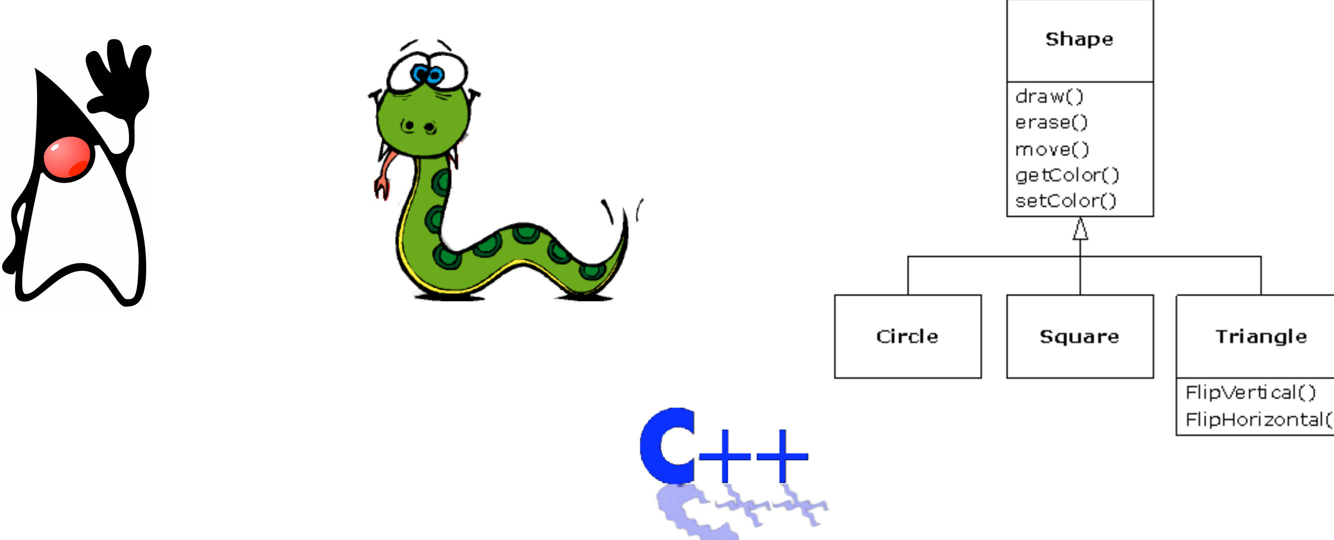
Algorithmen und Programmierung II

Abstrakte Klassen, Schnittstellen und **ADT**



Prof. Dr. Margarita Esponda

SS 2012

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 als Strukturierungsmittel

Interfaces in Java können als Unterinterfaces von anderen Interfaces definiert werden.

```
interface EventListener

interface ActionListener extends EventListener
```

Collections (Sammlungen)

Die Java Bibliotheken stellen eine Reihe von Schnittstellen und Klassen zur Verfügung, um Sammlungen von Objekten zu simulieren.

Es gibt drei Hauptgruppen von "Collections":

List: - Die Elemente sind sortiert
- Duplikate sind erlaubt
- Die Elemente sind indiziert

Set: - Duplikate sind nicht erlaubt

Map: - Zuordnung der Elemente zu eindeutigen Schlüsseln
- Elemente können mehrfach vorkommen

Collection-Klassen

Collection

List

Queue

Set

Vector

ArrayList

LinkedList

PriorityQueue

HashSet

SortedSet

TreeSet

Map

SortedMap

Hashtable

TreeMap

HashMap

Interfaces

Konkrete Klassen

Vektor-Klasse

Dynamische Datenstruktur

Serializable
Cloneable
Iterable
Collection
List
RandomAccess

Vector

"Serializable" bedeutet, dass ein Objekt in eine lineare, sequentielle Darstellung verwandelt werden kann.

"Iterable" bedeutet, dass Methoden zur Verfügung gestellt werden, mit denen eine Datenstruktur durchlaufen werden kann.

Die Objekte können an beliebigen Positionen eingefügt oder gelöscht werden.

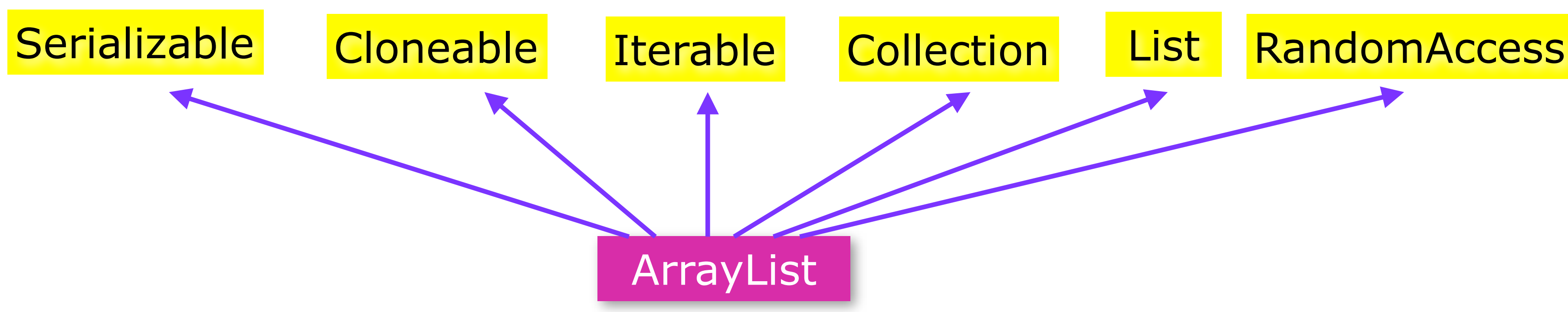
Interfaces

Konkrete Klassen

ConcurrentModificationException

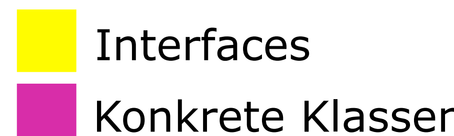
ArrayList-Klasse

Dynamische Datenstruktur



Die aktuelle Implementierung ist nicht synchronisiert

Operationen der Liste, die strukturelle Veränderungen der Liste verursachen, sind nicht gegen nebenläufige Zugriffe geschützt.



ConcurrentModificationException

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

Wrapper-Klassen

java.lang.*

Boolean
Byte
Character
Double
Float
Integer
Long
Short

hier werden zwei Objekt-Referenzen verglichen

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

Automatische *Unboxing*

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

Autoboxing/Unboxing

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);
}
…
```

automatisches Unboxing und Vergleich

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

Autoboxing/Unboxing

```
...
Integer n = new Integer( 5 );
Integer m = new Integer( 5 );

System.out.println( n >= m ); // Unboxing
System.out.println( n <= m ); // Unboxing
System.out.println( n == m ); // kein Unboxing
...
```

Ausgabe:

True
True
False

Der Vergleich mit == ist ein Referenzvergleich

Autoboxing/Unboxing

```
...
Integer n = 127;
Integer m = 127;
System.out.println( n == m );    // Unboxing
Integer n = 128;
Integer m = 128;
System.out.println( n == m );    // kein Unboxing
...
```

Ausgabe:

True

False

Automatisches *Unboxing* wird nur bei Objekten, die mit automatischem *Boxing* gebildet worden sind, nur innerhalb des Wertebereichs -128 bis 127 (1 Byte) liegen.

Beispiel mit Vector- und Wrapper-Klassen

```
...
Vector v = new Vector();
double d;
...
while( (d = readDouble() ) >= 0 )
        v.addElement( new Double(d) );
...
while( (d = readDouble() ) >= 0 )
        v.addElement( d );
...
```

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.

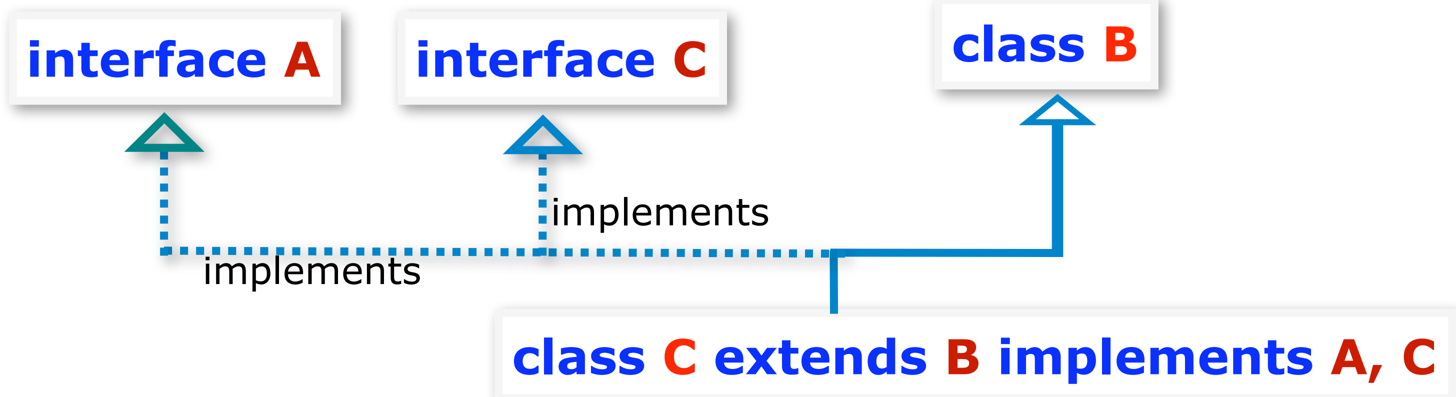
Interfaces als Strukturierungsmittel

Einfache und mehrfache Vererbung

Um die Probleme der Namenkollisionen zu vermeiden, wurde in Java mehrfache Vererbung abgeschafft.

Im Java wird eine beschränkte mehrfache Vererbung mit Hilfe von Interfaces simuliert.

Eine Klasse in Java kann von einer Oberklasse vererben und gleichzeitig mehrere Interfaces implementieren.



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 {
- Subtyp- Polymorphie
- Parametrische Polymorphie

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!

Vererbungspolymorphie

Unterklassen

Beispiel:

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

```java
public class Cat extends Animal {

   public void speak(){
      System.out.println("Miau Miau");
   }
}
```

```java
public class Cow extends Animal {

   public void speak(){
      System.out.println("Muh Muh");
   }
}
```

```java
public class Dog extends Animal {

   public void speak(){
      System.out.println("Wau Wau");
   }
}
```

Vererbungspolymorphie

Beispiel:

```
public class Zoo {

   Animal[] list;

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

   public void sound(){
       for(int i=0; i<list.length; i++)
       {
                list[i].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

- **unterschiedlicher Code** für unterschiedliche Argumente
  - **Methoden-überladung**
    - Verschiedene Methoden mit gleichem Namen aber verschiedenen Signaturen — **1**
  - **Typum-wandlung**
    - Dynamische Konversion von Argumenten für Operatoren — **2**
- **gleicher Code** für unterschiedliche Argumente
  - **Generizität**
    - Die Methoden verarbeiten eine Menge von Typen, die als Parameter spezifiziert sind — **3**
  - **Vererbung**
    - Ein Objekt kann zu mehreren Klassen gehören — **4**

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

2 Typumwandlung

3.0 + 5 → **8.0**

double int **double**

3.0 + " Kilogramm" → **"3.0 Kilogramm"**

double String **String**

Polymorphie

3 Generische Datentypen

Ab Java 1.5 werden Collection-Klassen als generisch betrachtet.

Die früher nur heterogenen Listen können 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-Objekte

Beispiel:

Es können nur Rechteck-Objekte in das ArrayList-Objekt 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 );
            }
    }
}
```

3 Generizität

Klassenschablonen

Ziel ist das Einsparen von Programmieraufwand.

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

In Java war dies schon von 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.

3 Generizität

Klassenschablonen

Beispiel:

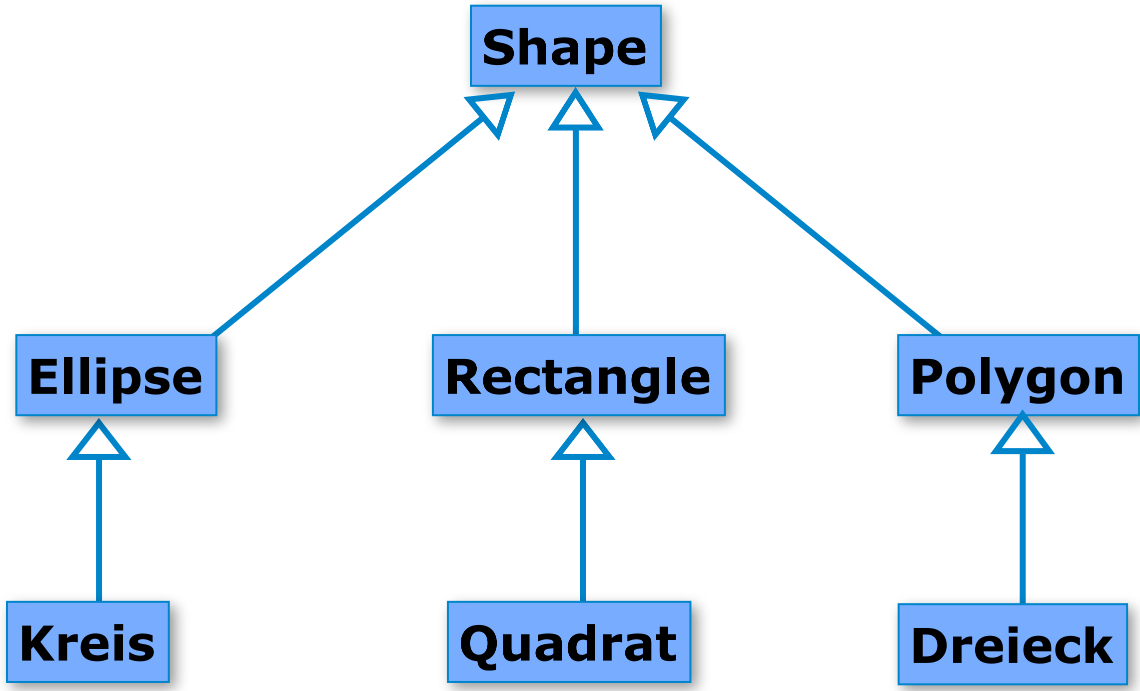
```
public class GenericTest <T> {

    private T wert;

    public GenericTest () {
    }

    public void setValue ( T wert ) {
          this.wert = wert;
    }

    public T getValue() {
        return wert;
    }
}
```

3 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() );
    }
}
```

4 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 paint(...);
}
```

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

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

Vererbungspolymorphie

Überschreiben von Methoden

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

```
...
Shape figure_1 = new Rectangle(0,0,10,10);
Shape figure_2 = new Circle(0.0,0.0,1.0);
...
...
figure_1.paint();
figure_2.paint();
...
```