Barry Linnert
Institut für Informatik

Freie Universität Berlin

# Programmieren

Barry Linnert

Sommersemester 2020

Freie Universität Berlin

# Gliederung der heutigen Vorlesung

- Kurze Wiederholung
- Polymorphie
  - Methodenüberladung
  - Typumwandlung
  - Vererbung
  - Generizität
- Zusammenfassung

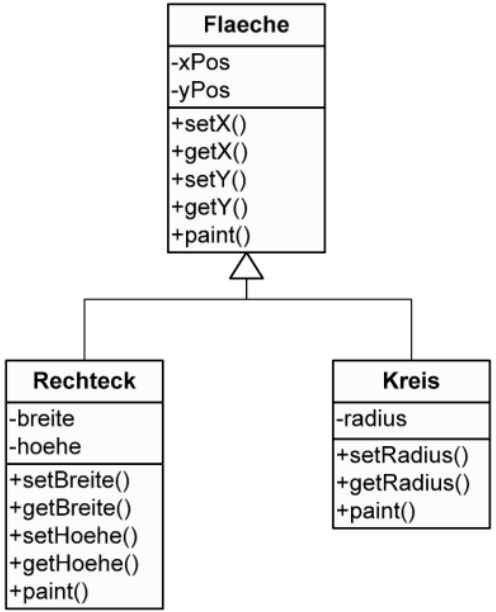
Freie Universität Berlin

Polymorphie

# WIEDERHOLUNG

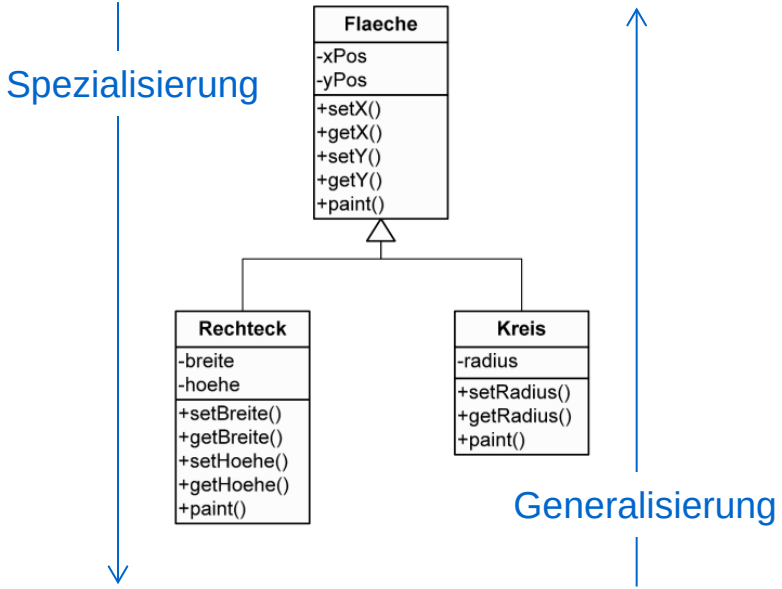
# Konzept der Vererbung

- Bei der Vererbung erbt eine Kindklasse alle Eigenschaften (Datenfelder und Methoden – sprich die Struktur und das Verhalten) ihrer Elternklasse.
- Die Kindklasse fügt den Eigenschaften der Elternklasse ihre eigenen individuellen Eigenschaften hinzu oder überschreibt Methoden der Elternklasse.
- Die Eigenschaften der Elternklasse müssen nicht in der Spezifikation der Kindklasse wiederholt werden.
- Die Kindklasse wird von der Elternklasse abgeleitet.



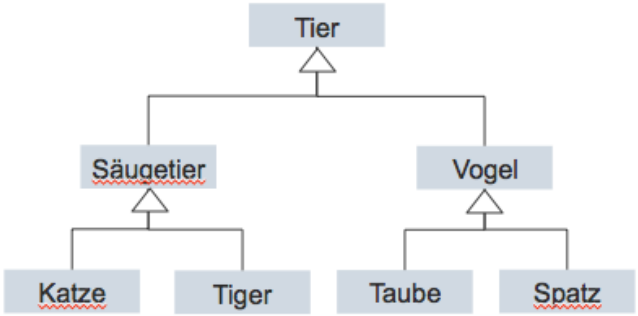
# Vererbungshierarchie mit Generalisierung und Spezialisierung

- Der Hauptzweck der Vererbung liegt in der Nutzung der Ähnlichkeit von neu zu schaffenden Klassen zu vorhandenen Klassen und zwar im Sinne von
  - **Spezialisierung, d.h.** Erweiterung - spezifische Attribute und Methoden legt man in einer Subklasse an und
  - **Generalisierung, d.h.** Abstraktion - allgemeingültige Attribute und Methoden gehören in eine Superklasse.



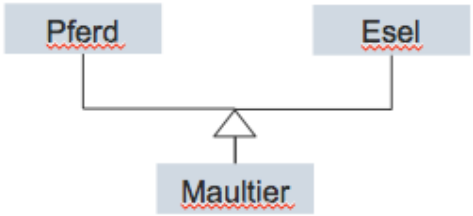
# Arten der Vererbung

- Eine **einfache** Vererbung liegt dann vor, wenn die Vererbungshierarchie ein Baum ist, d.h. außer der Basisklasse (Wurzel) hat jede Klasse genau eine direkte Oberklasse.




Java und C#

- Bei **mehrfacher** Vererbung hat mindestens eine Klasse mehr als eine direkte Oberklasse. Durch diese Eigenschaft ist es möglich, voneinander unabhängige Klassenhierarchien zusammenzuführen.




C++ und Python

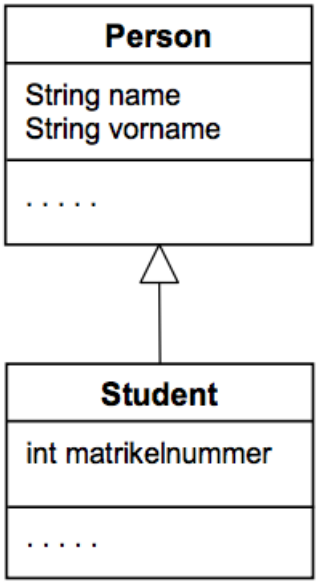
# Konstruktor

- Dient zum **Initialisieren eines Objektes.**
- Unterscheidet sich von einer normalen Methode unter anderem dadurch, dass der Konstruktor **ohne Rückgabetyp** (auch nicht void) deklariert wird.
- Unterscheidet sich weiterhin von einer normalen Methode dadurch, dass er **nicht** an eine abgeleitete Klasse **vererbt** wird.
- Wird vom Compiler dadurch erkannt, dass er den **gleichen Namen** trägt wie die Klasse selbst und keinen Rückgabewert hat.

# Konstruktoren bei abgeleiteten Klassen

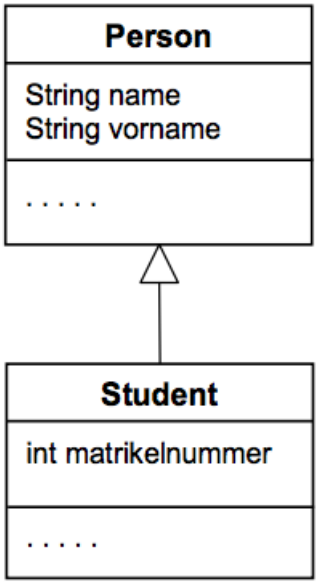
- Innerhalb eines Konstruktors einer abgeleiteten Klasse kann ein Konstruktor der Elternklasse mit dem Aufruf `super()` aufgerufen werden.

- Der Konstruktor der Elternklasse initialisiert die von der Elternklasse geerbten Anteile eines Objekts einer abgeleiteten Klasse.



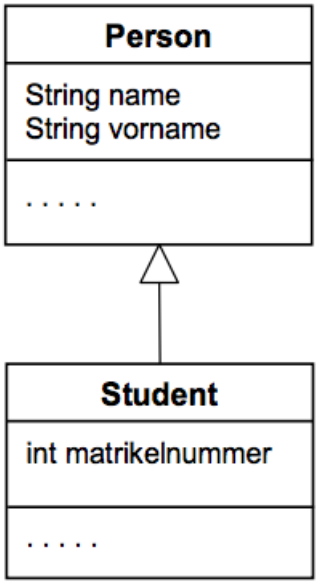
# Parameterloser Konstruktor

```
public class Person {
   private String name;
   private String vorname;
   public Person() {
      name = "Unbekannt";
      vorname = "Unbekannt”;}}

public class Student extends Person {
   private int matrikelnummer}
```



# Was macht der Compiler? (2)

```
public class Person extends Object {
    private String name;
    private String vorname;
    public Person() {
        super();
        name = "Unbekannt";
        vorname = "Unbekannt”;}}
```

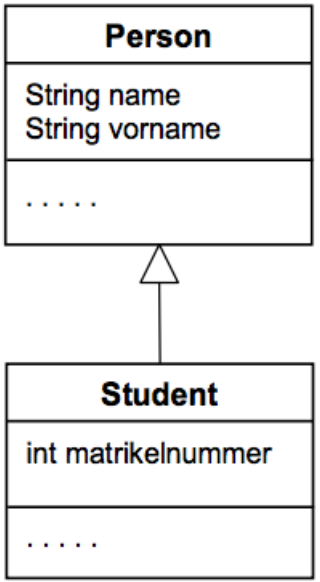



```
public class Student extends Person {
   private int matrikelnummer
   public Student() // Default-Konstruktor wird
   {                // vom Compiler eingefuegt.
     super();
   }}
```

# Was macht der Compiler? (3)

```
public class Person extends Object {
   private String name;
   private String vorname;
   public Person() {
      super();
      name = "Unbekannt";
      vorname = "Unbekannt”;}}
```

```
public class Student extends Person {
    private int matrikelnummer
    public Student() // Default-Konstruktor wird
    {                // vom Compiler eingefuegt.
      super();
    }}
```

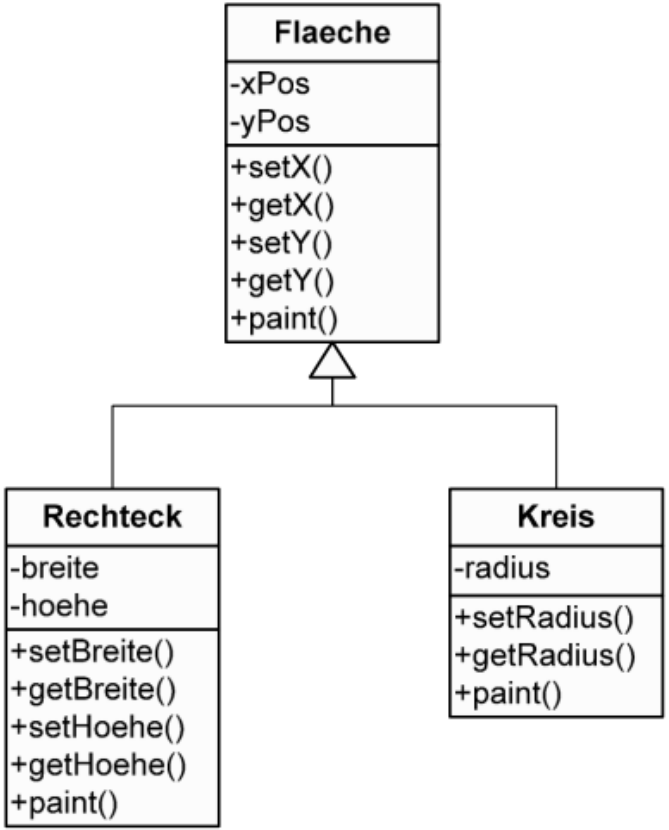


# Aufruf eines Konstruktors mit Parametern

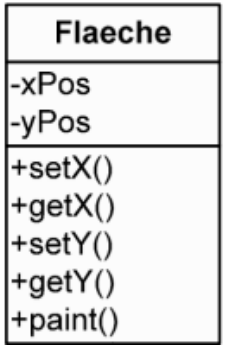
```
public class Person {
  private String name;
  private String vorname;
  public Person (String name, String vorname) {
     this.name = name;
     this.vorname = vorname;}}

public class Student extends Person {
  private int matrikelnummer;
  public Student (String name, String vorname, int m) {
       super (name, vorname);
       matrikelnummer = m; } }
```

Aufruf des Konstruktors der Superklasse

# Was ist das Objekt Flaeche?



```
public abstract class Flaeche {

// Variablen
  private int xPos;
  private int yPos;

// Konstruktor
public Flaeche(int x, int y) { setX(x); setY(y); }

// Methoden
public void setX(int x) { if (x >= 0) xPos = x; }
public int getX() { return xPos; }
public void setY(int y) { if (y >= 0) yPos = y; }
public int getY() { return yPos; }
public abstract void paint();
}
```



Freie Universität Berlin

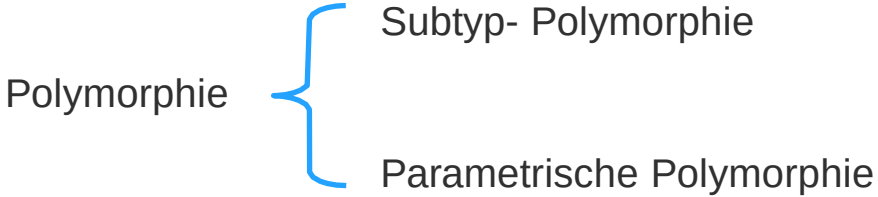
Polymorphie

# EINFÜHRUNG

# 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.

- Polymorphie (Vielgestaltigkeit) von Objekten bedeutet, dass ein Objekt in einer Vererbungshierarchie von verschiedenem Typ sein kann. Es kann vom Typ einer Basisklasse sein, aber auch vom Typ einer Unterklasse.

- 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

**unterschiedlicher** Code
für unterschiedliche Argumente

**gleicher** Code
für unterschiedliche Argumente

Methoden-
überladung

Verschiedene Methoden mit gleichem Namen, aber verschiedene Signaturen

1

Freie Universität Berlin

# 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 )
```

Freie Universität Berlin

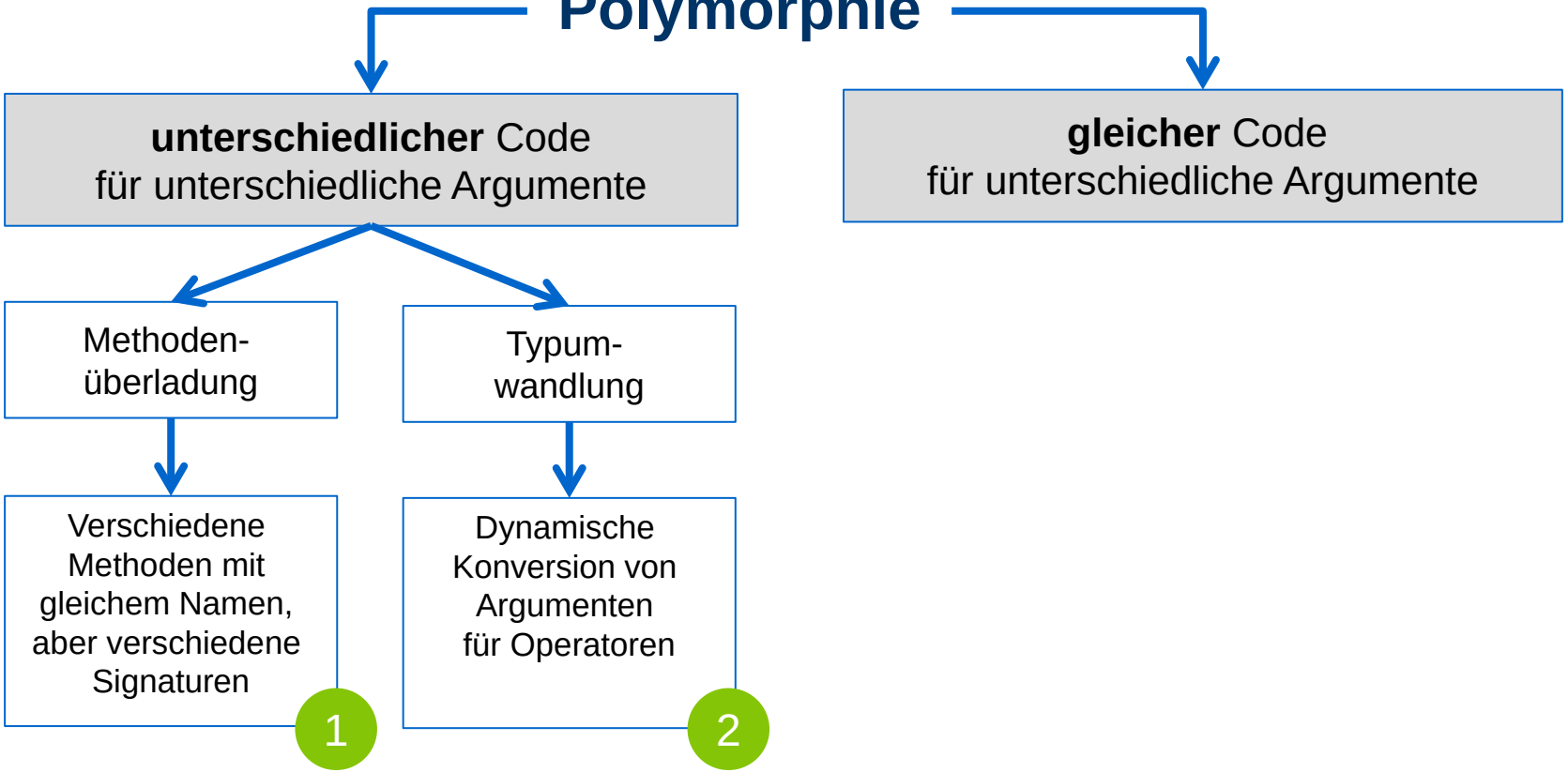
# Überladen von Konstruktoren

- Die Konstruktoren werden vom Übersetzer durch die Anzahl und den Typ der Parameter unterschieden.

```
public Person ( String vorname, String nachname, Date geburtsdatum )
public Person ( String vorname, String nachname )
public Person ( String vorname )
public Person ()
```

Signatur eines Konstruktors
Signatur = Methodenname + Parameterliste

# Polymorphie



# Konvertierung von Datentypen

**Explizite**
Typumwandlung

- Numerische Datentypen
- Referenztypen

**Implizite** (automatische)
Typumwandlung

- Numerische Datentypen
- Referenztypen
- Verknüpfungen der Klasse String
- Automatisches Boxing bzw. Unboxing

# Konvertierung von Datentypen

**Explizite Typumwandlung**

- Numerische Datentypen
- Referenztypen

**Implizite** (automatische) Typumwandlung

- Numerische Datentypen
- Referenztypen
- Verknüpfungen der Klasse String
- Automatisches Boxing bzw. Unboxing

# Explizite Typumwandlung

- Eine **explizite Typumwandlung** eines beliebigen Ausdrucks kann man mit dem **cast-Operator** (**Typkonvertierungsoperator**) durchführen.

- Durch (Typname) Ausdruck wird der Wert des Ausdrucks in den Typ gewandelt, der in den Klammern eingeschlossen ist

- Es kann nicht jeder Typ eines Operanden explizit in einen beliebigen anderen Typ umgewandelt werden. Möglich sind folgende Wandlungen
  - zwischen numerischen Datentypen und
  - zwischen Referenztypen.

Freie Universität Berlin

# Beispiele Explizite Typumwandlung

## Numerische Datentypen

```
int a = 1;           // a hat den Wert 1
Double b = 3.5;      // b hat den Wert 3.5
a = (int) b;         // Explizite Typkonvertierung in den Typ int

a = (int) 4.1;      // a bekommt den Wert 4 zugewiesen.
a = (int) 4.9;      // a bekommt ebenfalls den Wert 4 zugewiesen.
```

# Beispiele Explizite Typumwandlung

## Referenztypen

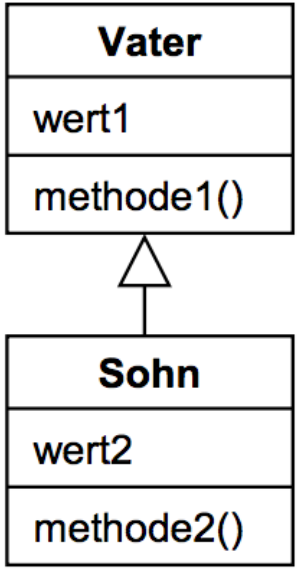
- Eine explizite Typkonvertierung von Referenzen mit Hilfe des cast-Operators muss immer dann erfolgen, wenn bei einer Zuweisung eine Referenzvariable vom Typ Eltern, die auf ein Objekt der Klasse Kind zeigt, einer Referenzvariablen vom Typ Kind zugewiesen wird.

- Ein **Down-Cast** bezeichnet einen Cast in einen Typ, der in der Vererbungshierarchie weiter unten liegt.

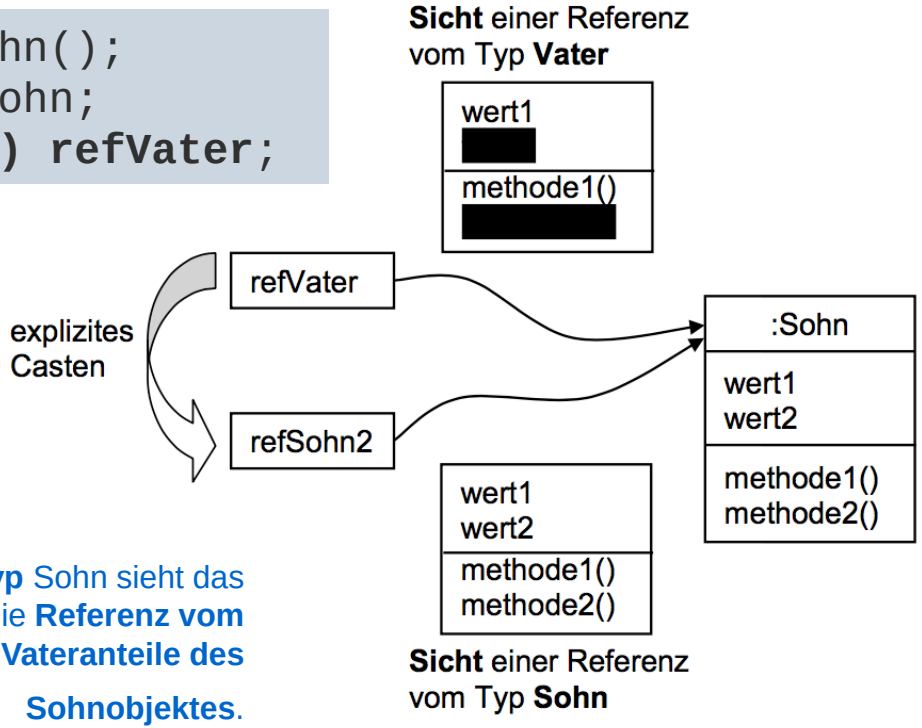
- Ein Down-Cast erfordert immer die explizite Angabe des cast-Operators.

# Beispiele Explizite Typumwandlung

## Referenztypen




```
Sohn refSohn = new Sohn();
Vater refVater = refSohn;
Sohn refSohn2 = (Sohn) refVater;
```




Die **Referenz vom Typ** Sohn sieht das **gesamte Objekt** und die **Referenz vom Typ** Vater sieht nur die **Vateranteile des Sohnobjektes**.

# Konvertierung von Datentypen

**Explizite**
Typumwandlung

- Numerische Datentypen
- Referenztypen

**Implizite** (automatische)
Typumwandlung

- Numerische Datentypen
- Referenztypen
- Verknüpfungen der Klasse String
- Automatisches Boxing bzw. Unboxing

# Implizite Typumwandlung

- Eine implizite Typkonvertierung findet statt, wenn ein Ausdruck in den Typ einer Variablen durch Zuweisung umgewandelt werden kann, so ist der Typ des Ausdrucks **zuweisungskompatibel** mit dem Typ der Variablen ist.

- Implizite Typkonvertierungen gibt es:
  - zwischen einfachen, numerischen (arithmetischen) Typen,
  - zwischen Referenztypen,
  - bei Verknüpfungen von Objekten der Klasse String mit Operanden anderer Datentypen
  - durch das automatische Boxing bzw. Unboxing zwischen einfachen numerischen Typen und Referenztypen numerischer Wrapper-Klassen.

Freie Universität Berlin

# Beispiele Implizite Typumwandlung

## Einfachen, numerischen Typen

```
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
```

# Übersicht Typumwandlung bei einfachen Datentypen

| | byte | short | int | long | float | double | boolean | char |
|---|---|---|---|---|---|---|---|---|
| byte | | e | e | e | e | e | x | e |
| short | i | | e | e | e | e | x | e |
| int | i | i | | e | e | e | x | i |
| long | i | i | i | | e | e | x | i |
| float | i | i | i | i | | e | x | i |
| double | i | i | i | i | i | | x | i |
| boolean | x | x | x | x | x | x | | x |
| char | e | e | e | e | e | e | x | |

Leserichtung: Der Typ in der Spalte wird in den Typ in der Zeile konvertiert.

Ausprägungen:

- (i) ein implizites (typsicheres) Casten ist möglich;
- (e) ein expliziter (nicht-typ-sicherer) Cast ist erlaubt
- (x) eine Typumwandlung ist grundsätzlich nicht gestattet.

# Beispiele Implizite Typumwandlung

## Referenztypen

- Ein impliziter **Up-Cast** bezeichnet einen Cast in einen Typ, der in der Vererbungshierarchie weiter oben liegt.

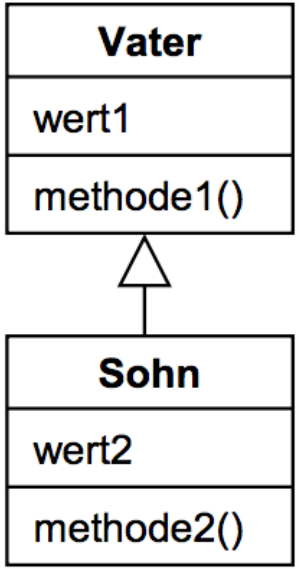
- Bei einem gültigen **Up-Cast ist es nie erforderlich, den cast-Operator anzugeben**, da der implizite Cast in die Superklasse eindeutig ist.

- Zeigen eine Referenz vom Typ Eltern und eine Referenz vom Typ Kind auf ein Objekt vom Typ Kind in einer Eltern-Kind-Hierarchie, so ist nach der Zuweisung der Referenz vom Typ Kind an die Referenz vom Typ Eltern nur der Eltern-Anteil des Objekts vom Typ Kind erreichbar.

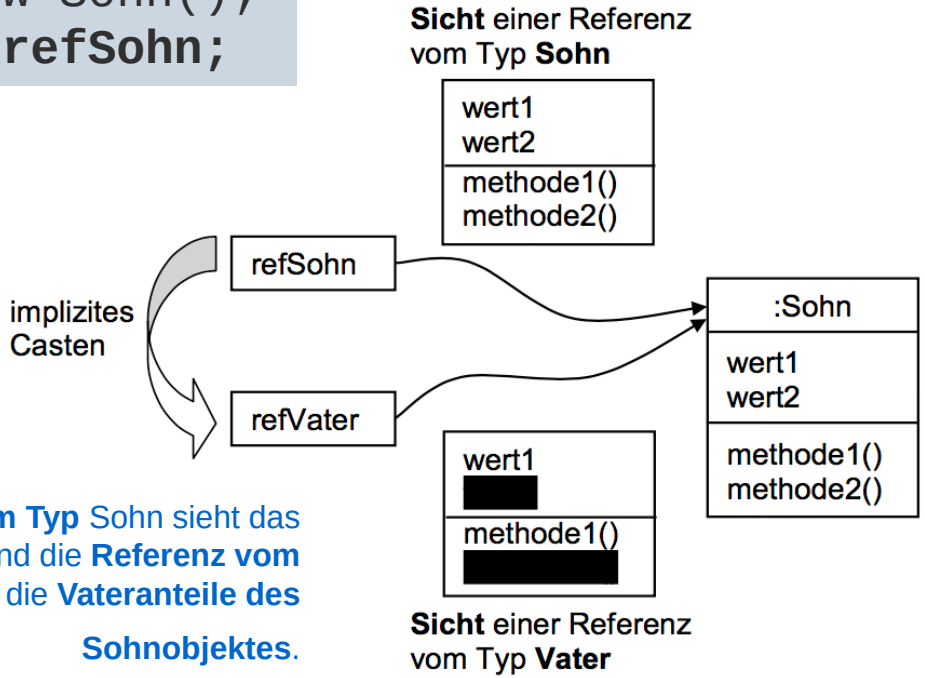
```
Kind refKind = new Kind();
Eltern refEltern = refKind;
```

# Beispiele Implizite Typumwandlung

## Referenztypen




```
Sohn refSohn = new Sohn();
Vater refVater = refSohn;
```




Die **Referenz vom Typ** Sohn sieht das **gesamte Objekt** und die **Referenz vom Typ** Vater sieht nur die **Vateranteile des Sohnobjektes**.

Freie Universität Berlin

# Beispiele Implizite Typumwandlung

## Verknüpfungen von Objekten der Klasse String

- Strings sind in Java kein Basistyp, sondern eine Bibliotheksklasse

`java.lang.String`

```
String s = "hallo";
```

äquivalent zu

```
String s = new String ("hallo");
```

- Methoden:

```
s.toUpperCase()          ····▶ "HALLO"
s.charAt( 4 )            ····▶ 'O'
s.length()               ····▶ 5
s.equals( s )            ····▶ true
s.replace('A', 'Ä')      ····▶ "HÄLLO"
s.substring(2, 4)        ····▶ "LL"
```

# Beispiele Implizite Typumwandlung

## Verknüpfungen von Objekten der Klasse String

- Strings in Java sind immer **konstant**, d.h. unveränderlich. Folgende Zuweisungen erzeugen jeweils neue String-Objekte und sind äquivalent:

```
String s  = "hallo";
```

```
s.length()  ⟶ 5
```

Wert

```
s  =  s + "Welt!";
```

||

```
s  =  new String(s + "Welt!");
```

# Beispiele Implizite Typumwandlung

## Automatische Boxing bzw. Unboxing mit Wrapper-Klassen

- Referenztypen können nicht auf primitive Datentypen gecastet werden und umgekehrt.

- Wenn man primitive Datentypen an einer Stelle verwenden will, wo nur Objekttypen erlaubt sind, kann man den Wert eines Basistyps in ein passendes “Wrapper”-Objekt einpacken.

- Für jeden primitiven Datentyp gibt es eine entsprechende Wrapper-Klasse, z.B. für `double` gibt es die Klasse `java.lang.Double`

# Beispiele Implizite Typumwandlung

## Automatische Boxing bzw. Unboxing mit Wrapper-Klassen

java.lang.*

- Boolean
- Byte
- Character
- Double
- Float
- Integer
- Long
- Short

```
...
public static void main(String[] args) {
    Integer num = 3;
    Integer num2 = 3;
    System.out.println( Integer.toBinaryString(num) );
    System.out.println( num==num2);
    System.out.println( num.intValue()==num2.intValue()
);
    Boolean bool = true;
    Double zahl = 3.0;
    System.out.println(num);
    System.out.println(bool);
    System.out.println(zahl);
}
```

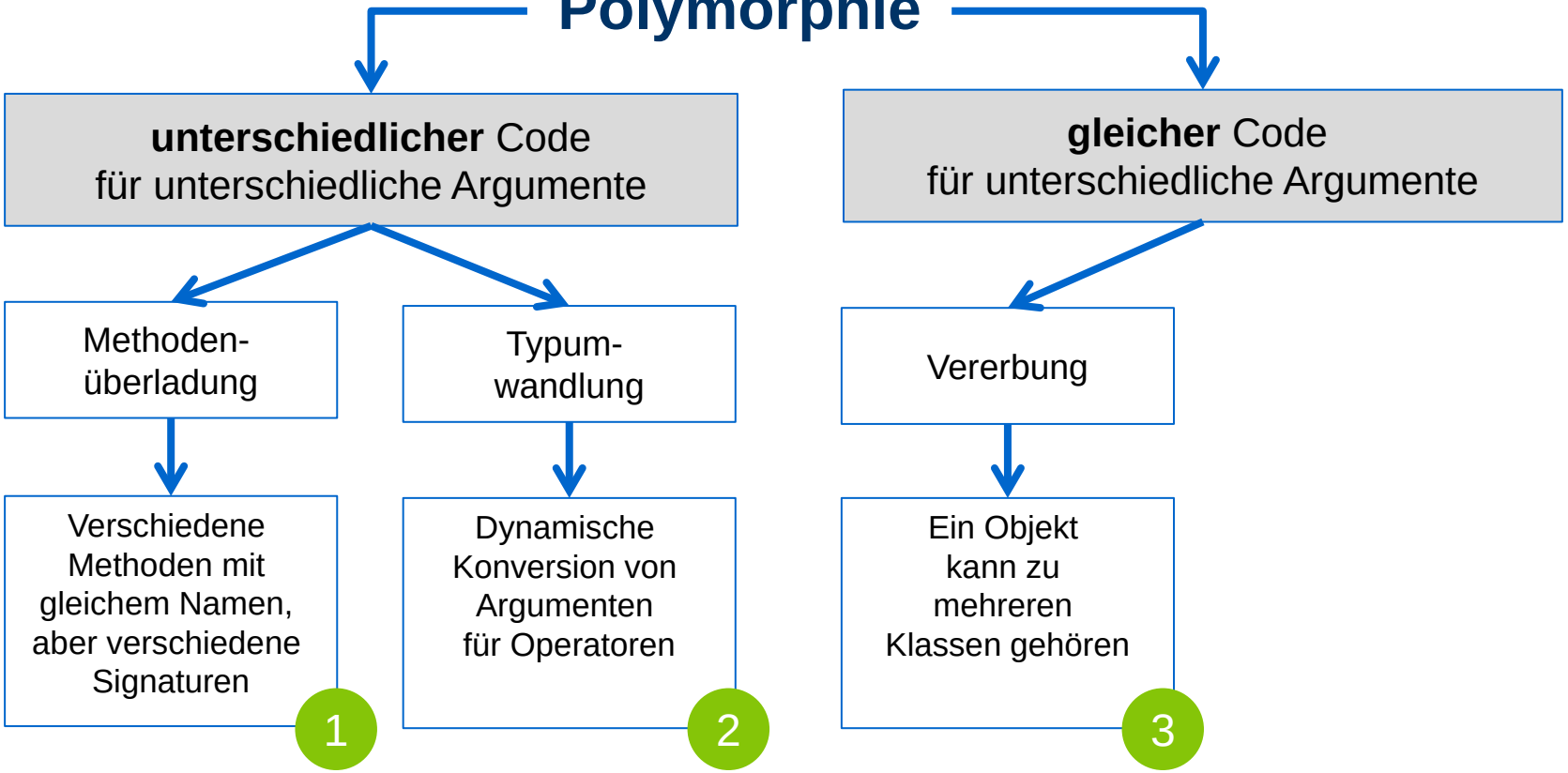
Autoboxing

Automatisches Unboxing und Vergleich

Ausgabe

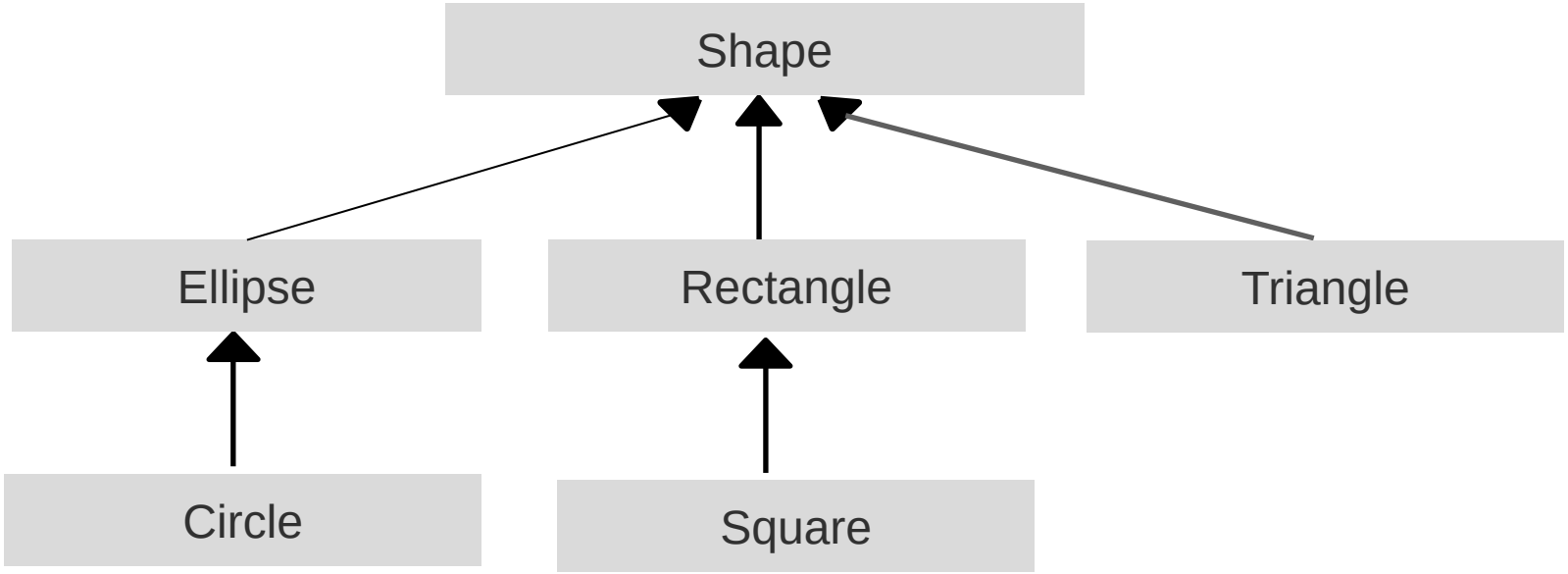
```
11
true
true
3
true
3.0
```

# Polymorphie



Freie Universität Berlin

# Vererbungspolymorphie (Subtyping)



# 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();
}
```

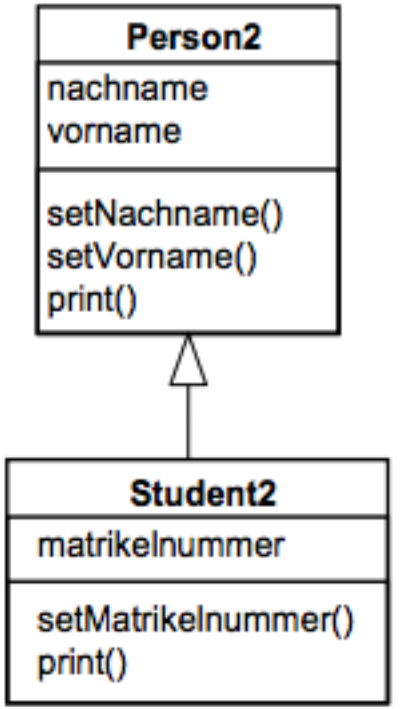
```
class Circle extends Shape {
...
   draw() {
       paintCircle( );
    }
…
}
```

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

# Überschreiben

- Eine abgeleitete Klasse enthält eine Methode mit der gleichen Signatur und mit gleichem Rückgabetyp wie eine Methode aus einer Basis(eltern)klasse, d.h.:

- Die Methode der Basisklasse überschreibt die Methode der abgeleiteten Klasse



# Gründe für das Überschreiben einer Methode

- **Verfeinerung**
  Eine abgeleitete Klasse verfeinert eine bestehende Methode der Elternklasse, z.B. durch zusätzlichen Datenfelder. In der Elternklasse sind diese zusätzlichen Datenfelder nicht bekannt, daher muss die Methode überschrieben werden.

- **Optimierung**
  Es kann nützlich sein, in einer abgeleiteten Klasse interne Datenstrukturen oder die Implementierung eines Algorithmus zu optimieren. Das Außenverhalten der Klasse darf sich dabei jedoch nicht ändern.

# Beispiel „Animal“

Unterklassen

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

```
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");
    }
}
```

# Beispiel „Animal“

```
public class Zoo {

    Animal[] list;

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


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

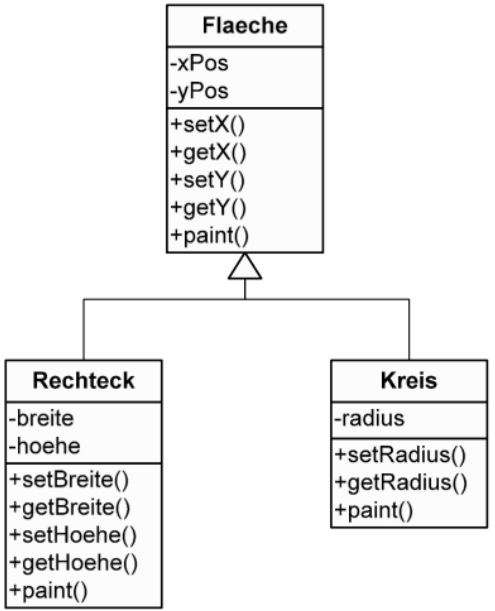
```
. . .
 public static void main( String[] args ) {
              Animal[] anim_list = {
                                    new Cat(),
                                    new Dog(),
                                    new Cow(),
                                    new Cat(),
                     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

    Zoo zoo = new Zoo( anim_list );

    zoo.sound();
```

Ausgabe:

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

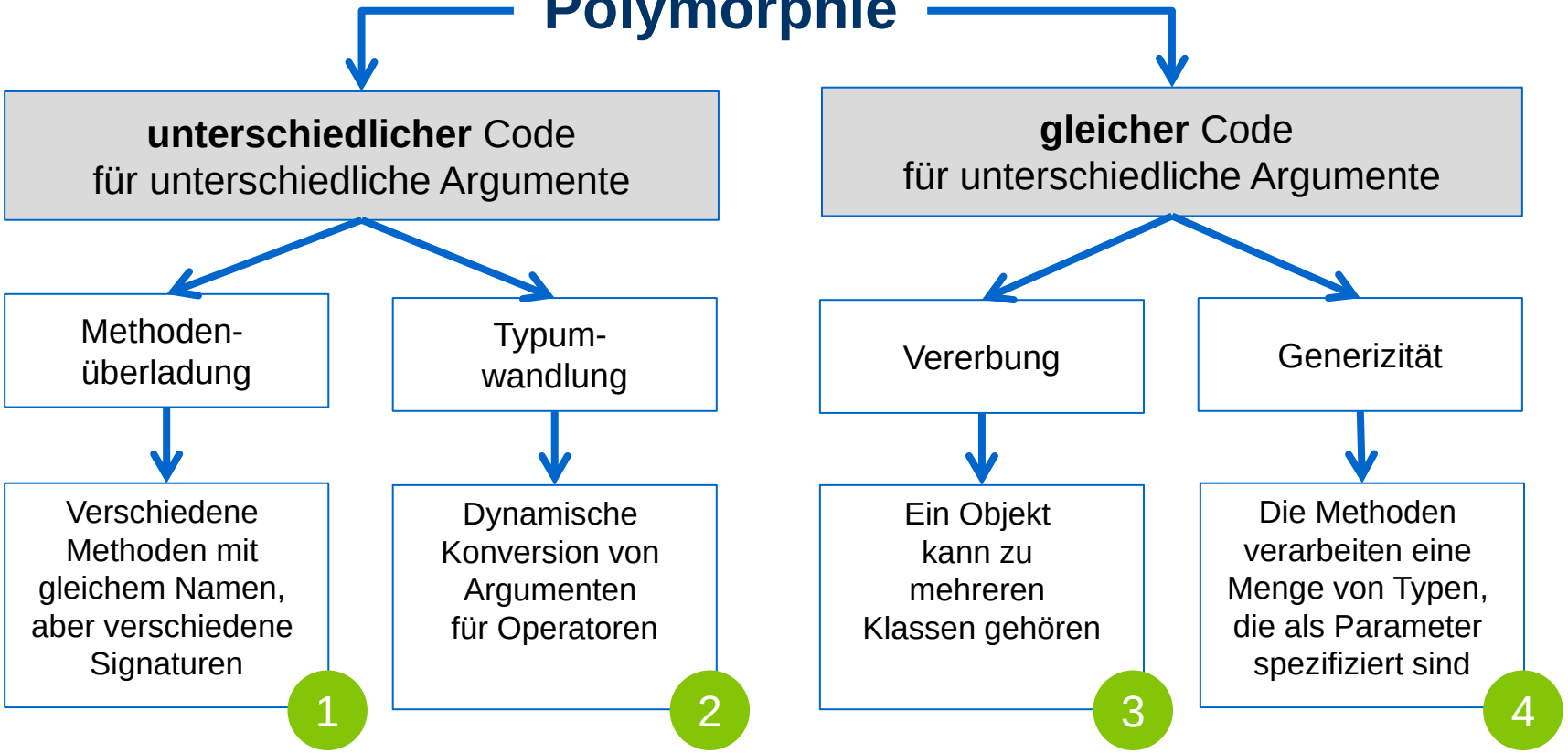
# Wiederholung: Überschreiben von Methoden

```
...
Flaeche figure_1 = new Rechteck(0,0,10,10);
Flaeche figure_2 = new Kreis(0.0,0.0,1.0);
...
figure_1.paint();
figure_2.paint();
...
```




- Der aktuelle Typ des aufgerufenen Objekts bestimmt, welche Methode tatsächlich benutzt wird.

# Polymorphie



# Generizität

- Generisch bedeutet in diesem Zusammenhang, dass Klassen, Schnittstellen und Methoden Parameter verwenden, welche einen Typ darstellen.

- Mit der Generizität von Klassen, Schnittstellen und Methoden werden die folgenden Ziele verfolgt:
  - Höhere Typsicherheit: Erkennen von Typ-Umwandlungsfehlern zur Kompilierzeit statt zur Laufzeit.
  - Wiederverwendbarkeit von Quellcode.
  - Vermeiden des expliziten Casts, der beim Auslesen aus einer Collection aus Elementen vom Typ Object notwendig ist.

# Beispiel Punkt-Klasse (herkömmlich)

```
public class Punkt {
   private Integer x;
   private Integer y;
   public Punkt (Integer x, Integer y) {
      this.x = x;
      this.y = y; }
   // weitere Elemente der Klasse Punkt
}
```

Freie Universität Berlin

# Beispiel Punkt-Klasse (generisch)

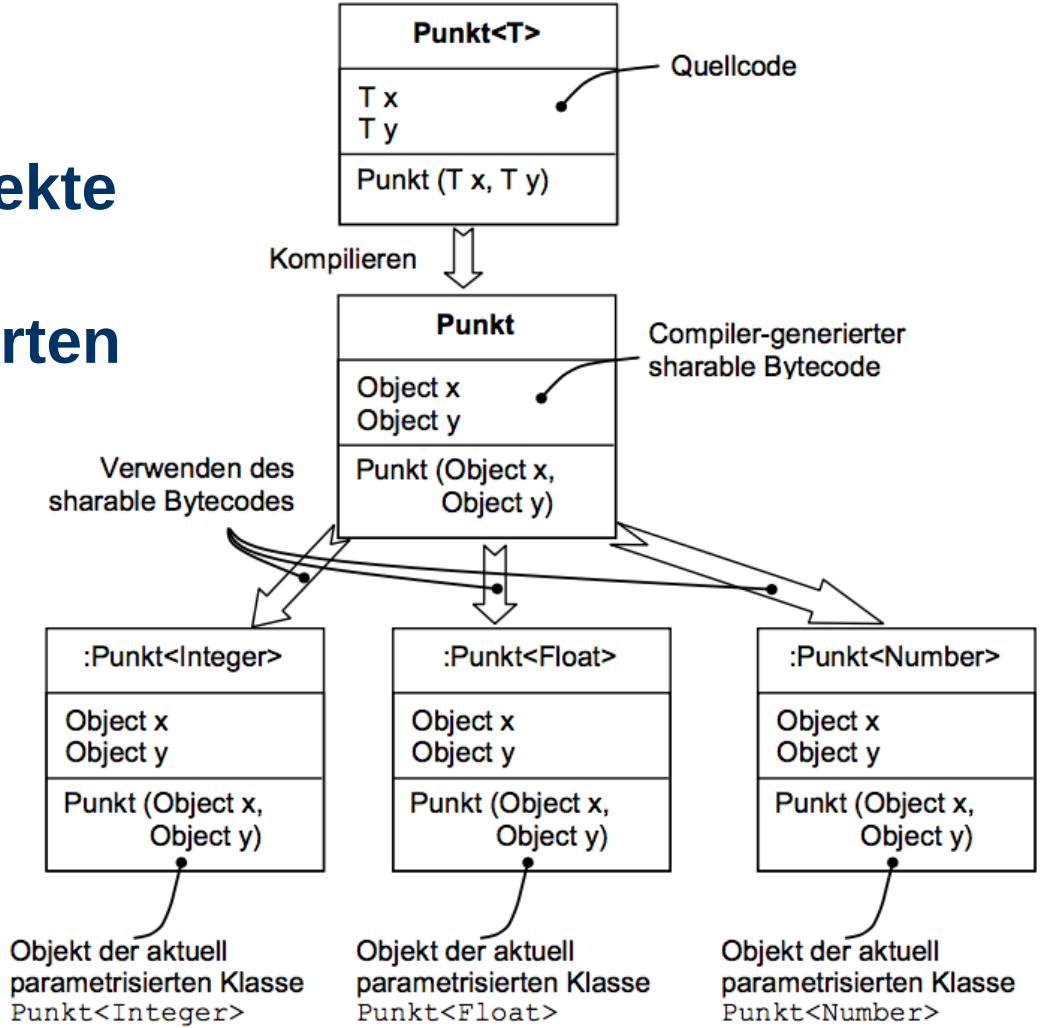
```
public class Punkt <T> {
   private T x;
   private T y;
   public Punkt (T x, T y) {
      this.x = x;
      this.y = y; }
   // weitere Elemente der Klasse Punkt<T>}
```

- Klassen können unabhängig von einem speziellen Typ generisch definiert werden. Der formale Typ-Parameter wird bei der **Verwendung** der Klasse dann durch den gewünschten **konkreten Datentyp** ersetzt.

# Beispiel Erzeugung von Objekten

```
// Aktueller Typ-Parameter Integer tritt an die Stelle von T
Punkt<Integer> integerPunkt = new Punkt<Integer> (1, 2);

// oder
Punkt<Integer> integerPunkt = new Punkt<>(1, 2);

// bzw. aktueller Typ-Parameter Double tritt an die Stelle von T
Punkt<Double> doublePunkt = new Punkt<Double> (1.0, 2.0);

// oder
Punkt<Double> doublePunkt = new Punkt<> (1.0, 2.0);
```

# Beispiel Objekte der aktuell parametrisierten Klassen



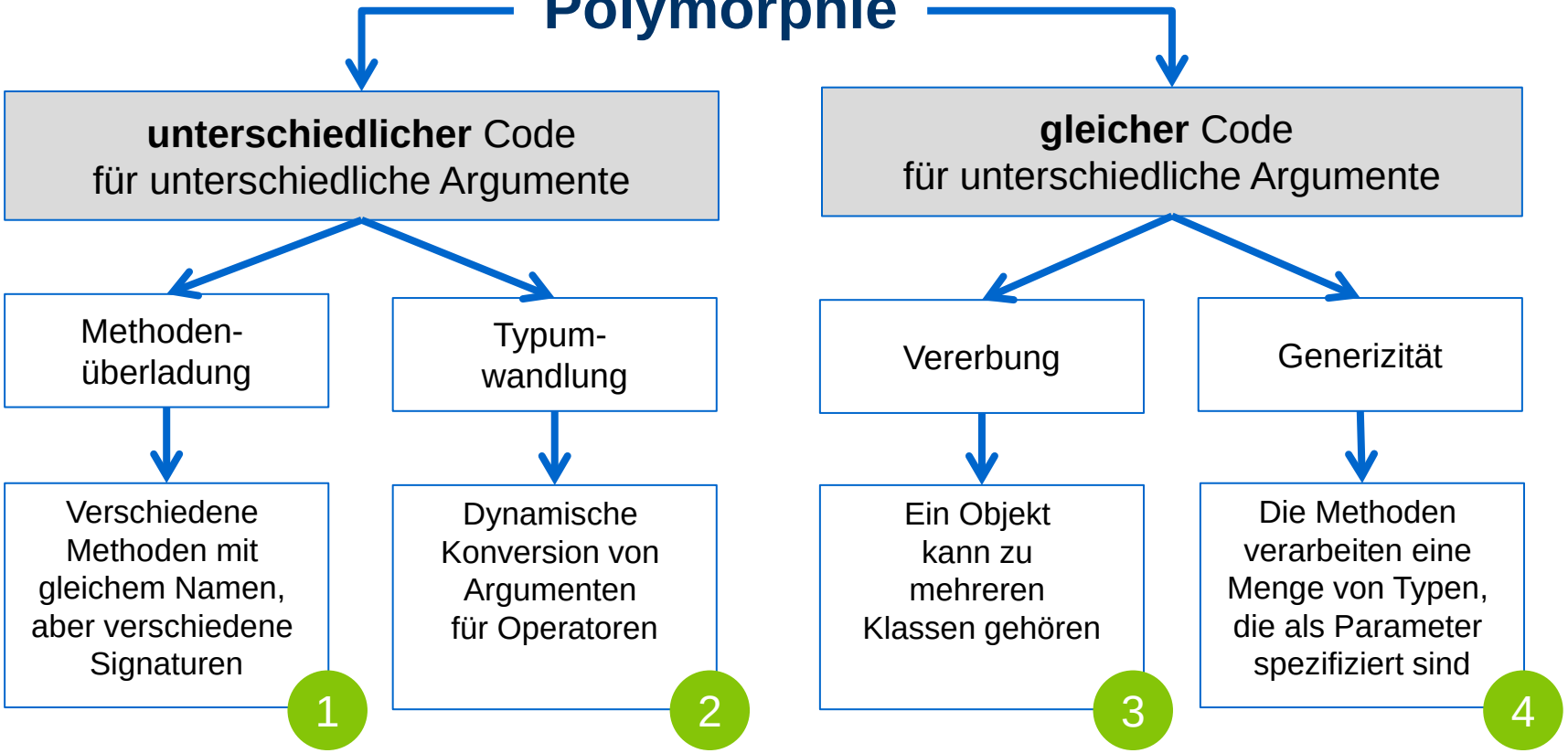

Klasse :Number
ist eine abstrakte Klasse

Freie Universität Berlin

Polymorphie

# ZUSAMMENFASSUNG

# Polymorphie



# Sie sollten wissen...

- was der Begriff der Polymorphie bedeutet.
- was die Idee hinter dem Überladen von Methoden ist und welche Auswirkungen das im Fall der Vererbung hat.
- welche Ansätze für implizite und explizite Typumwandlungen bestehen und welche unterschiedlichen Herangehensweisen für numerische und Referenztypen bestehen.
- was automatische Boxing bzw. Unboxing mit Wrapper-Klassen ist.
- wie Methoden überschrieben werden.
- was generische Klassen sind und welche Vorteile bei deren Nutzung bestehen.