

Ist es nicht offensichtlich,
dass die Gurkenscheiben
OO repräsentieren?



Objektorientierung

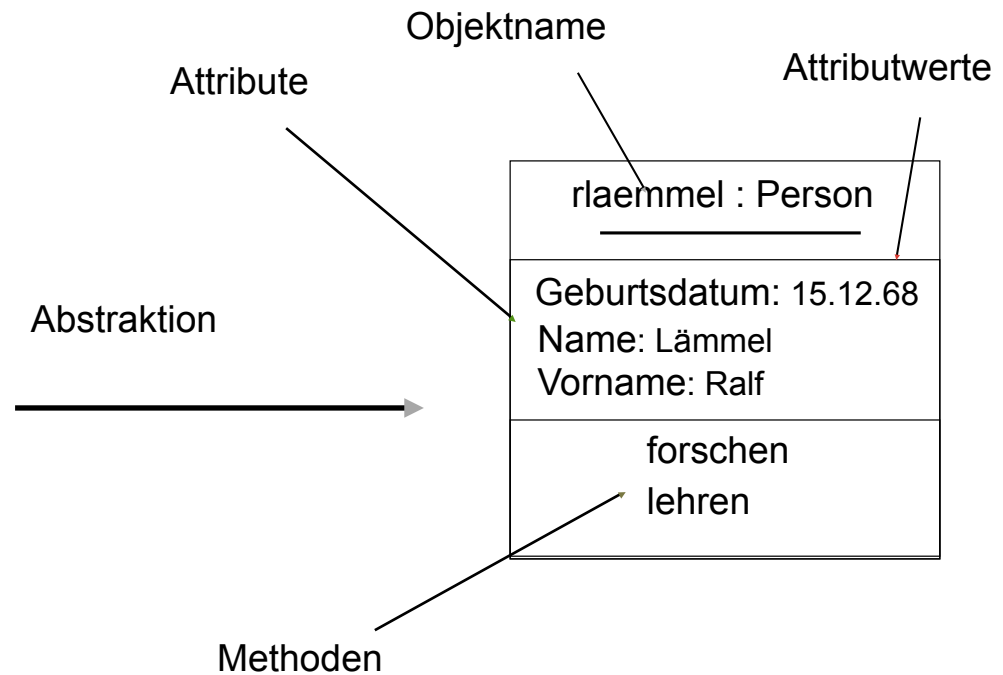
OOPM, Ralf Lämmel

7 neue bzw. ausgeweitete Begriffe

1. **Objekt** (mehr als die Instanz eines Datentyps)
2. **Klasse** (mehr als die Implementation eines Datentyps)
3. **Nachricht** (mehr als ein Methodenaufruf)
4. **Vererbung (Generalisierung)** (nicht nur für Schnittstellen)
5. Abstrakte Klassen (nicht instanziierbare Klassen)
6. Polymorphismus (weniger speziell als bisher)
7. Dynamische Bindung (weniger speziell als bisher)

Objekte und Nachrichten

Objekte als **Abstraktion**



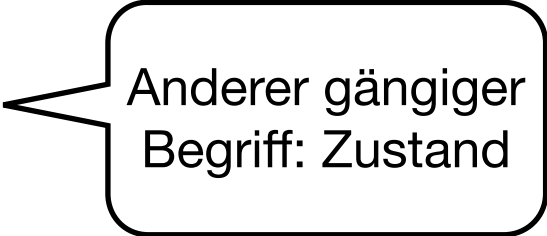
Was ist ein Objekt?

- * Abstraktionen von *physischen* Dingen
 - Personen, Studenten, Immobilien
- * Abstraktionen von *ideellen (virtuellen)* Dingen
 - Konten, Bestellung
- * Modellierung von rechnerbasierten Konzepten
 - Druckaufträge, Dialog

Beachte: Objekte nicht nur relevant in der Programmierung, sondern auch bei der Analyse und dem Entwurf. Zum Beispiel ist auch der Benutzer eines Informationssystems ein Objekt in der Analysephase.

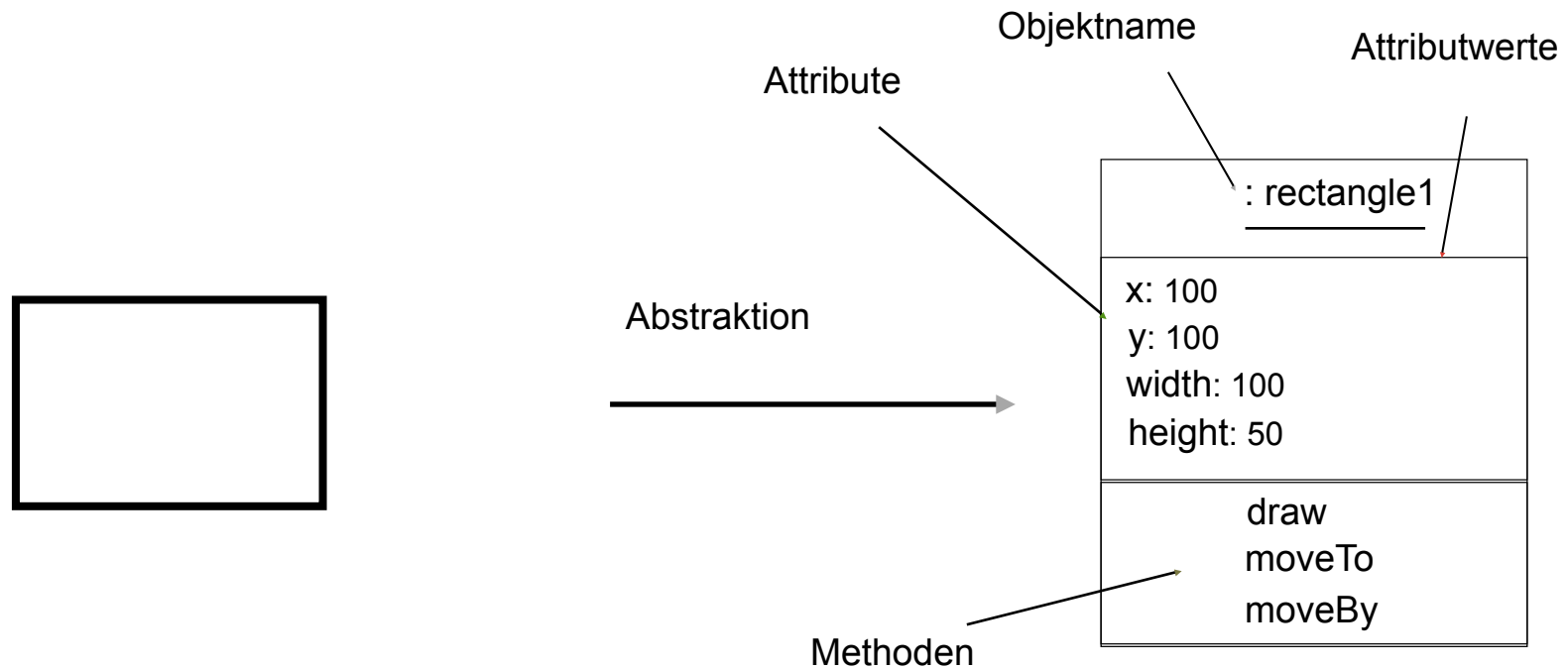
Definierende Eigenschaften eines Objektes

- * **Struktur** zusammengesetzt aus:
 - Attributen (primitiver Typen; typischerweise)
 - Beziehungen mit anderen Objekten
- * **Verhalten** beschrieben durch eine Menge von Operationen
- * **Identität** zur Unterscheidung von Objekten



Anderer gängiger
Begriff: Zustand

Objekte als Abstraktion



Objektidentität

- * Jedes Objekt besitzt einen eindeutigen Objektbezeichner (OID).
- * OID wird bei "Geburt" des Objekts vergeben.
- * OID ist unveränderlich.
- * OID ist unabhängig vom aktuellen Objektzustand.

Verschiedene
Objekte mit gleichen
Eigenschaften



Name = „Jackson“
Vorname = „Michael“



Name = „Jackson“
Vorname = „Michael“



Name = „Jackson“
Vorname = „Michael“

Objektidentität

- * Jedes Objekt besitzt einen eindeutigen Objektbezeichner (OID).
- * OID wird bei "Geburt" des Objekts vergeben.
- * OID ist unveränderlich.
- * OID ist unabhängig vom aktuellen Objektzustand.

Verschiedene
Objekte mit gleichen
Eigenschaften



Name = „Jackson“
Vorname = „Michael“



Name = „Jackson“
Vorname = „Michael“



Name = „Jackson“
Vorname = „Michael“

Nachricht

* Jedes Objekt versteht bestimmte Nachrichten.

■ Z.B.: ein Konto versteht die Nachricht für eine Abhebung.

* Bestandteile

■ Explizit

- Empfänger (ein Objekt)
- Nachrichtenname (Methodenname)
- Parameter (Objekte)

■ Implizit

- Absender (ein Objekt)

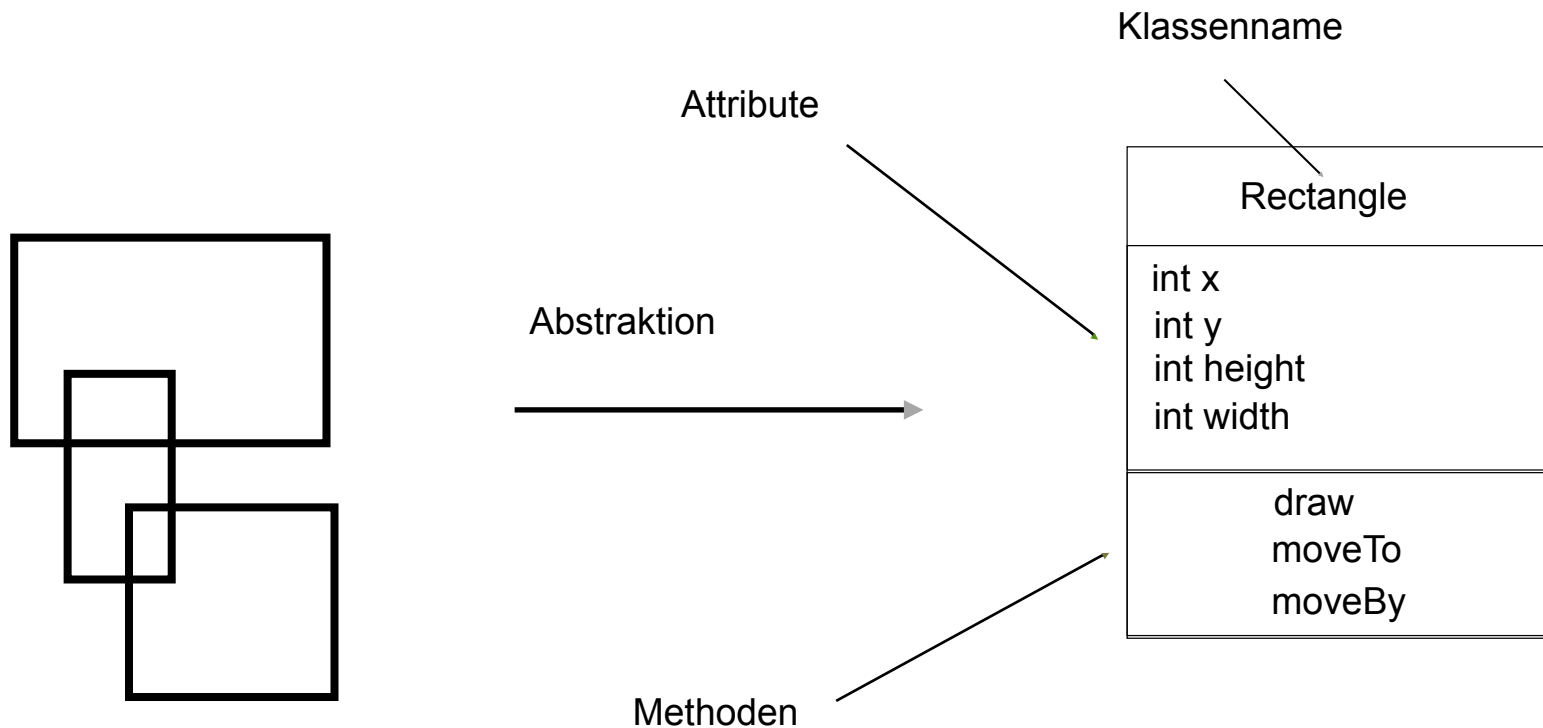
```
public class Transfer {  
    public Transfer(  
        Account from,  
        Account to,  
        float amount) { ... }  
    public execute() {  
        from.withdraw(amount);  
        to.deposit(amount);  
    }  
}
```

Potential für Nachrichtenaustausch

- * Ein Objekt kann Nachrichten an folgende Objekte senden:
 - an jedes “verknüpfte” Objekt.
 - an jedes Objekt aus den Parametern.
- * Abstrakt betrachten wir Verknüpfungen zwischen Objekten:
 - *Ein Konto ist mit dem Kontoinhaber verbunden.*
 - *Ein Kontoinhaber ist mit seinen Konten verbunden.*

Klassen und Vererbung

Vom Objekt zur Klasse



Ein beliebtes Beispiel

* Formen (“shapes”)

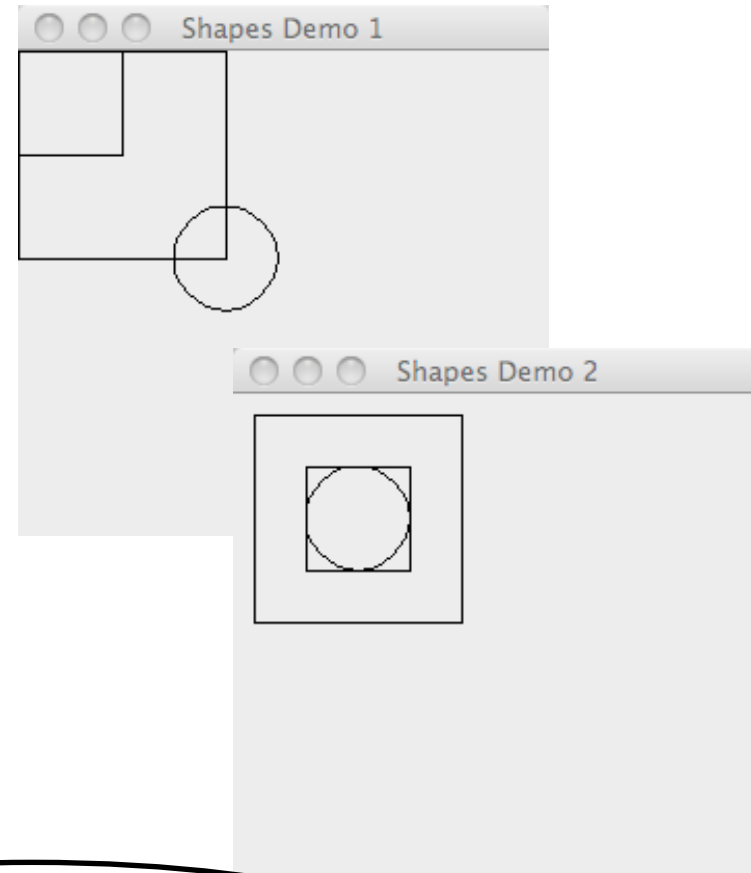
■ Rechtecke

■ Kreise

* Verhalten

■ Bewegen

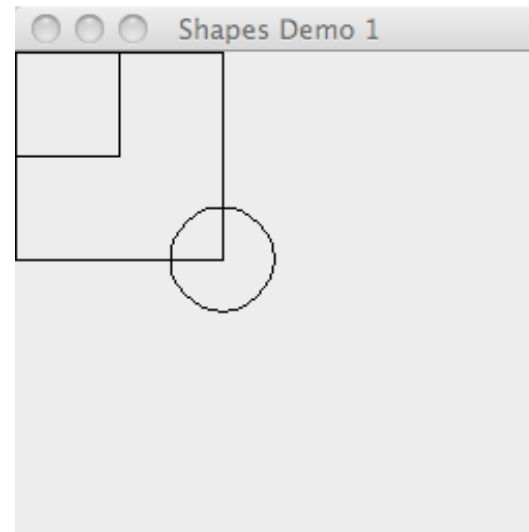
■ Zeichnen



Demo
[Repository: oo.shapes.awtish]

Nachrichtenaustausch im Computer-Aided Design (Bild-Aufbau)

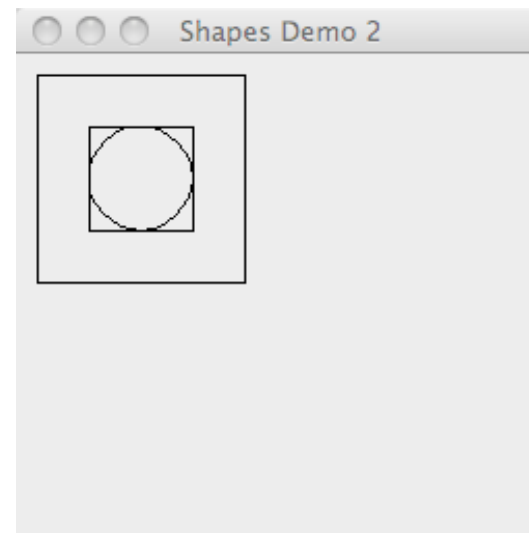
- * Instanziiere Abbildung a
- * An a : Konstruiere Rechteck $r1$ mit Abmessungen ...
- * An a : Konstruiere Rechteck $r2$ mit Abmessungen ...
- * An a : Konstruiere Kreis k mit Abmessungen ...
- * An a : Male Dich!
 - An $r1$: Male Dich!
 - An $r2$: Male Dich!
 - An k : Male Dich!



Nachrichtenaustausch im Computer-Aided Design (**Bild-*Mutation***)

- * An $r1$: Bewege Dich etwas vom Rand weg ...
- * An $r2$: Bewege Dich in die Mitte von $r1$...
- * An k : Bewege Dich in die Mitte von $r2$...

- * An a : Male Dich!
 - An $r1$: Male Dich!
 - An $r2$: Male Dich!
 - An k : Male Dich!



Instanz eines Entwurfsmusters: Eine Abbildung ist ein **Kompositum** und das Malen ist die Operation davon.

Klassen als Stempel von Objekten

- * Eine Klasse beschreibt eine Sammlung von Objekten mit gleichen Eigenschaften (Attributen), gemeinsamer Funktionalität (Methoden), gemeinsamen Beziehungen zu anderen Objekten und gemeinsamer Semantik.

- * Jedes Objekt ist Instanz genau einer Klasse.
- * Klassenzugehörigkeit ändert sich niemals.



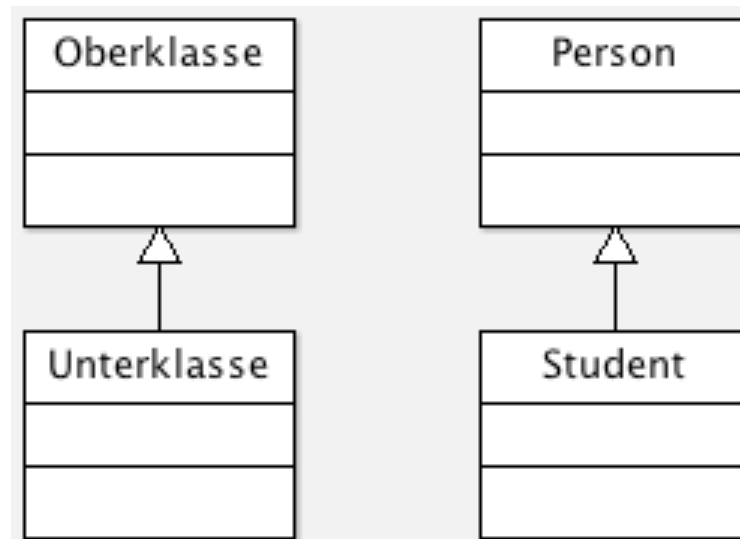
In Java ist das so!

X: 0	Y: 0
width: 50	
height: 50	

X: 50	Y: 50
width: 100	
height: 25	

Vererbung (Generalisierung)

- * Klassen werden in einer Hierarchie angeordnet.
- * Unterklasse erbt von Oberklasse
 - Attribute
 - Methoden
 - (Beziehungen)



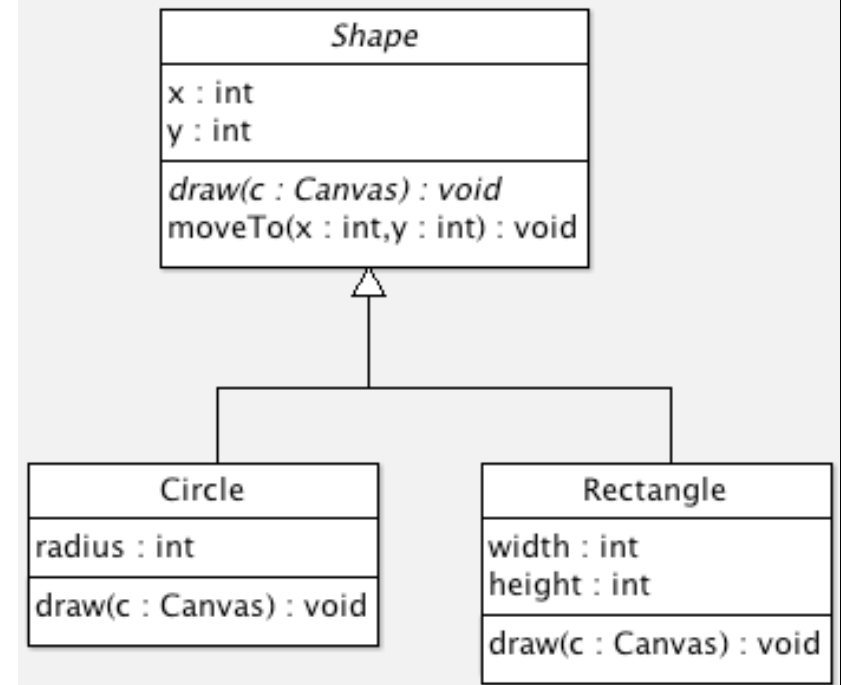
Abstrakte vs. konkrete Klassen

* Abstrakte Klasse

- nicht-instanziierbare Klasse
- Bereitstellung einer Schnittstelle
- Wurzel einer Fallunterscheidung für Vererbung
- Bereitstellung einer (unvollständigen) Implementation

* Konkrete Klasse

- instanziierebare Klasse
- weitere Spezialisierung (Vererbung) ist möglich.



Code zum Shapes-Beispiel

```
package oo.shapes.simple;
```

```
import java.io.PrintStream;
```

```
public abstract class Shape {
```

```
    private int x;  
    private int y;
```

```
    protected Shape(int x, int y) { moveTo(x, y); }
```

```
    public int getX() { return x; }
```

```
    public int getY() { return y; }
```

```
    public void setX(int x) { this.x = x; }
```

```
    public void setY(int y) { this.y = y; }
```

```
    public void moveTo(int x, int y) { setX(x); setY(y); }
```

```
    public void moveBy(int deltax, int deltay) {  
        moveTo(getX() + deltax, getY() + deltay);  
    }
```

```
    public abstract void draw(PrintStream s);
```

```
}
```

Die abstrakte Basisklasse für alle Formen

Privater Zustand für Position der Form

Konstruktor für Aufruf in den Unterklassen

Reguläre Getter und Setter

Die Methoden zum Bewegen können bereits implementiert

Das Malen muss von den Unterklassen implementiert werden.

```
package oo.shapes.simple;
```

```
import java.io.PrintStream;
```

```
class Circle extends Shape {
```

```
    private int radius;
```

```
    public Circle(int x, int y, int radius) {  
        super(x, y);  
        setRadius(radius);  
    }
```

```
    public int getRadius() { return radius; }
```

```
    public void setRadius(int radius) { this.radius = radius; }
```

```
    public void draw(PrintStream s) {  
        s.println(  
            "Drawing a Circle at:( "  
            + getX() + ", " + getY()  
            + "), radius " + getRadius());  
    }
```

```
}
```

Eine abgeleitete Klasse für Kreise

Zusätzlicher Zustand für Kreise

Ein echter Konstruktor

Aufruf vom Shape-Konstruktor

Reguläre Getter und Setter

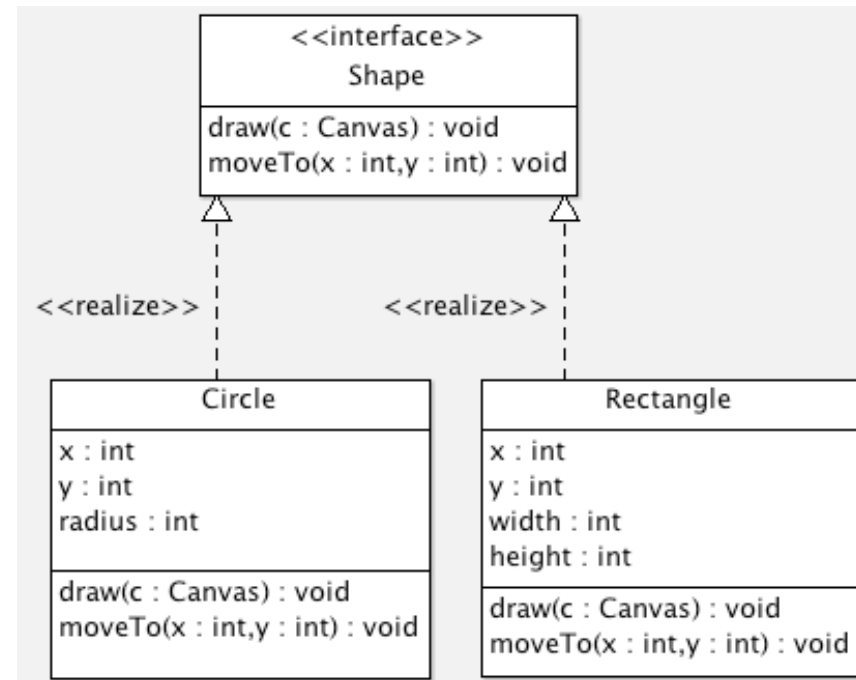
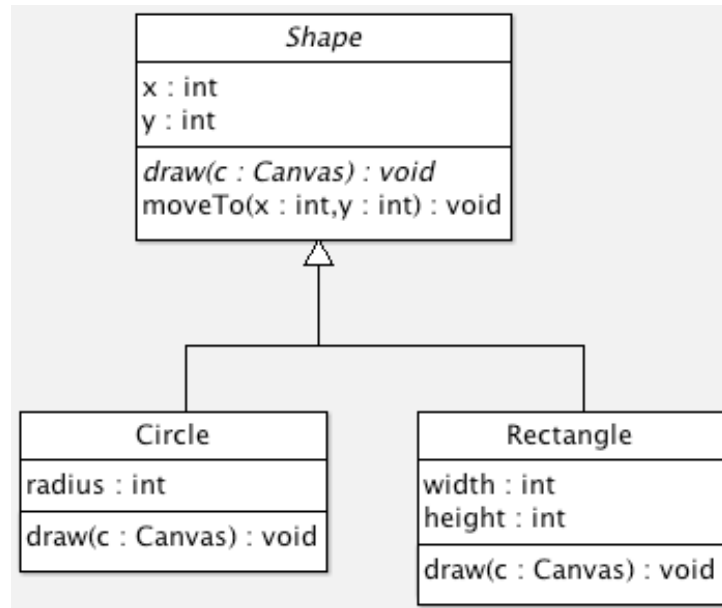
Wir "malen"
einen Kreis.

Drawing a Circle at:(15, 25), radius 8

```
class Rectangle extends Shape ...
```

**Extra Übungsaufgabe für Sie:
Versuchen Sie erst diese Klasse zu sketchen.
Sie können Ihr Ergebnis mit dem im Repository vergleichen.**

Abstrakte Klassen vs. Schnittstellen



* Wann nimmt man abstrakte Klassen?

* Wann nimmt man Schnittstellen?

Abstrakte Klassen vs. Schnittstellen

```
package oo.shapes.iop;

import java.io.PrintStream;

public interface Shape {
    void moveTo(int x, int y);
    void moveBy(int deltax, int deltay);
    void draw(PrintStream s);
}
```

Schnittstellen in Java können auch Attribute (Felder) beinhalten. Diese sind aber public, static und **final**.

Wir können mit dieser Schnittstelle (im Gegensatz zu der ursprünglichen, abstrakten Klasse) nicht den Zustand für die Position der Formen mit Attributen modellieren. Darum können wir auch nicht die bisherigen Implementationen für `moveTo` und `moveBy` angeben.

Abstrakte Klassen vs. Schnittstellen

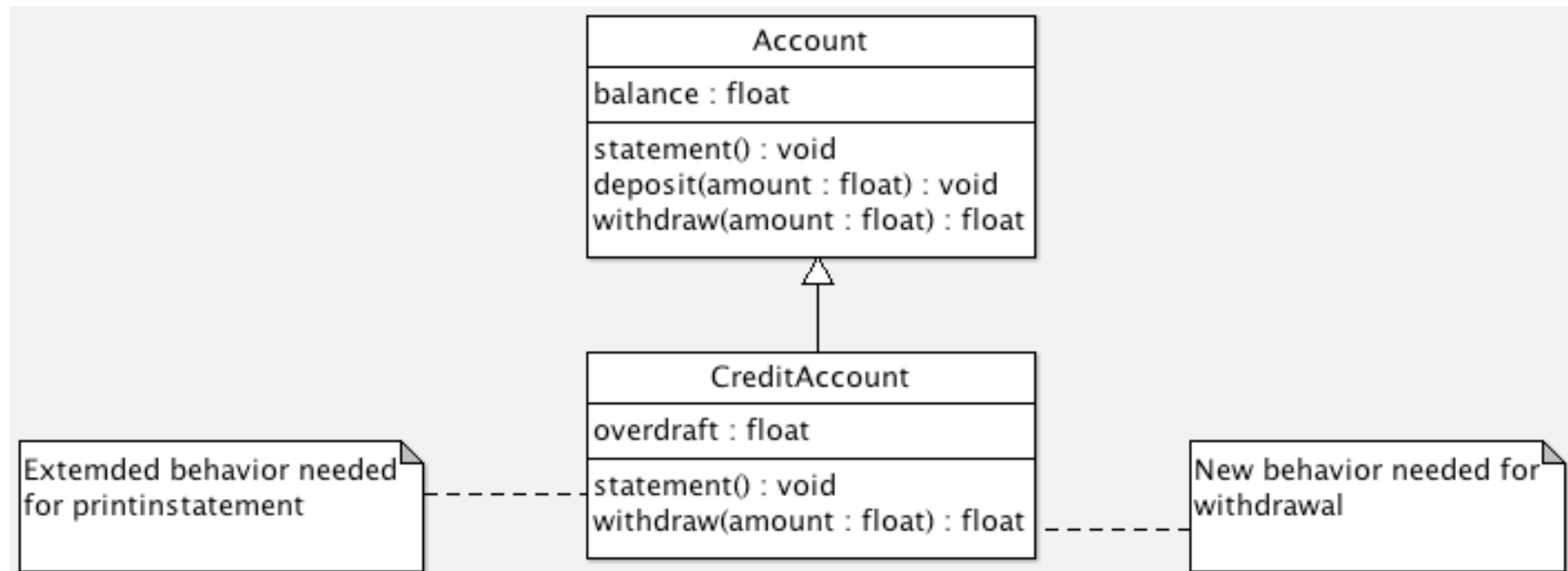
* Wann nimmt man abstrakte Klassen?

- Wurzel einer Fallunterscheidung in der Klassenhierarchie
- Faktorisierung von Attributen und Methodenimplementation
- Ermöglichung des Überschreibens von Methodenimplementationen

* Wann nimmt man Schnittstellen?

- Definition allein einer gemeinsamen Schnittstelle
- Ermöglichung von implementierenden Klassen ohne feste Oberklasse

Vererbung mit/ohne Überschreiben



Basisklasse für Konten

```
package oo.account;
```

```
public class Account {  
    (protected) float balance = 0;  
    public float getBalance() { return balance; }  
    public void statement() {  
        System.out.println("Account's balance is " + getBalance() + ".");  
    }  
    public void deposit(float amount) {  
        if (amount <= 0)  
            return;  
        balance += amount;  
    }  
    public float withdraw(float amount) {  
        float result = 0;  
        if (amount <= 0)  
            return result;  
        result = amount > balance ? balance : amount;  
        balance -= result;  
        return result;  
    }  
}
```

Wir ermöglichen auch
(schreibenden) Zugriff in
den Unterklassen.

Wir zahlen unter
Umständen nicht den
vollen Betrag aus
sondern nur soviel, dass
der Kontostand nicht
negativ wird.

Ein geändertes Konto

```
package oo.account;
```

```
public class CreditAccount extends Account {
```

```
    private float overdraft = 0;
```

```
    public void statement() {
```

```
        super.statement();
```

```
        System.out.println("Account's overdraft is " + getOverdraft() + ".");
```

```
    }
```

```
    public float getOverdraft() { return overdraft; }
```

```
    public boolean adjustOverdraft(float amount) { ... }
```

```
    public float withdraw(float amount) {
```

```
        float result = 0;
```

```
        if (amount <= 0)
```

```
            return result;
```

```
        result = balance - amount < -overdraft ?
```

```
            balance + overdraft
```

```
            : amount;
```

```
        balance -= result;
```

```
        return result;
```

```
    }
```

```
}
```

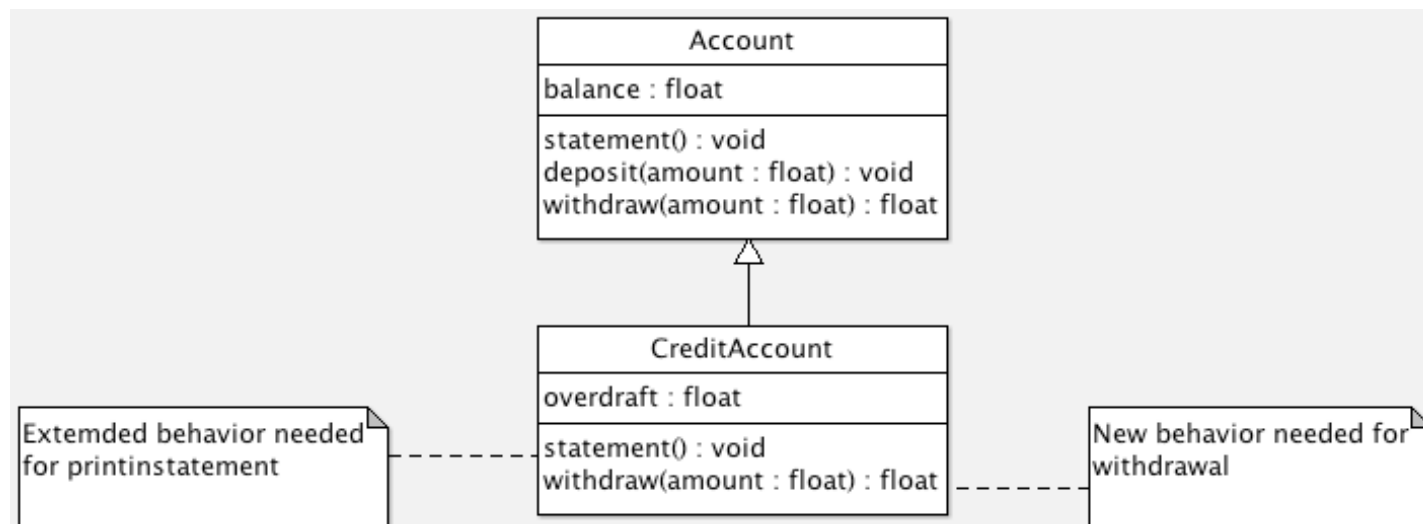
Ergänzung der vererbten
Implementation

Die geerbte Implementation
läßt keine negativen
Kontostände zu! Damit müssen
wir komplett überschreiben.

Vererbung mit/ohne Überschreiben

-- Zusammenfassung --

- * Optionen für Methodenvererbung
 - Vollständige Bewahrung
 - Vollständige Überschreibung (Ersetzung)
 - Teilweise Überschreibung (Ergänzung)
 - Java: “super” ruft Oberklassenmethode auf
- * Statische Methoden und Konstruktoren werden nicht vererbt.
 - Die Referenzierung mit “super” ist aber auch möglich.



Polymorphismus und dynamische Bindung

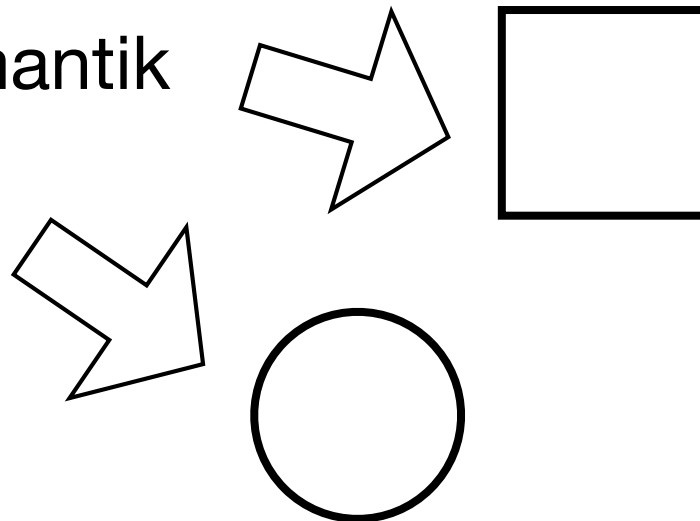
Polymorphismus

* Eine Methode ist polymorph, wenn sie in Unterklassen (Untertypen) unterschiedlich implementiert ist.

* Beispiel

■ Nachricht: `s.draw(...)` mit `s : Shape`

■ Optionen für Semantik



Dynamische Bindung

- * Dynamisches oder spätes Binden liegt vor, wenn erst beim Ablauf eines Algorithmus (zur Laufzeit) festgelegt werden kann, welche Implementation durch eine Nachricht aktiviert wird.
- * Beispiel: Verwalte einen Container mit Formen und biete eine Zeichen-Methode an, welche alle Formen des Containers zeichnet.

Polymorphismus mit Formen (einschließlich später Bindung)

```
// Construct a list of shapes
Shape[] scribble = new Shape[2];
scribble[0] = new Rectangle(10, 20, 5, 6);
scribble[1] = new Circle(15, 25, 8);

// Handle the shapes in the list polymorphically
for (int i = 0; i < scribble.length; i++) {
    scribble[i].draw(System.out);
    scribble[i].moveBy(100, 100);
    scribble[i].draw(System.out);
}
```

Drawing a Rectangle at:(10, 20), width 5, height 6

Drawing a Rectangle at:(110, 120), width 5, height 6

Drawing a Circle at:(15, 25), radius 8

Drawing a Circle at:(115, 125), radius 8

Fangfrage:

Wäre dies auch noch späte Bindung?

```
// Construct a list of shapes
Shape[] scribble = new Shape[2];
scribble[0] = new Rectangle(10, 20, 5, 6);
scribble[1] = new Circle(15, 25, 8);
```

```
// Handle the shapes in the list polymorphically
scribble[0].draw(System.out);
scribble[0].moveBy(100, 100);
scribble[0].draw(System.out);
scribble[1].draw(System.out);
scribble[1].moveBy(100, 100);
scribble[1].draw(System.out);
```

Drawing a Rectangle at:(10, 20), width 5, height 6

Drawing a Rectangle at:(110, 120), width 5, height 6

Drawing a Circle at:(15, 25), radius 8

Drawing a Circle at:(115, 125), radius 8

* Zusammenfassung

- OO Grundbegriffe
 - * Objekt, Klasse, Nachricht, Vererbung, Polymorphismus
- Erste Anfänge von UML
 - * Einfache Klassendiagramme

* Ausblick

- * OO-Spezifikationen (Verträge)
- * Beziehungen zwischen Klassen
- * Vererbung vs. Komposition
- * Ausgewählte UML-Diagramme im Detail
- * ...

Die “Erfinder” von OO?



* Prof. Kristen Nygaard

■ 27.8.1926 - 10.8.2002

■ Norwegischer Informatiker

■ Erfand die Programmiersprache SIMULA 67, was die Einführung des Klassenkonzeptes bedeutete (zusammen mit Ole-Johan Dahl).

■ Erfand die Programmiersprache BETA (zusammen mit B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen).

Die “Erfinder” von OO?



* Prof. David Parnas

- geb. 10.2.1941
- Amerikanischer Informatiker
- Propagierte “information hiding” (1971)
- Unterstützte modulare Spezifikationen (1972)
- Trennung von Spezifikation und Implementation
- Pionier des Software Engineering (seit 1975)