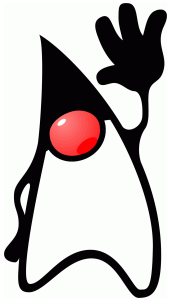
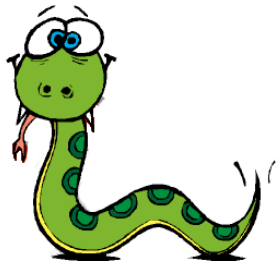
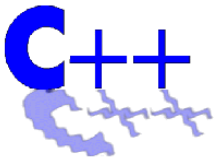
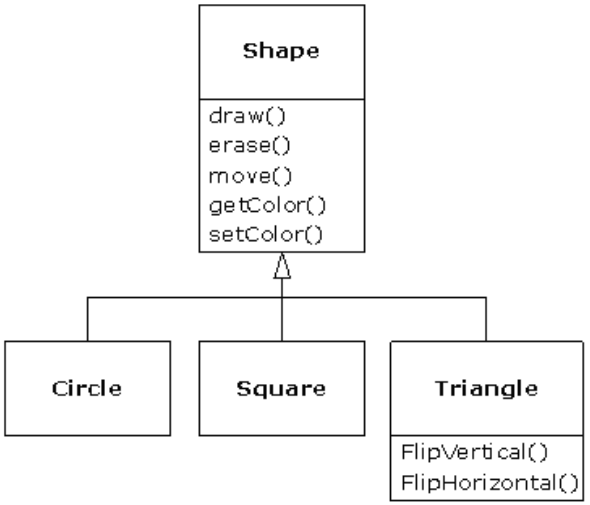
Objektorientiertes Programmieren

Polymorphie









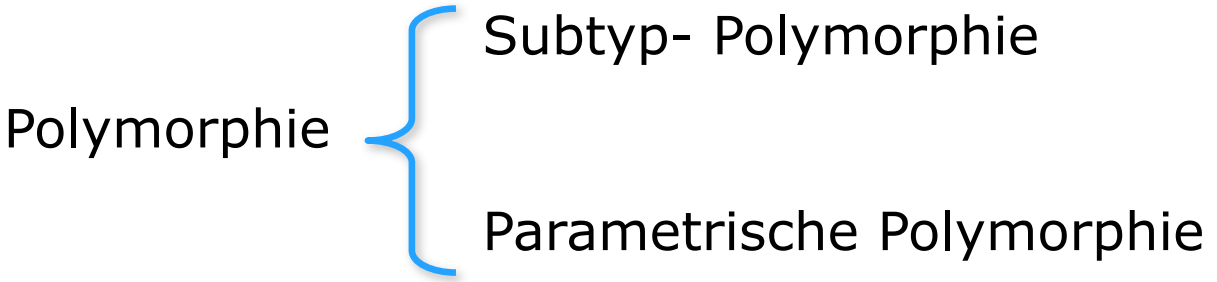
Prof. Dr. Margarita Esponda

SoSe 2020

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,

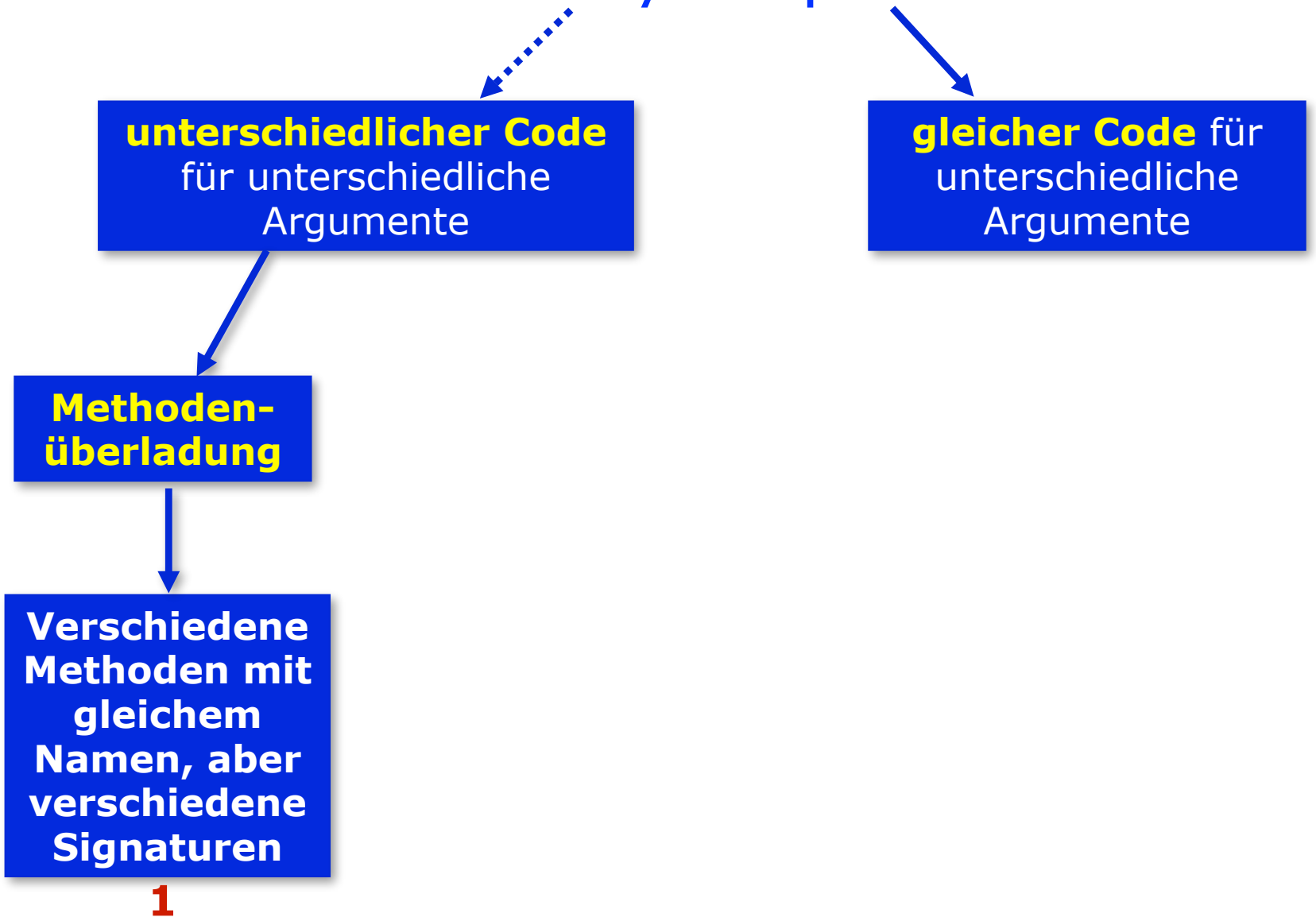
z.B. ein Schlaginstrument ist auch ein Musikinstrument.

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 )
...
```

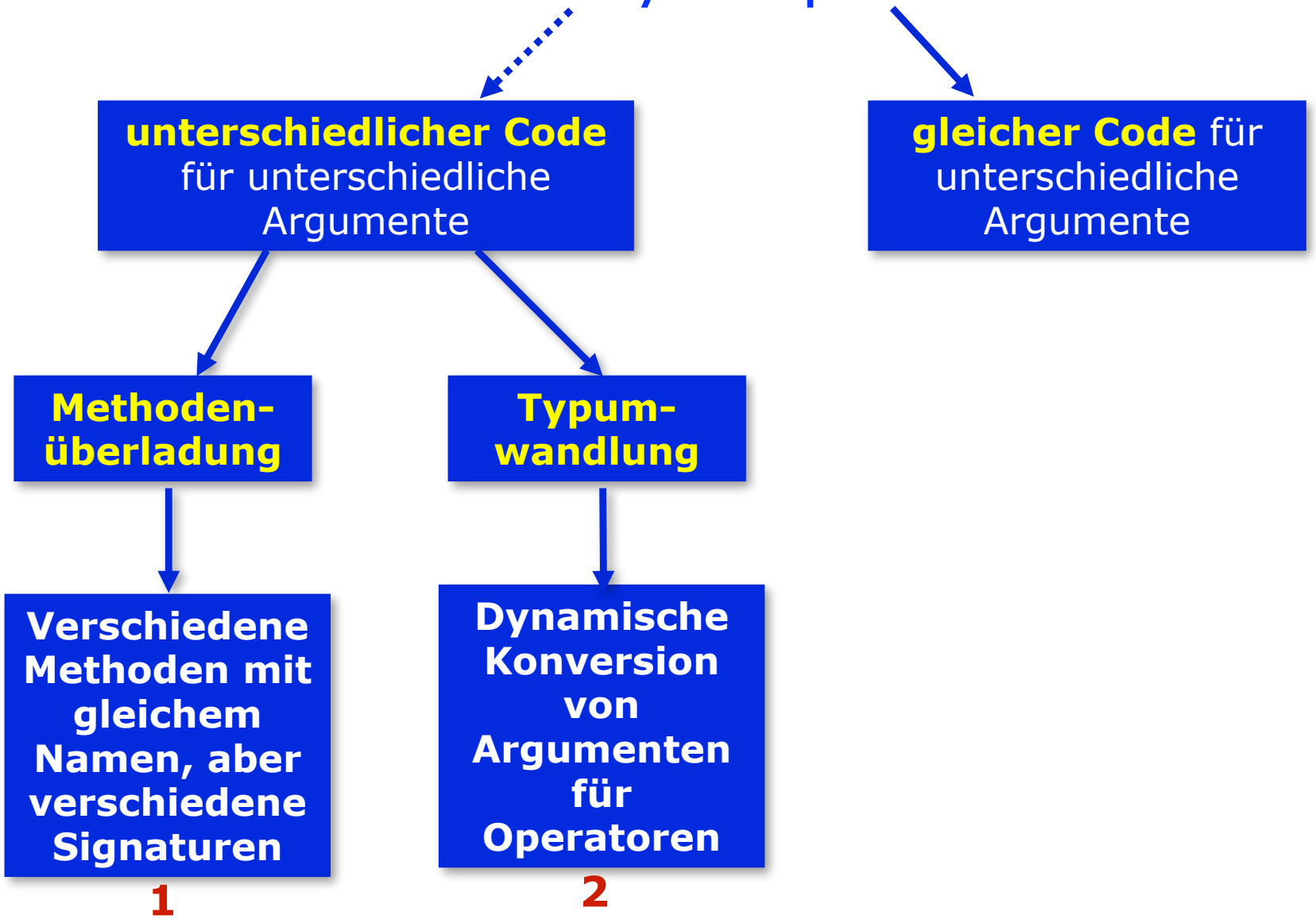
class Math

```
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

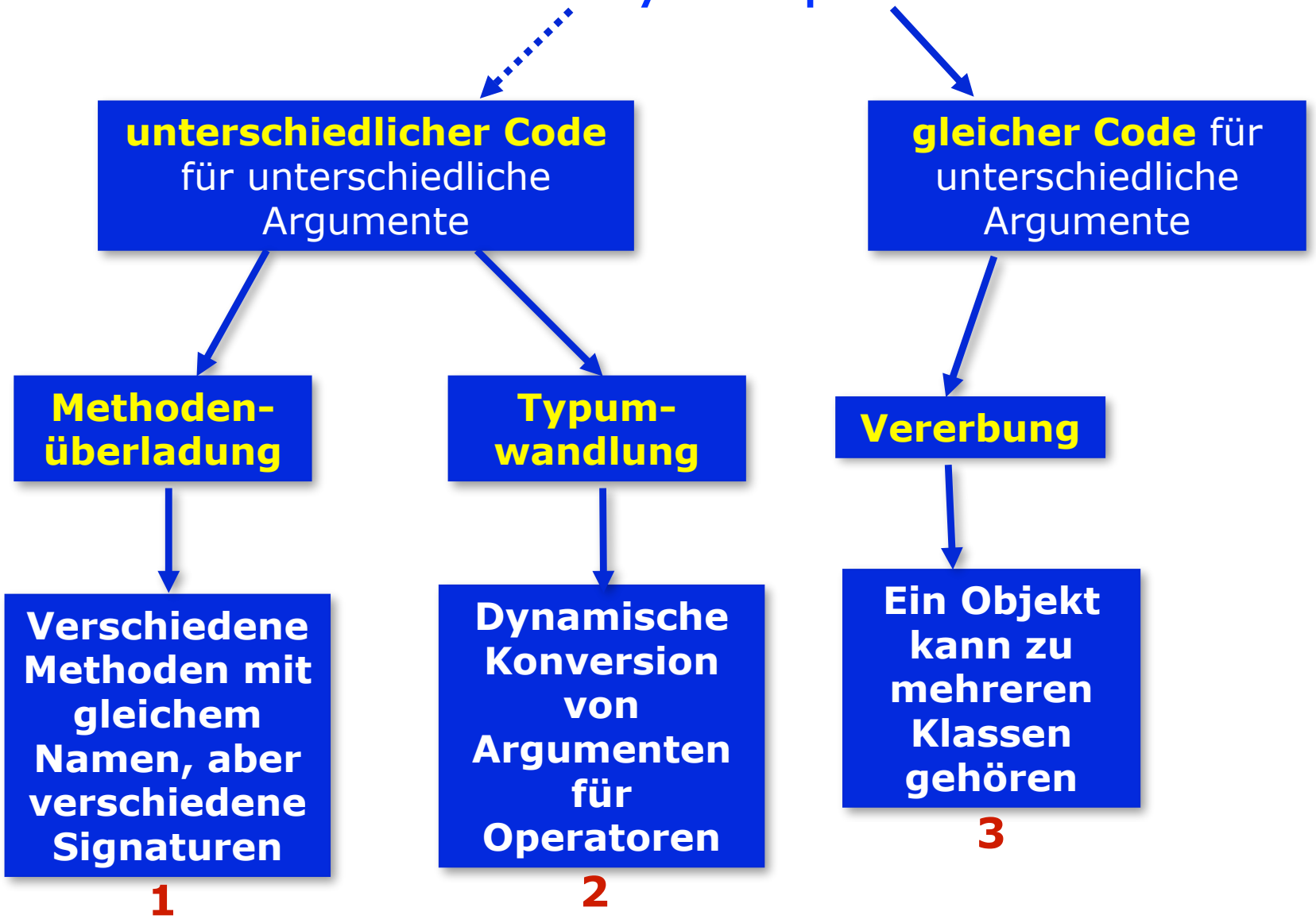
3.0 + " Kilogramm" → "3.0 Kilogramm"

double String String

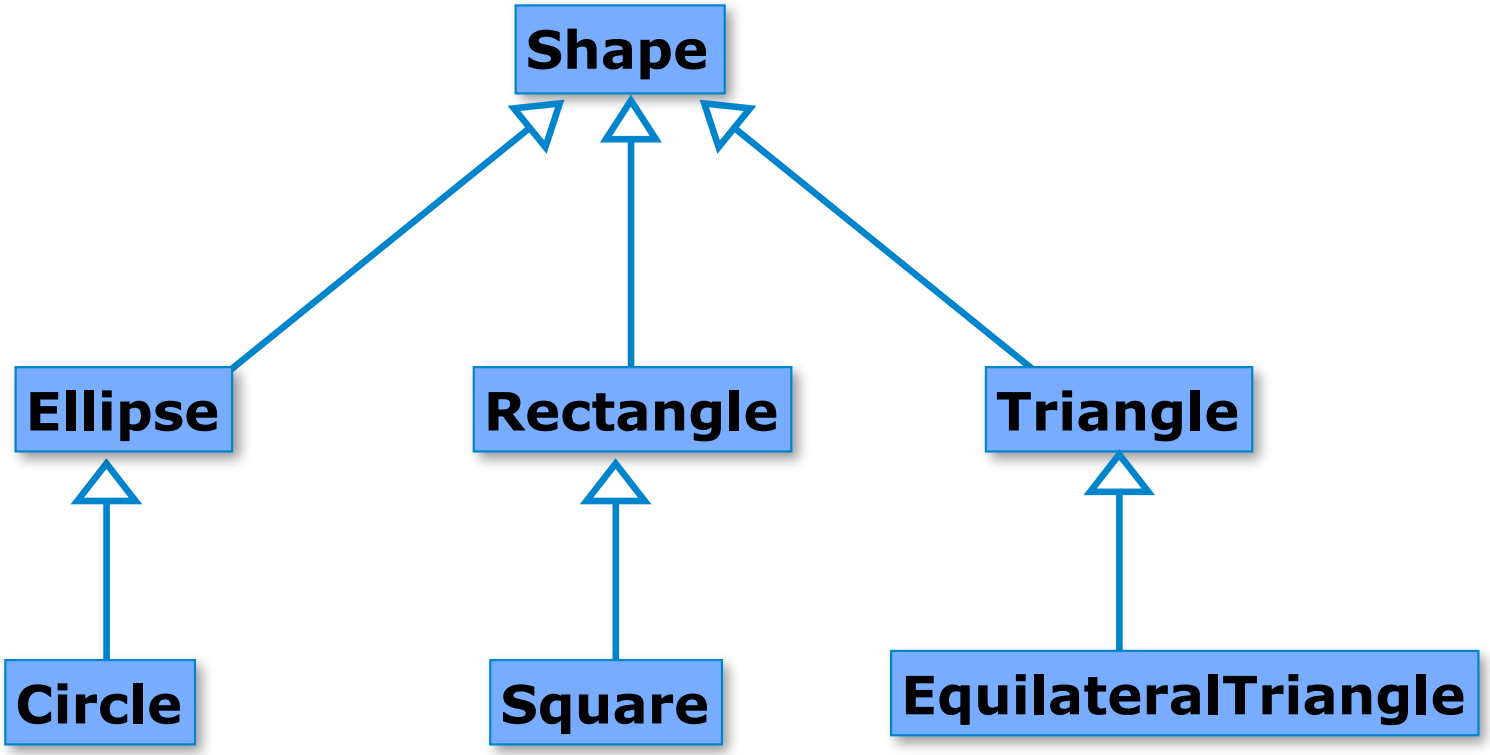
3.0 + 2 + " Kilogramm" → "5.0 Kilogramm"

double int String String

Polymorphie



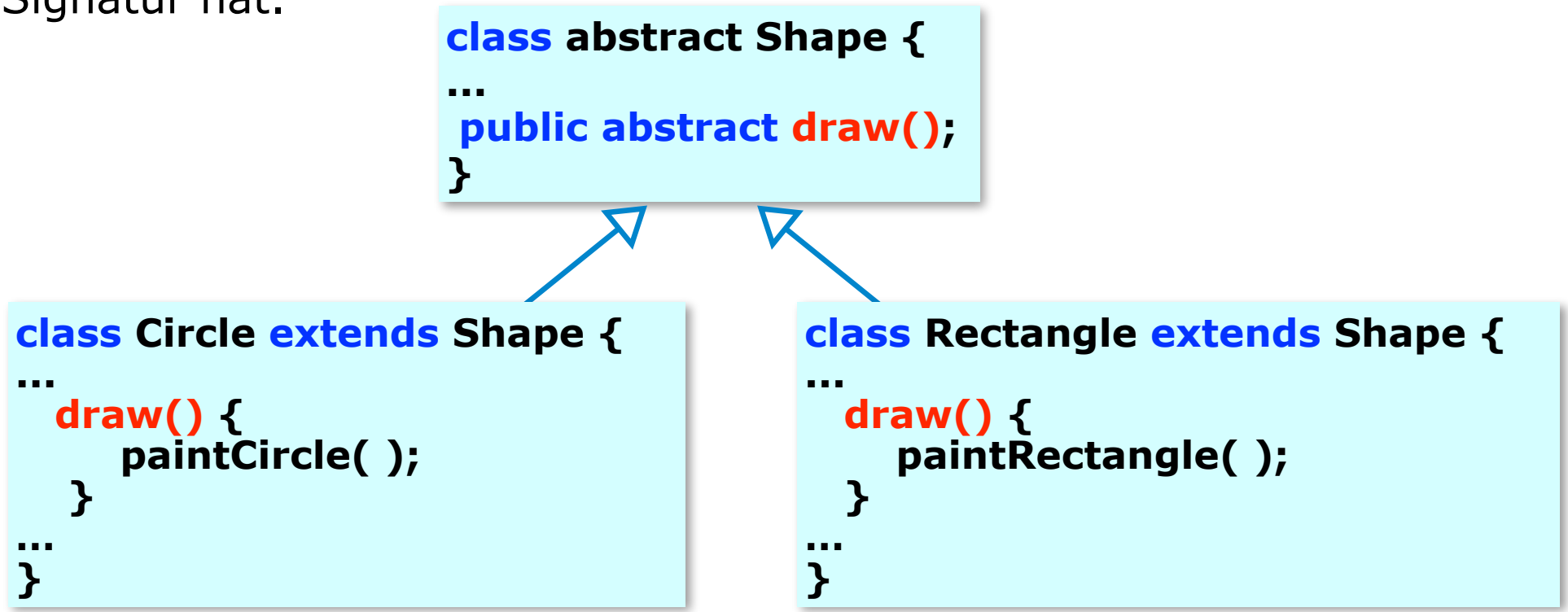
3 Vererbungspolymorphie (Subtyping)



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.



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:

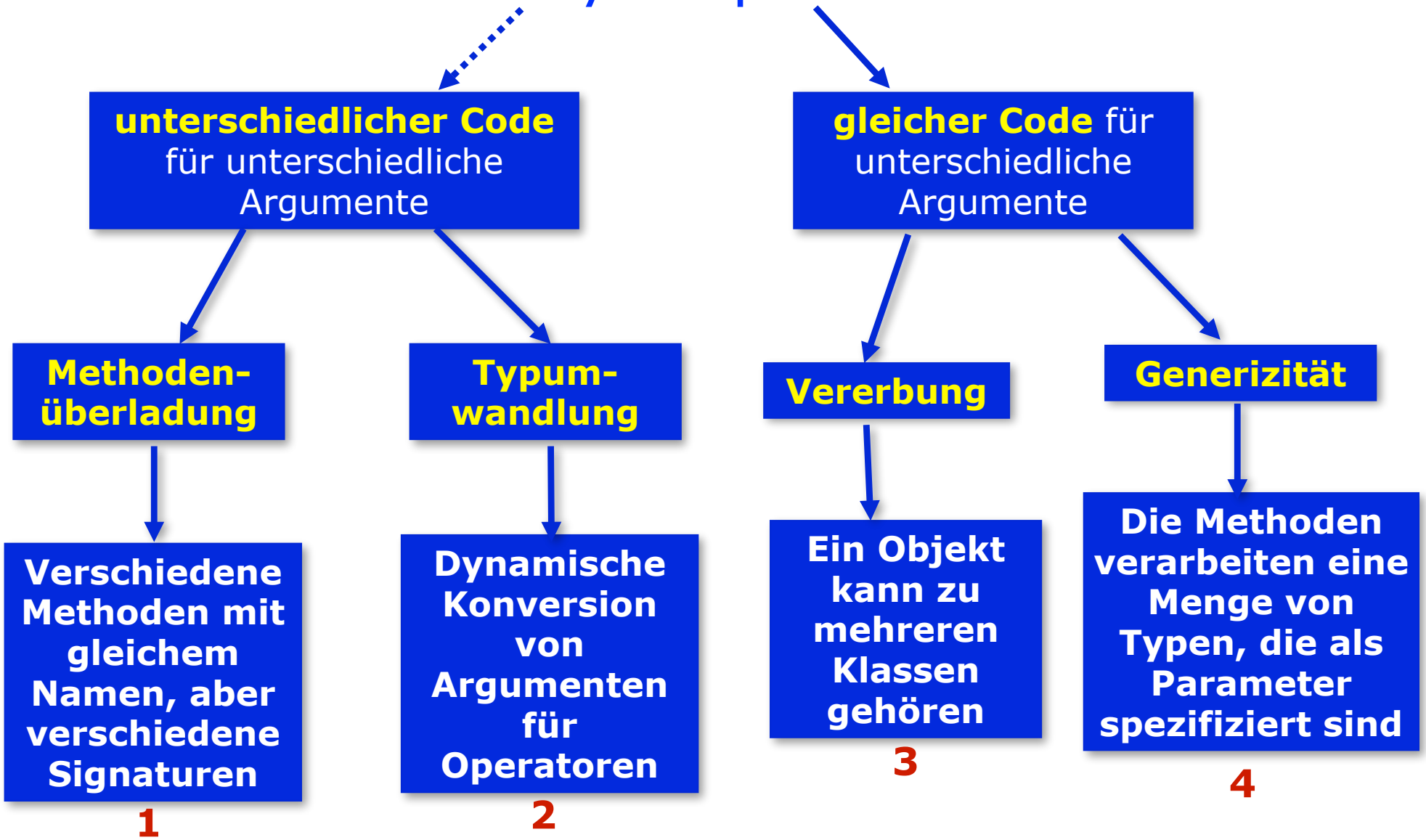
```
public class Zoo {

   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 )
    {...}
    . . .
}
```

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
```

Parametrisierte ArrayList-Objekt

Beispiel:

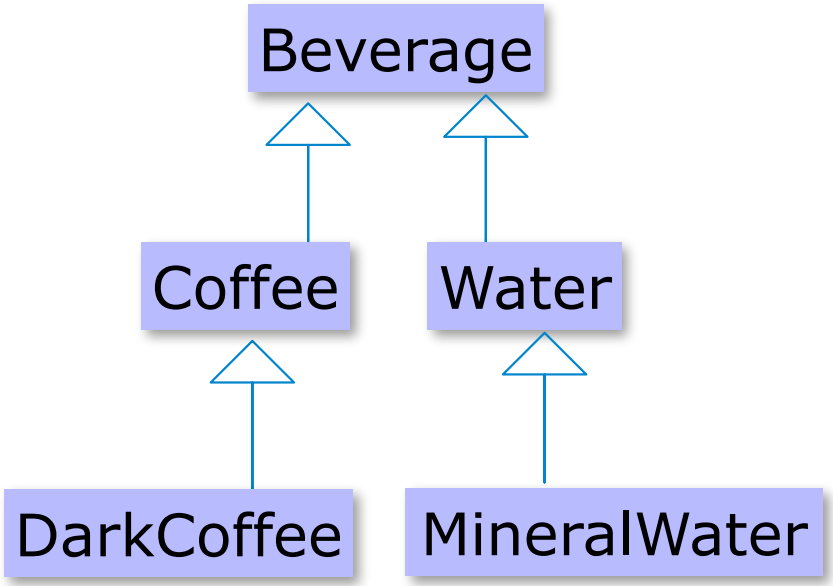
Es können nur Rechteck-Objekte in der ArrayList-Klasse eingefügt werden.

Die Cast-Operation ist nicht mehr nötig.

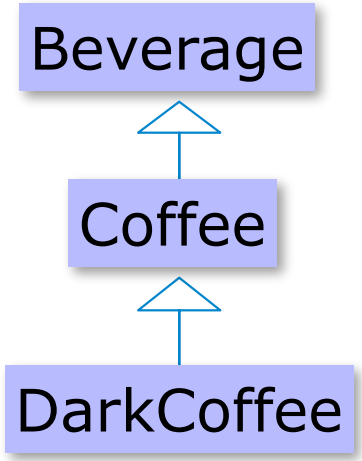
```
public class RechteckeWelt {

    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 Generische Datentypen in Java


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



Generische Datentypen und Vererbung in Java



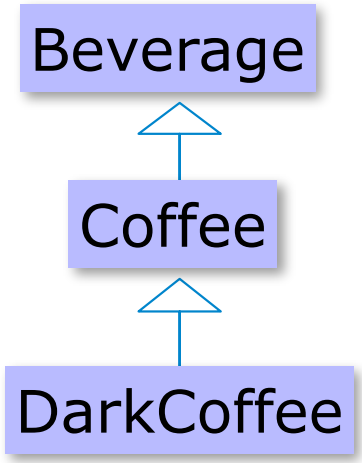
folgende Zuweisungen sind **legal**.

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

Cup<Coffee> cup = new Cup<Coffee>(coffee);
Cup<DarkCoffee> cup = new Cup<DarkCoffee>(coffee);
```

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 fromthe collection; return true if * the collection is changed. */
    boolean remove(T x);
...
}
```