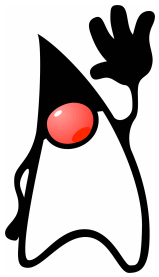
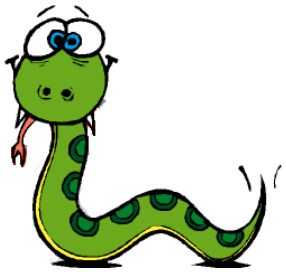
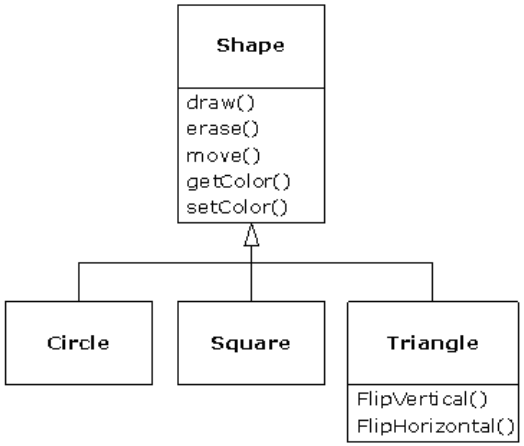
Freie Universität Berlin

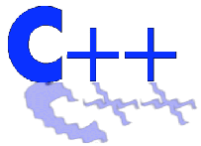
Algorithmen und Programmierung II

Vererbung (Teil II), Abstrakte Klassen, Schnittstellen









Prof. Dr. Margarita Esponda

SS 2012

Freie Universität Berlin

Haustier

```
public class Haustier {
        public static enum Zustand {
                                        TUT_NICHTS,
                                        SPIELT,
                                        SCHLAEFT,
                                        ISST,
                                        SPRICHT
                                    };
        public String name;
        public String lieblingsessen;
        public Person besitzer;
        private Zustand zustand;

        public Haustier(String name){
                this.name = name;
                this.zustand = Zustand.TUT_NICHTS;
        }
        . . .
```

Haustier

```
public class Haustier {
        . . .
        public void sprich(){
                zustand = Zustand.SPRICHT;
        }
        public void iss(){
                zustand = Zustand.ISST;
        }
        public void schlaf(){
                zustand = Zustand.SCHLAEFT;
        }
        public void ausruhen(){
                zustand = Zustand.TUT_NICHTS;
        }
}
```

Katze

```
public class Katze extends Haustier {

        public Katze( String name ){
                super( name );
                this.lieblingsessen = "Mäuse";
        }

        public void sprich(){
                super.sprich();
                System.out.println( "Miau! Miau!" );
        }
        ...
}
```

Hund

```
public class Hund extends Haustier {

        public Hund( String name ){
                super( name );
                this.lieblingsessen = "Fleisch";
        }

        public void sprich(){
                super.sprich();
                System.out.println( "Guau! Guau!" );
        }
       ...
}
```

Beispiel:

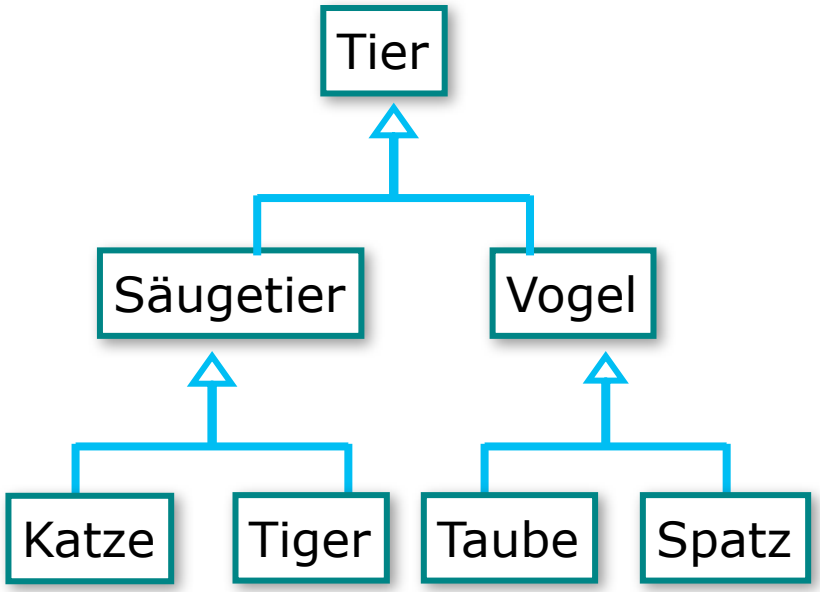
```
public class Rectangle {
    public int x, y;
    private int width, height;
    . . .
}//end of class Rectangle
```

```
import java.awt.Color;
import java.awt.Graphics;

public class DrawableRectangle extends Rectangle {
    private Color color = Color.BLACK;

    public Color getColor() {
        return color;
    }

    public void setColor(Color color) {
        this.color = color;
    }

    public void draw( Graphics g ){
        g.setColor(this.color);
        g.drawRect(x,y,getWidth(),getHeight());
    }
} // end of class ColorRectangle
```

Einfache und mehrfache Vererbung

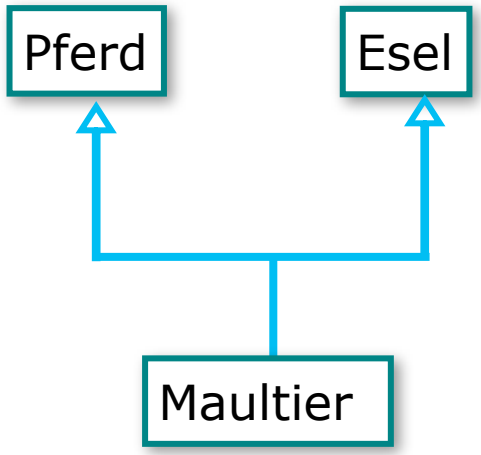
Beispiele:

Einfache Vererbung



Java und C#

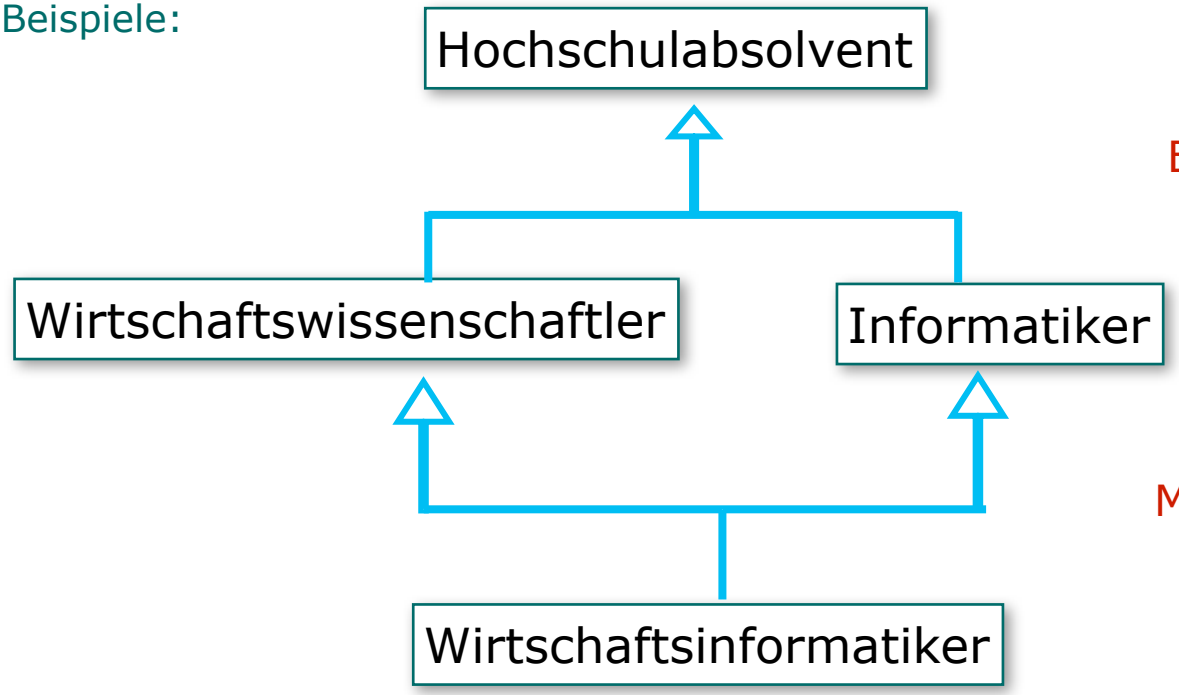
Mehrfache Vererbung



C++ und Python

Einfache und mehrfache Vererbung

Beispiele:



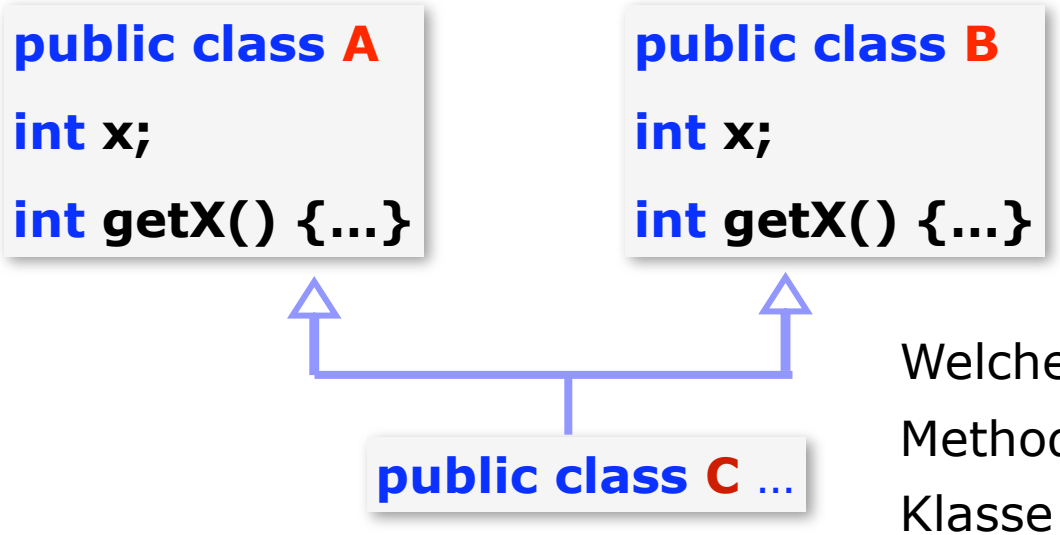
Einfache Vererbung

Mehrfache Vererbung

Einfache und mehrfache Vererbung

Probleme:

Bei mehrfacher Vererbung besteht die Gefahr der Namenskollisionen, weil Methodennamen oder Attribute mit gleichem Namen aus verschiedenen Oberklassen vererbt werden können.



Welche der beiden getX-Methoden sollen in der Klasse C verwendet werden?

Beispiel:

```
import java.awt.*;

public class Fenster extends JFrame {
        // Konstruktor
        public Fenster() {
                setTitle( "Fensterchen" );
                setBackground( new Color(0,190,190));
                setLocation( 500,300 );
                setSize( 200, 200 );
                setVisible( true );
        }
} // end of class MyFenster
```

Die Klasse Fenster vererbt alle Eigenschaften und Methoden, die die Oberklasse JFrame bereits besitzt.

```
public class Person {

    public static int anzahl = 0;

    public String vorname;
    public String nachname;
    private Date geburtsdatum;

    public Person (String vorname, String nachname, Date geburtsdatum) {
        this.vorname = vorname;
        this.nachname = nachname;
        this.geburtsdatum = geburtsdatum;
        anzahl++;
    }
    public Person () {
        this ( "", "", new Date() );
    }
...
```

```
...
    public Date getGeburtsdatum() {
            return geburtsdatum;
    }

    public void setGeburtsdatum( Date geburtsdatum ) {
            Date heute = new Date();
            if ( geburtsdatum.after(heute) )
                System.err.println( "Falsches Geburtsdatum" );
            else
                this.geburtsdatum = geburtsdatum;
    }

    public int alter(){
          Date heute = new Date();
          long time = heute.getTime() - geburtsdatum.getTime();
          time = time/1000;
          return  (int) (time/(365*24*3600));
    }
} // end of class Person
```

Vererbung (Beispiel)

```
public class Student extends Person {
      // Klassenvariablen
       public static long semesterDauer = 3600*24*183;
    // Instanzvariablen
       public String fachbereich;
       public int matrikelnr;
       private Date anfangsdatum;
   // Konstruktor
     public Student( Date anfangsdatum, String fachbereich,
                              int matrikelnr )
    {
            super();
            this.anfangsdatum = anfangsdatum;
            this.fachbereich = fachbereich;
            this.matrikelnr = matrikelnr;
    }
  . . .
```

Ein impliziter Aufruf des Konstruktors der Oberklasse wird hier vom Compiler angesetzt.

Vererbung

```
public class TestPersonStudent {

    public static void main(String[] args) {
        Date geburt1 = Datum.toDate( "23.02.1985" );
        Date geburt2 = Datum.toDate( "11.05.1983" );
        Person p1 = new Person( "Peter", "Meyer", geburt1 );
        System.out.println( p1.getGeburtsdatum().toString() );
        Student s1 = new Student( "Sandra","Smith", geburt2 );
        s1.setAnfangsdatum( Datum.toDate( "01.10.2004" ) );
        System.out.println( s1.getGeburtsdatum().toString() );
        System.out.println( s1.semester() );
        System.out.println( p1.alter() );
        System.out.println( "Anzahl der Personen = " + Person.anzahl );
    }
}
```

Vererbung (Beispiel)

```
public class Student extends Person {
        . . .
      // Instanzmethoden
        public Date getAnfangsdatum() {
                return anfangsdatum;
        }
        public void setAnfangsdatum(Date anfangsdatum) {
                this.anfangsdatum = anfangsdatum;
        }
        public int semester(){
                Date heute = new Date();
                long time = heute.getTime() - anfangsdatum.getTime();
                time = time/1000;
                return (int)(1 + time/semesterDauer);
        }
}
```

Vererbung

Beispiel in Python:

Klassenvariable

Instanz-Variablen

Unterklasse von Person

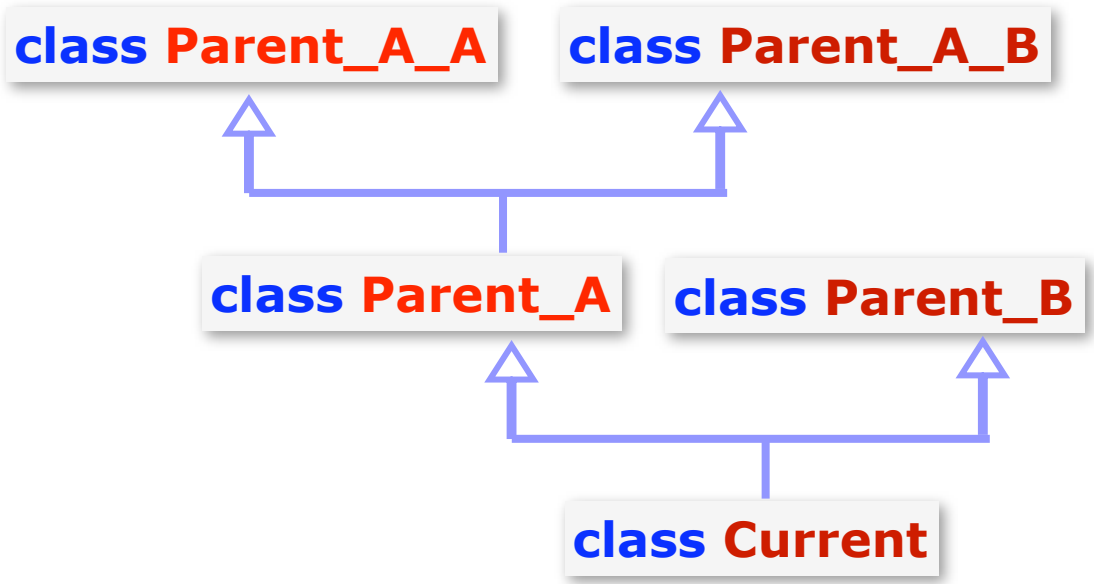
```
class Person:

    counter = 0

    def __init__(self, name, vorname):
        self.name = name
        self.vorname = vorname
        Person.counter += 1

    def sayYourName(self):
        print ( 'Hi. My name is ' , self.name)


class Student( Person ):

    def __init__(self, name, vorname, semester):
        super().__init__(name, vorname)
        self.semester = semester


s1 = Student('Peter', 'Meyer', 3)
s2 = Student('Nils', 'Meyer', 2)
print( s1.vorname )
print( Person.counter )
```

Ausgabe:

```
Meyer
2
```

Mehrfache Vererbung in Python

Eine Klasse in Python kann aus mehreren Klassen erben.



Python verwendet eine "**Tiefensuche**", um zu entscheiden, welche Methode der Oberklassen aufgerufen wird.

Wenn die Methode eines Objekts der Klasse Current aufgerufen wird, wird die Methode zuerst in **Current** gesucht. Wenn dort nichts gefunden wird, werden die Oberklassen in folgender Reihenfolge durchsucht:

Current, Parent_A, Parent_A_A, Parent_A_B und **Parent_B.**

Mehrfach-Vererbung

Beispiel in Python:

```
class D:
    def m(self):
        print('I am in D')

class A(D):
    pass

class B:
    def m(self):
        print('I am in B' )

class C(A, B):
    pass

x = C()
x.m()
```

Ausgabe:

```
I am in D
```

Überladen von Methoden

Die Methoden werden vom Übersetzer durch Anzahl und Typ der Parameter unterschieden.

```
public void draw ( String s )

public void draw ( int i )

public void draw ( double d )

public void draw ( double d, int x, int y )
```

Die Parametertypen bestimmen die *Signatur* einer Methode.

Überladen von Konstruktoren

Die Konstruktoren werden vom Übersetzer durch Anzahl und 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 des Konstruktors

Die Parametertypen bestimmen die *Signatur* des Konstruktors.

Überschreiben von Methoden

```
public class Person {
    ...
     public  String  name() {
       return  name;
     }
}
```

```
public class  Mann extends  Person {
    ...
    public  String  name() {
         return "Herr "+ super.name();
    }
}
```

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

Überschreiben von Methoden

- Eine als **final** markierte Methode kann nicht in Unterklassen überschrieben werden

  public final String nachname()

- Von einer **final** Klasse können keine Unterklassen gebildet werden

  public final class String { ... }

- Überschreibende Methoden können Zugriffsrechte weiter einschränken,

  public ------> protected

  aber nicht erweitern protected ------> public

Konstruktoren in Unterklassen

- Konstruktoren werden nicht vererbt, d.h. Unterklassen müssen jeweils eigene Konstruktoren angeben und die Oberklassenkonstruktoren benutzen.
- Wenn man keinen Konstruktor der Oberklasse verwendet, wird *nur* der implizite default-Konstruktor super() aufgerufen.
- Wenn man einen Konstruktor der Oberklasse verwenden will, muss der Aufruf am Anfang der jeweiligen Konstruktoren stehen.

super

```
public class Circle {
   double x, y, r;
   public Circle(){
   }
   ...
}
```

Der Konstruktor der Oberklasse wird implizit aufgerufen.

equiv.

```
public class Circle {
   double x, y, r;
   public Circle(){
      super();
   }
   ...
}
```

Der Konstruktor der Oberklasse wird explizit aufgerufen.

Wenn man keinen Konstruktor der Oberklasse verwendet, wird *nur* der implizite default-Konstruktor super() aufgerufen.

super

```
public class Person {
    String vorname;
    String nachname;

    public Person ( String vorname, String nachname ) {
        this.vorname = vorname;
        this.nachname = nachname;
    }
}
```

```
public class Student extends Person {

    String fachbereich;

    public Student ( String vorname, String nachname, String fachbereich ) {
        super ( vorname, nachnahme );
        this.fachbereich = fachbereich;
    }
}
```

Ein Konstruktor der Oberklasse muss hier explizit aufgerufen werden.

super

Das Schlüsselwort **super** dient nicht nur dazu, um Konstruktoren der Oberklasse aufzurufen sondern wird verwendet, um den Zugriff auf verdeckte Instanzvariablen und Methoden der Oberklasse zu ermöglichen.

Konstruktoren werden nicht vererbt, d.h. Unterklassen müssen jeweils eigene Konstruktoren angeben und die Oberklassenkonstruktoren benutzen.

super

Beispiel:

```
public class Klasse_O {
      int y;
      int x = 10;

      int q() { return x*x; }
}
```

```
. . .
Klasse_U u = new Klasse_U();
System.out.println( u.q1() );
System.out.println( u.q()  );
System.out.println( u.q2() );
. . .
```

```
public class Klasse_U extends Klasse_O
{
    int x = 2;

    int q()   { return x*x; }
    int q1() { return super.x*super.x; }
    int q2() { return  super.q(); }
}
```

Verdeckt die **x** der Oberklasse

Verdeckt die **q**-Methode der Oberklasse

Ausgabe:

```
100
4
100
```

Vererbung

```
class Object {
  . . .
}
```

```
class Car {
  . . .
}
```

```
class Ford extends Car {
  . . .
}
```

```
class FordFiesta extends Ford {
  . . .
}
```

Legale Zuweisungen sind:

```
. . .
Object obj = new Car();
Car car = new Ford();
Car car = new FordFiesta();
Object obj = new FordFiesta();
. . .
```

Illegale Zuweisungen sind z.B.:

```
. . .
Car obj = new Object();
FordFiesta car = new Ford();
. . .
```

Abstrakte Methoden

- Abstrakte Methoden enthalten nur die Deklaration des Methodenkopfes, aber keine Implementierung des Methodenrumpfes.
- Abstrakte Methoden haben anstelle der geschweiften Klammern mit den auszuführenden Anweisungen ein Semikolon. Zusätzlich wird die Definition mit dem Attribut **abstract** versehen.
- Abstrakte Methoden definieren nur eine Schnittstelle, die durch Überschreiben innerhalb einer abgeleiteten Klasse implementiert werden kann.

Beispiel:

```
public abstract void paint();
```

Abstrakte Klassen

- Eine Klasse, die mindestens eine abstrakte Methode enthält, muss als abstrakte Klasse deklariert werden.
- Die bereits implementierten Methoden in einer abstrakten Klasse können von anderen Klassen geerbt werden, welche die abstrakten Methoden zusätzlich implementieren können.
- Eine abstrakte Klasse wird in einer Unterklasse konkretisiert, wenn dort alle ihre abstrakten Methoden implementiert sind.
- Abstrakte Klassen werden mit dem Schlüsselwort **abstract** markiert.

Abstrakte Klassen

- Abstrakte Klassen sind künstliche Oberklassen, die geschaffen werden, um Gemeinsamkeiten mehrerer Klassen zusammenzufassen.
- Abstrakte Klassen dienen nur zur besseren Strukturierung der Software.
- Objekte können nicht aus einer abstrakten Klasse erzeugt werden.
- Die fehlende Implementierung wird in den Unterklassen "nachgeliefert", sonst sind diese auch abstrakt.

Abstrakte Klassen

Beispiel

```
public abstract class Figur {
    protected double x, y;
    public double getX() { return x; }
    public double getY() { return y; }
    public void setX( double x ) { this.x = x; }
    public void sety( double y ) { this.y = y; }
    public abstract double area();
}
```

Abstrakte Methode

Implementierung

```
public class Circle extends Figur {
    private double radio;
    static final private double PI = 3.1415926535897;
    public double area() {
        return PI*radio*radio;
    }
}
```

Eine abstrakte Klasse wird in einer Unterklasse konkretisiert, wenn alle ihre abstrakten Methoden implementiert werden.

Kapselung und Abstrakte Datentypen (ADT)

Klassen definieren neue Datentypen und die Operationen, die auf diesen Datentypen erlaubt sind.

Ein abstrakter Datentyp ist eine Typdefinition oder Spezifikation, die unabhängig von einer konkreten Implementierung ist.

In Java versucht man mit Hilfe von Interfaces konkrete Implementierungen von Datentypspezifikation zu trennen.

Interfaces (Motivation)

Beispiel:

Kunde

Schnittstelle
Interface

- Lautstärke
- Senderwechsel
- Farbjustierung

Implementierung

LED-Fernseher,
66 cm (26")
HD Ready

Die Kunden brauchen nur zu wissen, wie das Interface benutzt werden kann.

Die Implementierung muss nur wissen, welches Interface implementiert werden soll.

Die Implementierung kann vertauscht werden ohne die Kunden zu tauschen.

Interfaces

In Java sind Interfaces sowohl ein Abstraktionsmittel (zum Verbergen von Details einer Implementierung) als auch ein Strukturierungsmittel zur Organisation von Klassenhierarchien!

Eine Schnittstelle (interface) in Java legt eine **minimale Funktionalität** (Methoden) fest, die in einer implementierenden Klasse vorhanden sein soll.

Interfaces (Schnittstellen)

- Interfaces sind vollständig abstrakte Klassen.
- keine Methode ist implementiert.
- keine Instanzvariable ist deklariert.
- nur statische Variablen können deklariert werden.

```
public interface Collection {
        public void add(Object o);
        public void remove(Object o);
        public boolean contains(Object o);
}
```

Alle Methoden sind implizit **abstract** und **public**.

Interfaces

Verschiedene Implementierungen desselben Typs sind möglich, und die Implementierungen können geändert werden, ohne daß der Benutzer es merkt!

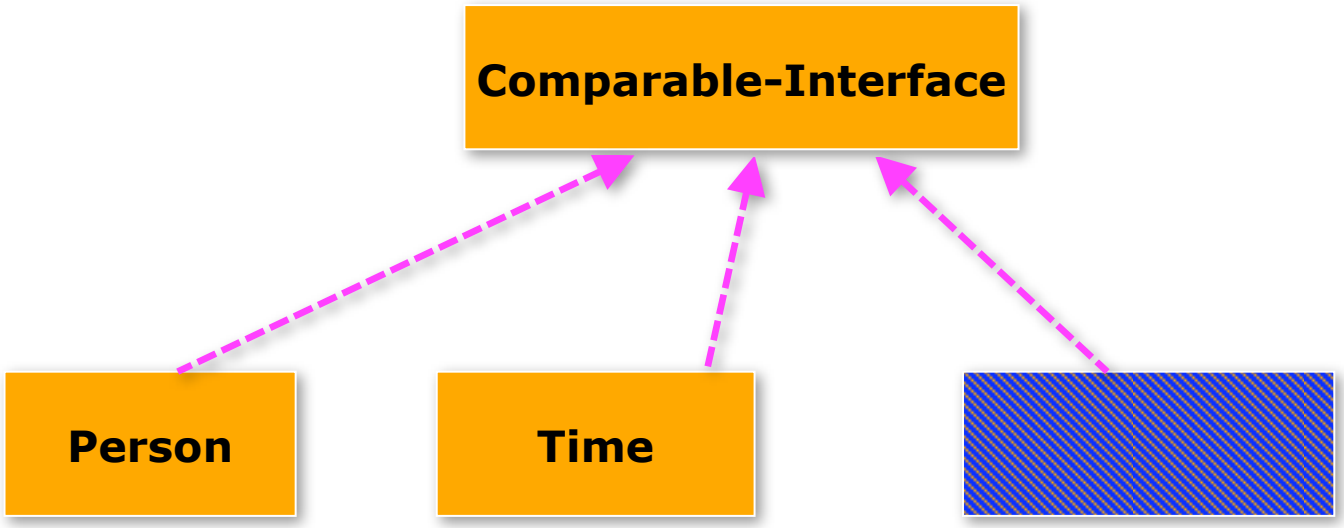
```
public class Set implements Collection {
    public void add( Object o ) { ... }
    public void remove( Object o ) { ... }
    public boolean contains( Object o ) {... }
}
```

Alle Methoden des Interface müssen implementiert werden.

Implementierung von Interfaces

Beispiel:

Comparable-Interface



Interfaces (Schnittstellen)

Freie Universität Berlin

```
interface Comparable {
      int compareTo( Object other );
}
```

```
public class Time implements Comparable {
        private int seconds;
        private int minutes;
        private int hours;
    public int compareTo( Object obj ) {
        Time time = (Time) obj;
        if ( this.toSeconds()<time.toSeconds() )
                return -1;
        else if ( this.toSeconds()>time.toSeconds() )
                return 1;
        else
                return 0;
    }
    . . .
```

Interfaces (Schnittstellen)

```
public class Sortierer {

    Comparable[] list;

    Sortierer( Comparable[] list ){ this.list = list; }

    public void bubbleSort(){
            boolean swap = true;
            Comparable temp;
            while ( swap ) {
                    swap = false;
                    for ( int i=0; i<list.length-1; i++ ) {
                            if ( list[i].compareTo( list[i+1] ) == 1 ) {
                                    temp = list[i];
                                    list[i] = list[i+1];
                                    list[i+1] = temp;
                                    swap = true;
                            }
                    }
            }
    }// end of class Sortierer
```

```
...
public static void main( String[] args ){

        Time[] zeit = new Time[6];

        zeit[0] = new Time(); zeit[0].setTime(3,10,20);
        zeit[1] = new Time(); zeit[1].setTime(7,10,20);
        zeit[2] = new Time(); zeit[2].setTime(5,10,20);
        zeit[3] = new Time(); zeit[3].setTime(4,10,20);
        zeit[4] = new Time(); zeit[4].setTime(2,10,20);
        zeit[5] = new Time(); zeit[5].setTime(0,10,20);

        Sortierer s = new Sortierer(zeit);

        s.bubbleSort();

        for (int i=0; i<zeit.length; i++){
                System.out.println("zeit["+i+"]="+zeit[i]);
        }
 }
```

```
zeit[0]=00:10:20
zeit[1]=02:10:20
zeit[2]=03:10:20
zeit[3]=04:10:20
zeit[4]=05:10:20
zeit[5]=07:10:20
```