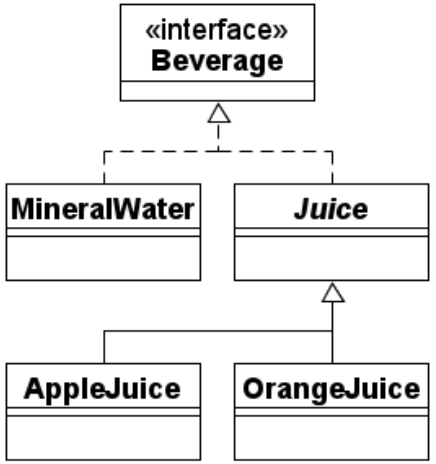
CARL VON OSSIETZKY universität OLDENBURG

Getränke

```
interface Beverage { }

class MineralWater implements Beverage { }

abstract class Juice implements Beverage { }

class AppleJuice extends Juice { }

class OrangeJuice extends Juice { }
```

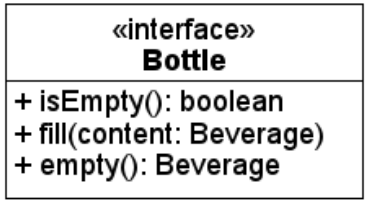


Flaschen



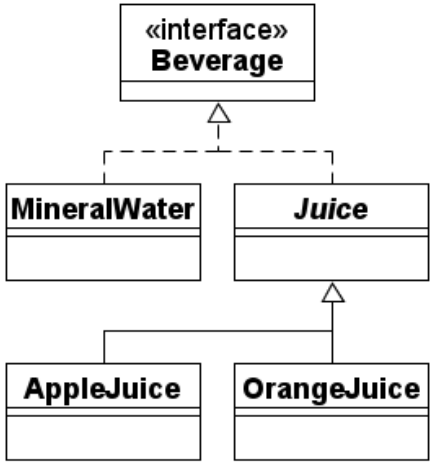

```
class BottleEmptyException extends Exception { }
class BottleNotEmptyException extends Exception { }
```

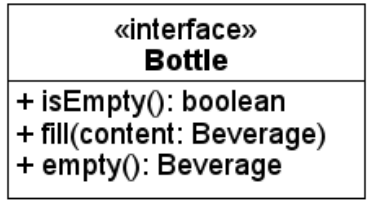
```
interface Bottle {

    boolean isEmpty();

    void fill(Beverage content) throws BottleNotEmptyException;

    Beverage empty() throws BottleEmptyException;

}
```



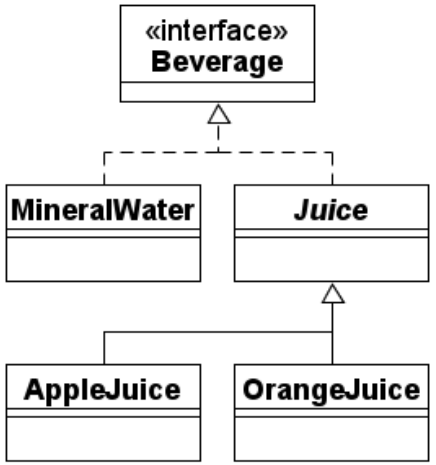
Flaschen

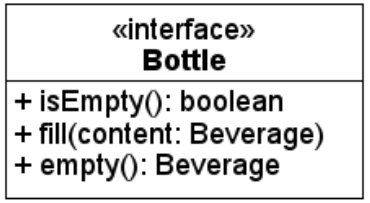
```
class BeverageBottle implements Bottle {

    private Beverage content;

    public boolean isEmpty() {
        return content == null;
    }

    public void fill(Beverage content) throws BottleNotEmptyException {
        if (this.content != null) { throw new BottleNotEmptyException(); }
        this.content = content;
    }

    public Beverage empty() throws BottleEmptyException {
        if (this.content == null) { throw new BottleEmptyException(); }
        Beverage content = this.content;
        this.content = null;
        return content;
    }

}
```





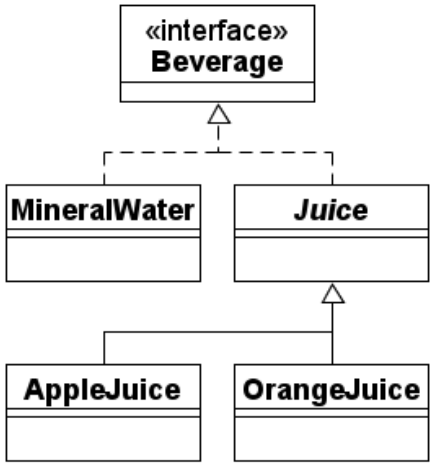
Flaschen

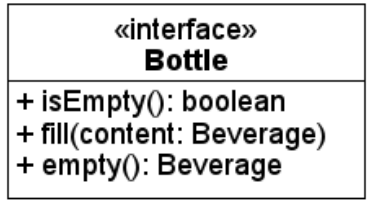





```
Bottle waterBottle = new BeverageBottle();
try {
    waterBottle.fill(new AppleJuice());
} catch (BottleNotEmptyException e) {
    System.out.println("Bottle was not empty!");
}
try {
    MineralWater water = (MineralWater) waterBottle.empty();
} catch (BottleEmptyException e) {
    System.out.println("Bottle was empty!");
}
```

Flaschen





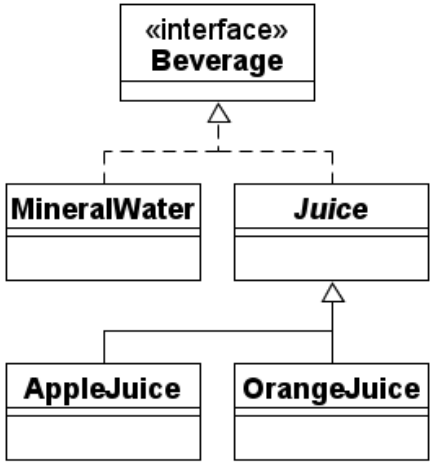

```
Bottle waterBottle = new BeverageBottle();
try {
    waterBottle.fill(new AppleJuice());
} catch (BottleNotEmptyException e) {
    System.out.println("Bottle was not empty!");
}
try {
    MineralWater water = (MineralWater) waterBottle.empty();
```

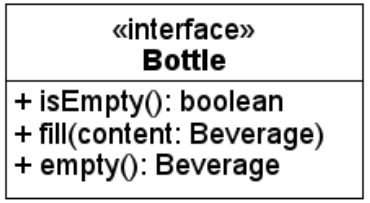
Exception in thread "main" **java.lang.ClassCastException**: class **AppleJuice** cannot be cast to class **MineralWater**

```
}
```

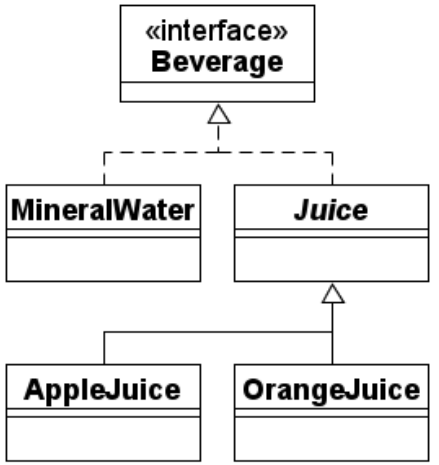
Flaschen (2)

```
class WaterBottle implements Bottle {

    private MineralWater content;

    public boolean isEmpty() {
        return content == null;
    }

    public void fill(MineralWater content) throws BottleNotEmptyException {
        if (this.content != null) { throw new BottleNotEmptyException(); }
        this.content = content;
    }

    public void fill(Beverage content) {
        // TODO implement this
    }

    public MineralWater empty() throws BottleEmptyException {
        if (this.content == null) { throw new BottleEmptyException(); }
        MineralWater content = this.content;
        this.content = null;
        return content;
    }
}
```

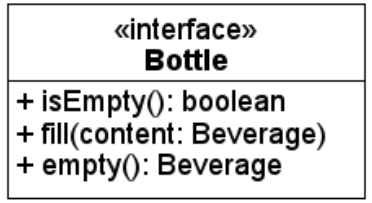




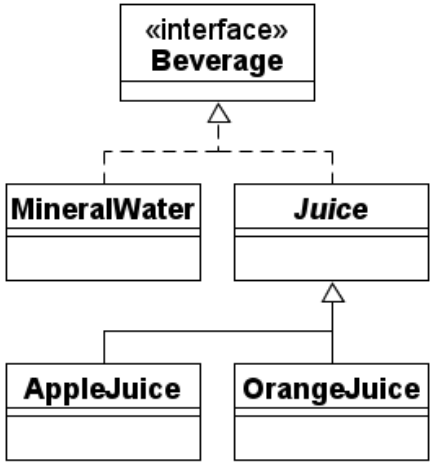
Flaschen (2)



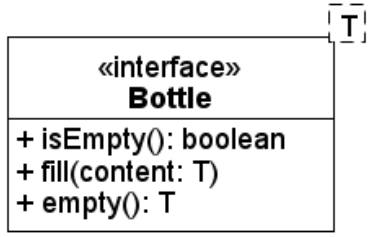

```
abstract class JuiceBottle implements Bottle { }

class AppleJuiceBottle extends JuiceBottle { }

class OrangeJuiceBottle extends JuiceBottle { }
```



Flaschen (3)

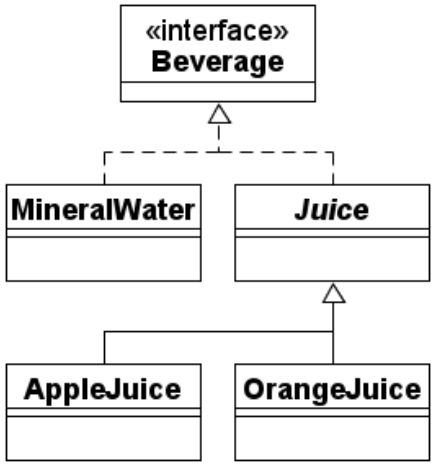


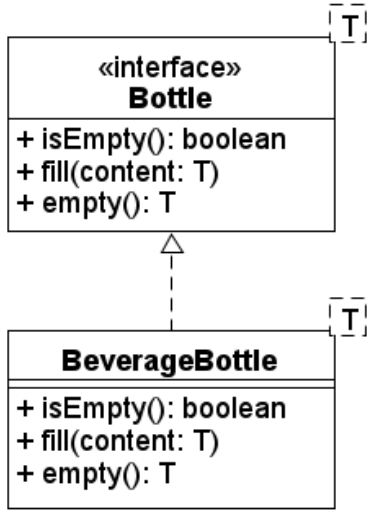

```
interface Bottle<T> {

    boolean isEmpty();

    void fill(T content) throws BottleNotEmptyException;

    T empty() throws BottleEmptyException;

}
```



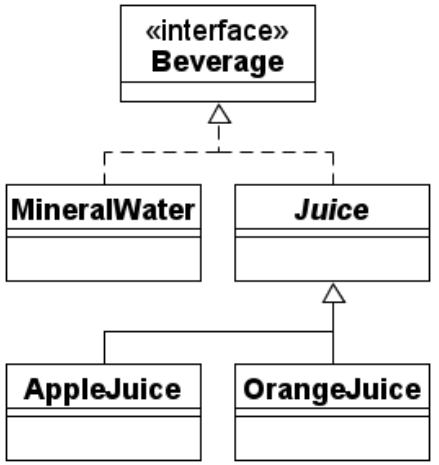
Flaschen (3)

```
class BeverageBottle<T> implements Bottle<T> {

    private T content;

    public boolean isEmpty() {
        return this.content == null;
    }

    public void fill(T content) throws BottleNotEmptyException {
        if (this.content != null) { throw new BottleNotEmptyException(); }
        this.content = content;
    }

    public T empty() throws BottleEmptyException {
        if (this.content == null) { throw new BottleEmptyException(); }
        T content = this.content;
        this.content = null;
        return content;
    }

}
```

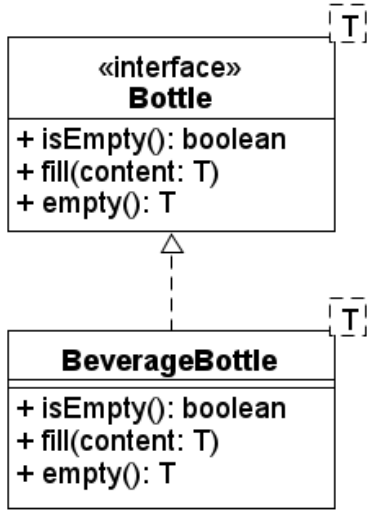




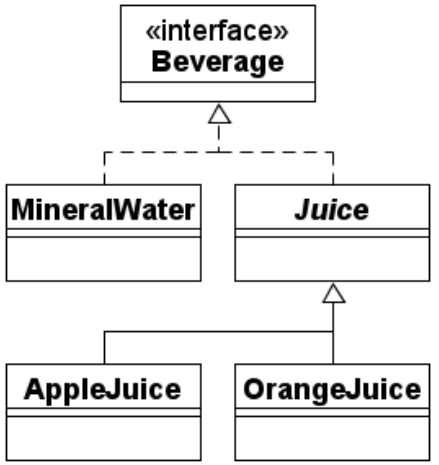
Flaschen (3)



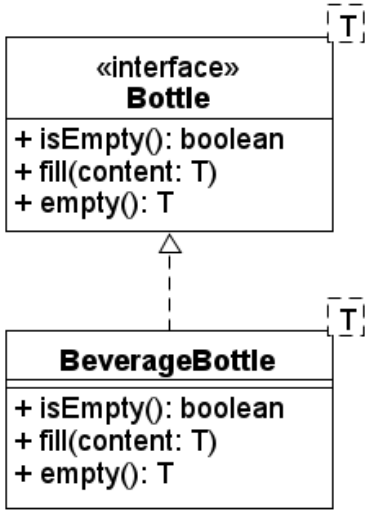

```
Bottle<MineralWater> waterBottle = new BeverageBottle<>();
try {
    waterBottle.fill(new MineralWater());
} catch (BottleNotEmptyException e1) {
    System.out.println("Bottle is not empty.");
}

Bottle<Integer> intBottle = new BeverageBottle<>();
```



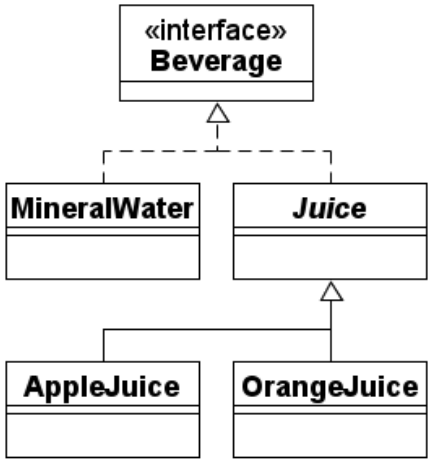
Flaschen (4)



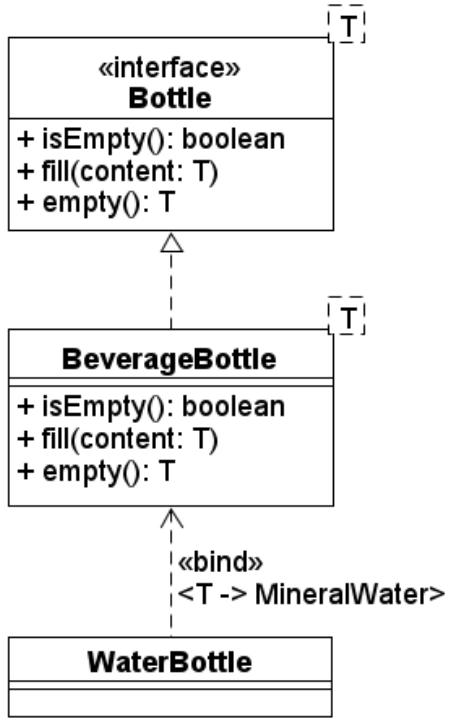

```
class BeverageBottle<T extends Beverage> implements Bottle<T> { }
```



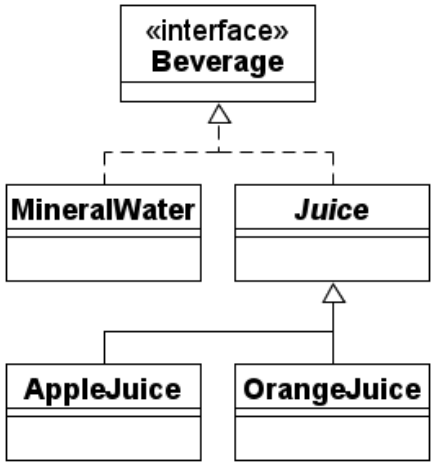
Flaschen (4)




```
class WaterBottle extends BeverageBottle<MineralWater> { }
```

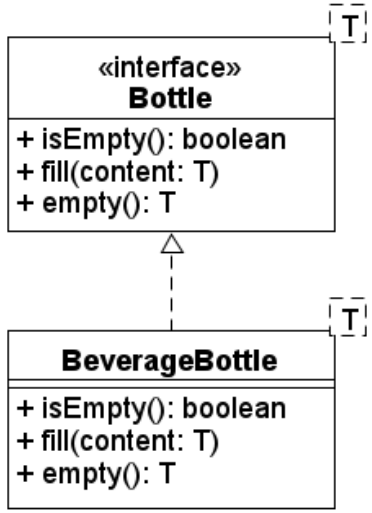


Getränkekisten



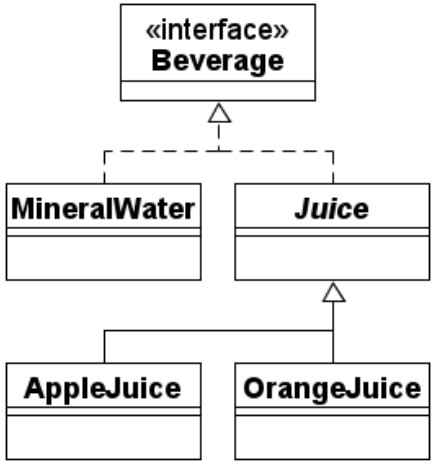

```
class WaterBottleCase {

    private Bottle<MineralWater>[] bottles = new Bottle<MineralWater>[12];
}
```

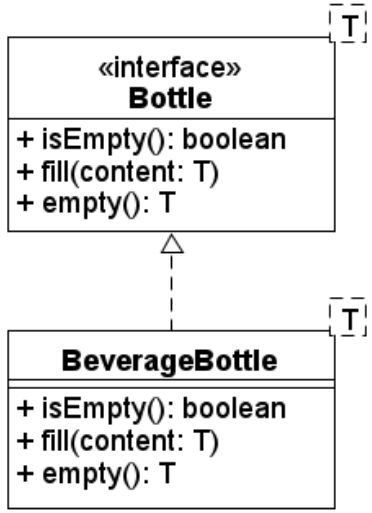
Cannot create a generic array of **Bottle<MineralWater>**



Getränkekisten (2)

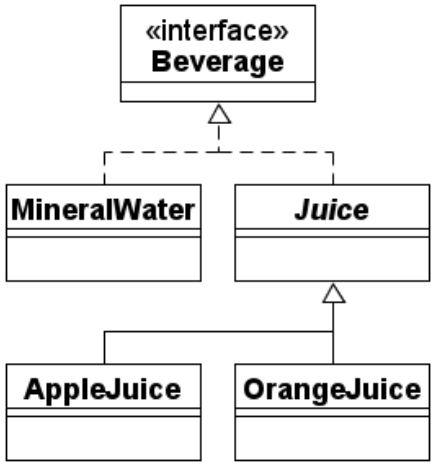
```
class WaterBottleCase {

    private Object[] bottles = new Object[12];
    private int count = 0;

    public boolean isFull() {
        return bottles.length == count;
    }

    public int getBottleCount() {
        return count;
    }

    public int getCapacity() {
        return bottles.length;
    }

    …
```

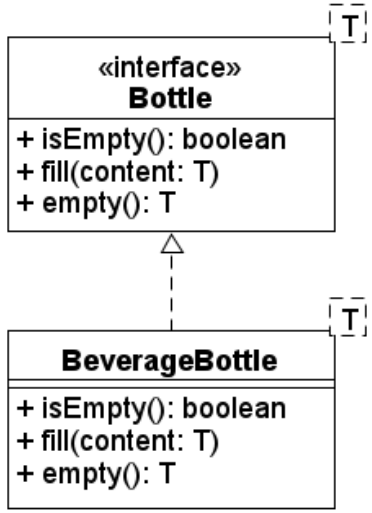




Getränkekisten (2)

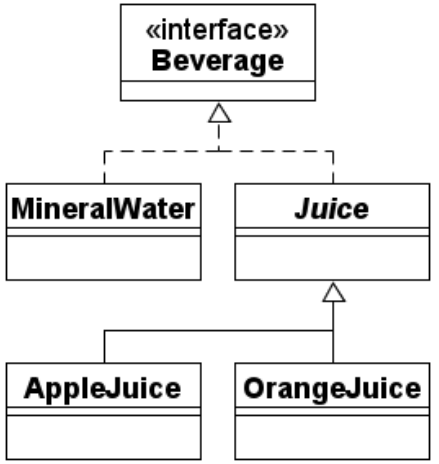
```
    public void add(Bottle<MineralWater> bottle) throws CaseFullException {
        if (isFull()) { throw new CaseFullException(); }
        bottles[count] = bottle;
        count++;
    }

    @SuppressWarnings("unchecked")
    public Bottle<MineralWater> getBottle(int index) {
        // safe type cast
        return (Bottle<MineralWater>) bottles[index];
    }

}
```

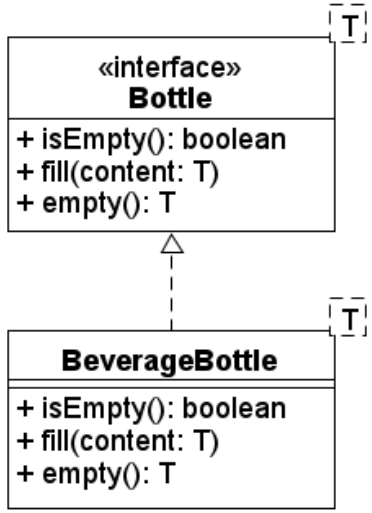




Getränkekisten (3)

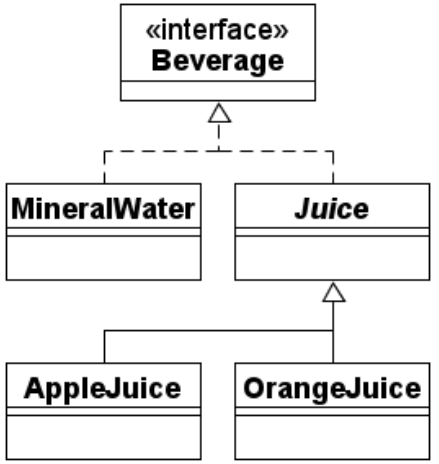
```
class BottleCase<T extends Beverage> {

    private Object[] bottles = new Object[12];
    private int count = 0;

    …

    public void add(Bottle<T> bottle) throws CaseFullException {
        if (isFull()) { throw new CaseFullException(); }
        bottles[count] = bottle;
        count++;
    }

    @SuppressWarnings("unchecked")
    public Bottle<T> getBottle(int index) {
        // safe type cast
        return (Bottle<T>) bottles[index];
    }
}
```

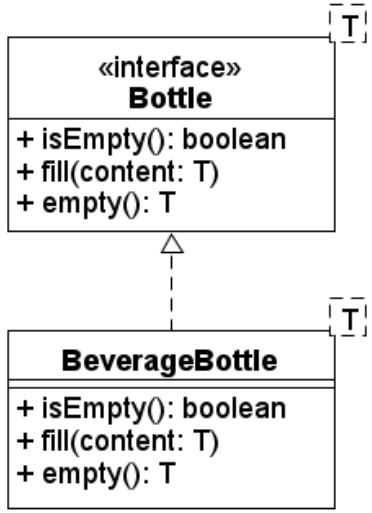




Getränkekisten (4)

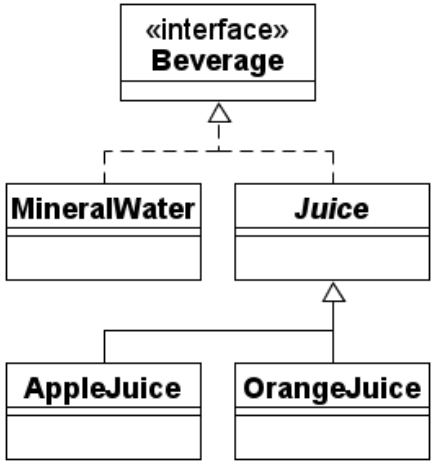
```
class BottleCase<T extends Beverage> {

    private Object[] bottles = new Object[12];
    private int count = 0;

    …

    public void add(Bottle<? extends Beverage> bottle)
            throws CaseFullException {
        if (isFull()) { throw new CaseFullException(); }
        bottles[count] = bottle;
        count++;
    }

    @SuppressWarnings("unchecked")
    public Bottle<? extends Beverage> getBottle(int index) {
        // safe type cast
        return (Bottle<? extends Beverage>) bottles[index];
    }
}
```

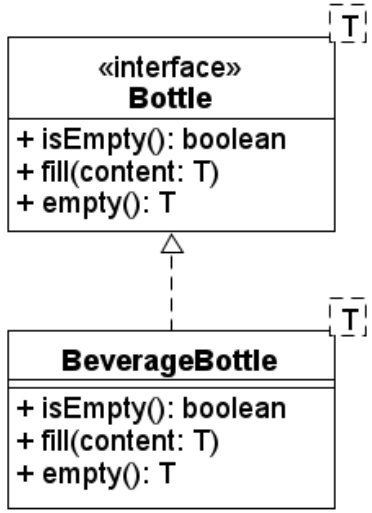




Getränkekisten (5)

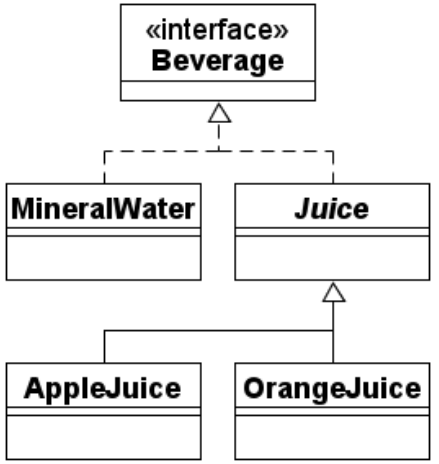
```
class BottleCase<T extends Bottle<? extends Beverage>> {

    private Object[] bottles = new Object[12];
    private int count = 0;

    …

    public void add(T bottle) throws CaseFullException {
        if (isFull()) { throw new CaseFullException(); }
        bottles[count] = bottle;
        count++;
    }

    @SuppressWarnings("unchecked")
    public T getBottle(int index) {
        // safe type cast
        return (T) bottles[index];
    }
}
```

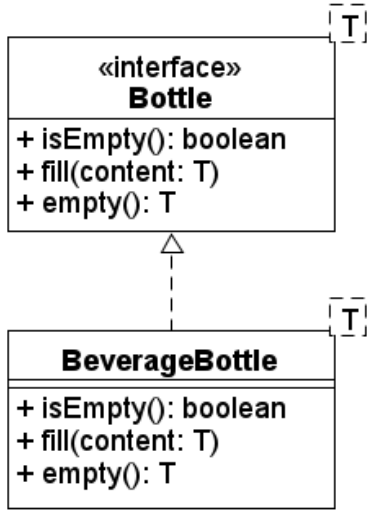




Getränkekisten (5)

```
BottleCase<Bottle<AppleJuice>> juiceBox = new BottleCase<>();
for (int i = 0; i < juiceBox.getCapacity(); i++) {
    Bottle<AppleJuice> bottle = new BeverageBottle<AppleJuice>();
    try {
        bottle.fill(new AppleJuice());
        juiceBox.add(bottle);
    } catch (BottleNotEmptyException e) {
        System.out.println("Bottle is not empty.");
    } catch (CaseFullException e) {
        System.out.println("Case is already full.");
    }
}
```





CARL VON OSSIETZKY universität OLDENBURG

Flaschen Umfüllen

```
class BottleTransfuser {

    public static <T extends Beverage> void transfuse(Bottle<T> source, Bottle<T> target)
            throws BottleNotEmptyException, BottleEmptyException {
        target.fill(source.empty());
    }

}
```

```
Bottle<MineralWater> waterBottle = new BeverageBottle<>();
try {
    waterBottle.fill(new MineralWater());
    Bottle<MineralWater> spareBottle = new BeverageBottle<>();
    BottleTransfuser.transfuse(waterBottle, spareBottle);
} catch (BottleNotEmptyException e) {
    System.out.println("Bottle is not empty.");
} catch (BottleEmptyException e) {
    System.out.println("Bottle is empty.");
}
```

CARL VON OSSIETZKY universität OLDENBURG

Lernziele

- Klassen mit Typparametern
- Methoden mit Typparametern