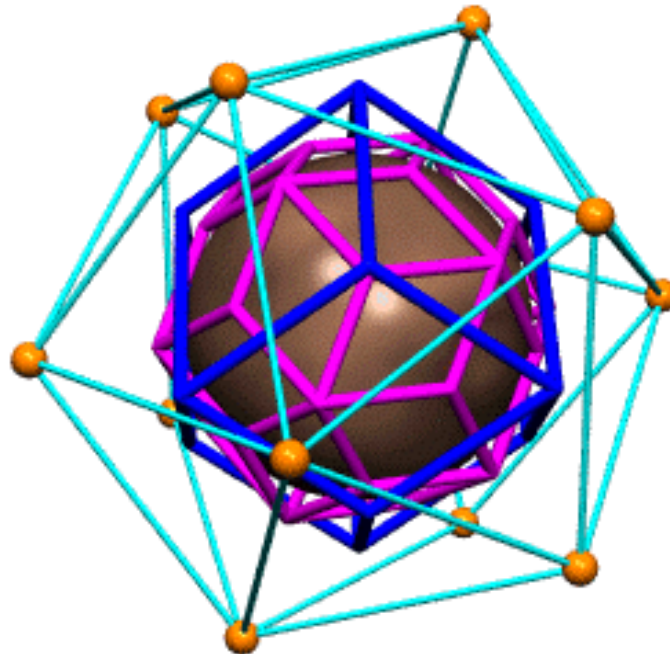


Innere Klassen in Java



SoSe 2018

Oliver Wiese

Innere Klassen

Klassen- oder Interfacedefinitionen können zur besseren Strukturierung von Programmen verschachtelt werden

Eine "**Inner Class**" wird innerhalb des Codeblocks einer anderen Klasse vereinbart.

Die bisher eingeführten Klassen werden auch **Top-Level-Klassen** genannt.

Einerseits erweisen sich innere Klassen als **elegante, sehr nützliche Zusätze** der Sprache, andererseits gibt es einige Regeln und Sonderfälle, die leicht verwirren können und deshalb vermieden werden sollten.

"Top-Level"-Klassen

```
public class TopLevelClass1 {  
    . . . .  
}
```



TopLevelClass1.java

```
public class TopLevelClass2 {  
    . . . .  
}
```



TopLevelClass2.java

Innere Klassen

```
public class OuterClass {  
    . . .  
    public class InsideClass {  
    }  
}
```

 OuterClass.java

Innere Klassen

Ziel:

Definition von Hilfsklassen möglichst nahe an der Stelle, wo sie gebraucht werden.

Motivation:

Geschachtelte und innere Klassen sind wesentlich motiviert durch das neue Ereignismodell im AWT von Java 1.1.

Alle Probleme lassen sich in Java mit externen Klassen lösen, aber oft verlässt man dadurch das **Konzept der Sichtbarkeit**.

Innere Klassen

Innere Klassen

- werden insbesondere in Zusammenhang mit der Programmierung von **Benutzeroberflächen** und **Iteratoren** eingesetzt.
- Innere Klassen **können auf Komponenten der sie umfassenden Klasse zugreifen**.
- innerhalb von Blöcken gelten die Zugriffsregeln für Blöcke.

Eine innere Klasse ist außerhalb des sie definierenden Blockes, außer mit Hilfe des voll qualifizierenden Namens, nicht sichtbar.

Innere Klassen

Der Name einer inneren Klasse setzt sich aus dem Namen der umfassenden Klasse, dem Trennzeichen \$ sowie dem Namen der Klasse zusammen.

OuterClass\$InsideClass.class

```
public class OuterClass {  
    . . .  
    public class InsideClass {  
  
    }  
    . . .  
}
```

Innere Klassen

Ab Java 1.1 gibt es vier Sorten von inneren Klassen.

- **Geschachtelte Klassen**

sind Top-Level-Klassen und Interfaces, die innerhalb anderer Klassen definiert sind, aber trotzdem Top-Level-Klassen sind.

- **Elementklassen**

innere Klassen, die in anderen Klassen definiert sind.

- **Lokale Klassen**

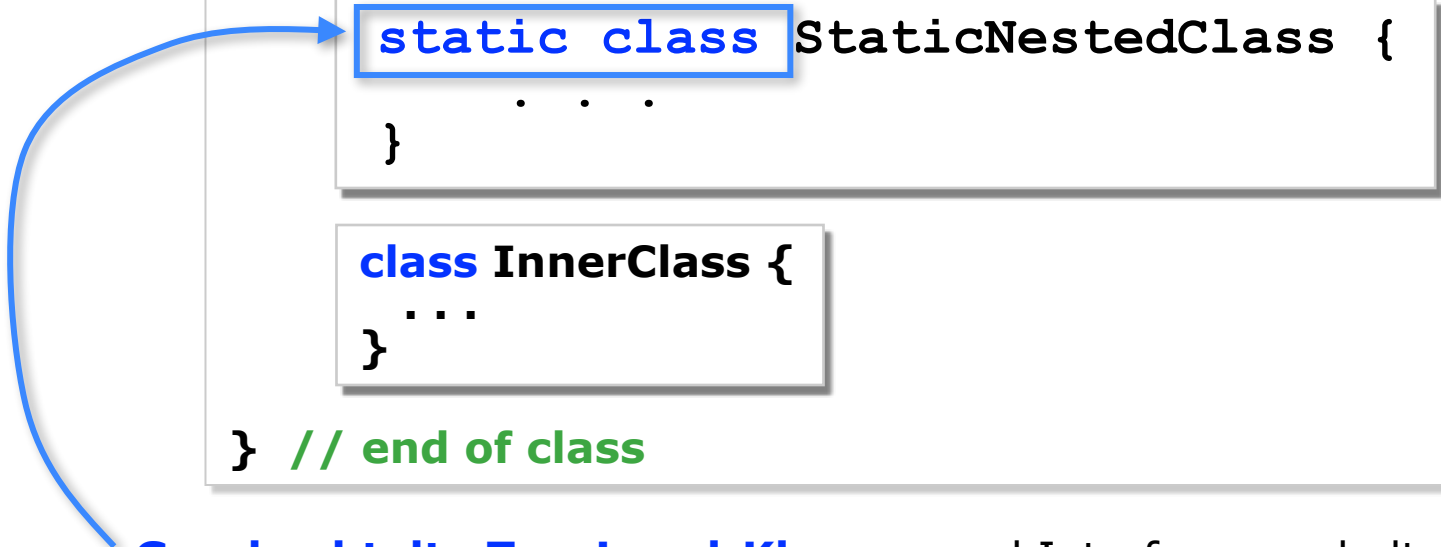
Klassen, die innerhalb einer Methode oder eines Java-Blocks definiert werden.

- **Anonyme Klassen**

Lokale und namenlose Klassen.

Geschachtelte Top-Level-Klassen

```
class EnclosingClass{
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
} // end of class
```



Geschachtelte Top-Level-Klassen und Interfaces verhalten sich wie andere Paket-Elemente auf der äußersten Ebene, außer dass ihrem Namen der Name der umgebenden Klasse vorangestellt wird. Sie werden als **static** deklariert

Geschachtelte Top-Level-Klassen

- * erlauben eine weitergehende Strukturierung des Namensraums von Klassen.

```
public class A {
    static int i = 4711;
    public static class B {
        int my_i = i;
        public static class C { ... }//end..C
    }// end of class B
}// end of class C
```

- * werden genauso verwendet wie normale *Top-Level*-Klassen

```
A a = new A();
A.B ab = new A.B();
A.B.C abc = new A.B.C();
```

Geschachtelte Top-Level-Klassen

```
public class A {
```

```
    static String a = "A";  
    String b = "B";
```

```
        public static class B {  
            void m() {  
                System.out.println( a );  
            }  
        } // end of class B
```

```
    } // end of class A
```

Zugriff nur auf
statische Variablen
der umrahmenden
Klassen.

Elementklassen

Elementklassen **echte innere Klassen** im Gegensatz zu den geschachtelten Top-Level-Klassen, die nur zur Strukturierung dienen.

Eine Elementklasse hat Zugriff auf alle Variablen und Methoden ihrer umgebenden Klasse.

Elementklassen werden analog gebildet und benutzt wie normale Klassen.

Der Compiler erzeugt beim Übersetzen einer Klassendatei für alle verschachtelte Klassen und Interfaces eigene **.class**-Dateien.

Elementklassen

Sie tragen den Namen der übergeordneten Klasse(n) und den eigenen, wobei diese mit **\$** unterteilt werden:

```
public class A {  
    ...  
    public class B {  
        ...  
        public class C {  
            ...  
        }  
    }  
}
```

javac A.java



A.class A\$B.class A\$B\$C.class

Auf diese Art wird vor der "Java Virtuelle Maschine" verborgen, dass diese Klassen eigentlich ineinander geschachtelt sind.

Elementklassen

Objekte von **Elementklassen** sind immer mit einem Objekt der umgebenden Klasse verbunden.

```
public class A {
    public static int i = 30;
    public class B {
        int j = 4;
        public class C {
            int k = i;
        }
    }
}
```

Objekterzeugung:

```
A a = new A();
A.B b = a.new B();
A.B.C c = b.new C();
```

Elementklassen

Jeder Instanz einer Elementklasse ist ein Objekt der umgebenden Klassen zugeordnet.

Damit kann das Objekt der Elementklasse implizit auf die Instanzvariablen der umgebenden Klasse zugreifen (**auch auf private Instanzvariablen**).

Elementklassen dürfen keine statischen Elemente (**Attribute, Methoden, Klassen, Interfaces**) besitzen.

Elementklassen

```
public class H {
```

```
    static String t = "text";
```

```
    String at = "another text";
```

```
    public class B {
```

```
        public void print() {
```

```
            System.out.println( at );
```

```
            System.out.println( t );
```

```
        }
```

```
    } // end of class B
```

```
} // end of class H
```

Zugriff auf alle
Variablen der
umgebenden
Klassen.

Lokale Klassen

Lokale Klassen sind innere Klassen, die nicht auf oberer Ebene in anderen Klassen verwendbar sind, sondern nur lokal innerhalb von Anweisungsblöcken von Methoden.

Lokale Klassen

Eine **Lokale Klasse** kann innerhalb einer Methode definiert und auch nur dort verwendet werden.

```
public class C {
    ...
    public void doSomething() {
        int i = 0;

        class X implements Runnable {
            public X() {...}
            public void run() {...}
        }
        new X().run();
    } // end of doSomething
}
```

Verwendung

Lokale Klassen

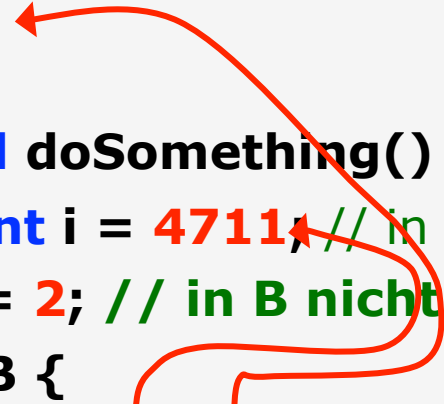
- Lokale Klassen dürfen folglich nicht als **public**, **protected**, **private** oder **static** deklariert werden.
- Lokale Klassen dürfen keine statischen Elemente haben (Felder, Methoden oder Klassen).
Ebenso dürfen sie nicht den gleichen Namen tragen wie eine der sie umgebenden Klassen.
- Eine Lokale Klasse kann im **umgebenden Codeblock** nur die mit **final** markierten Variablen und Parameter benutzen.

Lokale Klassen

```

public class A {
    int x = 42;

    public void doSomething() {
        final int i = 4711; // in B sichtbar
        int k = 2; // in B nicht sichtbar!
        class B {
            int j = i * x;
        } // end of class B
    } // end of method doSomething()
}
    
```



Lokale Klassen

```
public class H {  
    String t = "text";  
    public void m() {  
        final String mt = "in m";  
        class C {  
            void h() {  
                System.out.println( t );//Instanzvariable  
                System.out.println( mt );//mt ist final  
            }  
        }// end of class C  
        C in_m = new C();  
        in_m.h();  
    }// end of method m  
  
    public static void main( String[] args ) {  
        H h = new H();  
        h.m();  
    }  
}
```

Ausgabe:

```
text  
in m
```

Anonyme Klassen

"Einweg-Klassen"

Anonyme Klasse werden wie lokale Klassen innerhalb von Anweisungsblöcken definiert

haben keinen Namen. Sie entstehen immer zusammen mit einem Objekt

haben keinen Konstruktor

haben die gleichen Beschränkungen wie Lokale Klassen.

Anonyme Klassen

Die Syntax für Anonyme Klassen ist:

new-expression ***class-body***

Für die Anonyme Klasse kann man keine **extends-** oder **implements**-Klauseln angeben.

Anonyme Klassen

Beispiel:

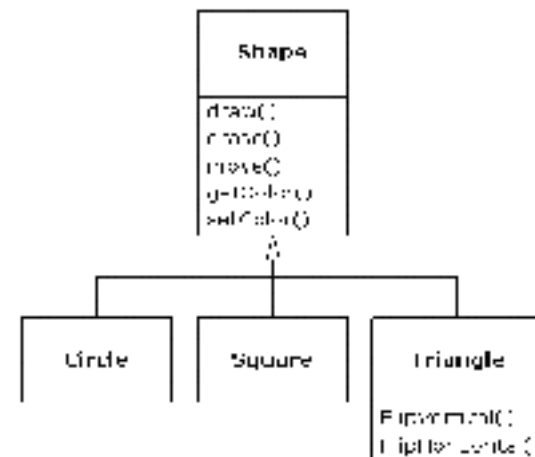
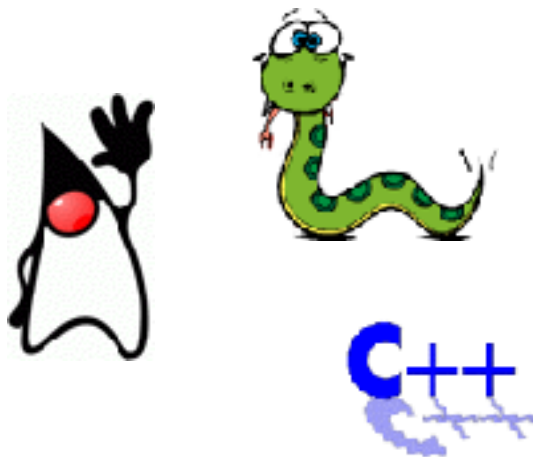
```
public void doSomething() {  
    Runnable r = new Runnable() {  
        public void run() { . . . }  
    };  
    r.run();  
}
```



Interface

Objektorientiertes Programmieren

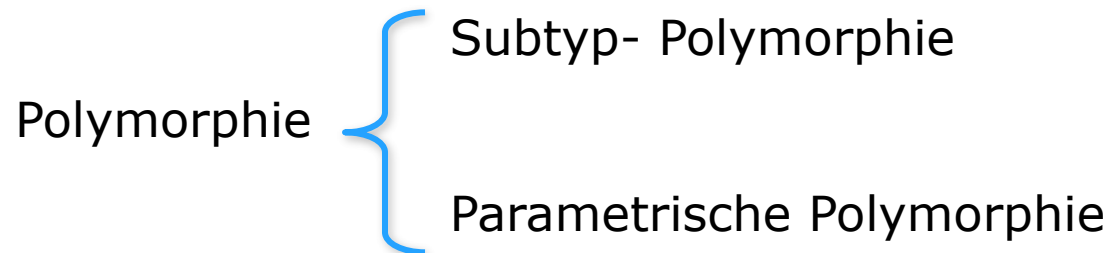
Polymorphie



Polymorphie

Polymorphie bedeutet Vielgestaltigkeit

Im Zusammenhang mit Programmiersprachen spricht man von Polymorphie, wenn Programmkonstrukte oder Programmteile für Objekte (bzw. Werte) mehrerer Typen einsetzbar sind.



Polymorphie

Polymorphie ist eines der wichtigsten Konzepte von OOP.

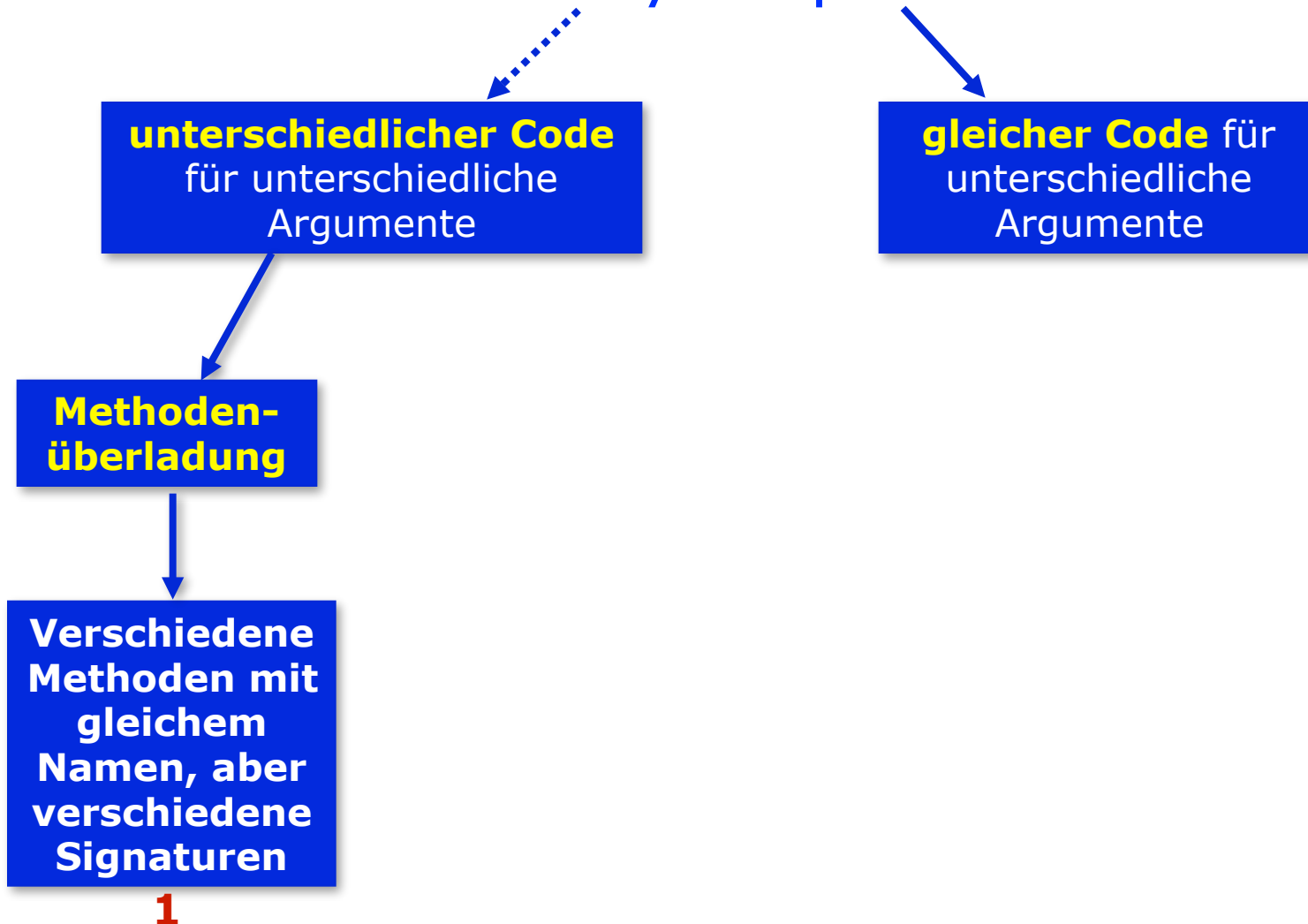
Objekte einer Unterklasse sind legale Exemplare der Oberklasse,
z.B. ein Schlaginstrument ist auch ein Musikinstrument.

Ein Musikinstrument-Objekt kann eine Instanz einer beliebigen Unterklasse von Musikinstrumenten beinhalten!

Wenn eine mehrfach überschriebene Methode mit einer Polymorph-Referenz aufgerufen wird, wird zur Laufzeit entschieden, welche Methode verwendet wird.

Entscheidend ist der aktuelle Objekttyp!

Polymorphie



Polymorphie

1 Methodenüberladung

In Java ist es erlaubt, Methoden mit dem gleichen Namen aber mit verschiedenen Argumentzahlen oder Argumenttypen zu implementieren.

Beispiele:

class PrintStream

```
public void println( int i )
public void println( double d )
public void println( boolean b )
public void println( char c )
public void println( String s )
public void println( Object o )
...
```

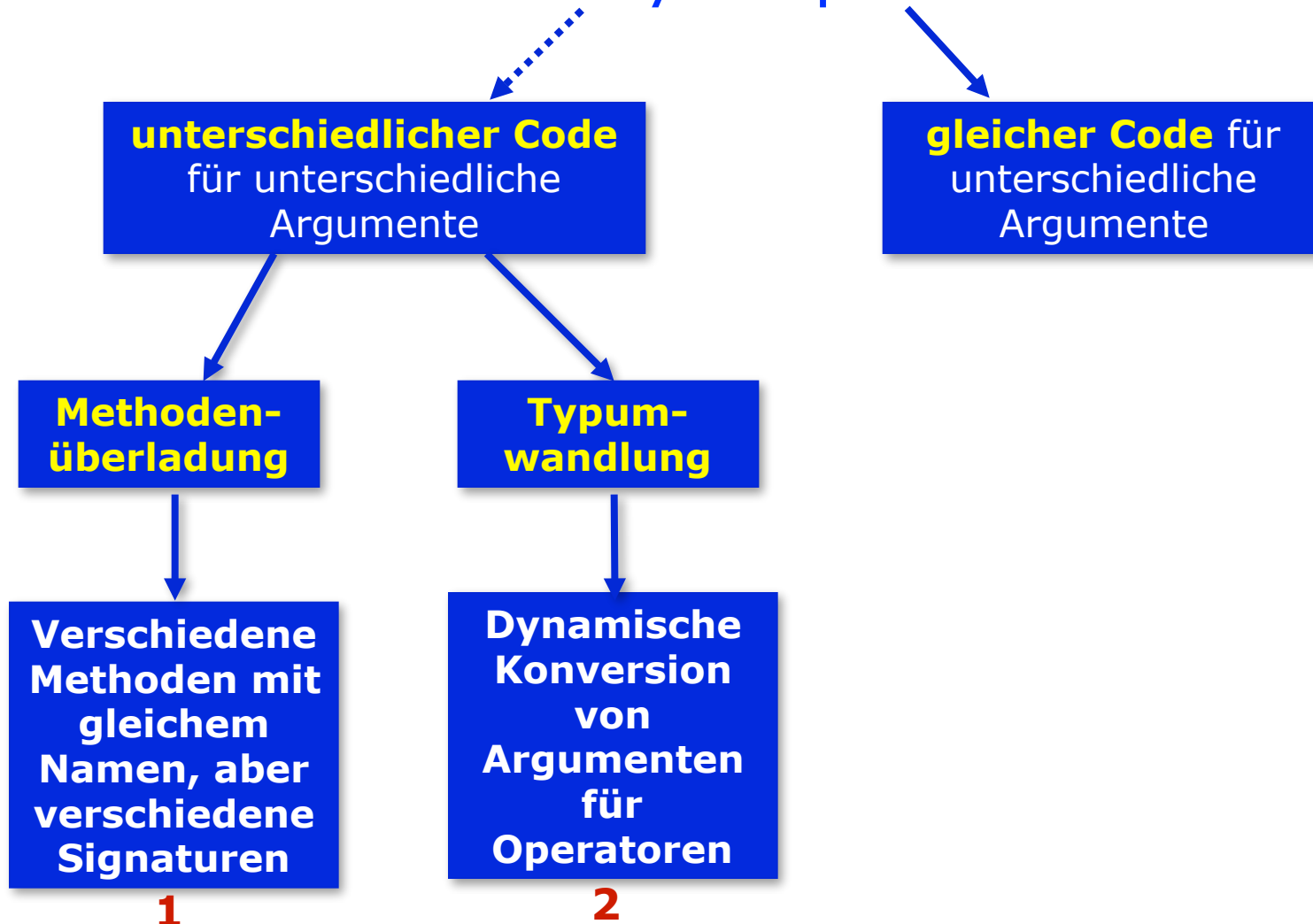
class Math

```
static double abs( double a )
static int abs( int a )
```

class Component

```
public void setSize( Dimension d )
public void setSize( int width, int height )
```

Polymorphie



Polymorphie

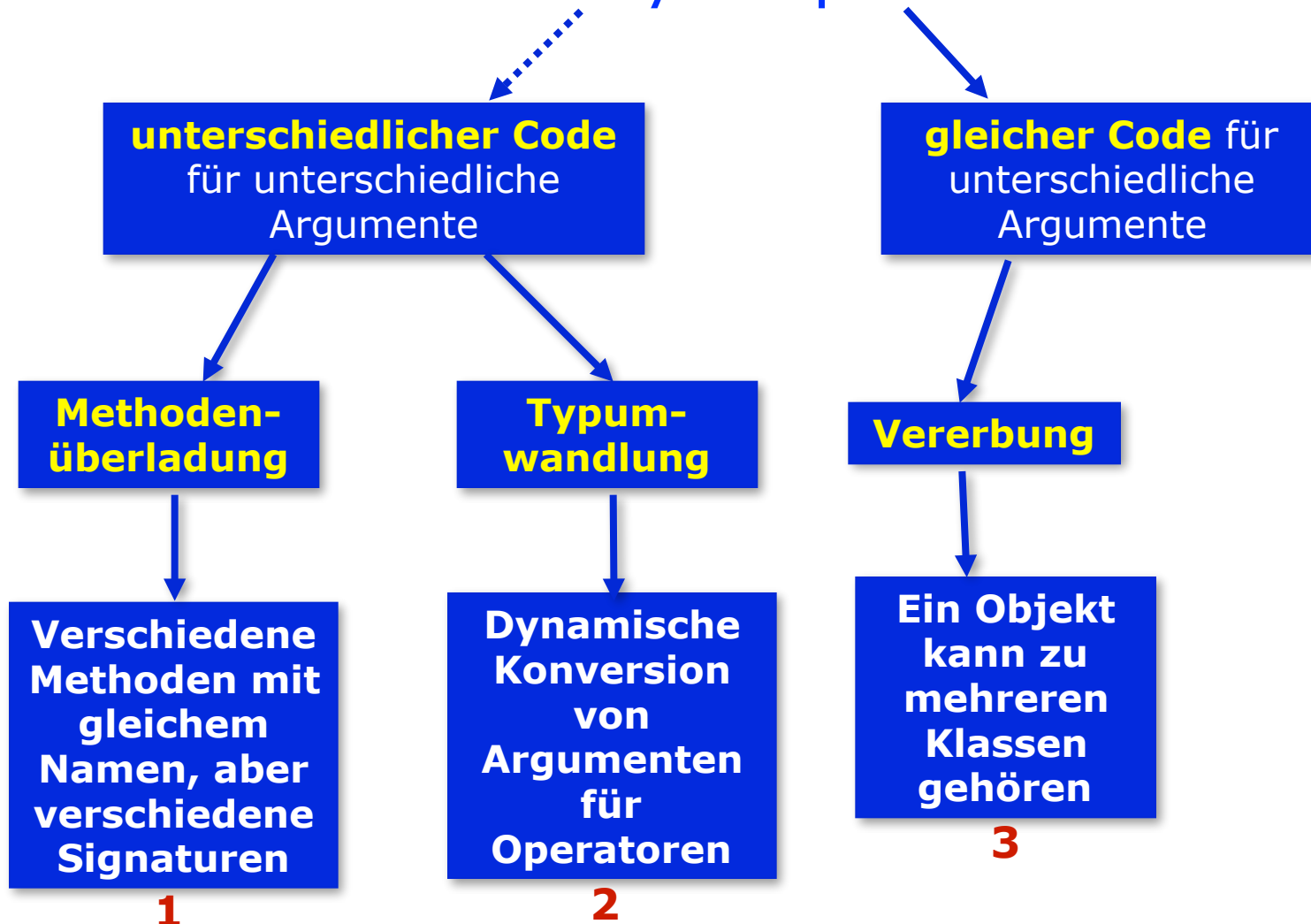
2 Typumwandlung

3.0 + 5 → 8.0
double int double

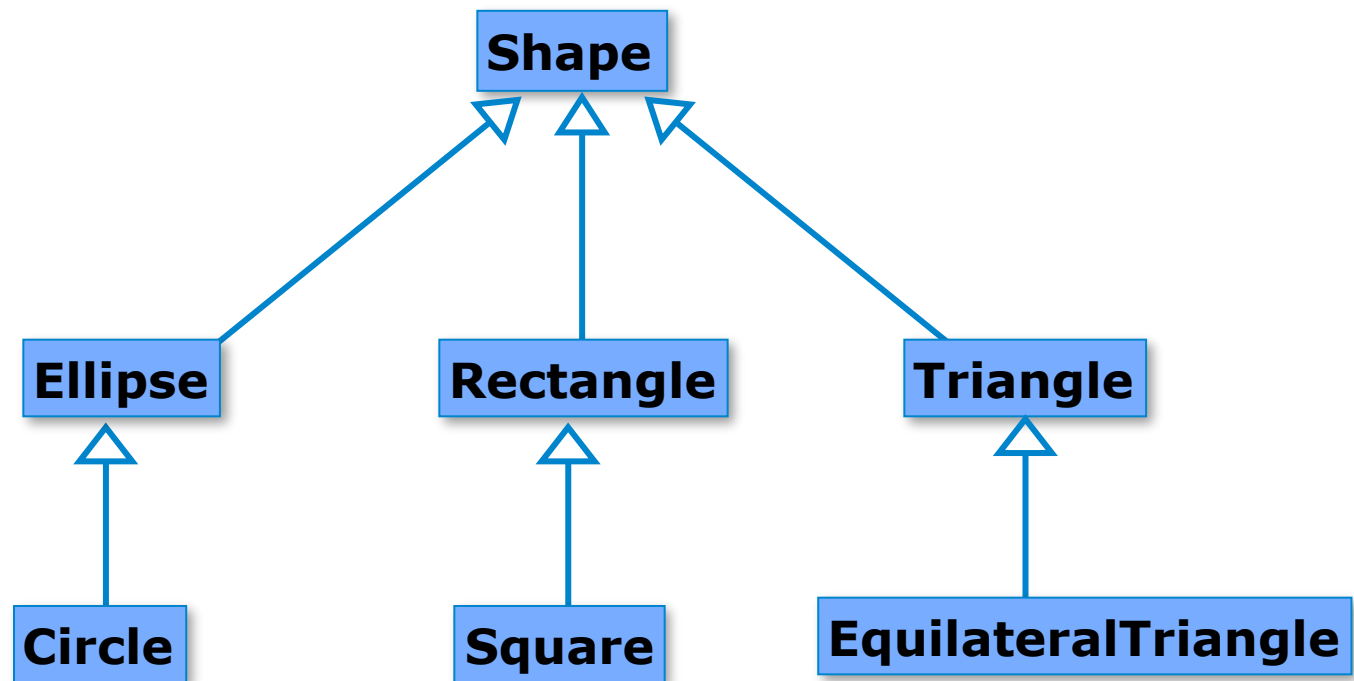
3.0 + " Kilogramm" → "3.0 Kilogramm"
double String String

3.0 + 2 + " Kilogramm" → "5.0 Kilogramm"
double int String String

Polymorphie



3 Vererbungspolymorphie (Subtyping)



3 Vererbungspolymorphie

Unterklassen können Methoden der Oberklasse **überschreiben** (overriding).

Der Name der Methode der Unterklasse "**verschattet**" den Namen der Methode der Oberklasse, wenn die überschriebene Methode dieselbe Signatur hat.

```
class Shape {
...
public abstract draw();
}
```

```
class Circle extends Shape {
...
draw() {
    paintCircle( );
}
...
}
```

```
class Rectangle extends Shape {
...
draw() {
    paintRectangle( );
}
...
}
```

3 Vererbungspolymorphie

Beispiel:

```
public abstract class Animal {
    ...
    public abstract void speak();
    ...
}
```

Unterklassen

```
public class Cat extends Animal {
    public void speak(){
        System.out.println("Miau Miau");
    }
}
```

```
public class Cow extends Animal {
    public void speak(){
        System.out.println("Muh Muh");
    }
}
```

```
public class Dog extends Animal {
    public void speak(){
        System.out.println("Wau Wau");
    }
}
```

3 Vererbungspolymorphie

Beispiel:

```
public class Zoo {

    Animal[] list;

    public Zoo( Animal[] list ){
        this.list = list;
    }

    public void sound(){
        for( Animal anim : list ){
            anim.speak();
        }
    }
    ...
}
```

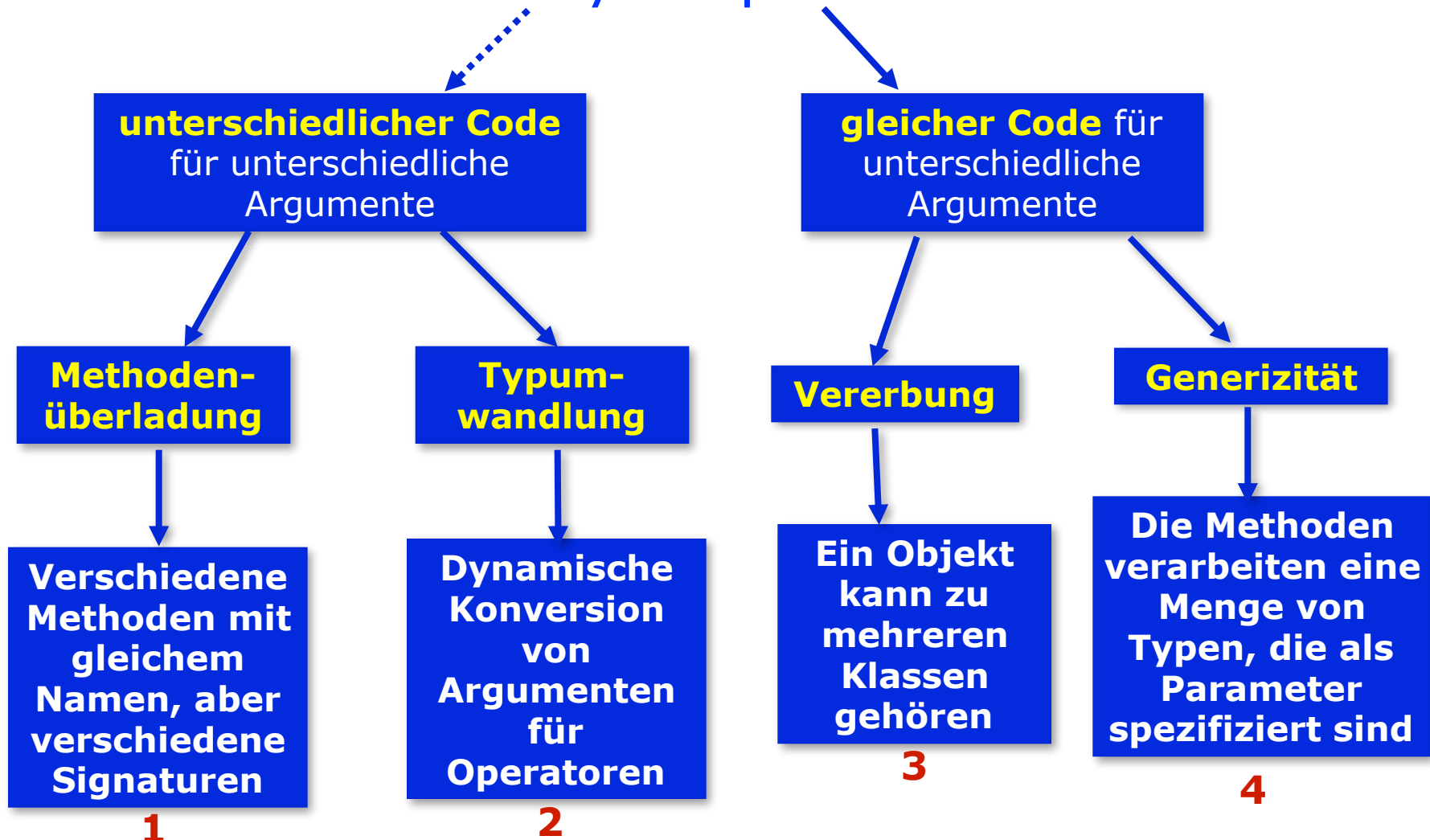
```
...
public static void main( String[] args ) {
    Animal[] anim_list = {
        new Cat(),
        new Dog(),
        new Cow(),
        new Cat(),
        new Dog()
    };

    Zoo zoo = new Zoo( anim_list );
    zoo.sound();
}
}
```

Ausgabe:

```
Miau Miau
Wau! Wau!
Muh, Muh ..
Miau Miau
Wau! Wau!
```

Polymorphie



Klassenschablonen

4 Generizität

Ziel ist das Einsparen von Programmieraufwand.

Klassen bzw. Methoden werden nur einmalig für viele verschiedene Objekt-Typen geschrieben.

In Java war dies schon seit Anfang an möglich, weil sämtliche Objekt-Typen von der Objekt-Klasse erben.

Das Problem ist, dass nicht sinnvolle Parameterübergaben oder Zuweisungen stattfinden können und zusätzliche Cast-Operationen notwendig sind.

Mit Generizität werden diese Probleme beseitigt.

Polymorphie

4 Generische Datentypen

Ab Java 1.5 werden **Collection**-Klassen als generisch betrachtet.

Die früher nur heterogenen Listen können jetzt parametrisiert werden, um daraus homogene Datenstrukturen zur Aufbewahrung von Objekten eines bestimmten Typs zu erzeugen.

```
class Entry<ET> {
    ET element;
    public Entry( ET element )
    {...}
    ...
}
```

Beispiel:

```
class Vector<ET> {
    ...
    public boolean add( ET o )
    {...}
    ...
}
```

Parametrisierte ArrayList-Klasse

Beispiel:

Es können nur
Rechteck-Objekte in
der ArrayList-Klasse
eingefügt werden.

Die Cast-Operation ist
nicht mehr nötig.

```
public class RechteckeWelt {  
  
    ArrayList <Rechteck> shapes;  
  
    public RechteckeWelt() {  
        shapes = new ArrayList <Rechteck> ();  
    }  
  
    public void add( Rechteck r ) {  
        shapes.add( r );  
    }  
  
    public void paint( Graphics g ) {  
        for ( int i=0; i<shapes.size(); i++ ) {  
            Rechteck r = shapes.get(i);  
            g.setColor( r.color );  
            g.fillRect( r.x, r.y, r.width, r.height );  
        }  
    }  
}
```

4 Generizität

Klassenschablonen

Beispiel:

```
public class GenericTest <T> {  
    private T wert  
    public GenericTest () {  
    }  
    public void setValue ( T wert ) {  
        this.wert = wert;  
    }  
    public T getValue() {  
        return wert;  
    }  
} // end of class GenericTest
```


4 Generizität

Beispiel:

```
public class TestGenericTest {

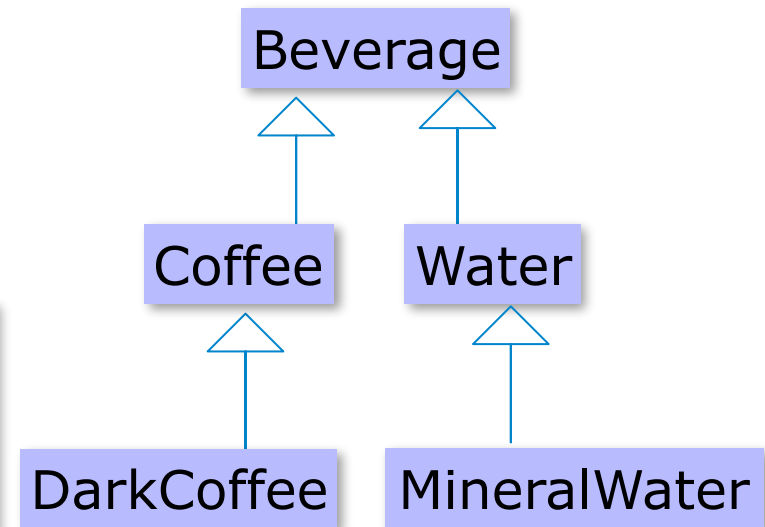
    public static void main(String[] args) {
        GenericTest<String> versuch = new GenericTest<String>();
        versuch.setValue( "hallo" );
        System.out.println( versuch.getValue() );
    }

} // end of class TestGenericTest
```

4 Generische Datentypen in Java

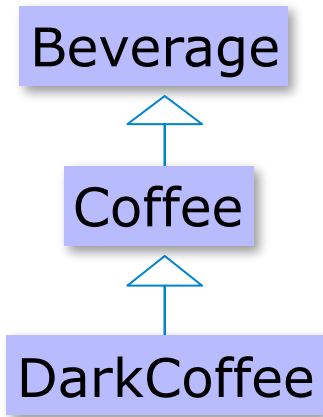


```
public class Cup<T> {  
    T beverage;  
    public Cup(T beverage) {  
        this.beverage = beverage;  
    }  
    ...  
}
```



Generische Datentypen und Vererbung in Java

folgende Zuweisungen sind **legal**.



```
DarkCoffee darkCoffee = new DarkCoffee();
```

```
Beverage beverage = new DarkCoffee();
```

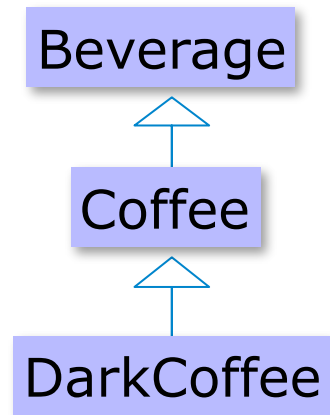
```
Cup<Coffee> cup = new Cup<Coffee>(darkCoffee);
```

```
Cup<DarkCoffee> cup = new Cup<DarkCoffee>(darkCoffee);
```

Typanpassungen

```
List<Integer> list = new ArrayList<Integer>();
```

Generische Datentypen und Vererbung in Java



folgende Zuweisung ist **illegal**

```
Cup<Coffee> cup2 = new Cup<DarkCoffee>(darkCoffee);
```

1. Fall Invarianz

- der Typparameter ist **eindeutig**
- **keine Einschränkung** auf den Typparameter
- aber **innerhalb von Zuweisungen keinerlei Flexibilität**
- **Typfehler können besser kontrolliert werden.**

erlaubt

```
Number n = new Integer(3);
```

nicht erlaubt!

```
ArrayList<Number> list = new ArrayList<Integer>();
```

 müssen gleich sein!

2. Fall Kovarianz

Referenzen müssen explizit mit der Syntax

<? extends T>

gekennzeichnet werden.

T ist der generellste Typ, der zugelassen ist (obere Einschränkung).

Beispiel:

```
Cup<? extends Beverage> cup =  
    new Cup<DarkCoffee>(darkCoffee);
```

```
Box<? extends Number> nBox =  
    new Box<Integer>(9);
```

Schlechte Erfahrung mit Arrays

Beispiel:

```
Number[] nums = new Integer[100];
```

ist OK, weil Integer von Number abgeleitet wird,

aber

```
nums[1] = 1.0;  
nums[2] = new Double(1.0);
```

verursachen ***ArrayStoreExceptions*** zur Laufzeit, weil

Integer <- Double

nicht zuweisungskompatibel sind.

Eingeschränkte Parametrisierung

Der Compiler kontrolliert, dass mindestens B und C Interfaces sind.



```
public class M<E extends A & B & C>  
{...}
```


3. Fall Kontravarianz

Referenzen müssen explizit mit der Syntax

<? super T>

gekennzeichnet werden.

untere Einschränkung

Beispiel:

```
Cup<? super DarkCoffee> cup =  
    new Cup<Beverage>(beverage);
```

```
Cup<? super DarkCoffee> cup =  
    new Cup<Object>(beverage);
```

4. Fall Bivarianz

Referenzen müssen explizit mit der Syntax

<?>

gekennzeichnet werden.

Beispiel:

Es gibt keine Einschränkung

Cup<?> cup = new Cup<String>(coffee);

```
Cup<?> cup5 = new Cup<String>("Hi");  
Cup<?> cup6 = new Cup<Object>(7);  
Cup<?> cup7 = new Cup<String>("Hi");  
cup5 = cup6;  
cup6 = cup7;
```

legale
Zuweisungen

Generische Datentypen

Eine Klasse kann mit mehreren Datentypen parametrisiert werden.

```
public class Generic <T1, T2, T3> { ... }
```

Anwendungsbeispiel:

```
public static void main(String[] args) {  
  
    Generic<String, Integer, Integer> sg =  
        new Generic<String, Integer, Integer>("Text", 3, 5);  
}
```

Generische Datentypen

Eine generische Klasse kann als Unterklasse einer anderen generischen Klasse definiert werden.

```
public class SpecialBox<E> extends Box<E> { ... }
```

Folgende Zuweisung ist dann legal.

```
Box<String> special = new SpecialBox<String>("Text");
```

Eingeschränkte Parametrisierung

```
public class MathBox<E extends Number>  
    extends Box<Number>  
{ ... }
```

Die Klasse **MathBox** kann mit beliebigen Datentypen, die als Unterklassen von **Number** definiert sind, parametrisiert werden.

Beispiele:

```
new MathBox<Integer>(5)
```

```
new MathBox<Double>(32.1)      Legal
```

```
new MathBox<String>("Zahlen");      Illegal
```

Methodenschablonen

```
public class Example {  
  
    public static <T> T randomChoose( T m, T n ){  
        return Math.random()>0.5 ? m : n;  
    }  
  
    public static void main(String args[]){  
        System.out.println(Example.randomChoose("Ja", "Nein"));  
        System.out.println(Example.randomChoose("Ja", 5));  
        System.out.println(Example.randomChoose(4.5, new Rectangle()));  
        String s = Example.randomChoose(new Rectangle(), "Text");  
    }  
}
```

Macht wenig Sinn!

Übersetzungsfehler nur hier!

Gebundene Typparameter

Innerhalb einer parametrisierten Klasse ist der Typparameter ein gültiger Datentyp, der innerhalb innerer Klassendefinitionen gebunden sein kann.

Beispiel:

```
public class OuterClass<T>{  
    ...  
    private class InnerClass<E extends T>{  
        ...  
    }  
}
```

Neue Collection Schnittstelle

```
interface Collection <T> {  
    /* Return true if the collection contains x */  
    boolean contains(T x);  
    /* Add x to the collection; return true if *the collection is changed. */  
    boolean add(T x);  
    /* Remove x from the collection; return true if * the collection is changed. */  
    boolean remove(T x);  
    ...  
}
```