

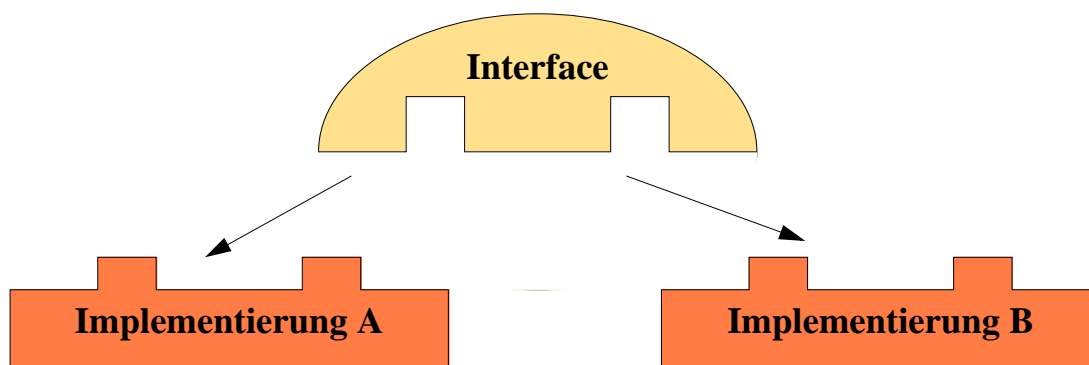
# Z.1 OO-Grundlagen mit Java

- Fundamentale Konzepte der objektorientierten Programmierung:

## Abstraktion und Kapselung

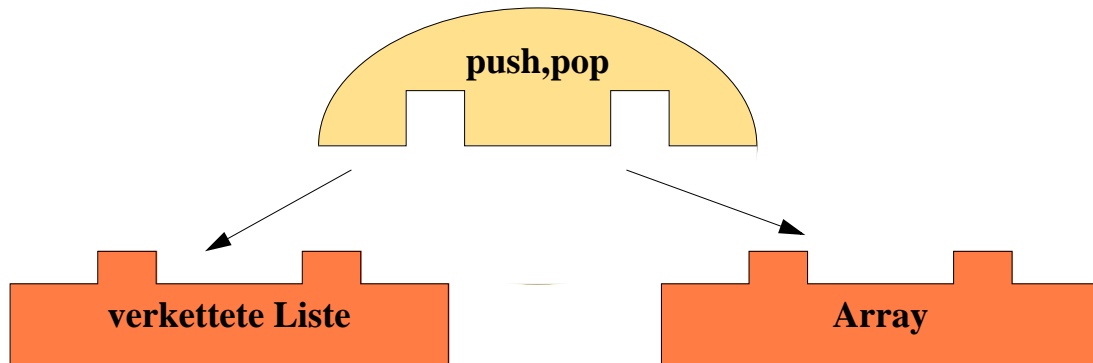
## 1 Abstraktion

- Trennung von:
  - ◆ Schnittstelle (interface): Was kann getan werden?
  - ◆ Implementierung: Wie wird es gemacht?



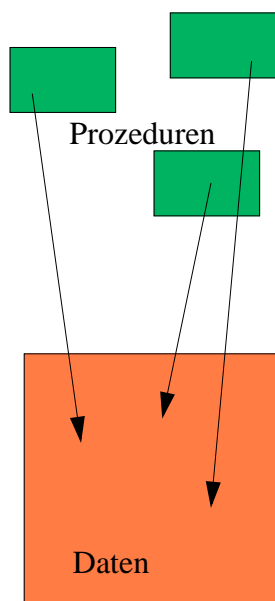
# 1 Abstraktion

- Beispiel: stack
  - ◆ Interface: push, pop
  - ◆ Implementierung: verkettete Liste, Array

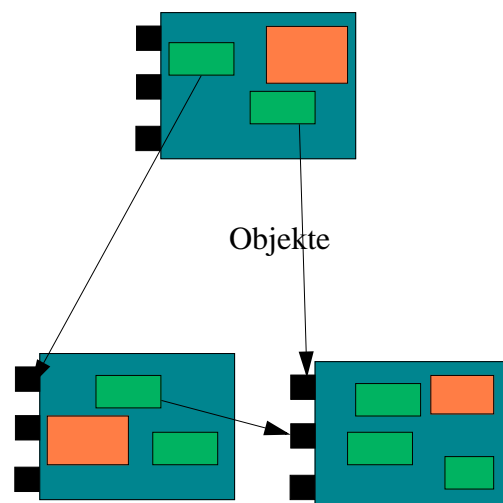


# 2 Kapselung

Prozedurale  
Programmierung

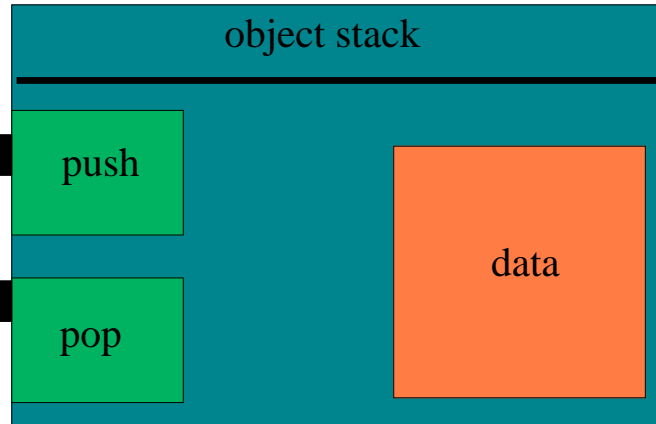


Objektorientierte  
Programmierung



## 2 Kapselung

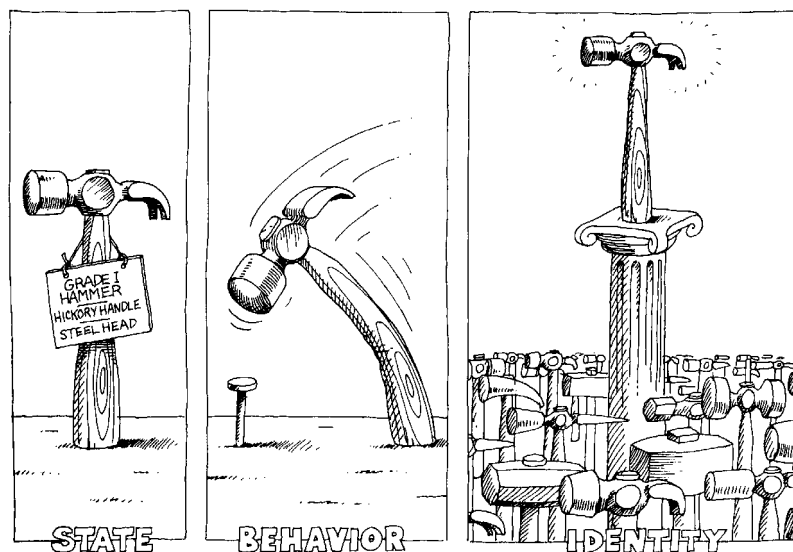
- Objekte: Gekapselte Datenstruktur, bestehend aus:
  - ◆ Daten (Instanzvariablen, Attribute)
  - ◆ Methoden (Operationen)



- Kapselung unterstützt die Bildung von Abstraktionen.

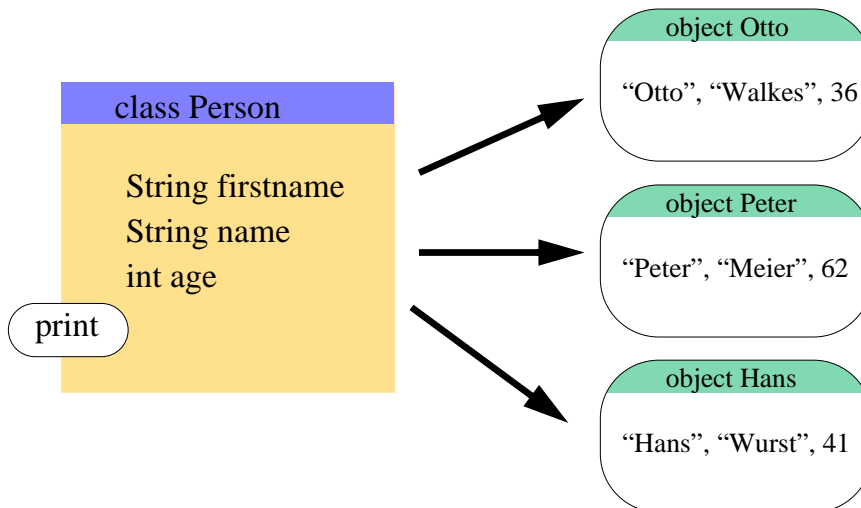
## Z.2 Objekte

### 1 Eigenschaften von Objekten



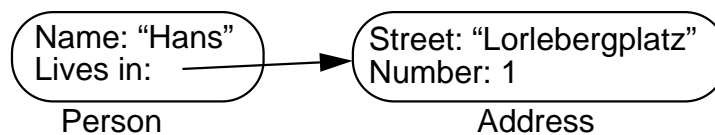
## 2 Klassen

- Objekte sind *Instanzen* einer Klasse.
- Die Klasse bestimmt die interne Struktur und die Schnittstelle eines Objekts.
- Beispiel:

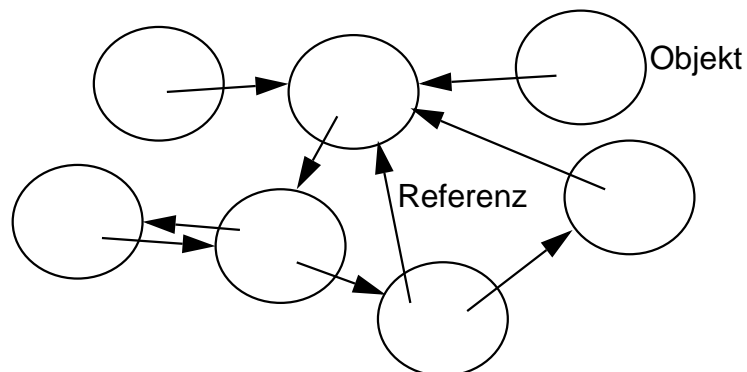


## 3 Das Objekt-Netz

- Objekte können andere Objekte referenzieren.
- ◆ Beispiel: eine Person kann eine Adresse referenzieren:



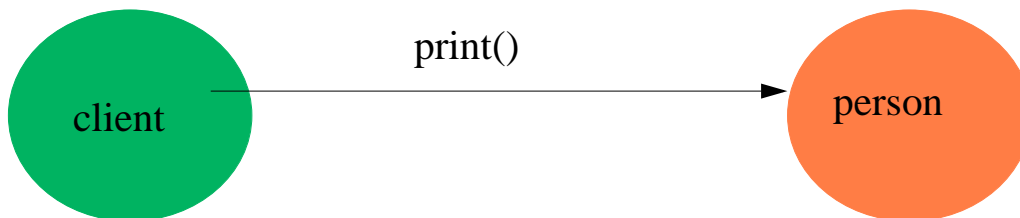
- Darstellung des Zustands eines objektorientierten Programms:



## 4 Nachrichten / Methoden

- Objekte kommunizieren mit Hilfe von Nachrichten.
  - ◆ Jedes Objekt legt sein eigenes Verhalten selbst fest.
  - ◆ Die Objektsemantik ist nicht über das ganze Programm verteilt.
- Nachricht = Methodenaufruf an einem Objekt

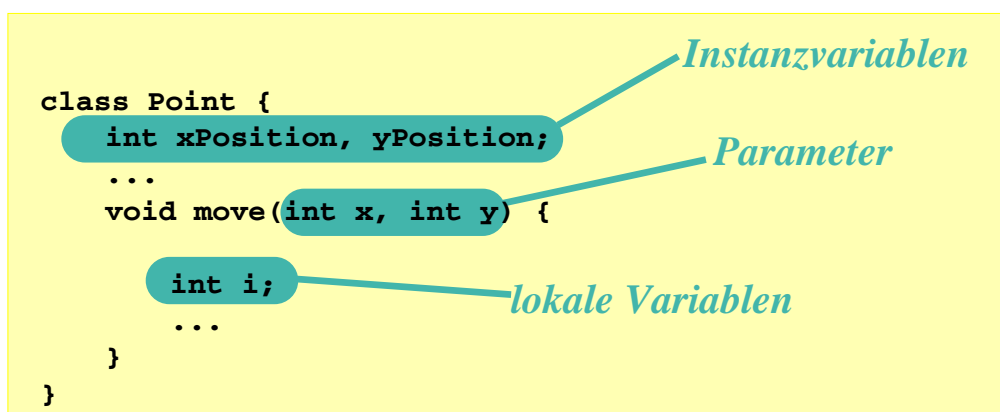
```
person.print()
```



- objektorientiertes Programm: mehrere Objekte kommunizieren miteinander, um eine bestimmte Aufgabe zu erfüllen.

## 5 Methoden und Variablen

- Methoden können auf 3 verschiedene Arten von Variablen zugreifen:
  - ◆ Parameter (Argumente)
  - ◆ lokale Variablen
  - ◆ Instanzvariablen



## 5 Variablen

- Parameter und lokale Variablen müssen unterschiedliche Namen haben.
- Parameter und lokale Variablen überdecken Instanzvariablen.

```
class Test {
    int a;
    void m(int a) {
        a = 5; // does not change instance variable a
    }
}
```

## 5 Variablen

- Zugriff auf Instanzvariablen mit:
  - ◆ *instanceName.variableName*

```
class Person {
    String firstname;
    String name;
    int age;

    boolean sameName(Person otherPerson) {
        if (name == otherPerson.name) return true;
        return false;
    }
}
```

*Instanzvariable*      *Instanzvariable*

## 6 Der Parameter *this*

- Jede Methode hat einen *impliziten* Parameter **this**.
- **this**: Referenz auf die Instanz, an der die Methode aufgerufen wurde:

```
class Person {
    String name;
    ...
    void print() {
        System.out.println(this.name);
    }
}
```

- **this** kann bei Eindeutigkeit weggelassen werden.
- Beispiel für Mehrdeutigkeit:

```
class Person {
    ...
    boolean compare(String name) { return this.name == name; }
}
```

## 7 Überladen

- Methoden mit unterschiedlichen Parametern können den gleichen Namen haben.
- Beispiel:

```
class Date {
    ...
    void print(PrintStream stream) { stream.println(...); }
    void print() { print(System.out); }
}
```

- Hinweis: Überladen funktioniert nur mit Parametern nicht mit dem Typ des Rückgabewerts:

```
class Income {
    ...
    int computeIncome() { ... }
    float computeIncome() { ... } // Error !!
}
```

## 8 Objekt-Initialisierung

- Erzeugen eines Objekts bedeutet Reservierung von Speicher.
- Dieser Speicher muss initialisiert werden.
- Eine Möglichkeit:
  - ◆ Explizites Aufrufen einer Initialisierungsmethode.
  - ◆ Nachteil: fehleranfällig.

## 9 Konstruktoren

- Konstruktoren dienen der Initialisierung des Objekts.
- Name des Konstruktors = Name der Klasse.
- Der Konstruktor wird automatisch nach der Objekterzeugung aufgerufen.

## 9 Konstruktoren (2)

- Mehrere Konstrukturen sind möglich
- Aufruf eines anderen Konstruktors mit **this(...)**:

```
class Person {
    String name;
    int age;

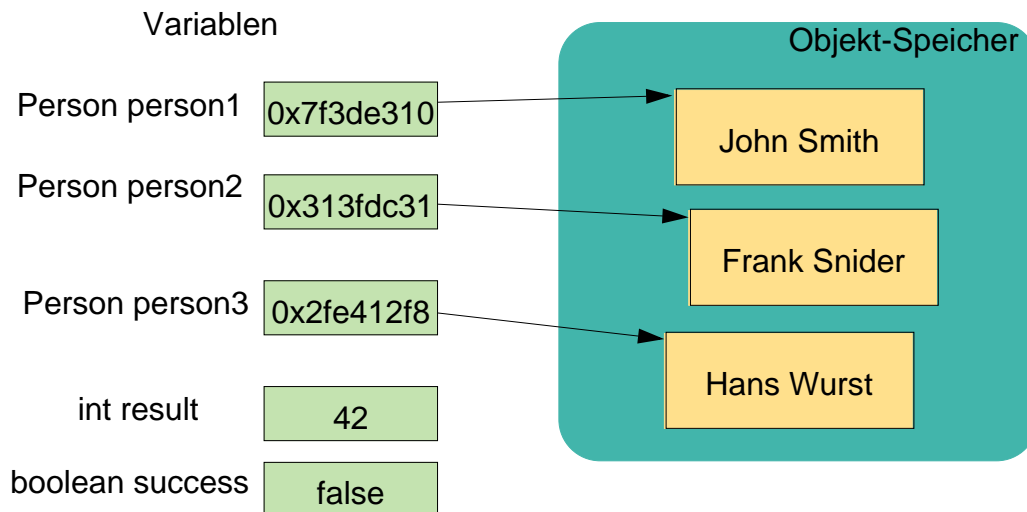
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    Person(String name) {
        this(name, 18);
    }
    ...
}
```



## 10 Objekte und Referenzen

- Java-Variablen bezeichnen keine Objekte, sondern Referenzen.

```
Person p; // Deklaration einer Referenz auf ein Objekt
          // der Klasse Person
p.print(); // Fehler: Methodenaufruf an einer null-Referenz
```



## 11 Zuweisungen

- = weist einer Variable eine Referenz zu
- == vergleicht zwei Referenzen
- Einer Variable eines primitiven Datentypes kann keine Referenz zugewiesen werden.
- Einer Variable, welche eine Objektreferenz ist, kann niemals der Wert eines primitiven Datentypes zugewiesen werden.
- Beispiel:

```
Person p; // Deklaration einer Referenz-Variablen
int i=42; // Deklaration und Initialisierung einer
          // Variable eines primitiven Datentypes
p = i; // Fehler: Zuweisung zwischen Referenz und
        // primitiven Datentyp
```

## 12 Aufrufsemantik von Methoden

- Objekt-Parameter werden als Referenz übergeben.
- Primitive Datentypen (int, float, etc.) werden als Wert übergeben.
- Beispiel:

```
void meth(int a, Person k) {
    a = 5;                // a: passed by value
    k.setAge(25);        // k: passed by reference
}
```

## 13 Gleichheit und Identität

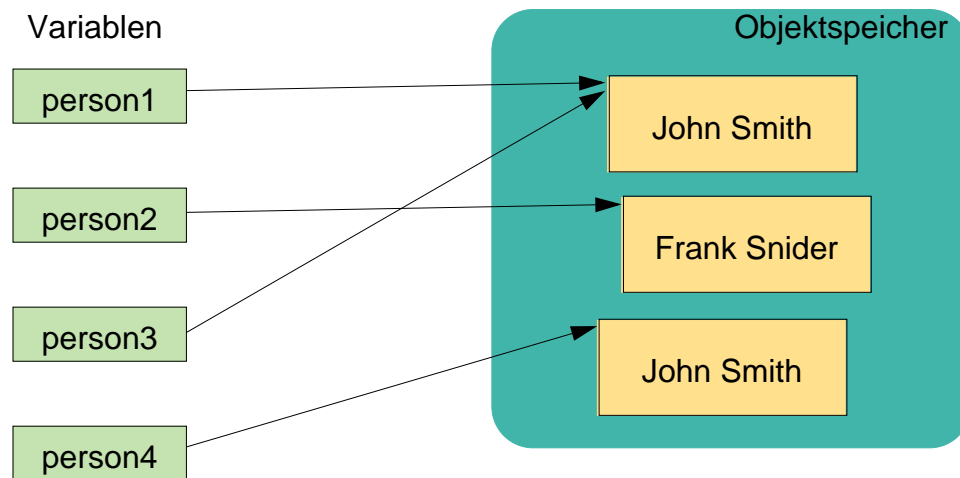
- Unterschied zwischen *gleichen Objekten* und *identischen Objekten*:

```
class Date {
    int day, month, year;
    Date(int day, int month, int year) {
        this.day = day; this.month = month; this.year = year;
    }
}
...
Date d = new Date(1,3,98);
Date d1 = new Date(1,3,98);
Date d2 = d;
```

- ◆ d und d1 sind gleich
- ◆ d und d2 sind identisch

## 14 Identität und Referenzen

- Identität: gleiche Referenz
- Gleichheit: Inhalt der referenzierten Objekte ist gleich



- Welche Personen sind identisch, welche gleich?

## 15 Testen von Gleichheit und Identität

- Identität kann mit dem Operator `==` getestet werden:

```
if (d == d1) { ... }
```

- Gleichheit kann mit der Methode `equals` getestet werden:

```
if (d.equals(d1)) { ... }
```

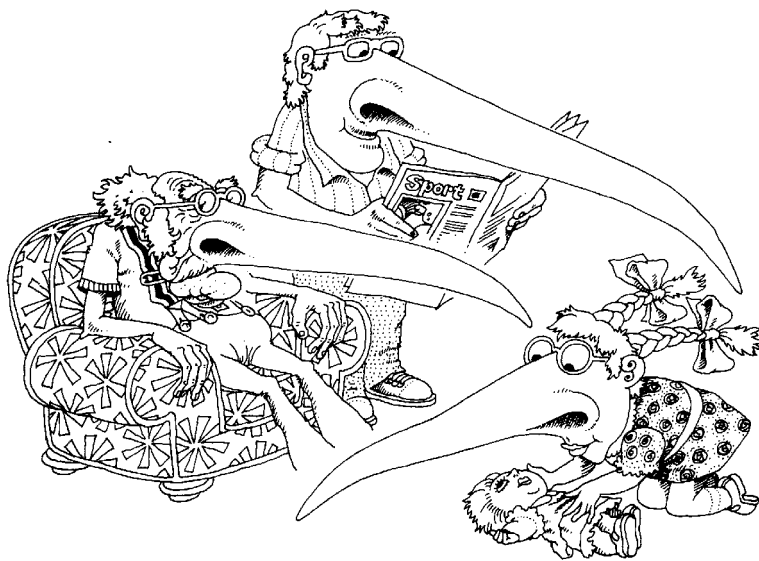
- Die Methode `equals` muss selbst implementiert werden:

```
class Date {
    ...
    public boolean equals(Object o) {
        if (! (o instanceof Date)) return false;
        Date d = (Date)o;
        return d.day == day && d.month == month && d.year == year;
    }
}
```

## Z.3 Vererbung

- Definition von Objekten durch Verweise auf andere Objekte.
  - ◆ Beispiel:
    - Definition eines Tieres: Ein Ding welches atmet und isst.
    - Definition einer Kuh: Ein Tier, dass “muuuu” macht und Milch gibt.
    - Kuh *erbt* die Eigenschaften “atmen” und “essen” von Tier.

### 1 Vererbung in der echten Welt



## 2 Vererbung in Java

- Vererbung: Definition eines neuen Objekts auf der Basis einer spezialisierten Definition eines existierenden Objekts.
- Neue Klassen können von existierenden Klassen *abgeleitet* werden.
- Beispiel: Ein Kunde ist eine Person.

```
class Customer extends Person {
    int number;
    ...
}
```

- **Customer** ist eine *Unterklasse (subclass)* von **Person**
- **Person** ist die *Oberklasse (superclass)* von **Customer**.

## 3 Unterklassen

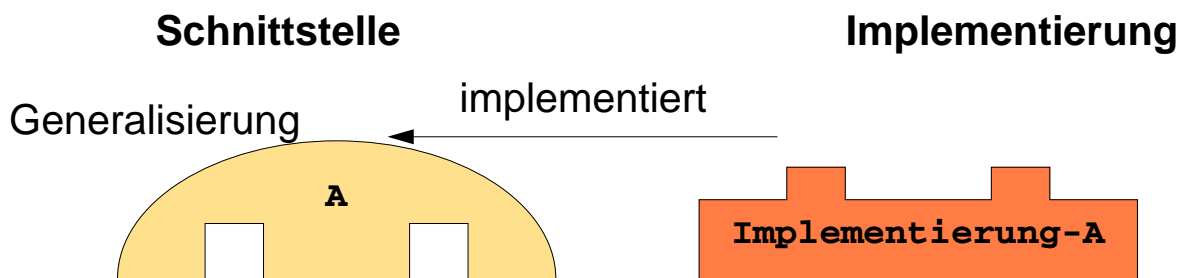
- Unterklassen erben
  - ◆ Zustand (Instanzvariablen) und
  - ◆ Verhalten (Methoden) von der Oberklasse.
- Unterklassen können:
  - ◆ neue Instanzvariablen einführen
  - ◆ neue Methoden einführen
  - ◆ geerbte Instanzvariablen verdecken (vorsicht!)
  - ◆ geerbte Methoden überschreiben

## 4 Das Ersetzungsprinzip

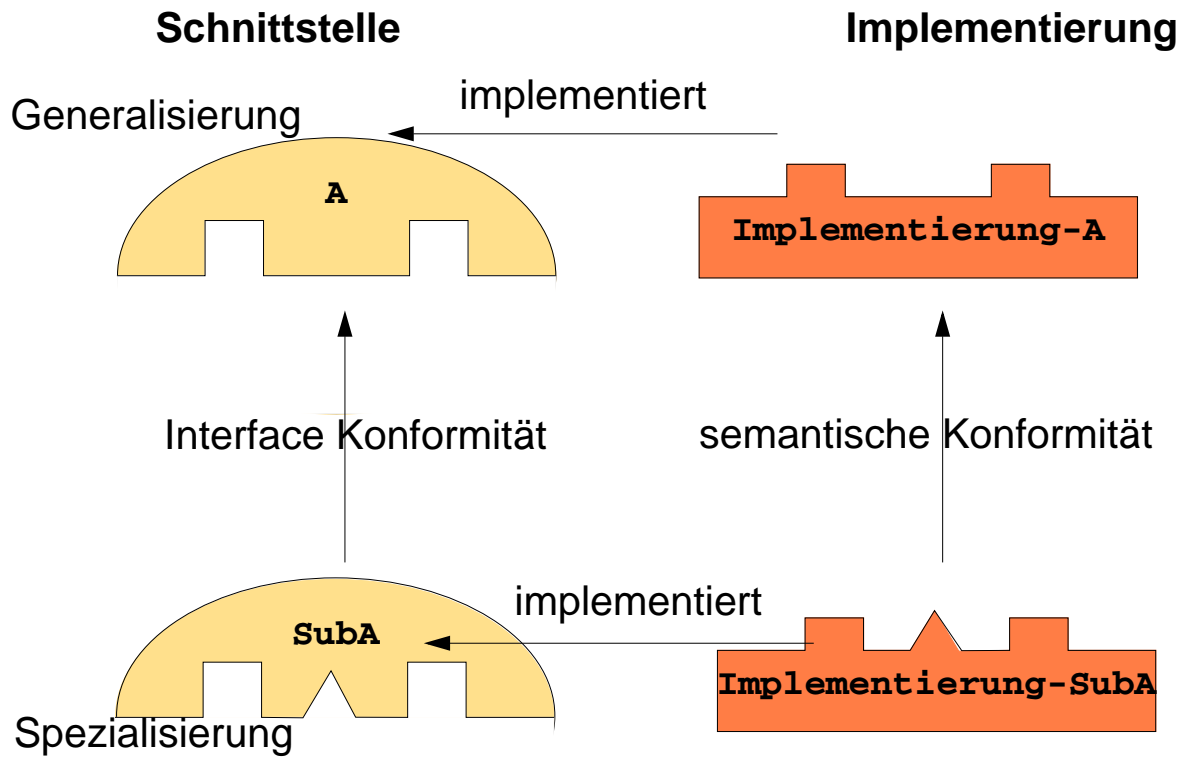
- Wenn ein Objekt der Oberklasse erwartet wird, kann immer auch ein Objekt einer Unterklasse verwendet werden.
  - ◆ wichtigster Typ des Polymorphismus
- Vererbung ist Spezialisierung ("ist ein" Relation)
- alles was für die Generalisierung gilt muss auch auf die Spezialisierung zutreffen.
  - ◆ Die Spezialisierung muss alle Anforderungen der Generalisierung erfüllen.

→ **Abstraktion**

## 5 Vererbung ist Spezialisierung



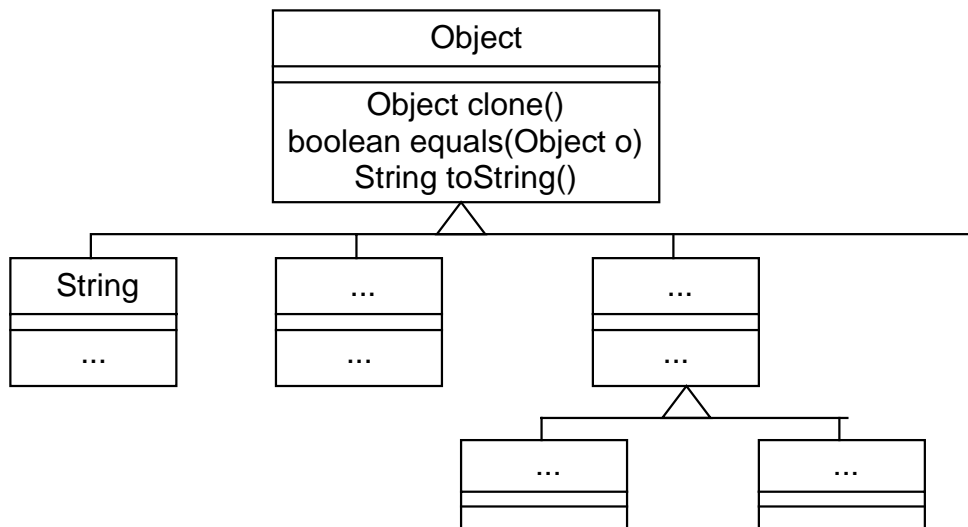
## Z.3 Vererbung ist Spezialisierung



MW - Übung

## 6 Vererbung in Java

- Klassen mit einfacher Vererbung
- Baum Hierarchie mit der Klasse `Object` als Basisklasse aller anderen Klassen



- primitive Typen (`int`, `float`, ...) sind außerhalb des Klassenbaumes

MW - Übung

## 7 Die Klasse Object

- Die Klasse `Object`: Basisklasse aller anderen Klassen

```
class Person extends Object {... }
```

- ◆ `extends Object` kann wegfallen

```
class Person {... }
```

- Stellt Basisfunktionalität zur Verfügung, zum Beispiel:

- ◆ `boolean equals(Object o)` // Test auf Gleichheit
  - Standardimplementierung vergleicht die Referenzen
  - Jede Klasse sollte eine eigene Implementierung bereitstellen
- ◆ `String toString()` // Stringdarstellung eines Objekts
  - Standardimplementierung: Klassenname und Objekt ID
  - Jede Klasse sollte eine eigene Implementierung bereitstellen

## 8 Überschreiben

- Unterklassen können für geerbte Methoden eine neue Implementierung bereitstellen.
- Die neue Implementierung *überschreibt* die geerbte Implementierung:

```
class Person {
    String name;
    ...
    void print() {
        System.out.println("Person: " + name);
    }
}
class Customer extends Person {
    int number;
    ...
    void print() {
        System.out.println("Person: " + name);
        System.out.println("Customer number: " + number);
    }
}
```



## 8 Überschreiben (2)

- Die Implementierung der Oberklasse kann mit `super.method()` aufgerufen werden.

- Beispiel:

```
class Customer extends Person {
    int number;
    ...
    void print() {
        super.print();
        System.out.println("Customer number: " + number);
    }
}
```

## 9 Überladen vs. Überschreiben

- Um eine Methode zu überschreiben müssen die Typen der Parameter und des Rückwerts exakt übereinstimmen, ansonsten wird die Methode überladen.
- Um dem Ersetzungsprinzip gerecht zu werden würde es ausreichen, wenn:
  - ◆ die Typen der Parameter von einer Oberklasse der ursprünglichen Parameter sind
  - ◆ der Typ des Rückgabewertes von einer abgeleiteten Klasse des ursprünglichen Rückgabetyps ist.
- Java unterstützt das jedoch nicht!!!

## 9 Überladen vs. Überschreiben (2)

- häufiger Fehler:

```
class Object {
    boolean equals(Object o) { ... }
    ...
}

class Customer {
    int number;
    boolean equals(Customer c) { return number == c.number; }
    ...
}
```

*equals von Object wird nicht überschrieben*

## 10 Dynamisches Binden

- Die Methoden werden erst bei einem Aufruf gebunden.
- *dynamischer Typ*: Typ/Klasse des referenzieren Objekts  
(die Klasse, die bei **new** verwendet wurde)
- *statischer Typ*: Typ der Referenz
- Durch den statischen Typ wird festgelegt, welche Methoden aufgerufen werden können.
- Der dynamische Typ legt fest, welche Methode verwendet wird:

```
Customer c = new Customer("Max", 1234);
Person p = c; // dynamischer Typ von p ist Customer,
              // statischer Typ ist Person

c.print();
p.print(); // obwohl die Referenz p den Typ Person hat, wird
           // die print() Methode von Customer verwendet
```

## 11 Sichtbarkeit und Vererbung

- Die Sichtbarkeit von Methoden darf in Unterklassen nicht eingeschränkt werden (Ersetzbarkeit!):.

```
class Person {
    public String getName() { ... }
}
class Customer extends Person {
    private String getName() { ... }
}
Error
```

## 12 Konstruktoren und Vererbung

- Konstruktoren werden **nicht** vererbt
- Aufrufen eines Konstruktors der Oberklasse mittels **super (...)**
- super (...)** muss die erste Anweisung in einem Konstruktor sein
- Falls die erste Anweisung nicht **super (...)** ist, so fügt der Compiler automatisch eine **super ()** Anweisung ein.
- Standardkonstruktor:
  - wird vom Compiler erzeugt, falls *kein* Konstruktor definiert wird
  - enthält eine **super ()** Anweisung

## 12 Konstruktoren und Vererbung (2)

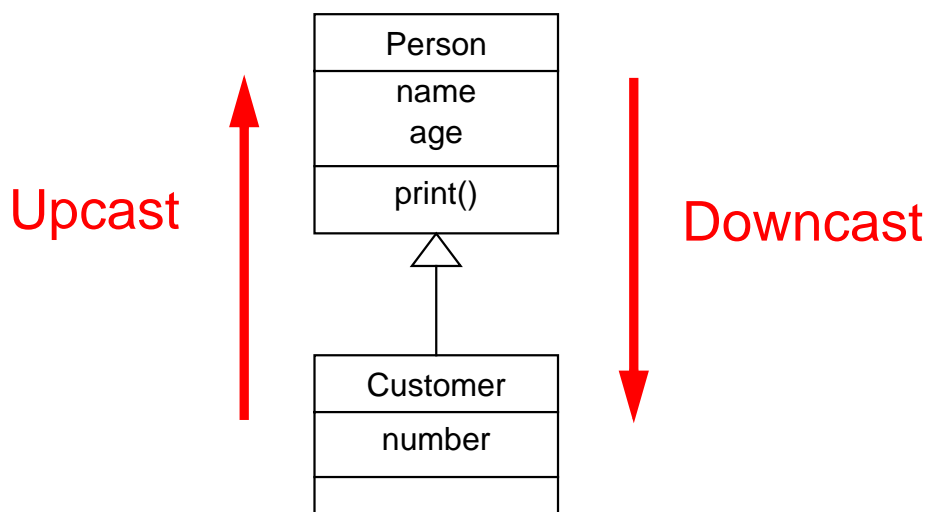
- Beispiel:

```
class Customer extends Person {
    int number;
    Customer(String name, int number) {
        super(name);
        this.number = number;
    }
    ...
}
```

## 13 Typenkonvertierung: Upcast

- Typenkonvertierung von einer Unterklassen zur Oberklasse (*upcast*) erfolgt automatisch:

```
Person p = new Customer(...);
```



## 13 Typenkonvertierung: Downcast

- Typenkonvertierung von der Oberklasse zu einer Unterklasse (*downcast*) explizit mittels Cast-Operator:

```
Customer c = new Customer(...)
Person p = c; // implizite Typenkonvertierung
Customer c2 = (Customer) p; // explizite Typenkonvertierung
```

- Wenn das Objekt und die Variable nicht typkonform sind, wird eine `ClassCastException` generiert:

```
Customer c = new Customer(...)
Person p = c; // implizite Typenkonvertierung
Employee e = (Employee) p;
// erzeugt eine ClassCastException zur Laufzeit
```

## 14 Typ Ermittlung

- `instanceof` Operator:

```
Customer c = new Customer(...)
Person person = c; // upcast
if (person instanceof Employee) {
    Employee employee = (Employee) person;
    ...
} else if (person instanceof Customer) {
    Customer customer = (Customer) person;
    ...
}
```

- `instanceof` mit der Oberklasse des dynamischen Typs ist ebenfalls `true`:

```
person instanceof Person
```

## 15 Die Klasse Class

- Die Klasse `class`: Die Klasse aller Klassen.
- Ein `class` Objekt repräsentiert eine Klasse oder ein Interface.
- Mit Hilfe eines `class` Objekts können neue Instanzen erzeugt werden:

```
Customer c = new Customer();
...
Class aClass = c.getClass();
System.out.println("Class of c is:"+aClass);

Object o = aClass.newInstance(); // ein neues Customer Objekt

Person p = (Person) o;
```

- Ein `class` Objekt kann aus dem Namen einer Klasse generiert werden:

```
Class aClass = Class.forName("Employee");
```

## Z.4 Abstrakte Klassen

- Szenario:
  - ◆ Zeichen mit geometrischen Formen (Kreis, Rechteck, Linie)
- Problem:
  - ◆ Zeichenbereich `Sheet` kann nur mit `Shape` Objekten umgehen
  - ◆ `Sheet` muss `draw()` am `Shape` aufrufen
  - ◆ `Shape` kann keine sinnvolle Implementierung von `draw()` bereitstellen
- `Shape` ist eine *abstrakte* Klasse

## Z.4 Abstrakte Klassen

- ...werden verwendet um gemeinsame Eigenschaften von Klassen herauszuarbeiten und zusammenzufassen.
- ...können **nicht** zur Erzeugung von Objekten verwendet werden.
- ...enthalten Methoden ohne Implementierung (*abstrakte Methoden*)
- Abstrakte Klassen und Methoden werden mit dem Schlüsselwort **abstract** deklariert:

```
abstract class Shape {
    public abstract void draw();
}
```

- Wenn eine konkrete Klasse von einer abstrakten Klasse abgeleitet wird, so müssen alle abstrakten Methoden implementiert sein:

```
class Circle extends Shape {
    public void draw() { ... }
}
```

## Z.5 Interfaces

- Java trennt das Klassenkonzept vom Typkonzept
- Interfaces repräsentieren Typen
- Interfaces enthalten
  - ◆ Methodennamen und -signaturen
  - ◆ Konstanten (**static final**)
- alle Methoden sind (implizit) abstrakt
- Klassen können zu Interfaces kompatibel sein (Schlüsselwort **implements**)
- Diese Klassen müssen alle Methoden des Interfaces implementieren.
- Definition einer Schnittstelle mit dem Schlüsselwort **interface**

## 1 Beispiel

1. Definition einer Schnittstelle:

```
public interface Printable {
    public void print();
}
```

2. Definition einer Klasse, welche die Schnittstelle implementiert:

```
public class Account implements Printable {
    ...
    public void print() {
        System.out.println("balance="+balance());
    }
}
public class Person implements Printable { ... }
```

## 1 Beispiel (2)

3. Verwendung der Interfaces:

```
public class PrintQueue {
    public void add(Printable p ) { ... }
}
....
PrintQueue queue = new PrintQueue();
Printable p=new Person(...);
queue.add(p);
Account account = new Account(...);
queue.add(account);
```



## 2 Vererbung von Interfaces

- Interfaces können mehrere andere Interfaces *erben*, Klassen können mehrere Interfaces *implementieren*.
- Beispiel:

```
interface Streamable extends FileIO, Printable {
    // zusätzliche Methoden
}

class Test implements Streamable, TestInterface {
    ...
}

class Test1 extends Test {
    ...
}
// Test1 ist zu FileIO, Printable, Streamable,
// TestInterface und zur Klasse Test kompatibel
```

## 3 Abstrakte Klassen vs. Interfaces

- Abstrakte Klassen können eine partielle Implementierung einer Abstraktion bereitstellen.
- Abstrakte Klassen können Instanzvariablen enthalten.
- Ein abstrakte Klasse sollte verwendet werden, wenn nur ein Teil der Implementierung offen bleiben soll.
- Ein Interface ist gut geeignet um bestimmte Eigenschaften zu repräsentieren. (Printable, Clonable,...)

## Z.6 Packages

- Klassen lassen sich in Pakete (packages) zusammenfassen.
- Paket = Programm-Modul
  - ◆ mit eindeutigen Namen (z.B. `java.lang` oder `java.awt.image`)
  - ◆ enthält eine oder mehreren Klassen
- Pakete partitionieren den Namensraum.

### 1 Schlüsselwort package

- Packages werden mit **package** deklariert:

```
package test;
public class TestClass ...
```

- **package** muss die erste Anweisung in einer Datei sein.
- Hierarchien von Packages sind möglich:

```
package test.unittest1;
```

## 2 Schlüsselwort: import

- Klassen anderer Packages können mit **import** verwendet werden:

```
import java.util.*; // use all classes from package java.util
import java.io.File; // use class File from package java.io
```

- Bei der Suche von Klassen wird der Package-Name als Verzeichnis verwendet.
- Beispiel:
  - ◆ Package **bank** mit Klasse **Customer**
  - ◆ kann verwendet werden mit **import bank.Customer**
  - ◆ Bytecode-Datei wird gesucht als **bank/Customer.class**
- Package **java.lang.\*** wird automatisch importiert.
- Zugriff auf Klassen ohne import: durch Verwendung des vollständigen Klassennamens, inklusive Package.

## 2 Schlüsselwort: import - Beispiel

Datei  
editor/shapes/X.java

```
package editor.shapes;
public class X {
    public void test() {
        System.out.println("X");
    }
}
```

Datei  
editor/filters/Y.java

```
package editor.filters;
import editor.shapes.*;
class Y {
    X x;
    void test1() { x.test();}
}
```

Datei  
editor/filters/Z.java

```
package editor.filters;
class Z {
    editor.shapes.X x;
    void test1() { x.test();}
}
```

### 3 Packages und der CLASSPATH

■ Klassen werden mit Hilfe der Umgebungsvariable CLASSPATH gesucht.

■ Beispiel:

- ◆ Package **bank** mit Klasse **Customer**
- ◆ wird verwendet mit **import bank.Customer;**
- ◆ Compiler und Interpreter suchen die Bytecode-Datei:  
**bank/Customer.class**
- ◆ CLASSPATH enthält **/proj/test:/tmp**
- ◆ gesucht wird nach:
  - **/proj/test/bank/Customer.class**
  - **/tmp/bank/Customer.class**
- ◆ beim kompilieren kann man das Zielverzeichnis angeben:
  - **javac -d /proj/test Customer.java**

### 4 Standard Java Pakete

- **java.lang**: fundamentale Java Klassen (Thread, String,...)
- **java.io**: Ein- / Ausgabe Unterstützung (Files, Streams,...)
- **java.net**: Netzwerk Unterstützung (Sockets, URLs, ...)
- **java.awt**: GUI Unterstützung (Abstract Windowing Toolkit)
- **java.applet**: Applet Unterstützung
- **java.util**: Hilfsklassen (Random) und Datenstrukturen (Vector, Stack)
- **java.rmi**: Remote Method Invocation
- **java.security**: kryptographische Unterstützung
- Nähere Informationen in der API Documentation:  
**[http://www4/Services/Doc/Java/jdk-1.4/docs/api/index.html](http://www4.Services/Doc/Java/jdk-1.4/docs/api/index.html)**

## Z.7 Sichtbarkeitsattribute

- *Kapselung* ist eines der Grundprinzipien objektorientierter Programmierung.
- Kapselung wird zum verstecken unnötiger Information verwendet (*information hiding*).

### 1 Sichtbarkeitsattribute - Klassen

- Eine Klasse kann öffentlich (**public**) oder nicht öffentlich sein.

```
public class X { ... }

class X { ... }
```

- **public** Klassen sind außerhalb des Package verfügbar.
- Klassen ohne **public**-Deklaration (private Klassen) sind nur innerhalb desselben Package sichtbar.
- Eine public-Klasse muss in einer eigenen Datei deklariert werden.  
Dateiname := Klassenname + ".java"  
Beispiel: Klasse **x** muss in der Datei **x.java** definiert werden.

## 2 Sichtbarkeitsattribute - Klassenelemente

- Sichtbarkeitsattribute für Methoden und Variablen:
  - ◆ `public`, `default`, `protected`, `private`
- Wirkung:
  - ◆ **public**: global sichtbar
  - ◆ **default**: innerhalb des gleichen Packages sichtbar
  - ◆ **protected**: innerhalb des gleichen Packages und in Unterklassen sichtbar.
  - ◆ **private**: nur innerhalb der gleichen Klasse sichtbar
- Sichtbarkeitsattribute müssen bei *jeder* Methode bzw. Instanzvariable extra angegeben werden.

## 2 Sichtbarkeitsattribute - Klassenelemente (2)

- Übersicht:

	Sichtbarkeitsattribute			
sichtbar in	<b>public</b>	<b>protected</b>	<b>default</b>	<b>private</b>
gleiche Klasse	ja	ja	ja	ja
gleiches Package	ja	ja	ja	nein
Unterklassen	ja	ja	nein	nein
andere Packages	ja	nein	nein	nein

## 3 Kapselung

- ohne Kapselung:

```
class Person {
    public String name; // "name" can be modified/read globally
}
```

- besser:

```
class Person {
    private String name; // only Person can access "name"
    public String getName() { // access method
        return name;
    }
}
```

## Z.8 statische Elemente

### 1 Klassenvariablen und Klassenmethoden

- Klassen können Variablen und Methoden enthalten:  
statische Variablen und Methoden
- Diese können ohne Objekt der Klasse genutzt werden:

```
class Test {
    private static int counter = 0;
    public Test() { counter++; }
    public static int howMany() { return counter; }
}
Test t = new Test();
System.out.println("No. of Test objects: " + Test.howMany());
```

- weiteres Beispiel:
  - ◆ **System.out** ist eine statische Variable der Klasse **System**.

## 2 Klassenkonstruktor

- Klassen haben einen Zustand (**static** Variablen) und müssen deshalb durch einen Klassenkonstruktor initialisiert werden.

- Beispiel:

```
class Test {
    static int counter;
    static {
        counter = 9;
    }
}
```

## Z.9 Konstanten

- Variablen können konstant (*final*) sein:
  - ◆ Sie müssen entweder bei der Deklaration (JDK 1.0) oder später (JDK 1.1: *blank finals*) initialisiert werden.
  - ◆ Dieser initiale Wert kann nicht verändert werden.

- Beispiel:

```
class Test {
    public static final int x=5; // konstante Klassenvariable
    private final int t=10;     // konstante Instanzvariable
}
```



## Z.9 Konstanten (2)

- Seit Java 1.1 können die Parameter von Methoden und lokale Variablen konstant sein.

- Beispiel:

```
class Test {
    String name;
    void setName(final String name) {
        final int i = 42;
        this.name = name;
    }
}
```

## 1 Final Methoden

- Wenn Methoden als **final** deklariert werden, können sie nicht überschrieben oder verdeckt werden
  - ◆ Sicherheit
  - ◆ Effizienz (inlining ist möglich)

```
class Test {
    final void hello() {...}
}

class Test2 extends Test {
    void hello() { ... } // Fehler!! hello ist final in Test
}
```

## 2 Final Klassen

- Wenn Klassen als **final** deklariert werden, kann man keine Klassen davon ableiten.

Beispiel:

```
public final class Test {
    ...
}

public class Test2 extends Test { //Fehler!! Test ist final
    ...
}
```

## Z.10 Innere Klassen

- **local inner class**: nur von der umschließenden Klasse nutzbar
- **inner class with method scope**: nur innerhalb der Methode nutzbar
- **anonymous inner class**: nur bei der Definition nutzbar
- **static inner class**: global nutzbar

# 1 Lokale Innere Klassen

- Innere Klassen können auf Instanzvariablen der umschließenden Klasse zugreifen:

```
class Test {
    private String array[] = { "hans", "otto", "max"};
    class Inner {
        String method(int i) { return "Name:+" + array[i]; }
    }
}
```

- Sichtbarkeitsattribute wie für Methoden und Instanzvariablen (**private**, **default**, **protected**, **public**)

```
class Test {
    private String array[] = { "hans", "otto", "max"};
    private class MyEnum implements Enumeration {
        private int counter = 0;
        public Object nextElement() { return array[counter++]; }
        public boolean hasMoreElements() { return counter < array.length; }
    }
    public Enumeration enumerate() { return new MyEnum(); }
}
```

# 2 Innere Klassen innerhalb einer Methode

- Klasse, die nur innerhalb einer Methode gebraucht wird:

```
class Test {
    private String array[] = { "hans", "otto", "max"};
    public Enumeration enumerate() {
        class MyEnum implements Enumeration {
            private int counter = 0;
            public Object nextElement() { return array[counter++]; }
            public boolean hasMoreElements() { return counter < array.length; }
        }
        return new MyEnum();
    }
}
```

## 2 Innere Klassen innerhalb einer Methode (2)

- Kann auf konstante (**final**) Parameter und konstante (**final**) lokale Variablen der umschließenden Methode zugreifen
- Zugriff auf **this** der umschließenden Klasse **x** mit **x.this**:

```
class Test {
    public void test(final String msg) {
        class Inner {
            public void output(String hello) { System.out.println(hello+msg); }
        }
        ...
    }
}
```

## 3 Anonyme Klassen

- Der Klassenname **MyEnum** im vorherigen Beispiel enthält keinerlei Information → Einsatz einer anonymen inneren Klasse:

```
class Test {
    private String array[] = { "hans", "otto", "max"};

    Enumeration enumerate() {
        return new Enumeration() {
            int counter = 0;
            public boolean hasMoreElements() {
                return counter < array.length; }
            public Object nextElement() {
                return array[counter++]; }
        };
    }
}
```

*Ende des return Statements*

## 4 Statische Innere Klassen

- Statische innere Klassen können nur auf statische Variablen und Methoden der umschliessenden Klasse zugreifen:

```
class Test {
    private static String array[] = { "hans", "otto", "max" };

    static class E implements Enumeration() {
        int count = 0;
        public boolean hasMoreElements() {
            return count < array.length; }
        public Object nextElement() {
            return array[count++]; }
    }
}
...
Enumeration e = new Test.E();
System.out.println(e.nextElement());
```