

***Einführung in die Informatik I
Objekt-orientierte Programmierung***

Prof. Bernd Brügge, Ph.D
Technische Universität München

Wintersemester 2000/2001

29. und 30. Januar 2001

Verschiedenes

❖ Deutschsprachige Literatur:

- Goos: Vorlesungen über Informatik, Band 2, Springer Verlag 2. Auflage, 1999, ISBN 3-540-64340-0.
- Küchlin, Weber: Einführung in die Informatik, Springer Verlag, 2000, ISBN 3-540-67384-9.

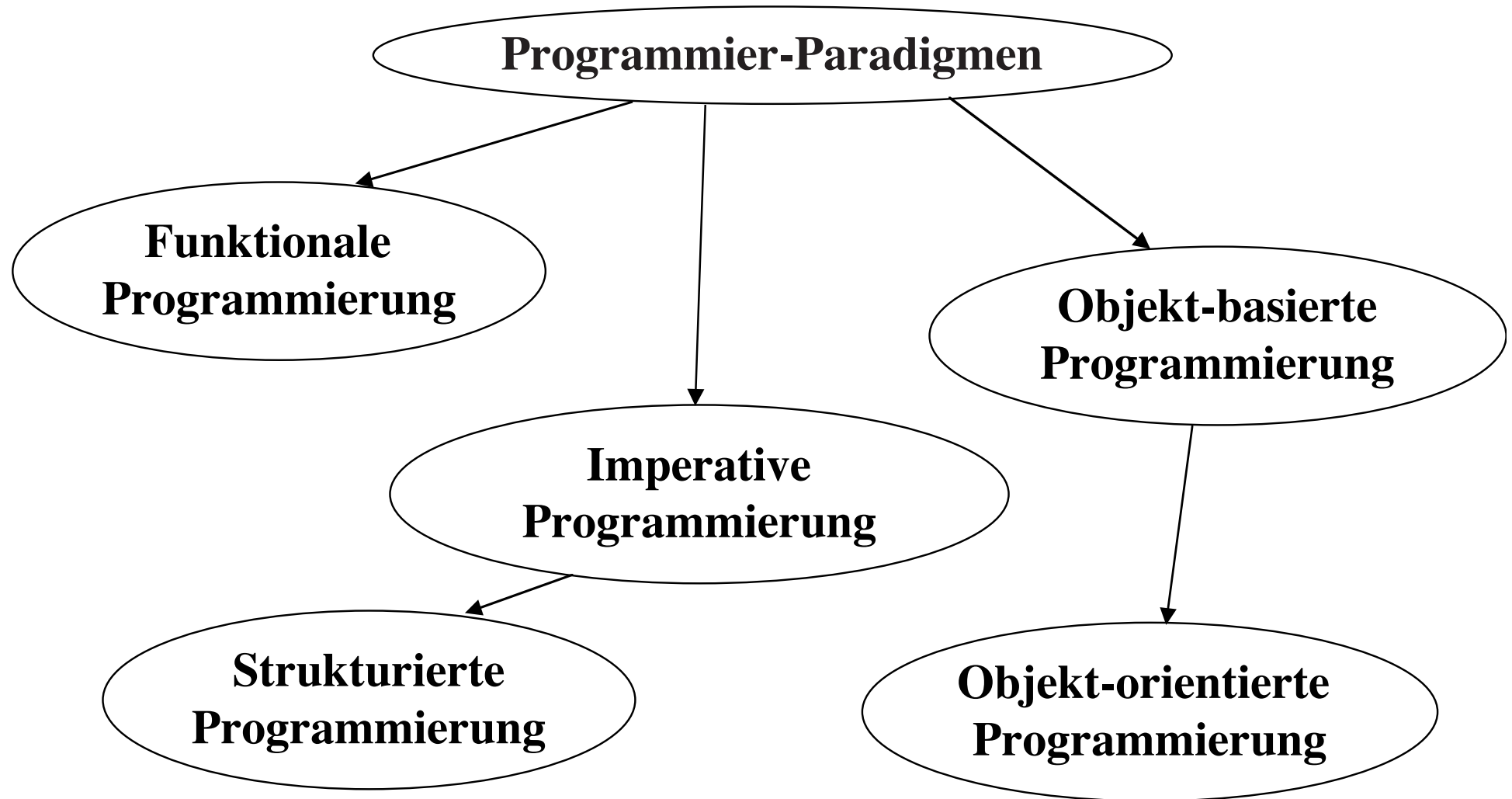
❖ Englisch-sprachige Literatur:

- Morelli: Java, Java, Java: Object-Oriented Problem Solving, Prentice Hall, 2000, ISBN 0-13-011332-8.
- Brügge, Dutoit: Object-Oriented Software Engineering, Prentice Hall, Oct 1999, ISBN-0-13-489725-0.
- David Flanagan: Java in a Nutshell, O'Reilly, 1997, ISBN 1-56592-304-9.

Überblick: Wo stehen wir?

- ❖ Funktionale Programmierung: Ein Programm ist eine Menge von Funktionen. Die Ausführung des Programms besteht in der Berechnung eines Ausdrucks mit Hilfe der Funktionen und liefert Ergebniswerte
- ❖ Imperative Programmierung: Ein Programm besteht aus einer Menge von Daten und Routinen (Prozeduren, Funktionen). Die Ausführung des Programms verändert die Werte dieser Daten. Die Routinen sind nicht an die Daten gebunden.
 - Strukturierte Programmierung: Alle imperativen Programme können mit Zuweisungs-, **if**-, **for**-, **while**- und **do-while**-Anweisungen realisiert werden.
- ❖ *Zwei neue Begriffe:*
- ❖ Objekt-basierte Programmierung: Ein Programm ist eine Menge von kooperierenden Klassen, die Daten und Routinen (Methoden genannt) enthalten. *Methoden können nicht außerhalb von Klassen existieren.*
- ❖ Objekt-orientierte Programmierung: Ein Programm ist eine Menge von kooperierenden und *wiederverwendbaren* Klassen. Klassen können ihre Eigenschaften *vererben*.

Überblick über die Programmier-Paradigmen



Überblick über diesen Vorlesungsblock

❖ Themen:

- Weitere Konzepte der objekt-basierten Programmierung
 - Klassenvariablen und Klassenmethoden
 - Klasse String
- Konzepte der objekt-orientierten Programmierung
 - Vererbung
 - Abstrakte Klassen und Schnittstellen (interfaces)
 - Generische Klassen

❖ Ziele:

- Sie verstehen die Gründe der objekt-orientierten Programmierung
- Sie können aktiv mit Vererbung umgehen, insbesondere mit Vererbung durch Spezifikation und Vererbung durch Implementation (extends).
- Sie kennen den Unterschied in Java zwischen Klasse, abstrakter Klasse und Schnittstelle.

Systeme, Klassen und Objekte

- ❖ Ein System ist eine Menge von Objekten, die miteinander interagieren.
 - Durch die Interaktion wird der Zustand des Gesamtsystems verändert, der aus den Zuständen der einzelnen Objekte besteht. Der Zustand eines Objektes wiederum wird durch die Werte seiner Attribute bestimmt.

Objekt-Basierte Programmierung

- ❖ Jedes Objekt gehört zu einer Menge von Objekten mit gleichen Merkmalen. Wir nennen diese Menge auch Klasse.
- ❖ **Definition Klasse:**
 - Die Menge aller Objekte mit gleichen Merkmalen, d.h. mit gleichen Attributen und Operationen.
- ❖ **Definition Instanz:**
 - Ein Objekt ist eine Instanz einer Klasse K , wenn es Element der Menge aller Objekte der Klasse K ist.
- ❖ Die subtile Unterscheidung zwischen Klassen und Typen ist für uns dabei zunächst unwesentlich:
 - Für uns bedeuten Klassen und Typen dasselbe.

Java unterstützt objekt-basierte Programmierung

- ❖ Zentrales Konstruktionsmittel bei der **Analyse** von Objekten ist der Klassenbegriff.
- ❖ Java unterstützt den Klassenbegriff auch während der **Implementierung**.
 - Eine Java Klassendeklaration spezifiziert die Mitglieder der Klasse, d.h. die Instanzvariablen (instance variables), die Konstruktoren (constructors) und die Methoden der Klasse.
 - Die Attribute der Klasse heißen in Java auch Felder (fields)
- ❖ Ein Objekt ist die Instanz einer Klasse.
 - Bei der Erzeugung eines Objektes (mit **new**), wird ein passendes Stück Speicher reserviert, auf dem neue Instanzen für die Felder eingerichtet werden.
 - Jedes Objekt hat also eigene Werte für die in der Klasse spezifizierten Attribute.

Beispiel einer Java-Klasse (class)

```
public class Person {  
    private String name = "no name";  
  
    public void Person(String s) {  
        name = s;  
    }  
    public void setName(String s) {  
        name = s;  
    }  
    public String getName() {  
        return name;  
    }  
} // Person
```

Bisher: Instanzvariablen vs Lokale Variablen

- ❖ In Vorlesung 09, Folie 39 hatten wir gesagt:
 - Innerhalb einer Klasse unterscheidet Java
 - **Instanzvariablen** (instance variables), die auf Klassenebene als Attribute deklariert werden und in Objekten instantiiert werden.
 - **Lokale Variablen** (local variables), die innerhalb von Methoden, oder allgemeiner innerhalb einer Verbundanweisung, deklariert werden können.
 - Eine Instanzvariable kann einen Modifizierer (modifier) haben (`private`, `public`), eine lokale Variable nicht.
- ❖ Jetzt führen wir noch einen weiteren Typ von Variablen ein:
 - Klassenvariablen (class variables).

Klassenvariablen und Klassenmethoden

- ❖ In Java gibt es auch Klassenvariablen und Klassenmethoden, die nicht an Objekte, sondern an Klassen gebunden sind.
- ❖ Diese Variablen und Methoden werden mit dem Modifikator **static** bezeichnet.

Warum Klassenvariablen?

- ❖ Instanzvariablen werden bei jeder Instantiierung einer Klasse instantiiert, d.h. *jedes Objekt* vom Typ der Klasse hat *seine eigenen Instanzen der in der Klasse definierten Attribute*.
- ❖ Manchmal wollen wir allerdings von einem Attribut nur eine einzige Kopie haben.
- ❖ Beispiel:
 - Wir entwickeln eine Klasse Student, und wollen in der Testphase wissen, wie oft die Klasse instantiiert wird.
 - Eine Möglichkeit ist die Anzahl der Objekte vom Typ Student zu zählen. Dazu brauchen wir natürlich eine Variable, die mit der Klasse assoziiert ist, und nicht mit einer bestimmten Instanz.
 - Genau das macht eine Klassenvariable.
- ❖ Klassenvariablen werden also typischerweise für globale Buchhaltungsaufgaben für Klassen eingesetzt, z.B. um die Anzahl aller erzeugten Instanzen einer Klasse zu zählen.

Beispiel: Klassenvariable

Klassenvariable

```
public class Person {  
    static int num_person = 0; // Zählt die Instanzen von Person
```

Instanzvariable

```
    private String name;  
  
    public void Person(String s) { //Konstruktor  
        num_person++;  
        name = s;  
    }  
    public void setName(String s) {  
        name = s;  
    }  
    public String getName() {  
        return name;  
    }  
} // Person
```

Klassenvariablen

- ❖ **Wichtig:** Ist ein Feld in einer Java Klasse statisch (**static**) deklariert, dann gibt es dieses Feld nur einmal, egal wieviele Objekte von dieser Klasse existieren.
 - Der Name statisch bezieht sich auf die Speicherallokation, die für Klassenvariablen statisch (zur Übersetzungszeit) geschehen kann, und nicht dynamisch (zur Laufzeit), wenn Objekte instantiiert werden.
 - Statisch deklarierte Variablen haben nichts "statisches", ihre Werte können während der Laufzeit variieren, genauso wie die Werte anderer Variablen (Instanzvariablen, lokale Variablen).

Klassenmethoden

- ❖ Ist eine Methode in einer Java Klasse statisch (**static**) deklariert, dann gibt es diese Methode genau einmal, egal wieviele Objekte von dieser Klasse existieren.
 - Die Methode wird dann nicht auf einem Objekt aufgerufen, sondern der Aufruf ist von der Art:
 - **Klassenname.Methodenname (...)**

- ❖ Beispiel: Die Klassendefinition

```
class Time {  
    public static void stick (Time t) {  
    }  
}
```

erlaubt den Aufruf `Time.stick(t)`

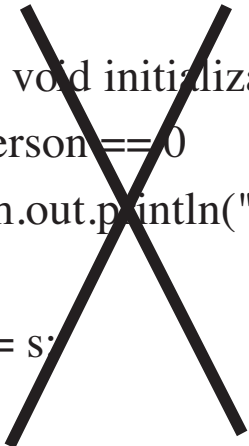
- ❖ Wir müssen also kein Objekt vom Typ `Time` instantiieren, um `stick` aufzurufen!

Implementation von Klassenmethoden

- ❖ Klassenmethoden dürfen bei der Implementation nur auf Klassenvariablen und andere Klassenmethoden direkt zugreifen; Zugriffe auf Instanzvariablen und -methoden sind nur über eine existierende Instanz möglich (Punkt-Notation)!

Geht nicht: Eine Klassenmethode darf keine Instanzvariablen benutzen!

```
public class Person {  
    static int num_person = 0; // Zählt die Instanzen von Person  
  
    private String name;  
  
    public void Person(String s) {  
        //Konstruktor  
        num_person++;  
        name = s;  
    }  
  
    public static void initialization (String s) {  
        if num_person == 0  
            System.out.println("Error");  
        else  
            name = s;  
    }  
} // Person
```



main ist nichts anderes als eine Klassenmethode!

```
public class PersonenSystem
{
    public static void main (String[] argv)
    {
        Person person1;
        Person person2;
        person1= new Person("Socrates");
        person2= new Person("Plato");
    } // main()
} // PersonenSystem
```

String[] argv: **Eine Reihung von Objekten vom Typ String (Argumente der Kommandozeile)**

public static void main:
main ist eine öffentliche (public) Klassenmethode (static), die kein Ergebnis zurückliefert (void).

•Wichtig:

- Es gibt nur ein **main**, egal wie oft PersonenSystem instantiiert wird.
- **main** wird vom Benutzer aufgerufen, indem er den Java-Interpreter mit der Klasse aufruft, deren **main**-Methode ausgeführt werden soll:
 - **% java PersonenSystem a1 a2 a3...**

Zeichenketten in Java (Strings)

- ❖ String ist eine vordefinierte Java-Klasse zur Modellierung von Zeichenketten:
 - 2 private Attribute (Instanzvariablen)
 - value: Eine Folge von Zeichen (vom Typ char)
 - count: Die Länge der Zeichenkette
 - Etliche Methoden:
 - Zur Manipulation von Zeichenketten.
- ❖ Zeichenketten haben gewisse Eigenschaften von primitiven Datentypen.
 - Beispiel: Zeichenketten können direkt als Werte angegeben werden (Literele).
 - Jede Folge von null oder mehr Zeichen in Gänsefüßchen ist ein Literal:
 - “Socrates” , “ Julia Berger “ , “”

Java-Klasse für Zeichenketten: String

- ❖ In Java bezeichnet **String** die Klasse aller unveränderlichen Zeichenketten mit gleichbleibender Länge.
 - Die Klasse **StringBuffer** bezeichnet die Menge aller variablen Zeichenketten → Info II.
- ❖ Zeichenketten werden durch Gänsefüßchen gekennzeichnet:
"foobar", **"Hello World"**, **"**, ...
- ❖ Zeichenketten-Literale werden vorwiegend benutzt, um Variablen vom Typ **String** einen Wert zuzuweisen. Als Zuweisungsoperator benutzen wir das Zeichen **=**
- ❖ Beispiel:
 - **String s;**
 - **s = "foobar";**
- ❖ *String-Zeichenketten sind nicht veränderlich!*
(aber: einer Variable vom Typ **String** können während des Programmablaufs verschiedene Zeichenketten zugewiesen werden)

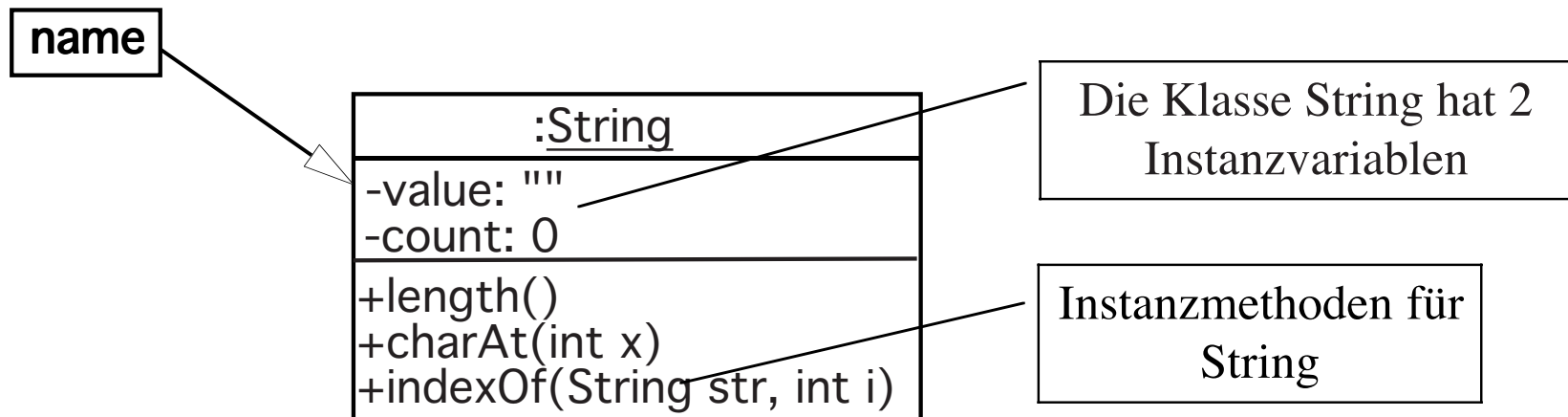
Konstruktion von Zeichenketten in Java

- ❖ Für die Klasse **String** offeriert Java insgesamt 11 Konstruktoren. Zwei davon sind:

```
public String(); // Kreiert das leere Wort
public String(String initial_value); // Kreiert die Kopie einer
// Zeichenkette initial_value
```

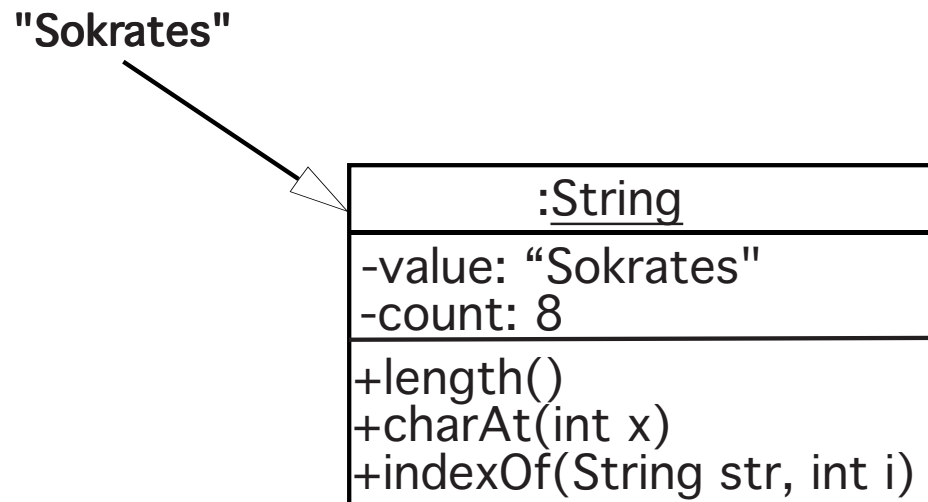
- ❖ Die folgende Zuweisung kreiert ein **String**-Objekt für das leere Wort und speichert die Referenz zu diesem Objekt unter *name* ab:

```
String name = new String();
```



Konstruktion von Zeichenketten in Java

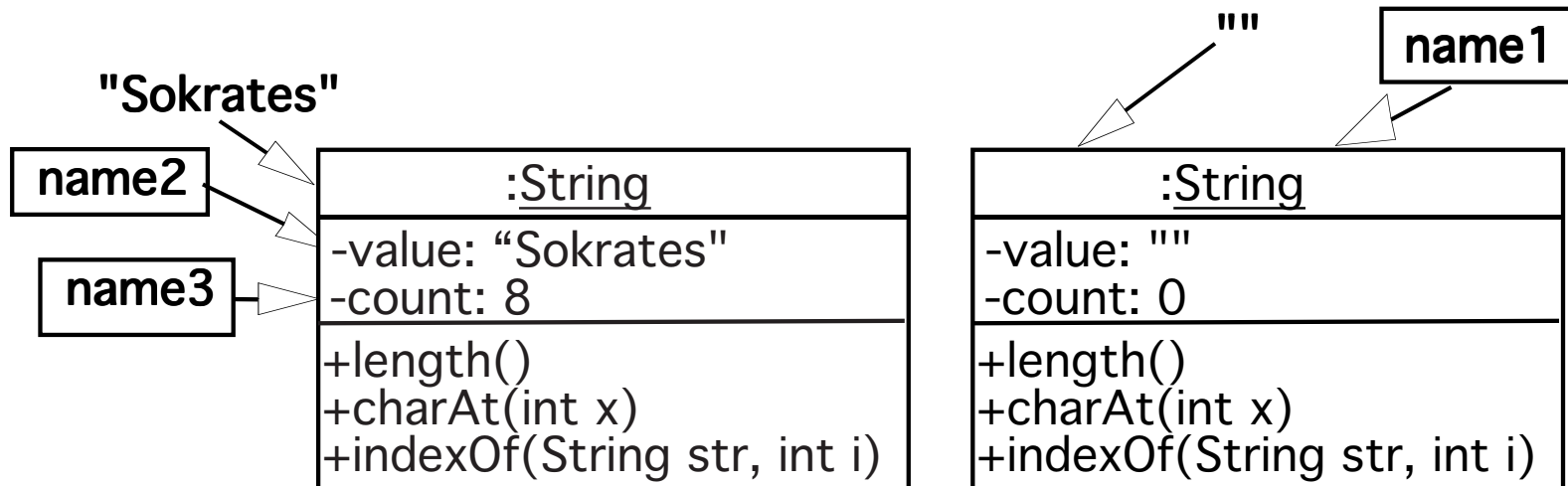
- ❖ Ein Literal, z.B. “Sokrates”, wird als Verweisvariable auf ein Objekt vom Typ **String** behandelt. Alle Instanzen von “Sokrates” in einem Java-Programm beziehen sich auf ein und dasselbe Objekt.



Konstruktion von Zeichenketten in Java

- ❖ Wenn ein Literal einer Variablen vom Typ String zugewiesen wird, zeigen die Werte dieser Variable auch auf das Objekt, das durch das Literal beschrieben wird.

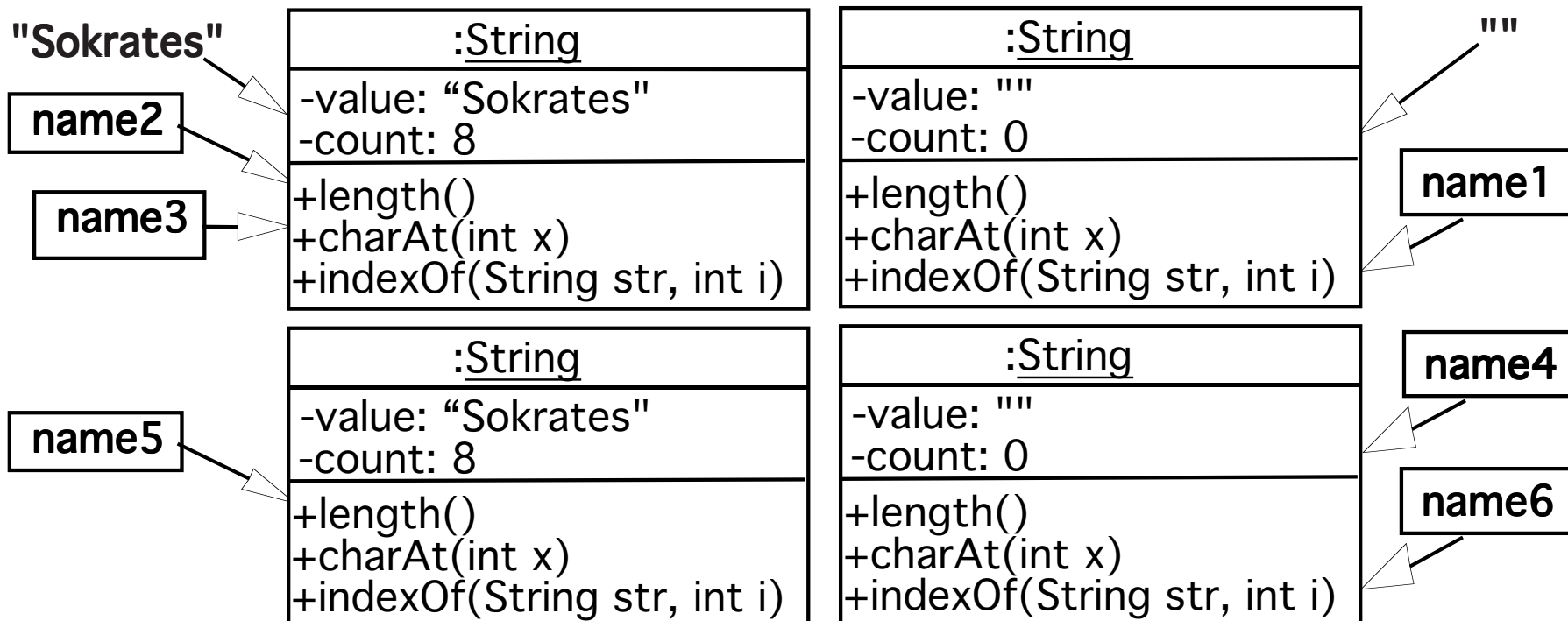
```
String name1 = "";           // Referenz auf das leere Wort  
String name2 = "Sokrates";  // Referenzen auf "Sokrates"  
String name3 = "Sokrates";
```



Konstruktion von Zeichenketten in Java

- ❖ Neue **String**-Objekte werden dann erzeugt, wenn der **String**-Konstruktor aufgerufen wird:

```
String name4 = new String();  
String name5 = new String("Sokrates");  
String name6 = name4;
```



Konkatenation von Zeichenketten

- ❖ Der **+**-Operator konkateniert Zeichenketten:

```
String lastName = "Onassis";  
String jackie = new String("Jacqueline " + "Kennedy " + lastName);
```

“Jacqueline Kennedy Onassis”

- ❖ Werte von anderen Typen werden automatisch in Zeichenketten konvertiert, wenn sie mit Zeichenketten konkateniert werden.
Beispiel: Konkatenation eines Zeichenketten-Literals mit einer Variablen vom Typ **int**:

```
int i = 5;  
System.out.println("The square root of 25 = " + i);
```

The square root of 25 = 5

Unterschied: Literal und String-Variable

- ❖ Ein Java-Compiler alloziert Speicherplatz für jede Zeichenkette, die im Java-Programm vorkommt. Nehmen wir das folgende Beispiel:

- **String s, t;**
- **s = "blabla";**
- **s = "foobar";**

- ❖ Was passiert?

- Der Speicherplatz für die Referenzvariable s wird alloziert.
 - s hat als Wert eine Referenz auf **null**.
- Der Speicherplatz für die Referenzvariable t wird alloziert.
 - t hat als Wert eine Referenz auf **null**.

- Der Speicherplatz für das String-Objekt "blabla" wird alloziert.
 - "blabla" ist die Referenz auf dieses Objekt.

- s bekommt als Wert die Adresse des "blabla"-Objektes

- Der Speicherplatz für das String-Objekt "foobar" wird alloziert.

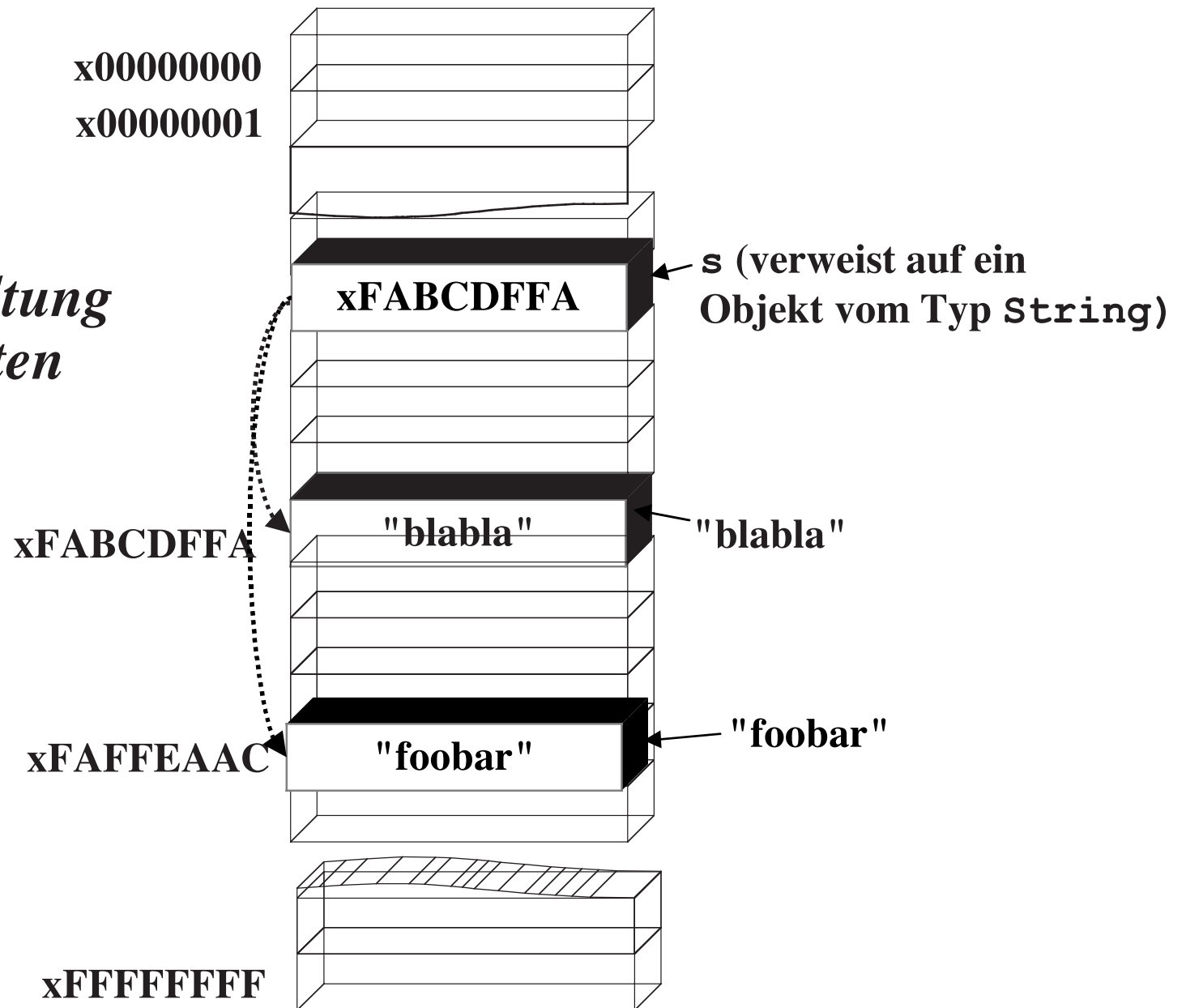
- "foobar" ist die Referenz auf dieses Objekt.

- s bekommt als Wert die Adresse des "foobar" Objektes

- ❖ Fazit:

- s zeigt auf das "foobar"-Objekt
- t ist immer noch null
- Das "blabla"-Objekt wird nicht mehr referenziert

Speicherverwaltung für Zeichenketten



Indizierung von Zeichenketten

- ❖ Die Länge einer Zeichenkette ist die Anzahl ihrer Zeichen.

```
String string1 = "";           // string1.length() ==> 0
String string2 = "Hello";     // string2.length() ==> 5
String string3 = "World";    // string3.length() ==> 5;
String string4 = string2 + " " + string3; // string4.length() ==> 11;
```

- ❖ Die Position eines Zeichens innerhalb einer Zeichenkette ist der *Index*.

Zeichenketten sind - wie Reihungen - *Null-indiziert*: das erste Zeichen hat den Index 0.

Das letzte Zeichen in einer Zeichenkette von 8 Buchstaben hat den Index 7.

Methoden zum Finden von Teilworten

- ❖ Die Methoden `indexOf` and `lastIndexOf` sind Instanzmethoden, mit denen man die Indexposition eines Teilwortes (*substring*) innerhalb einer Zeichenkette finden kann:

Index eines Teilwortes:

```
public int indexOf(String substring, int fromIndex);
```

```
public int lastIndexOf(String substring, int fromIndex);
```

Beispiele von Zeichenoperationen

- ❖ Die `indexOf`-Methode sucht von links nach rechts innerhalb der Zeichenkette. Die `lastIndexOf`-Methode sucht von rechts nach links.
- ❖ Beispiel: Suche nach einem einzelnen Zeichen "o"
Wir schreiben das gesuchte Zeichen als String

```
String string1 = "";  
String string2 = "Hello";  
String string3 = "World";  
String string4 = string2 + " " + string3;
```

```
string1.indexOf("o", 0) ==> -1  string1.lastIndexOf("o", 0) ==> -1  
string2.indexOf("o", 0) ==> 4   string2.lastIndexOf("o", 0) ==> 4  
string3.indexOf("o", 0) ==> 1   string3.lastIndexOf("o", 0) ==> 1  
string4.indexOf("o", 0) ==> 4   string4.lastIndexOf("o", 0) ==> 7
```

Das Resultat -1 bedeutet, dass das Zeichen nicht in der Zeichenkette ist.

Beispiele von Teilwortoperationen

```
String string1 = "";  
String string2 = "Hello";  
String string3 = "World";  
String string4 = string2 + " " + string3;
```

```
string1.indexOf("or", 0) ==> -1   string1.lastIndexOf("or", 0) ==> -1  
string2.indexOf("or", 0) ==> -1   string2.lastIndexOf("or", 0) ==> -1  
string3.indexOf("or", 0) ==> 1    string3.lastIndexOf("or", 0) ==> 1  
string4.indexOf("or", 0) ==> 7    string4.lastIndexOf("or", 0) ==> 7
```

Konvertierung von Daten in Zeichenketten

- ❖ Zusätzlich zu den Instanzmethoden offeriert die Klasse **String** eine Reihe von *Klassenmethoden* `String.valueOf(...)`, die primitive Typen (**int**, **double**, **bool**, **char**) in Objekte vom Typ `String` konvertieren:

```
static public String valueOf(primitiver Typ);
```

```
String number = new String (String.valueOf(128)); // "128"  
String truth = new String (String.valueOf(true)); // "true"  
String bee = new String (String.valueOf('B')); // "B"
```

Klassenmethoden werden aufgerufen, indem man den Klassennamen zur Qualifizierung benutzt (siehe Folie 15)

Man beachte den Unterschied zwischen Zeichen 'B' und Zeichenkette "B"

Beispiel: Schlüsselwortsuche

- ❖ Das Finden von Schlüsselworten oder reservierten Worten ist eine wichtige Aufgabe für Compiler, Browser und Dokument-Editoren.
- ❖ Problembeschreibung: Finde alle Instanzen des Schlüsselwortes *K* innerhalb einer Zeichenkette *S*.

Pseudocode für den Algorithmus: Wir definieren eine Indexvariable *pos*, einen Zähler sowie eine Zeichenkette *R* für das Resultat.

Setze *pos* auf den ersten Index von *K* in *S*

While (*pos* != -1) {

Erhöhe den Zähler

Konkatenierte *pos* mit dem Result *R*

Setze *pos* auf den nächsten Index von *K* in *S*

}

Konkatenierte den Wert von Zähler mit *R*

***R* ist das Ergebnis**

Java Implementation: Schlüsselwortsuche

```
public String keywordSearch(String s, String keyword) {  
    String result = " ";  
    int count = 0;  
    int pos = s.indexOf(keyword, 0);  
    while (pos != -1) {  
        count++;  
        result = result + pos + " ";  
        pos = s.indexOf(keyword, pos + 1);    // Finde die nächste Position  
    }  
    result = count + ": " + result;    // Konkateniere den Zähler  
    return result;                    // Resultat eine Zeichenkette  
} // keywordSearch()
```

Test

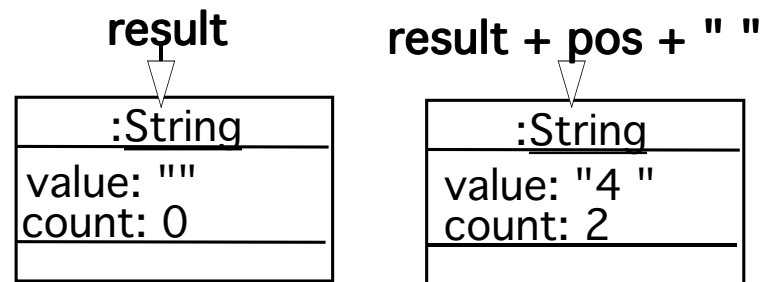
```
keywordSearch( "this is a test","is")  
keywordSearch( "able was i ere i saw elba","a")  
keywordSearch( "Dies ist ein Test","taste")
```

Resultat

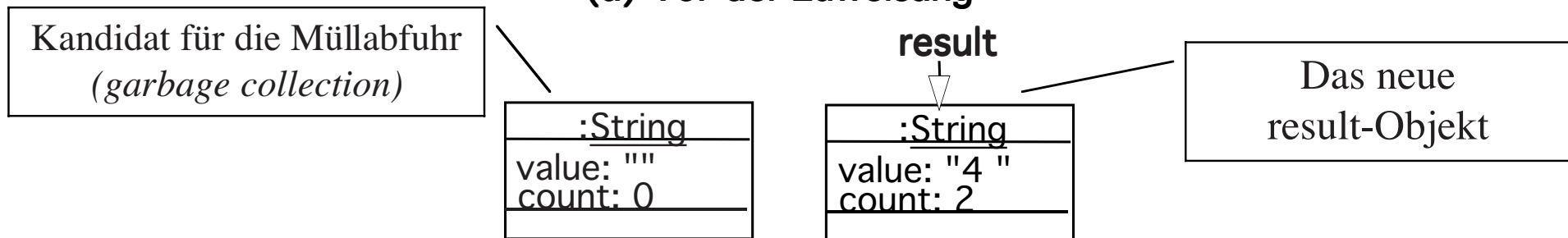
```
2: 2 5  
4: 0 6 18 24  
0:
```

Zeichenkettenoperationen erzeugen viele "Waisen"

- ❖ Java-Strings können nicht modifiziert werden. Immer, wenn ein neuer Wert einer String-Variablen zugewiesen wird, erzeugt Java ein neues String-Objekt und überlässt das alte der Müllabfuhr.
- ❖ Beispiel:
`result= result + pos + " "; // mit pos == 4`



(a) Vor der Zuweisung



(b) Nach der Zuweisung

Die Kommandozeile und der argv-Parameter in main

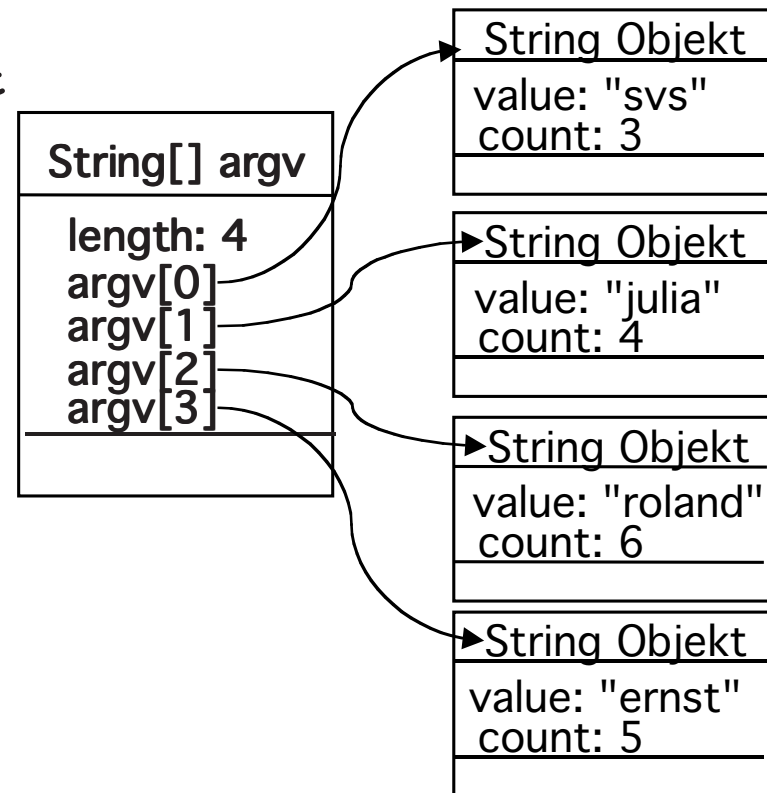
❖ **argv**, der einzige Parameter von **main**, ist eine Reihung von Zeichenketten (array of String), die die Argumente der Kommandozeile enthalten, die beim Aufruf von java mitgeliefert wurden. Beispiel:

❖ `% java Test svs julia roland ernst`

❖ Da **argv** eine Reihung von Strings ist, können wir leicht die Anzahl der Kommandozeilenargumente bestimmen:

– `int argc = argv.length;`

❖ **argv[0]** ist das erste Argument der Kommandozeile, d.h. "**svs**" (nicht der Name der Anwendung **Test** wie in C oder C++).



Wo stehen wir jetzt?

- ❖ **Objekt-basierte Programmierung:** Bietet Konzepte zur Konstruktion neuer Datentypen mit Hilfe von Klassen
- ❖ **Objekt-orientierte Programmierung:** Hauptziel ist die Wiederverwendung (reuse) von Bausteinen (components). Drei Konzepte werden wir jetzt kennenlernen:
 1. **Implementationsvererbung** (implementation inheritance), auch reale Vererbung genannt:
 - Wiederverwendung von Implementationen
 2. **Spezifikationsvererbung** (interface inheritance), auch virtuelle Vererbung genannt:
 - Wiederverwendung von Schnittstellen
 3. **Generische Klassen:**
 - Wiederverwendbarkeit von Datenstrukturen mit unterschiedlichen Basistypen.
- ❖ Im folgenden besprechen wir Javakonstrukte für diese Konzepte.

Das Wiederverwendungsproblem kommt in vielen Gestalten

❖ **"Weißer-Kasten"-Wiederverwendung (white box reuse):**

- Ich habe Zugriff auf die Entwicklungsprodukte (Modellierung, Systementwurf, Objektentwurf)
- Ich habe Zugriff auf die Klassenhierarchie und den Java-Code des Programms.
- Ich muss für dieses Programm eine neue Komponente entwickeln oder in einer anderen Umgebung benutzen oder an einen neuen Kunden anpassen.

❖ **"Schwarzer-Kasten"-Wiederverwendung (black box reuse)**

- Ich habe keinen Zugriff auf die Modelle und Entwürfe.
- Ich habe nur Zugriff auf das lauffähige Programm (binary code).
- Ich komme eventuell an die Schnittstelle ran.
- Ich muss dieses Programm an ein anderes anbinden.

In Info I machen wir "Weißer-Kasten"-Wiederverwendung

- ❖ **1. Implementationsvererbung:** Wiederverwendung von Code
- ❖ **2. Spezifikationsvererbung:** Wiederverwendung von Schnittstellen
- ❖ **3. Generische Klassen:** Wiederverwendung von Klassen

Implementationsvererbung: Wiederverwendung von Code

- ❖ Es gibt schon eine Liste für **int**-Elemente; ich brauche eine doppelt verkettete Liste für Addressbucheinträge.
- ❖ Mein Kunde möchte einen Keller haben
 - Die Liste bietet die Operationen Insert(), Find(), Delete()
 - Der Keller muss die Operationen Push(), Pop(), Top() anbieten
 - Wie kann ich die **List**-Klasse dafür verwenden?
- ❖ Ich bin in einer Elektrofirma angestellt und muss einen neuen Mikrowellenherd entwerfen. Kann ich die alte Software verwenden?

Generische Klassen: Wiederverwendung von Klassen

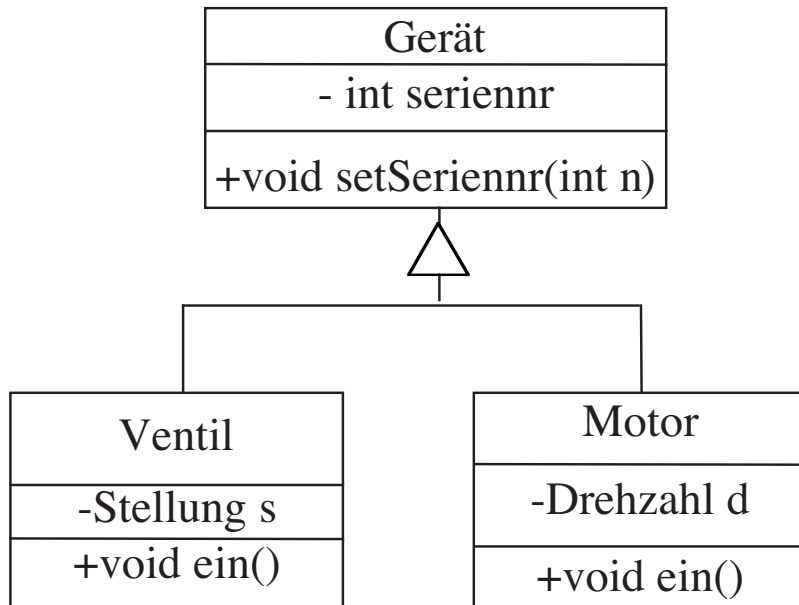
- ❖ Ich habe schon eine **List**-Klasse für Elemente vom Typ **int** und für Elemente vom Typ **double** geschrieben.
 - Sie sehen sich sehr ähnlich, unterscheiden sich nur bezüglich des Typs.
- ❖ Wie kann ich ohne großen Aufwand Listen bei der Implementation von Adressbüchern, Bauteilkatalogen, Flugzeugreservierungen einsetzen?
- ❖ Kann ich die **Adressbuch**-Klasse, die ich entwickelt habe, als Subsystem in mein kommerzielles E-Mail-Programm einbinden?
 - Oder sogar in die Abrechnungssoftware meiner Arztkunden, oder in das Kontaktmanagement-System meiner Autoverkäufer?

Das Konzept der Vererbung

- ❖ Definition Vererbung (Vorlesung 03):
 - Zwei Klassen stehen in einer **Vererbungsbeziehung** (inheritance relationship) zueinander, falls die eine Klasse
 - **Unterklasse** genannt - alle Merkmale der anderen Klasse
 - **Oberklasse** genannt - besitzt, und darüber hinaus noch zusätzliche Merkmale.
 - Umgekehrt verallgemeinert die Oberklasse die Unterklasse dadurch, dass sie spezialisierende Eigenschaften weglässt. Wir nennen das auch **Verallgemeinerungsbeziehung** (generalization relationship).
- ❖ **Alternativer Sprachgebrauch (Java):** Eine von einer **Superklasse** (super class) **B** abgeleitete Klasse **A** erbt die Attribute und Methoden, die **B** anbietet.

Beispiel für Vererbung

❖ Modell:



```
// Irgendwo in Main() oder in einer anderen Klasse:
....
Ventil v = new Ventil();
v.setSeriennr(1508)
Geraet g = new Ventil(); // ein Ventil ist ein Gerät
```

Java Code:

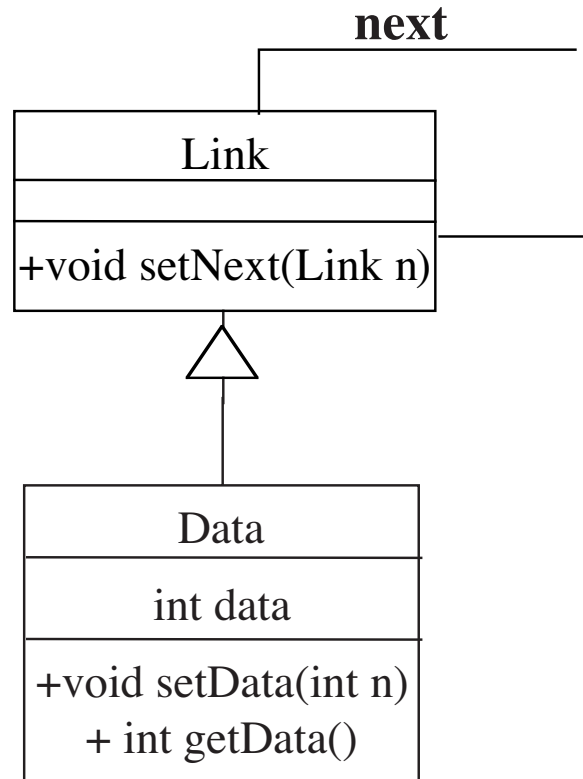
```
class Geraet {
    private int seriennr;
    public void setSeriennr(int n) {
        seriennr = n;
    }
}

class Ventil extends Geraet {
    private Stellung s;
    public void ein() {
        s.an = true;
    }
}

class Motor extends Geraet {
    private Drehzahl d;
    public void ein() {
        d.on = true;
    }
}
```

Noch ein Beispiel

❖ Modell:



Java Code:

```
class Link {
    Link next;
    public void setNext(Link n) {
        next = n;
    }
}

class Data extends Link {
    int data;
    public void setData(int d) {
        data = d;
    }
}

.....
Link l = new Data();
l.setData(5000);
.....
```

- ❖ Data erweitert Link um ein neues Feld `data` und zwei neue Methoden `setData()` und `getData()`, die auf Objekten vom Typ `Data` aufgerufen werden können.

Überschreibbare Methoden

- ❖ Falls die Unterklasse tatsächlich die Implementierung einer Methode aus der Superklasse wiederverwendet, sprechen wir von **Implementationsvererbung** (implementation inheritance).
- ❖ Oft ist Flexibilität gefordert: Die Implementierung der Methode aus der Superklasse muss an die Besonderheiten der Unterklasse angepasst werden, und zwar durch eine Reimplementierung.
- ❖ Methoden, die eine Reimplementierung zulassen, heißen **überschreibbare Methoden**.
 - In Java sind überschreibbare Methoden Standard, d.h. es gibt kein Schlüsselwort, um überschreibbare Methoden zu kennzeichnen.
 - Wenn eine Methode nicht überschrieben werden darf, muss sie mit dem Schlüsselwort **final** gekennzeichnet sein.

Beispiel für die Überschreibung einer Methode

Ursprünglicher Java-Code:

```
class Geraet {
    int seriennr;
    public final void help() {...}
    public void setSeriennr(int n) {
        seriennr = n;
    }
}
class Ventil extends Geraet {
    Stellung s;
    public void ein() {
        s.an = true;
    }
}
```

Diese Definition reimplementiert ("überschreibt") die Methode `setSeriennr()` aus der Klasse `Geraet`

Neuer Java-Code :

```
class Geraet {
    int seriennr;
    public final void help() {...}
    public void setSeriennr(int n) {
        seriennr = n;
    }
}
class Ventil extends Geraet {
    Stellung s;
    public void ein() {
        s.an = true;
    }
    public void setSeriennr(int n) {
        seriennr = n + s.seriennr;
    }
} // class Ventil
```

nicht überschreibbar

Überschreibbare Methoden werden oft mit leerem Methodenrumpf geschrieben

```
class Geraet {  
    int seriennr;  
    public void setSeriennr(int n) {}  
}  
class Ventil extends Geraet {  
    Stellung s;  
    public void ein() {  
        s.an = true;  
    }  
    public void setSeriennr(int n) {  
        seriennr = n + s.seriennr;  
    }  
} // class Ventil
```

Ich erwarte, dass die Methode `setSeriennr()` überschrieben wird. Also implementiere ich sie nicht, sondern schreibe nur einen leeren Methodenrumpf ("stub")

Überschreibung der Methode `setSeriennr()` aus der Klasse `Geraet`

Abstrakte Methoden und Abstrakte Klasse

- ❖ Wenn eine Methode noch keine Implementation hat, nicht einmal einen leeren Methodenrumpf, dann sprechen wir von einer **abstrakten Methode** (abstract method)
 - Eine Klasse, die mindestens eine abstrakte Methode enthält, ist eine **abstrakte Klasse** (abstract class). Sie wird mit den Schlüsselworten **abstract class** gekennzeichnet.

Beispiel einer abstrakten Methode

Ursprünglicher Java Code:

```
class Geraet {  
    int seriennr;  
    public void setSeriennr(int n) {  
        seriennr = n;  
    }  
}
```

Abstrakte Methode: hat keinen Rumpf, nicht einmal Klammern {}

```
class Ventil extends Geraet {  
    Stellung s;  
    public void ein() {  
        s.an = true;  
    }  
}
```

Implementation der abstrakten Methode setSeriennr ()

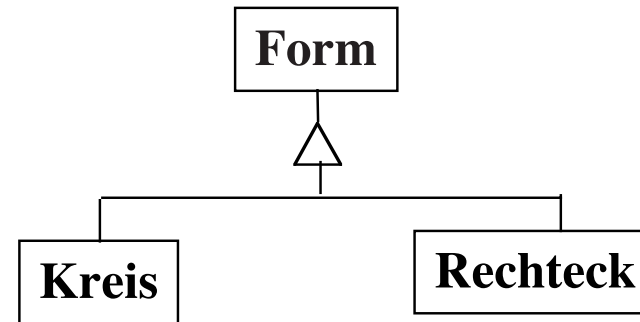
Java Code mit abstrakter Methode:

```
abstract class Geraet {  
    int seriennr;  
    public abstract void setSeriennr(int n);  
}
```

```
class Ventil extends Geraet {  
    Stellung s;  
    public void ein() {  
        s.an = true;  
    }  
    public void setSeriennr(int n) {  
        seriennr = n;  
    }  
} // class Ventil
```

Noch ein Beispiel für abstrakte Methoden

```
public abstract class Form {  
    public abstract double flaeche();  
    public abstract double umfang();  
}  
  
class Kreis extends Form {  
    private double r;  
    private PI = 3.14159265358979323846;  
    public Kreis() { r = 1.0; }  
    public Kreis(double radius) { r = radius; }  
    public double flaeche() { return PI * r * r; }  
    public double umfang() { return 2 * PI * r; }  
    public double getRadius() { return r; }  
}
```



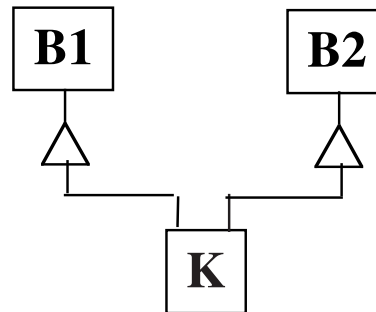
```
class Rechteck extends Form {  
    private double b, h;  
    public Rechteck() { b = 0.0; h = 0.0; }  
    public Rechteck(double breite, double hoehe)  
        { b = breite; h = hoehe; }  
    public double flaeche() { return b * h; }  
    public double umfang() { return 2 * (b + h); }  
    public double getBreite() { return b; }  
    public double getHoehe() { return h; }  
}
```

Eigenschaften von Klassen mit abstrakten Methoden

- ❖ Wenn wir eine Methode als abstrakt deklarieren, dann überlassen wir die Implementation einer Unterklasse, die von der Superklasse erbt.
 - Die Signatur der Methode in der Unterklassenimplementation muss mit der Signatur der abstrakten Methode identisch sein.
- ❖ Eine Klasse, die eine abstrakte Methode enthält, ist automatisch auch abstrakt, und muss deshalb als abstrakt deklariert werden.
- ❖ **Abstrakte Klassen können nicht instantiiert werden:**
 - Nur Unterklassen, in denen alle Methoden implementiert sind, können instantiiert werden.
- ❖ Wenn eine Unterklasse einer abstrakten Klasse nicht alle abstrakten Methoden implementiert, dann ist die Unterklasse selber auch abstrakt.

Mehrverfachvererbung

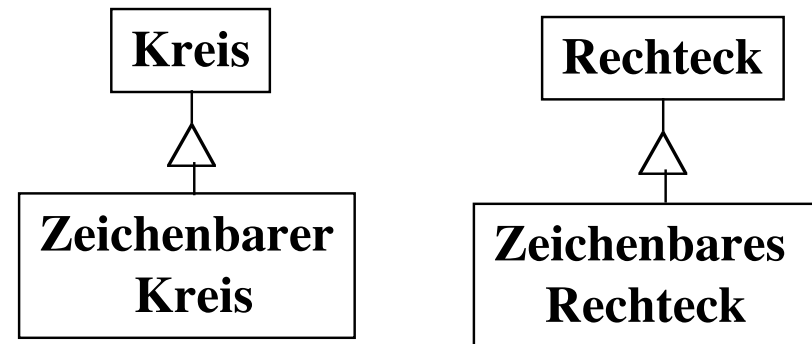
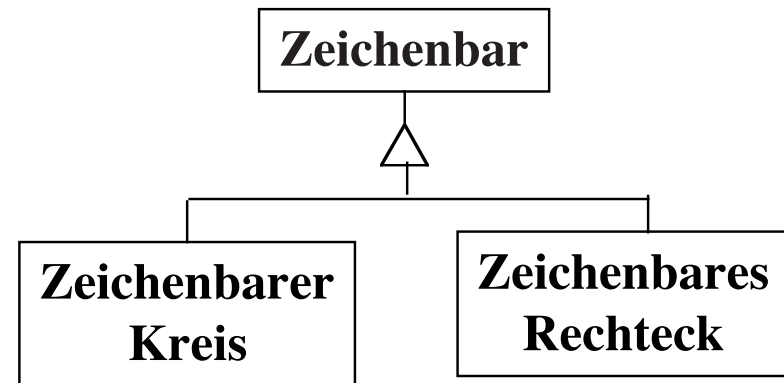
- ❖ **Definition Mehrfachvererbung:** Eine Klasse K erbt von mehreren Superklassen B_1, \dots, B_n .



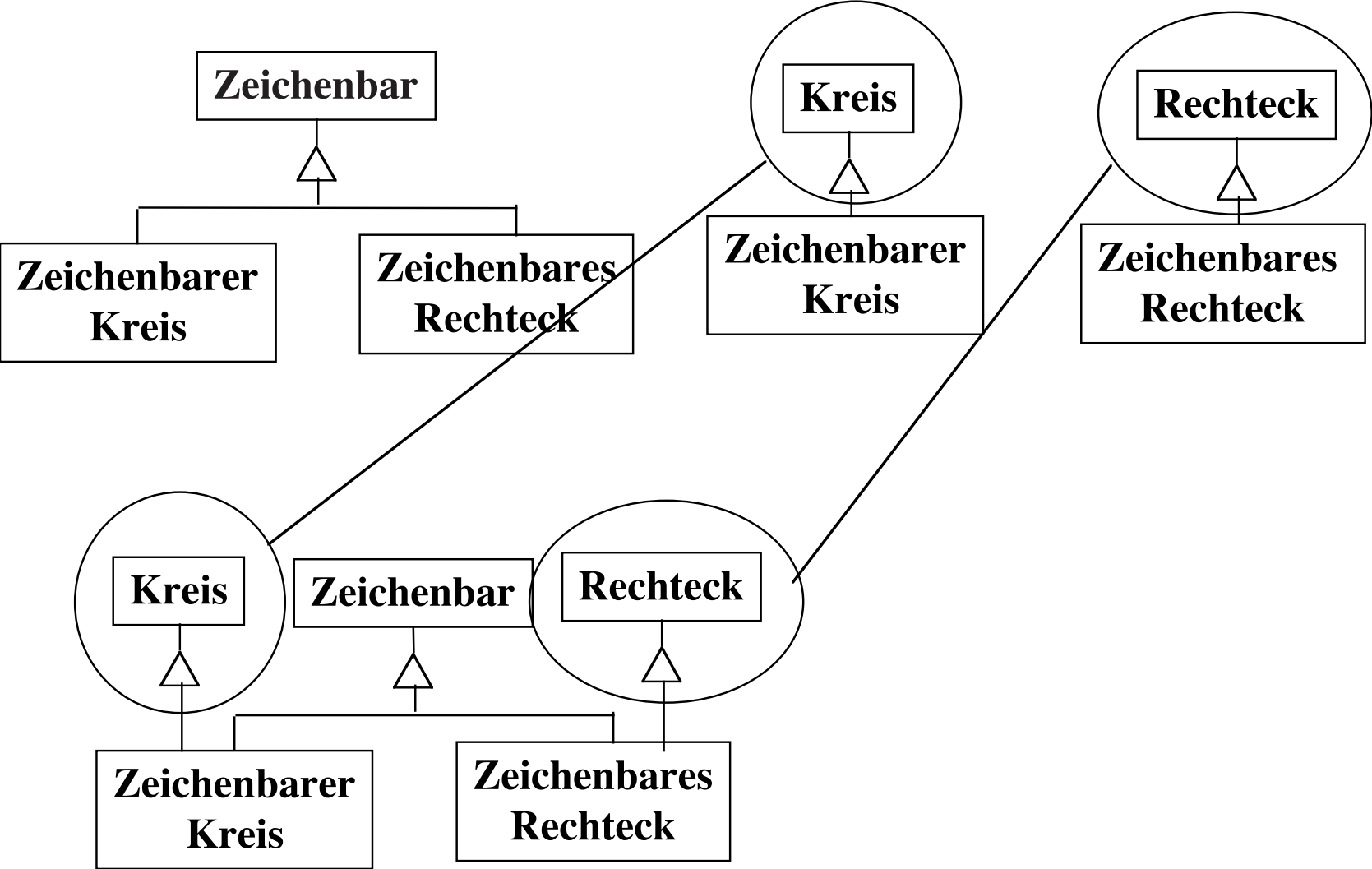
- ❖ Mehrfachvererbung kommt in der Modellierung ganz natürlich vor. Schauen wir uns noch einmal die Klasse **Form** an.
 - Wir wollen diese Klasse so erweitern, dass wir eine Anzahl von Formen auf dem Bildschirm malen können.

Modellierung von Zeichenbaren Formen

- ❖ Wir könnten eine abstrakte Klasse **Zeichenbar** definieren, und dann wieder verschiedene Unterklassen definieren, wie z.B. **ZeichenbarerKreis**, **ZeichenbaresRechteck**, usw.
- ❖ Wir wollen aber auch, daß diese **Zeichenbar** Typen die Methoden **flaeche** und **umfang** anbieten.
 - Um diese Methoden zu reimplementieren, würden wir gern **ZeichenbarerKreis** zu einer Unterklasse von **Kreis** machen, und **ZeichenbaresRechteck** zu einer Unterklasse von **Rechteck**.

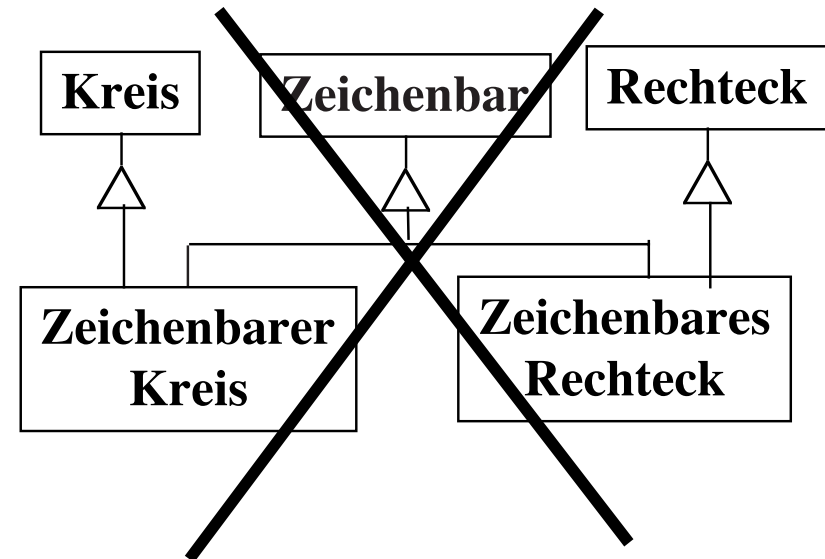


Was wir wollen



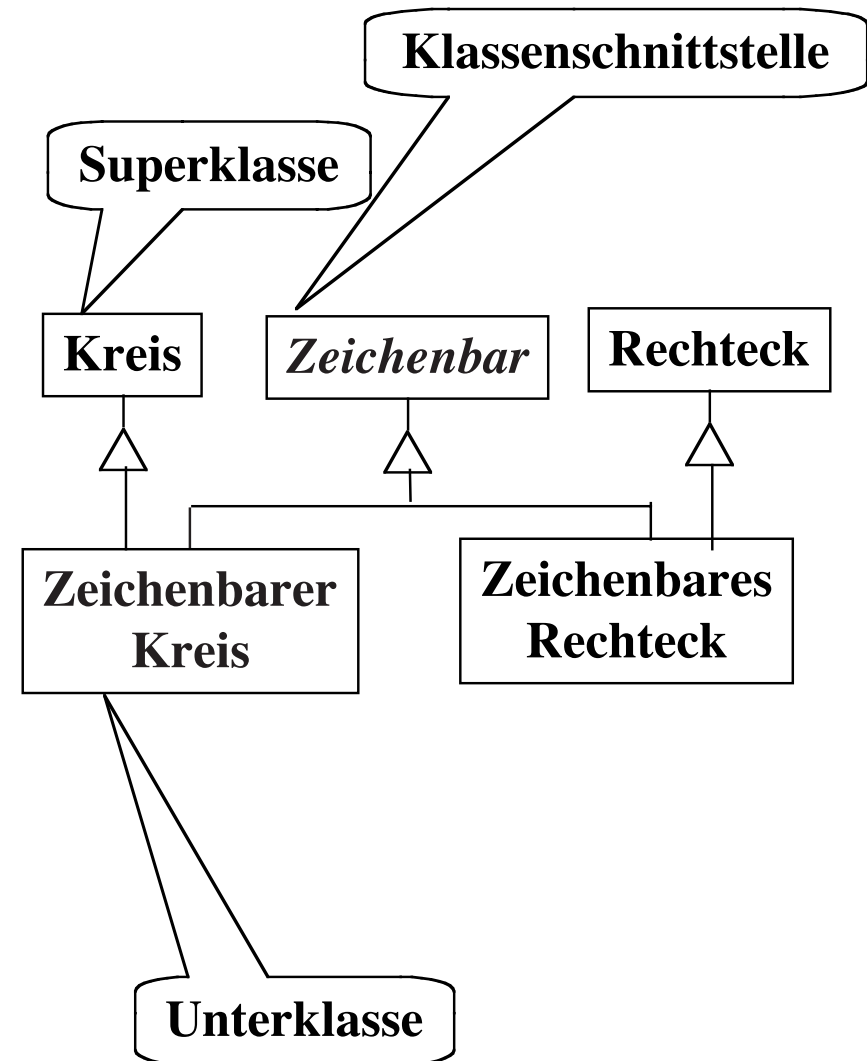
Mehrfachvererbung ist nicht erlaubt

- ❖ Java verbietet Mehrfachvererbung.
 - In Java können Klassen nur eine Superklasse haben.
 - Wenn **ZeichenbarerKreis** Unterklasse von **Kreis** ist, dann kann sie nicht gleichzeitig Unterklasse von **Zeichenbar** sein.



Java-Implementation mit Klassenschnittstelle

- ❖ Java hat eine andere Lösung zu diesem Problem: die **Klassenschnittstelle** (interface). Eine Klassenschnittstelle ist wie eine abstrakte Klasse mit folgenden Unterschieden:
 - Sie benutzt das Schlüsselwort "**interface**" anstatt "**abstract class**".
 - Sie darf keine Instanzvariablen haben, nur Konstanten ("**final static**").
- ❖ In der Modellierung zeichnen wir Klassenschnittstellen besonders aus, und zwar durch Schrägschrift.



Beispiel einer Klassenschnittstelle in Java

```
public interface Zeichenbar {  
    public void setFarbe(Farbe f);  
    public void setPosition(double x, double y);  
    public void zeichne();  
}
```

Schlüsselwort: interface

Keine Instanzvariablen: nicht erlaubt!

Ein Interface enthält nur abstrakte Methoden: keinen Rumpf, keine Klammern {}

❖ Sprachgebrauch:

- Eine Klasse erweitert (**extends**) ihre Superklasse
- Eine Klasse implementiert (**implements**) eine Schnittstelle.

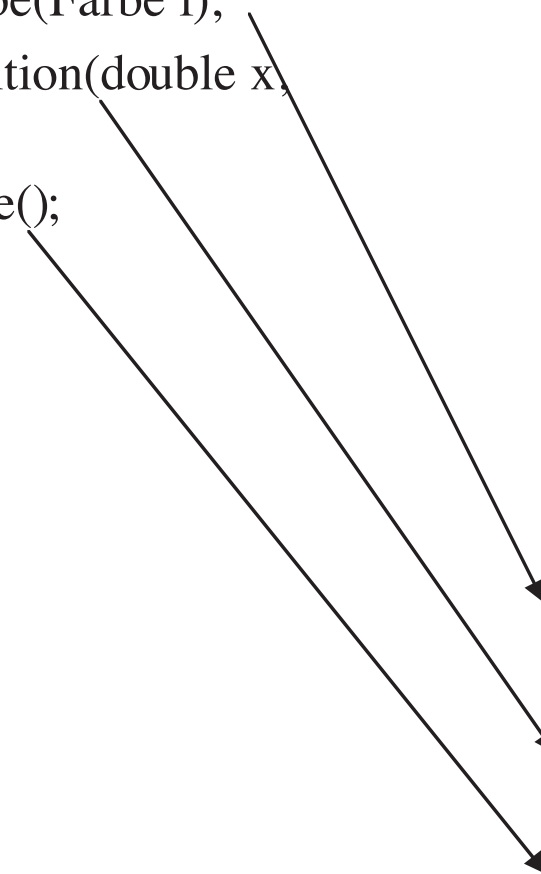
❖ Beispiel:

- `public class ZeichenbaresRechteck extends Rechteck implements Zeichenbar {....`

Implementation einer Java-Schnittstelle

```
interface Zeichenbar {  
    public void setFarbe(Farbe f);  
    public void setPosition(double x,  
        double y);  
    public void zeichne();  
}
```

```
public class ZeichenbaresRechteck extends  
    Rechteck implements Zeichenbar {  
    private Farbe lf;  
    private double lx, ly;  
    private zeichneRechteck () {  
        // Zeichnet ein Rechteck in Farbe lf in  
        // Position lx, ly. Breite und Höhe  
        // bekommt man durch Aufruf von  
        // getBreite() und getHoehe() (öffentliche  
        // Methoden in Rechteck, siehe Folie 44).  
    };  
    // Implementationen von Zeichenbar:  
    public void setFarbe(Farbe f)  
        { lf = f; }  
    public void setPosition(double x, double y)  
        { lx = x; ly = y; }  
    public void zeichne()  
        { zeichneRechteck(); }  
}
```

The diagram consists of three arrows pointing from the interface methods on the left to the corresponding implementation methods in the class on the right. The first arrow points from 'setFarbe(Farbe f)' in the interface to the implementation in the class. The second arrow points from 'setPosition(double x, double y)' in the interface to the implementation in the class. The third arrow points from 'zeichne()' in the interface to the implementation in the class.

Namenskonflikte

- ❖ Was ist, wenn der Implementierer von **ZeichenbaresRechteck** ein Feld **Farbe f** definiert hat, und eine abstrakte Methode mit formalem Parameter **Farbe f** benutzen muss?

```
interface Zeichenbar {  
    public void setFarbe(Farbe f);  
    public void setPosition(double x, double y);  
    public void zeichne();  
}
```

**Namenskonflikt
zwischen
Instanzvariable f
und Formalparameter f**

```
public class ZeichenbaresRechteck extends Rechteck implements Zeichenbar{  
    private Farbe f;    private double lx, ly;  
    public ZeichenbaresRechteck () { f = red; }  
    private zeichneRechteck () {}  
    public void setFarbe(Farbe f) { f = f; }  
    public void setPosition(double x, double y) { lx = x; ly = y; }  
    public void zeichne() { zeichneRechteck(); }  
}
```

Wir wollen dem Feld f den Wert des Parameters f zuweisen. Aber nicht so!

Bindung und Bindungsbereich

- ❖ **Definition Bindung:** Sei x ein beliebiger Bezeichner (Variable, Konstante), t ein Typ und E ein Ausdruck dieses Typs.
Dann heißt
 - $t\ x = E$;die **Deklaration des Bezeichners x** .
Durch die Deklaration ist x an den Wert von E **gebunden**.
- ❖ **Definition Bindungsbereich:** Eine Bindung bezieht sich immer auf einen Bereich, den sogenannten Bindungsbereich. In Java werden Bindungsbereiche durch geschweifte Klammern "{" und "}" gekennzeichnet.
- ❖ Beispiele von Bindungsbereichen:
 - Formaler Parameter: Methodenrumpf
 - Lokale Variable: Methodenrumpf ab Stelle der Definition
 - Instanz- und Klassenvariable: Innerhalb des Klassenrumpfes
 - Zählvariable: Schleifenkörper

Bindungsbereiche

```
public class ZeichenbaresRechteck extends Rechteck  
    implements Zeichenbar {  
    private Farbe f;    ...  
    ...
```

**Bindungsbereich
der Instanzvariable f**

```
public void setFarbe(Farbe f)
```

```
{
```

```
....
```

```
}
```

**Bindungsbereich des
formalen Parameters f**

```
private foo () {
```

```
    for (int f = 0; f < 1000; f++) {
```

```
        ....
```

```
    }
```

```
}
```

**Bindungsbereich der
Schleifenvariable f**

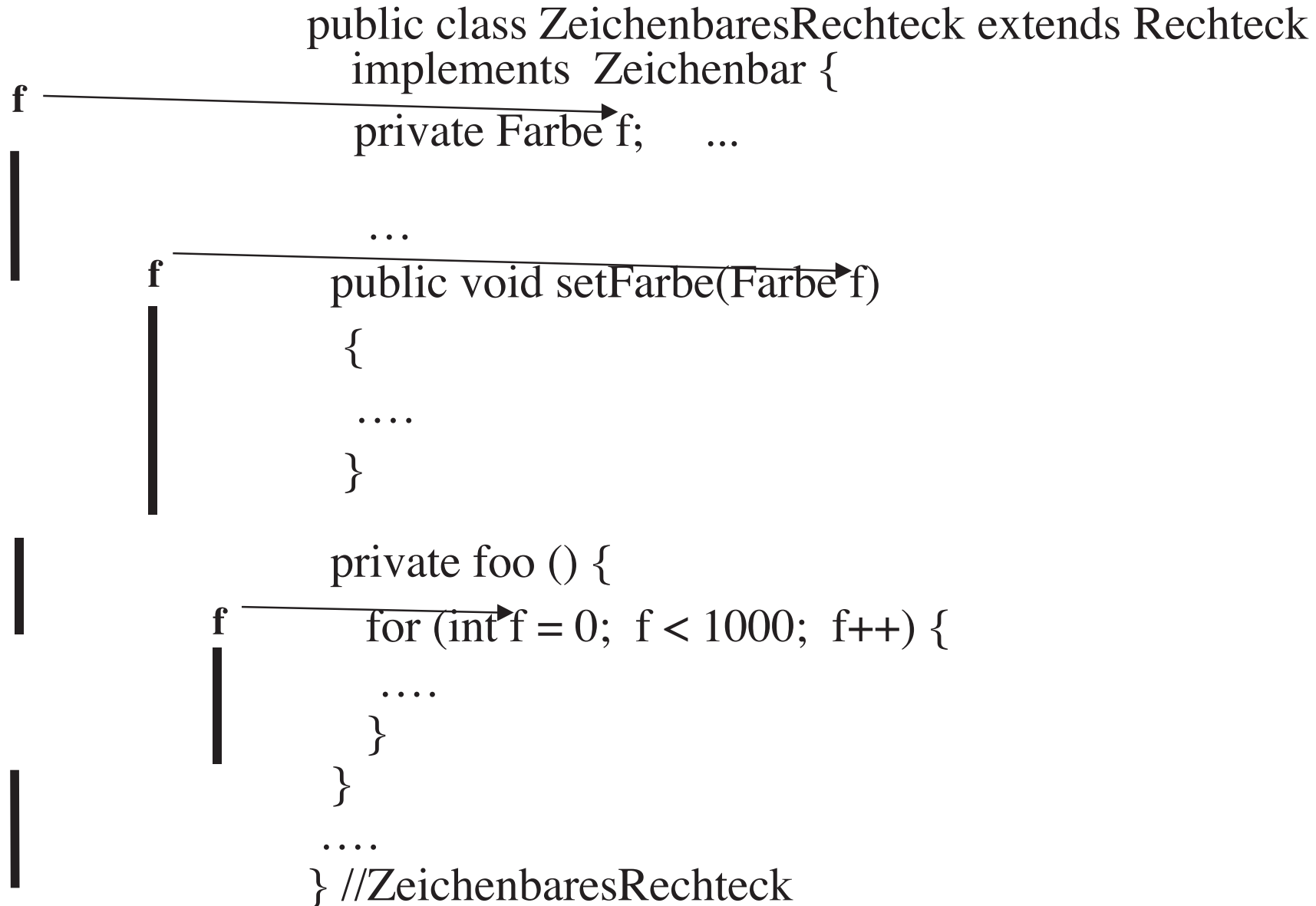
```
....
```

```
} //ZeichenbaresRechteck
```

Lebensdauer und Gültigkeitsbereich einer Bindung

- ❖ In einem Java-Programm kann ein Bezeichner also mehrfach vereinbart werden, gebunden sein, und mit verschiedenen Werten belegt sein.
Wir unterscheiden deshalb zwischen Lebensdauer und Gültigkeitsbereich einer Bindung.
- ❖ **Lebensdauer einer Bindung:**
Der Bindungsbereich des Bezeichners.
- ❖ **Gültigkeitsbereich einer Bindung (scope):**
Die Lebensdauer eines Bezeichners abzüglich aller Bindungsbereiche, in denen der Bezeichner erneut gebunden wird.
- ❖ **Verschattung:**
Durch einen Gültigkeitsbereich wird eine andere, außerhalb vorhandene Bindung für denselben Bezeichner außer Kraft gesetzt.
Man sagt, die Bindung wird verschattet (überlagert).

Gültigkeitsbereiche des Bezeichners *f*



Die "this"-Referenz

- ❖ Schauen wir uns noch einmal an, wie Instanzmethoden in Java benutzt werden:

```
{ ....  
  class Link {  
    private int data;  
    ...  
    public int getData() { return data};  
  } // class Link  
  
  ....  
  Link current;  
  
  ....  
  current.getData();  
  
  ....  
}
```

- ❖ Jede Instanzmethode bezieht sich implizit auf eine Instanz **this**; **this** entspricht genau der Instanz, für welche die Methode aufgerufen ist.
- ❖ Die Deklaration
 public int getData() { return data; }
ist also eigentlich eine Kurzschreibweise von
 public int getData() { return **this**.data; }
wobei **this** das aktuelle Objekt referenziert.
- ❖ Im Normalfall wird **this** nicht benötigt: Methoden greifen implizit auf die Felder zu, die in der Klasse deklariert sind:
 - **return data**
- ❖ Nur im Falle von Verschattungen benutzen wir **this** explizit:
 - **return this.data**

Lösung des Namenskonfliktes

- ❖ Mit **this.f** können wir auf die Instanzvariable **f** in **ZeichenbaresRechteck** zugreifen, obwohl sie verschattet ist.

```
interface Zeichenbar {
    public void setFarbe(Farbe f);
    public void setPosition(double x, double y);
    public void zeichne();
}

public class ZeichenbaresRechteck extends Rechteck implements Zeichenbar{
    private Farbe f;    private double lx, ly;
    public ZeichenbaresRechteck () { f = red; }
    private zeichneRechteck () {}
    public void setFarbe(Farbe f) { this.f = f; }
    public void setPosition(double x, double y) { lx = x; ly = y; }
    public void zeichne() { zeichneRechteck(); }
}
```



**Wir weisen der
Instanzvariable f den
Wert des Parameters f zu.**

Wie benutzt man Klassenschnittstellen in Java?

- ❖ Im Gegensatz zur Vererbung von Klassen, bei denen in Java nur die Einfachvererbung zulässig ist, können auch mehrere (Klassen-)Schnittstellen mittels der **implements**-Klausel aufgeführt und implementiert werden.
- ❖ Eine Klasse, die eine Schnittstelle realisiert, muss alle in der Schnittstelle genannten Methoden implementieren.
- ❖ Eine Schnittstelle ist in Java ein Referenztyp (reference type), d.h. man kann Variablen oder Parameter mit einer Schnittstelle als Typ deklarieren.
 - Dies wird oft bei der Programmierung generischer Klassen gemacht.

Konzepte für Wiederverwendbarkeit und Erweiterbarkeit in Java

- ✓ Implementations-Vererbung
- ✓ Spezifikations-Vererbung
- ✓ Abstrakte Klassen
- ✓ Klassenschnittstellen
- Generische Klassen

Generische Klassen

- ❖ Wir haben die Klassen **LinkedList** und **Tree** in den letzten Vorlesungen eingeführt, um grundsätzliche Konzepte für Listen- und Baumoperationen zu erklären.
 - Aus diesem Grund hatten wir uns auf Listen und Bäume beschränkt, deren Knoten nur applikationspezifische Daten vom Typ **int** speichern konnten.
 - Was uns jetzt interessiert, ist die Frage, ob wir diese Strukturen auch für Abstraktionen aus der Applikationsdomäne (Personen, Autoteile, Flugzeugreservierungen, ...) nehmen können.
- ❖ Wir wollen deshalb jetzt Listen- und Baum-Klassen entwickeln, die eine generelle Knotenklasse benutzen, in der wir beliebige Daten speichern und verarbeiten können.
- ❖ Fangen wir mal mit der Liste von **int**-Knoten an. Schauen wir uns noch einmal die Implementation des **int**-Listenelementes an.

*Java-Implementation von **int**-Listenelementen*

```
class Link {  
    private    int    data;    // Datum  
    private Link next; // Nächster Eintrag  
                        // in Liste  
  
    public Link ( int    value) {  
        data = value; // Initialisiere Datum  
        next = null;  // Initialisiere next  
    }  
  
    public    int    getData () {  
        return data;  
    }  
  
    public void setData (int    value) {  
        data = value;  
    }  
}
```

```
    public Link getNext () {  
        return next;  
    }  
  
    public void setNext (Link n) {  
        next = n;  
    }  
  
    public void displayLink ()    {  
        System.out.print("{ " + data + " } ");  
    }  
} // end class Link
```

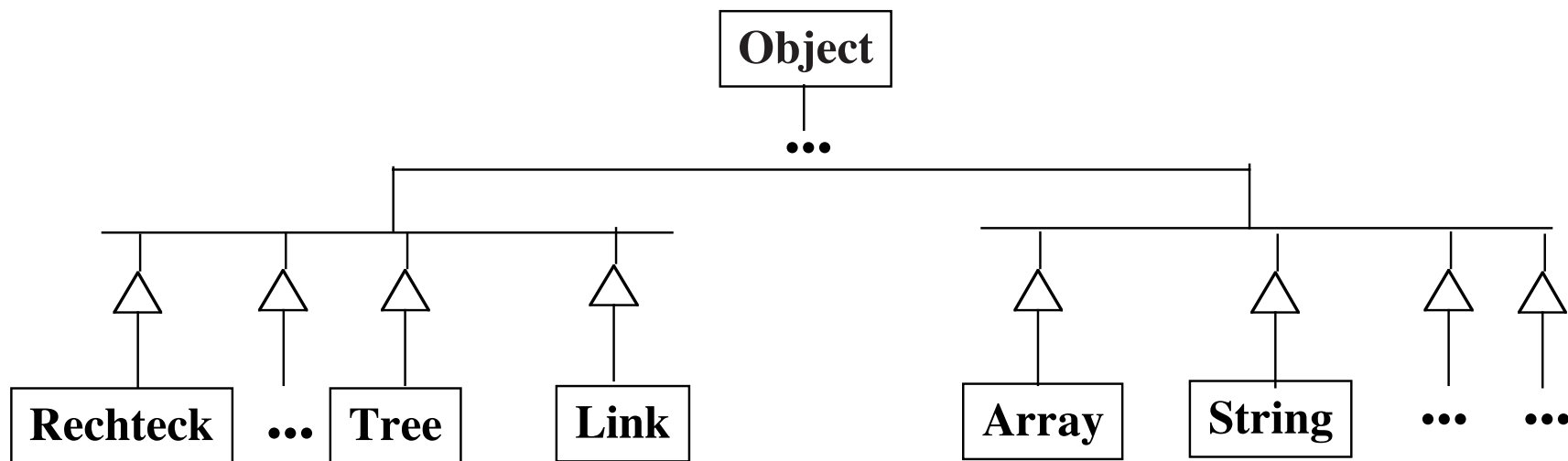
Java-Implementation von generischen Listenelementen

```
class Link {  
    private Object data; // Datum  
    private Link next; // Nächster Eintrag  
                        // in Liste  
  
    public Link (Object value) {  
        data = value; // Initialisiere Datum  
        next = null; // Initialisiere next  
    }  
  
    public Object getData () {  
        return data;  
    }  
  
    public void setData (Object value) {  
        data = value;  
    }  
}
```

```
    public Link getNext () {  
        return next;  
    }  
  
    public void setNext (Link n) {  
        next = n;  
    }  
  
    public void displayLink () {  
        System.out.print(data.toString());  
    }  
} // end class Link
```

Java's Klassenhierarchie

- ❖ In Java unterscheiden wir benutzerdefinierte Klassen und vordefinierte Klassen.
 - Benutzerdefinierte Klassen: **Rechteck**, **Link**, **Tree**,...
 - Vordefinierte Klassen: **Array**, **String**,...
- ❖ Alle Klassen in Java bilden eine Klassenhierarchie.
 - Die Superklasse der Klassenhierarchie heißt **Object**
 - Jede Java-Klasse, bis auf **Object**, hat eine Superklasse.



Die Java-Klasse Object

```
public class Object {  
    ....  
    // Instanzmethoden  
    public boolean equals(Object obj);  
        // true, wenn beide Objekte gleich sind.  
    ....  
    public String toString();  
        // Konvertiert die Werte der Felder eines Objektes in eine Zeichenkette  
    ....  
}
```

**Vollständige Definition von Object
=> Java Referenz-Manual**

Java-Implementation: Verkettete Liste von `int`-Elementen

```
class LinkedList {
    private Link head;           // Referenz auf das erste Element in der Liste

    public LinkedList () {      // Konstruktor
        head = null;           // Noch keine Elemente in der Liste
    }

    public boolean isEmpty () { // wahr, wenn Liste leer ist
        return (head == null);
    }

    public void insertFirst (int value) { // Erzeuge ein neues Element am Anfang
        newLink = new Link(value);
        newLink.setNext(head); // newLink zeigt auf den alten Wert von head
        head = newLink;        // head zeigt auf das neue Element
    }
}
```

Java-Implementation einer generischen Liste 1/30/01

```
class LinkedList {  
    private Link head;           // Referenz auf das erste Element in der Liste  
  
    public LinkedList () {       // Konstruktor  
        head = null;           // Noch keine Elemente in der Liste  
    }  
  
    public boolean isEmpty () {  // wahr, wenn Liste leer ist  
        return (head == null);  
    }  
  
    public void insertFirst (Object value) {}  
    public Link find(String key) { return null; }  
  
} // class LinkedList
```

Wir schreiben diese Methoden mit leeren Rümpfen (stubs), weil wir sie ohnehin überschreiben wollen.