

## Teil 2. Objektorientierte Konzepte

---

Objekte und Klassen

# Vorlesung „Objektorientierte Softwareentwicklung“

Kapitel 2: Objektorientierte Konzepte und ihre Ausprägung in Java

---

## Objekte und Klassen

---

Gekapselte, dynamisch erzeugbare „Module“ → Objekte

Schablonen zur Objekterzeugung → Klassen

Klassen- versus Instanzmitglieder

Zustand und Verhalten

# Von Modulen zu Objekten (1)

- Module sind statisch!
  - ◆ Man kann nur eine endliche Anzahl davon aufschreiben
- Die Welt ist dynamisch!
  - ◆ Man muss oft neue „Einheiten“ nach Bedarf erzeugen
  - ◆ Man weiß oft nicht in vorhinein wann und wie viele
  - ◆ Beispiel: Es gibt nicht nur ein Auto in der Welt!
- Idee: **Alle Autos im gleichen Modul darstellen**
  - ◆ **Komplexität!**? – Alle Exemplare, aller Varianten, aller Typen aller Hersteller in einem Modul?
  - ◆ **Verständlichkeit und Wartbarkeit!**? – Verquickung von verschiedenster Spezialfälle in einem Modul?
  - ◆ **Schlechte Idee!!!**

# Von Modulen zu Objekten (2)

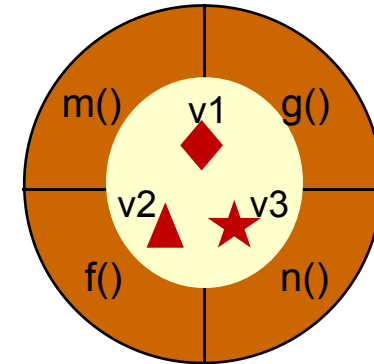
- Module sind statisch!
  - ◆ Man kann nur eine endliche Anzahl davon aufschreiben
- Die Welt ist dynamisch!
  - ◆ Man muss oft neue „Einheiten“ nach Bedarf erzeugen
  - ◆ Man weiß oft nicht in vorhinein wann und wie viele
  - ◆ Beispiel: Es gibt nicht nur ein Auto in der Welt!
- Idee: **Modulbeschreibung und Speicherzuteilung trennen**
  - ◆ Klasse bleibt ein statisches Modul
  - ◆ Sie beschreibt aber zusätzlich die **gemeinsame Struktur** einer beliebigen Menge „**dynamisch erzeugbarer Module**“ → **Objekte**
  - ◆ Daraus werden nach Bedarf „Module“ erzeugt die der Struktur entsprechen aber jeweils einen **eigenen Speicherbereich** haben

# Objekte

- Objekte

- ◆ Gekapselte, dynamisch erzeugte Einheiten von Daten und Operationen

- Variablen → Zustand
- Operationen → Verhalten



- Zustand eines Objektes

- ◆ Werte der Variablen des Objektes zu einem gewissen Zeitpunkt → Zustand kann sich ändern

- Verhalten eines Objektes

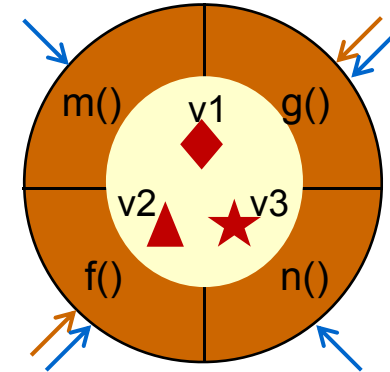
- ◆ Menge der Reaktionen auf Operationsaufrufe
- ◆ Reaktionsmöglichkeiten: Eigene Zustandsübergänge und Aufruf von Operationen anderer Objekte

# Objekte

- Objekte

- ◆ **Gekapselte**, dynamisch erzeugte Einheiten von Daten und Operationen

- Variablen → Zustand
- Operationen → Verhalten



- Schnittstelle

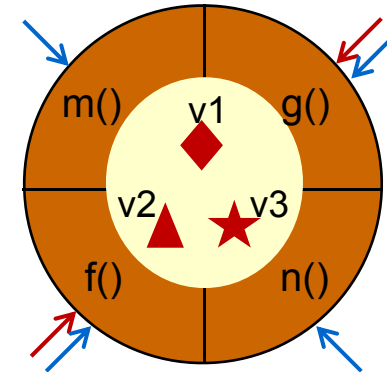
- Menge für einen bestimmten Benutzerkreis aufrufbaren Operationen
- Verschiedene Arten von „Benutzern“ können evtl. unterschiedliche Schnittstellen angeboten bekommen (,public‘, ,package‘, ...)

# Objekte: Kapselung

- Objekte

- ◆ **Gekapselte**, dynamisch erzeugte Einheiten von Daten und Operationen

- Variablen → Zustand
- Operationen → Verhalten



- Kapselung

- ◆ Sprache stellt sicher, dass der Zustand eines Objektes nur über die in seiner **Schnittstelle** spezifizierten Operationen manipuliert wird
- ➔ Bei gleichbleibendem Interface wirken sich Änderungen der lokalen Implementierung nicht auf andere Objekte aus

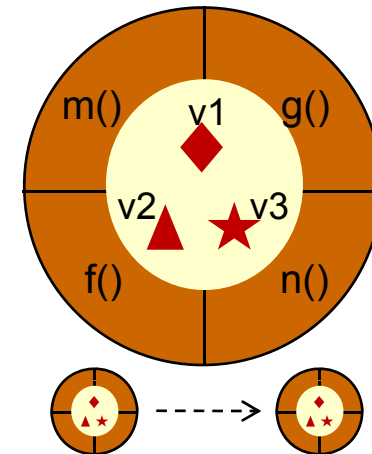
➔ **Wartbarkeit und Zugriffs-Synchronisation!**

# Wie entstehend Objekte? - Spezifikation und Erzeugung

- Objekt-/Prototypbasierte Sprachen

- ◆ Erzeugung durch „hinschreiben“:  
Die Variablen + Methoden können pro Objekt einzeln spezifiziert werden

- ◆ Erzeugung durch kopieren („clonen“):  
Objekte werden durch Kopieren von bestehenden Objekten erzeugt und danach verändert



- Beispiele

- ◆ Self – der Urvater aller prototypbasierten Sprachen

- ◆ Newton-Script – Die Sprache des Urvaters aller PDAs

- ◆ Java-Script – Die Sprache für dynamische Webseiten



# Wie entstehend Objekte? - Spezifikation und Erzeugung

- **Klassensbasierte Sprachen**

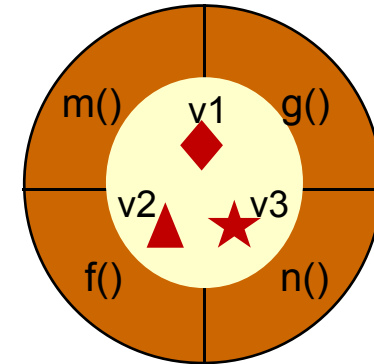
- ◆ **Spezifikation durch „hinschreiben“:**  
Die Variablen + Methoden können pro Objekt einzeln spezifiziert werden

- Dadurch entsteht aber noch kein Objekt!

- ◆ **Erzeugung durch „Instantiierung“:**  
Objekte werden durch Aufruf einer speziellen Operation aus der Spezifikation erzeugt

- ◆ Klasse dient gleichzeitig als Modul mit Klassenvariablen und Methoden und als Schablone für Objekterzeugung durch „Instanziierung“

- ◆ Objekte werden dementsprechend als „Instanzen“ bezeichnet



# Beispiele klassenbasierter Sprachen

- Simula (1968)
  - ◆ Der Urvater aller objektorientierten Sprachen
- Smalltalk (1970)
  - ◆ Die erste „rein objektorientierte“ Sprache
    - „Rein“ heißt hier: „Alles ist ein Objekt!“ – Auch „elementare“ Datentypen!
- C++ (1979 / 1983)
  - ◆ Die erste effiziente Implementierung einer oo Sprache
- Eiffel (1986)
  - ◆ Die erste oo Sprache die für bestimmte Modellierungsprinzipien spezielle Sprachkonstrukte anbot

# Klassen als „Objektschablonen“

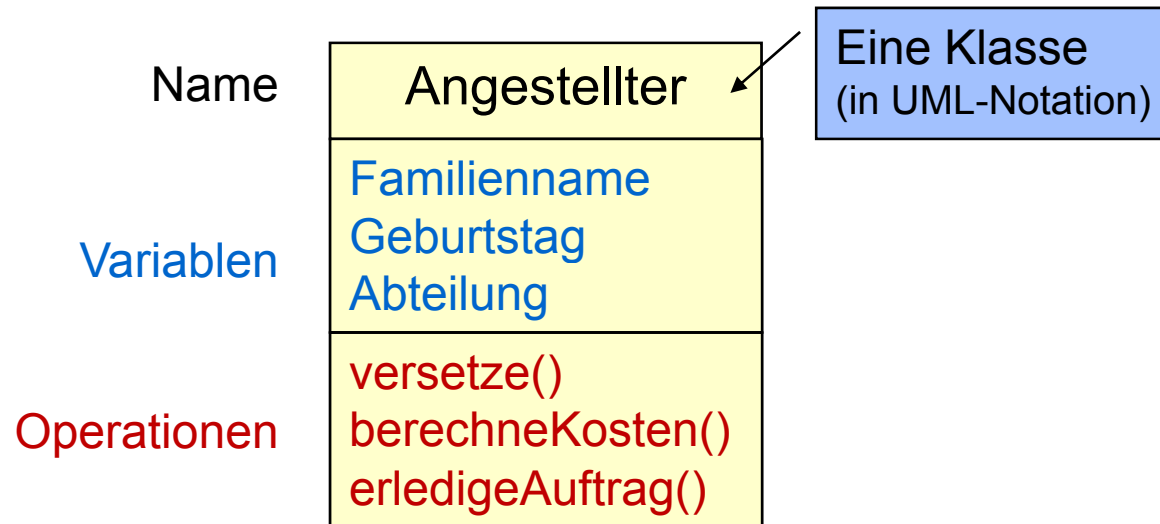
- Eine Klassendeklaration spezifiziert zusätzlich zu ihrem statischen Teil **Instanz-Mitglieder**:
  - ◆ **Instanz-Felder**: Variablen aus denen sich Objekte des Typs zusammensetzen
    - Auch „Attribute“, „Eigenschaften“, „Instanzvariablen“ (properties, instance variables, data members) genannt
  - ◆ **Instanz-Methoden** (instance methods) des Datentyps die auf den Objekten operieren
    - Auch Mitgliedsfunktionen (member functions) genannt
  - ◆ **Die Konstruktoren** (constructors), mit denen neue Objekte des Typs initialisiert werden

# Instanz-Felder und Methoden

- Jede Instanz hat eigene Kopien der in ihrer Klasse spezifizierten Instanz-Variablen
  - ◆ Bei der Erzeugung eines Objekts (mit `new`) wird ein Speicherbereich reserviert, in dem neue Inkarnationen der Felder eingerichtet werden
- Jede Instanz hat in ihrer Klasse implementierten Instanz-Methoden
  - ◆ Diese werden jeweils beim Aufruf an ein Objekt „gebunden“ und operieren auf diesem Objekt
  - ◆ Wenn sie auf ein Feld einer Klasse zugreifen, dann ist für die Dauer des Aufrufs die Inkarnation des Feldes im gebundenen Objekt gemeint → siehe „this“

# Klassifikation und Instantiierung: UML

Klassen definieren Schnittstelle, Operationen und Variablen ...



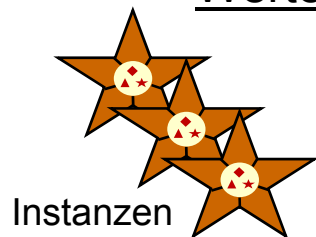
... für ihre Instanzen. Instanzen enthalten Variablen-Werte und das Wissen zu welcher Klasse sie gehören.



# Klassifikation und Instantiierung: Java

- Klasse beschreibt Menge „gleichartiger“ Objekte

- ◆ **gleiche** Schnittstelle
- ◆ **gleiche** Implementierung
- ◆ **gleiche** Variablen
- ◆ **verschiedene** Variablen-Werte



Aufruf eines anderen Konstruktors der gleichen Klasse.

```
class Bike {  
    // Instanz-Variablen:  
    Bremse vorne, hinten;  
    int gang = 18;  
  
    // Instanz-Methoden:  
    void schalten() {gang++;}  
  
    // "Konstruktor"-Methoden:  
    Bike(Bremse v, Bremse h) {  
        vorne = v; hinten = h;  
    }  
    Bike() {  
        this(..., ...)  
    }  
}
```

- Klasse ist Schablone für Objekterzeugung

```
Bike meinRad = new Bike();  
Bike deinRad = new Bike(v, h);
```

# Klassen-Variablen und Klassen-Methoden

## ● Variablen und Methoden, die

- ◆ nur ein mal pro Klasse existieren
- ◆ für alle Instanzen zugreifbar sind
- ◆ durch Nachrichten an die Klasse auch von außen zugreifbar sind:
  - Bike.verkaufe(10);
  - siehe auch Abschnitt über Sichtbarkeit

## ● Benutzung

- ◆ gemeinsame Eigenschaften aller Instanzen
  - z.B. klassenspezifische Konstanten
- ◆ Informationen über die Instanzen (Metainformationen)
  - z.B. Anzahl der Instanzen

```
class Bike {  
    // Instanz-Variablen:  
    ...; int gang = 18;  
  
    // Klassen-Variable:  
    static final int gänge = 21;  
  
    // Instanz-Methode:  
    void schalten() {  
        if (gang < gänge) {gang++;}  
    }  
  
    // Klassen-Variable:  
    static int nrOfBikes = 0;  
  
    // Klassen-Methode:  
    static verkaufe(int anz) {  
        nrOfBikes = nrOfBikes - anz;  
    }  
}
```

# Klassendeklaration: Beispiel

- Wir deklarieren eine **Klasse Time** für den Gebrauch in einer elektronischen Stoppuhr
- Die **Instanzvariablen sec, min, hrs** stellen die gemessene Zeit der *jeweiligen* Uhr dar
- Die **Instanzmethode tick** implementiert die Operation des Tickens des Sekundenzählers

```
class Time {
    byte sec=0; //seconds, 0<=sec<60
    byte min=0; //minutes, 0<=min<60
    int hrs=0; //hours, 0<=hrs

    void tick() {
        sec++;
        if(sec >= 60) {
            sec -= 60;
            min++;
            if(min >= 60) {
                min -= 60;
                hrs++;
            }
        }
    }
}
```



# Klassen als Typen

- Typdeklaration
  - ◆ Sei  $\kappa$  der Name einer Klasse. Dann ist  $\kappa \ v;$  die Deklaration einer Variablen  $v$  vom Typ  $\kappa$
- Objekterzeugung
  - ◆ Der Ausdruck  $\text{new } \kappa()$  erzeugt ein neues Objekt vom Typ  $\kappa$  und sein Wert ist die Referenz auf dieses Objekt
- Objektinitialisierung
  - ◆ Als Nebeneffekt der Ausführung von  $\text{new}$  wird das neue Objekt initialisiert
- Zugriff auf Instanz-Variablen und -Methoden
  - ◆ Man kann auf die Mitglieder des von  $v$  referenzierten Objekts mittels des Punkt-Operators  $.$  zugreifen

# Zugriff auf Instanz-Variablen und -Methoden

- Eine Instanzvariable wird folgendermaßen angesprochen

**objektausdruck.variablenname**

```
Time t = new Time();  
t.sec = 21; // Setze sec in t auf 21  
t.min = 35; // Setze min in t auf 35  
t.hrs = 2;  // Setze hrs in t auf 2  
t.tick();  // Rufe tick() von Objekt t auf
```

- Eine Instanzmethode wird folgendermaßen aufgerufen

**objektausdruck.methodenname(arg1, ..., argn)**

# Initialisierung und Konstruktoren

---

Objekterzeugung und Initialisierung → new-Operator und Konstruktoren

Initialisierung von Klassen

# Initialisierung und Konstruktoren

- Der **new**-Operator

- ◆ Der Aufruf **new K()** reserviert den für eine neue Instanz der Klasse **K** den benötigten Speicherplatz
- ◆ Er initialisiert die Instanzvariablen mit Varianten von Null, je nach dem Variablentyp
  - `0`, `0d`, `0f`, `\u0000`, `false` oder `null`

- Konstruktoren

- ◆ Oftmals genügt die Initialisierung einer Variablen mit Null nicht
- ◆ Zu diesem Zweck können in der Objekt-Klasse **Konstruktoren (constructors)** definiert werden

# Initialisierung und Konstruktoren

- **Konstruktoren** haben Ähnlichkeit mit Klassen-Methoden, gelten aber nicht als Methoden
  - ◆ Alle Möglichkeiten der Zugriffskontrolle für Methoden gibt es auch für Konstruktoren
  - ◆ Aber sie haben keinen expliziten Ergebnistyp
    - Auch nicht den leeren Typ `void`!
    - Ihr impliziter Ergebnistyp ist die enthaltende Klasse
- Konstruktoren tragen den Namen ihrer Klasse
  - ◆ Verschiedene Konstruktoren einer Klasse unterscheiden sich lediglich in der Anzahl bzw. dem Typ ihrer Parameter
    - „overloading“ von Konstruktoren ist erlaubt

# Initialisierung und Konstruktoren

- Ein Konstruktor kann einen anderen in derselben Klasse mit **this(...)** explizit aufrufen (explicit invocation)
  - ◆ Er kann einen Konstruktor der unmittelbaren Oberklasse mit **super(...)** aufrufen
- Ist in einer Klasse K kein Konstruktor definiert, so erzeugt der Übersetzer einen **parameterlosen Standard-Konstruktor**
  - ◆ `class K { K() {super();} ... }`
  - ◆ Dieser ruft den entsprechenden Konstruktor der Oberklasse auf.

# Initialisierung und Konstruktoren: Beispiel A

- Die Klasse `Date` hat Konstruktoren der Stelligkeit 0 bis 3
- Der Nullstellige setzt das Datum auf den 1.1.0
- Die Konstruktoren der Stelligkeit  $n > 0$  rufen zunächst den  $(n-1)$ -stelligen Konstruktor explizit auf und überschreiben entweder
  - ◆ das Jahr,
  - ◆ Monat und Jahr,
  - ◆ oder Tag, Monat und Jahr

```
public class Date {
    private byte day, month;
    private short year;

    // Konstruktoren:
    public Date() {
        day=1; month=1;
    }
    public Date(short y) {
        this(); year=y;
    }
    public Date(byte m, short y) {
        this(y); month = m;
    }
    public Date(byte d, byte m, short y){
        this(m,y); day=d;
    }
    ...
}
```

# Initialisierung und Konstruktoren: Beispiel A

- Damit sind folgende Initialisierungen von Variablen vom Typ Date möglich:

```
Date aday =
    new Date();
    // Der 1.1.0

Date bday =
    new Date((short) 1970);
    // Der 1.1.1970

Date cday =
    new Date((byte) 8, (short) 1975);
    // Der 1.8.1975

Date dday =
    new Date((byte)23, (byte)8, (short)2002);
    // Der 23.8.2002
```

```
public class Date {
    private byte day, month;
    private short year;

    // Konstruktoren:
    public Date() {
        day=1; month=1;
    }
    public Date(short y) {
        this(); year=y;
    }
    public Date(byte m, short y) {
        this(y); month = m;
    }
    public Date(byte d, byte m, short y){
        this(m,y); day=d;
    }
    ...
}
```



# Initialisierung von Klassenvariablen

- Klassen werden beim ersten Gebrauch angelegt
  - ◆ incl. der Initialisierung ihrer Klassenvariablen
  - ◆ noch bevor Instanzen davon erzeugt werden.
- Initialisierung von Klassenvariablen
  - ◆ Vorinitialisierung mit (Varianten von) Null
  - ◆ Initialisierungsanweisungen von Klassenvariablen ausführen
    - Beispiel: `static int i=1;`
  - ◆ Statische Initialisierungsblöcke (static initializers) ausführen
    - Erlauben mehr Freiheit in der Initialisierung, inklusive des Aufrufs von Klassenmethoden. Beispiel: `static { i=f()+2; }`

# Vorlesung „Objektorientierte Softwareentwicklung“

Kapitel 2: Objektorientierte Konzepte und ihre Ausprägung in Java

---

## Objektidentität

---

Objektidentität

Sharing

Aliasing

# OOP: Objekt-Identität (OID)

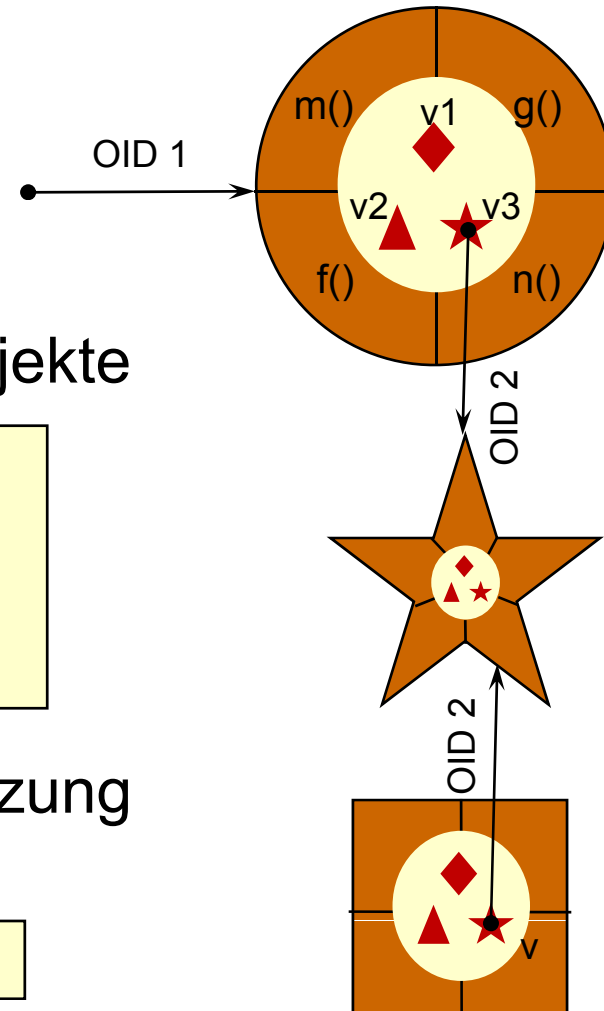
- OID = Objekt-Referenz, die unabhängig ist von
  - ◆ Zustand des Objekts
  - ◆ Ort der Speicherung
  - ◆ ...

- Variablen speichern OIDs, keine Objekte

```
public class Kreis {  
    ...  
    Stern v3 = new Stern(...);  
    ...  
}
```

- OIDs ermöglichen gemeinsame Nutzung von Objekten („Sharing“)

```
Stern v = v3;
```

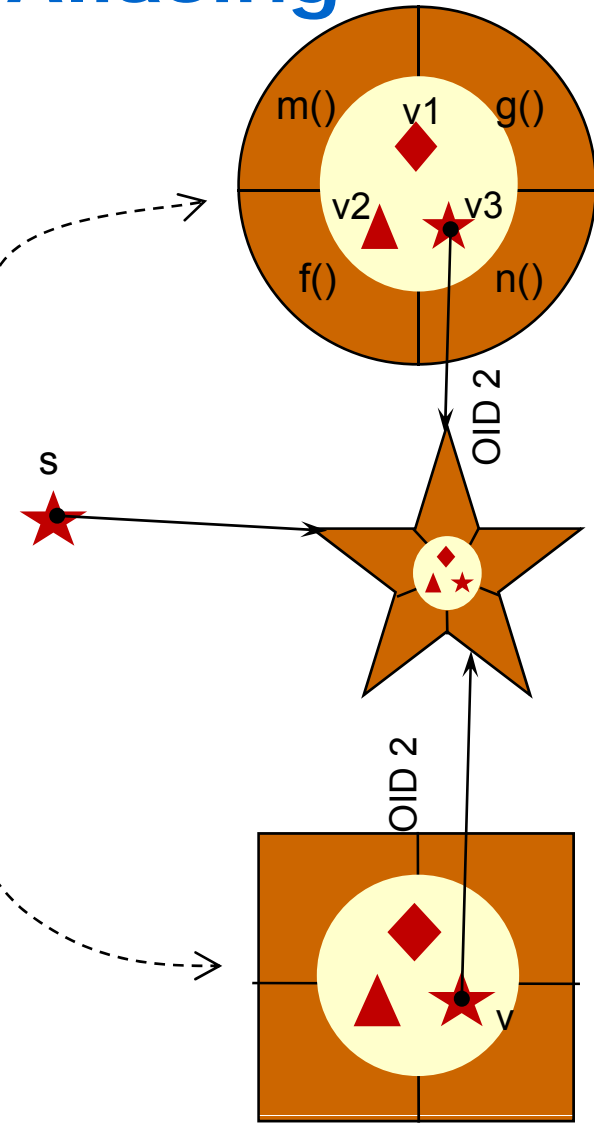


# Entstehung von Sharing / Aliasing

```
public class Kreis {
    ...
    Form v3;
    public Kreis(Form f) {v3 = f;}
    ...
}
```

```
...
Stern s = new Stern();
new Kreis(s);
...
new Quadrat(s);
```

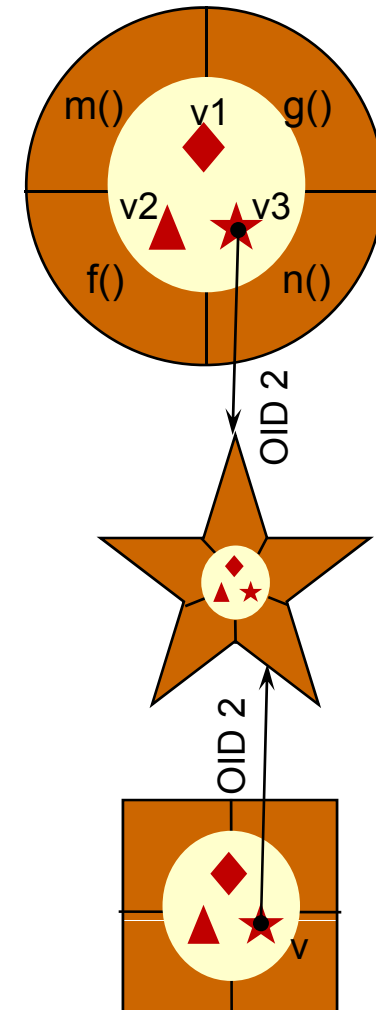
```
public class Quadrat {
    ...
    Form v;
    public Quadrat(Form f) {v = f;}
    ...
}
```



# Objekt-Identität → Sharing / Aliasing

Peter Grogono (Concordia University):  
„Copying, Sharing and Aliasing“

- Nutzen: „Sharing“
  - ◆ Möglichkeit, explizit zu machen, dass zwei unterschiedliche Variablen das **selbe** Objekt referenzieren.
- Gefahr: „Aliasing“
  - ◆ Risiko, dass unterschiedliche Variablen das **selbe** Objekt referenzieren, ...
  - ◆ ... ohne dass man sich dessen bewusst ist.
  - ◆ Versehentliche Änderungen
- Sharing = Aliasing
  - ◆ unterschiedliche Seiten der gleichen Medaille.



# Vorlesung „Objektorientierte Softwareentwicklung“

Kapitel 2: Objektorientierte Konzepte und ihre Ausprägung in Java

---

## Selbstbezug: this

---

Bezug auf das „aktuelle“ Objekt → this / self / current

Umsetzung als impliziter Parameter

Typisierung von this

# Selbstbezug: this

## ● Problem

- ◆ Die selbe Instanz-Methode kann von jeder Instanz einer Klasse ausgeführt werden!
- ◆ Wie nimmt man in der Methode auf das Objekt Bezug, für das die Methode zur Laufzeit ausgeführt werden wird?

## ● Lösung

- ◆ Variable, deren Wert nur vom Compiler / Laufzeitsystem gesetzt wird
- ◆ Java, C++: „**this**“
- ◆ Smalltalk: „**self**“
- ◆ Eiffel: „**current**“

```
void registrierenBei(Amt a) {
    a.fahrradRegistrieren(this)
}

void schalten(int gang) {
    if (gang>0 && gang<gänge) {
        this.gang = gang;
    }
}
```

## ● Konvention

- ◆ **this** ist implizites Zielobjekt eines Zugriffs, wenn kein explizites Ziel angegeben ist
  - "schalten(3)" ist gleichbedeutend mit "this.schalten(3)"

# Umsetzung von „this“: Impliziter Parameter einer jeden Instanz-Methode

- Ursprünglicher Java-Code

```
void registrierenBei(Amt a) { a.fahrradRegistrieren(this) }
```

- Prinzip

- ◆ „This“ wird vom Compiler jeder Instanz-Methode als **erster Parameter** hinzugefügt.
- ◆ Jeder „this“-Zugriff ist ein Zugriff auf diesen Param.
- ◆ Bei jeder Nachricht, wird das **Zielobjekt** als Wert von „this“ in der aufgerufenen Methode übergeben.

- Explizit gemachter „this“ Parameter

```
void registrierenBei(... this, Amt a) { a.fahrradRegistrieren(a, this) }
```

In diesem Aufruf von fahrradRegistrieren() wird das Amt **a** der Wert des „this“-Parameters sein.





# Der Typ von this

- Der statische Typ von `this` ist jeweils die enthaltende Klasse
  - ◆ In Klasse  $\kappa$  hat `this` den statischen Typ  $\kappa$
- Der dynamische Typ von `this` ist jeweils die Klasse des Objektes für das eine Methode gerade ausgeführt wird
  - ◆ Durch „Vererbung“ kann es sein dass diese beiden Klassen unterschiedlich sind → Später

# Interfaces, Subtypen und Nachrichten

---

Abstrakte Datentypen → Implementierung der von Methoden abstrahieren

Sehr abstrakte Datentypen → Interfaces → auch Datenstrukturen abstrahieren

Nutzen → Verschiedene Implementierungen des gleichen Datentyps → implements-Beziehung

Subtypen "revisited": Objektorientierte Subtypen

Unterschied von statischem und dynamischem Typ

# Bisher: Konkrete Datentypen

- Klassennamen können in Typdeklaration verwendet werden
- Problem: Das ist zu starr
  - ◆ Klasse legt komplette Definition von Zustand und Verhalten fest (Datenstrukturen und Implementierung)
  - ◆ Andere Implementierungen des gleichen Konzeptes werden somit ausgeschlossen
- Idee: **Abstrakte Datentypen**
  - ◆ Ein abstrakter Datentyp legt nur eine Datenstruktur („Trägermenge“) und die Signaturen der darauf arbeitenden Operationen fest
  - ◆ Verschiedene konkrete Datentypen können einen abstrakten Typ unterschiedlich implementieren

# Schnittstellen: Abstrakter als abstrakt!

- Kritik an abstrakten Datentypen
  - ◆ Sie legen eine Datenstruktur („Trägermenge“) fest!
  - ◆ Somit besteht nicht mehr die Freiheit, die gleiche Funktionalität auf einer ganz anderen Datenstruktur zu implementieren
  - ◆ Beispiel: Suchen auf Listen, Bäumen, ...
- Idee: Auch von Datenstruktur abstrahieren!
- Schnittstellen (Interfaces)
  - ◆ Benannte Menge von Operationsspezifikationen
  - ◆ Jeweils **Name**, **Parametertypen**, **Ergebnistyp**

# Interface (Schnittstelle)

- Enthält Typen und Namen von Operationen aber keine Implementierung!
- Dient der Angabe der Operationen die ein Objekt mindestens bieten muss, um in einem bestimmten Kontext benutzbar zu sein
- ... unabhängig von seiner konkreten Implementierung

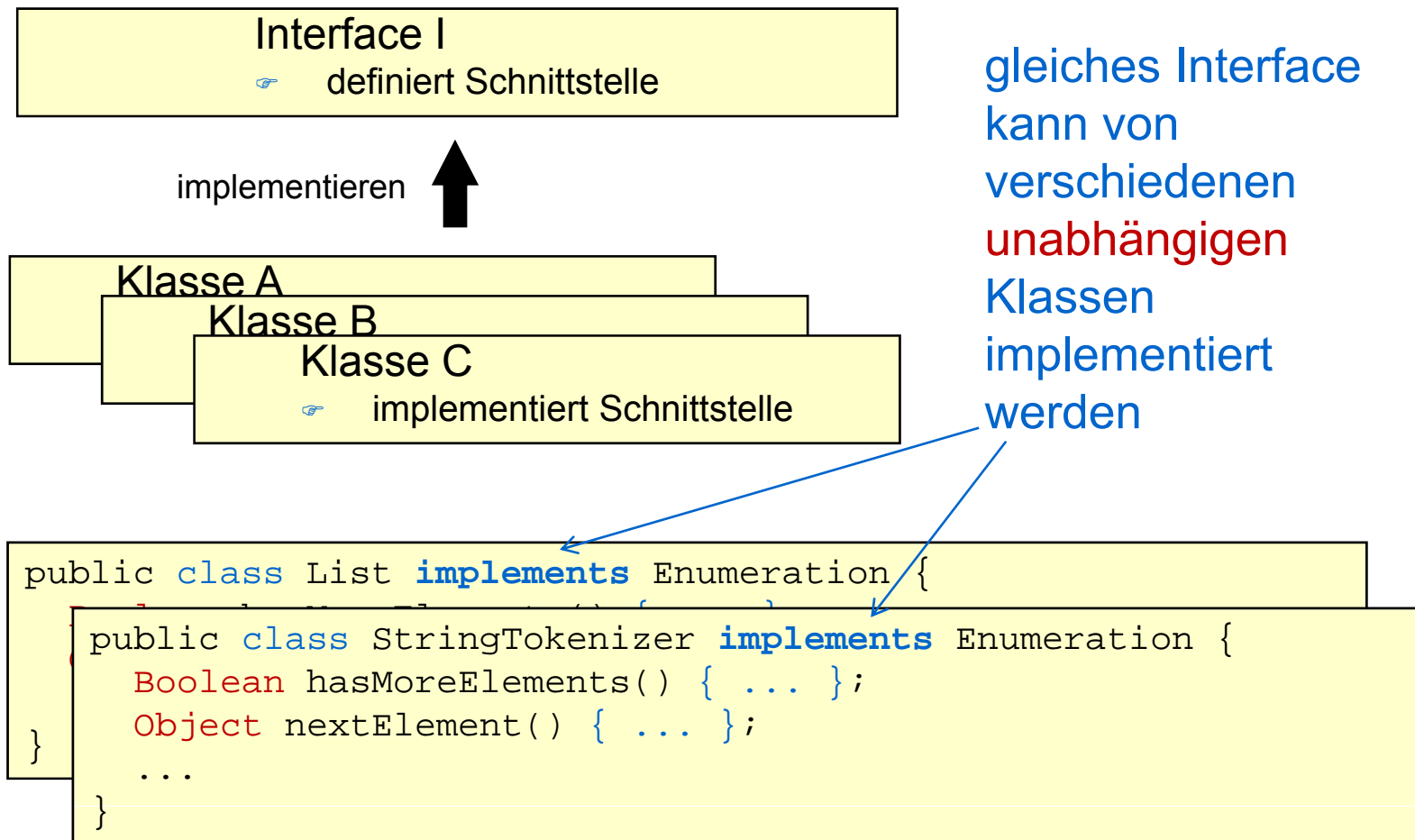
```
// Druckbare Objekte  
  
public interface Printable {  
    void print();  
}
```

```
// x referenziert druckbare Objekte  
  
Printable x;  
...  
x.print();
```

Alle Operationen im Interface sind implizit „public“

# „Implementierungs“-Beziehung

- Klassen implementieren Interfaces



# Die Implementierungs-Beziehung und die Subtyp-Beziehung

- **Ersetzbarkeitsprinzip:** **B** ist ein Subtyp von **A**  $\Leftrightarrow$  Instanzen von **B** sind immer für Instanzen von **A** einsetzbar
- **Teilmengen-Kriterium:** Die Menge der Instanzen von **B** ist eine Teilmenge der Instanzen von **A**

## Konsequenz

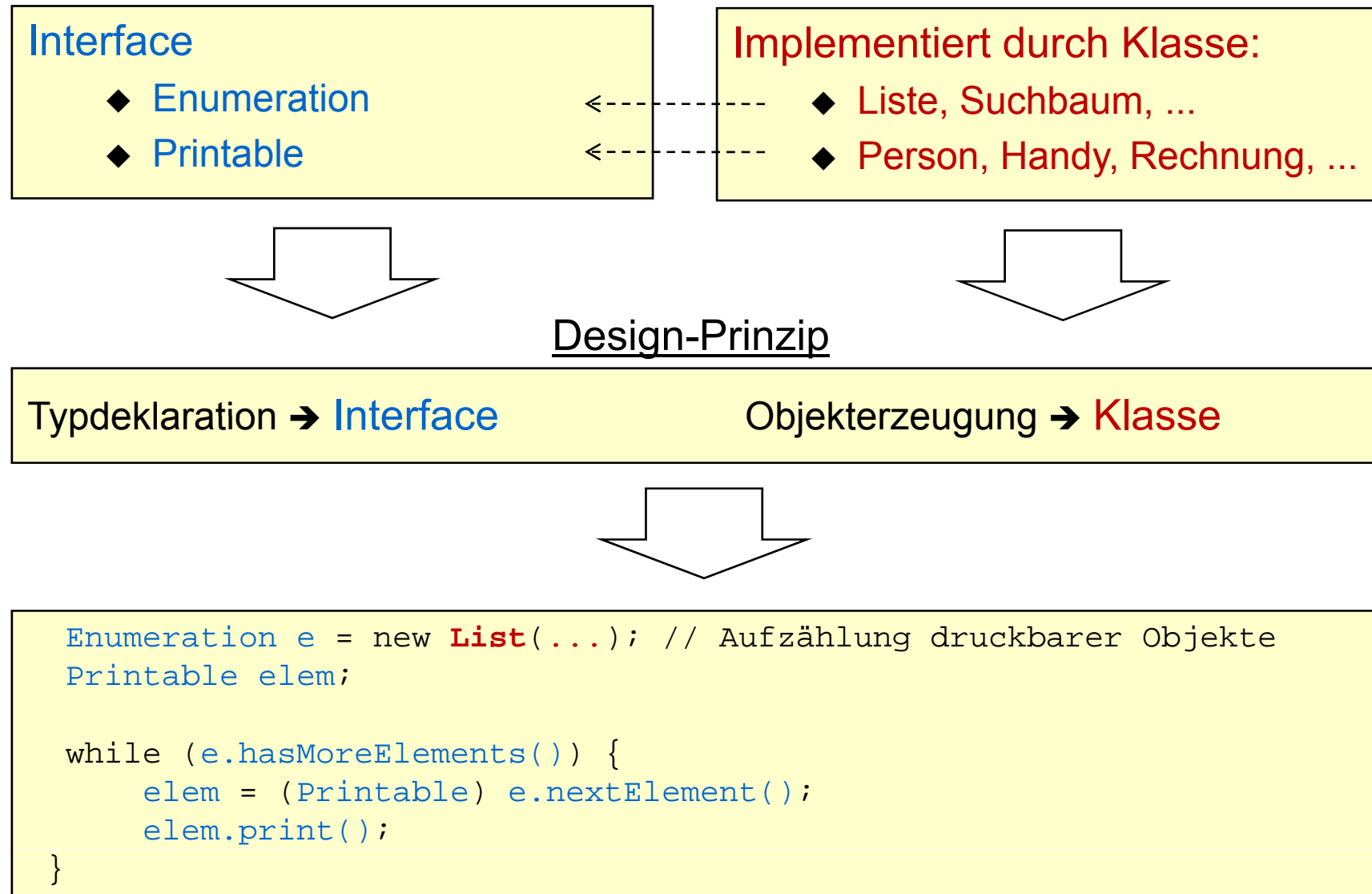
- Implementierende Klassen sind Subtypen der implementierten Interfaces
  - ◆ **Ersetzbarkeit:** In jeden Kontext einsetzbar wo das Interface erwartet wird
  - ◆ **Teilmenge:** Die Instanzen des Interfaces sind die Vereinigung der Instanzen aller implementierenden Klassen.

# Interfaces versus Klassen

- Empfehlung
  - ◆ Zuerst möglichst eine Interface-Hierarchie entwerfen
  - ◆ Interface-Namen als Typdeklarationen verwenden
  - ◆ Klassen (die die Interfaces implementieren) nur zur Objekt-Erzeugung benutzen (siehe Beispiel auf der nächsten Seite)
- Warnung
  - ◆ Wenn ein Klassenname **C** als Typdeklaration verwendet wird (**C var;**), wird die Menge der einsetzbaren Untertypen auf die Unterklassen von **C** eingeschränkt
  - ◆ Instanzen von Klassen, die die gleiche Schnittstelle wie **C** implementieren, aber nicht von **C** erben, könnten nicht an **var** zugewiesen werden



# Interfaces versus Klassen: Beispiel



# Vorlesung „Objektorientierte Softwareentwicklung“

## Kapitel 2: Objektorientierte Konzepte und ihre Ausprägung in Java

---

# Dynamisches Binden (Dynamic Binding)

---

Subtyping mit statischem Binden wäre sinnlos.

Nachrichten und Dynamisches Binden

Nachrichtenauflösung → Bestimmung der Signatur + dynamisches Binden

Implementierung von dynamischem Binden

Unterschied static versus nicht static für methoden

# Nutzen der erweiterten Subtypbeziehung?

Typ- und Untertyp-Deklarationen

```
interface Point { int getX(); int getY() }  
  
class MyPoint implements Point {  
    private int x; public int getX(){...}  
    private int y; public int getY(){...}  
}
```

Benutzung

```
Point p;           // Variablendeklaration  
MyPoint mp;       // Variablendeklaration  
...  
p = mp;           // ok: Point ist Obertyp von MyPoint  
...  
p.getX();         // ok: getX() ist in Point enthalten
```

Was soll hier passieren? Welche Methode soll aufgerufen werden?

Hier wird die Ersetzbarkeit ausgenutzt

# Dilemma: Bisher nur „statisches Binden“

Typ- und Untertyp-Deklarationen

```
interface          Point { int getX(); int getY() }

class MyPoint implements Point {
    private int x; public int getX(){...}
    private int y; public int getY(){...}
}
```

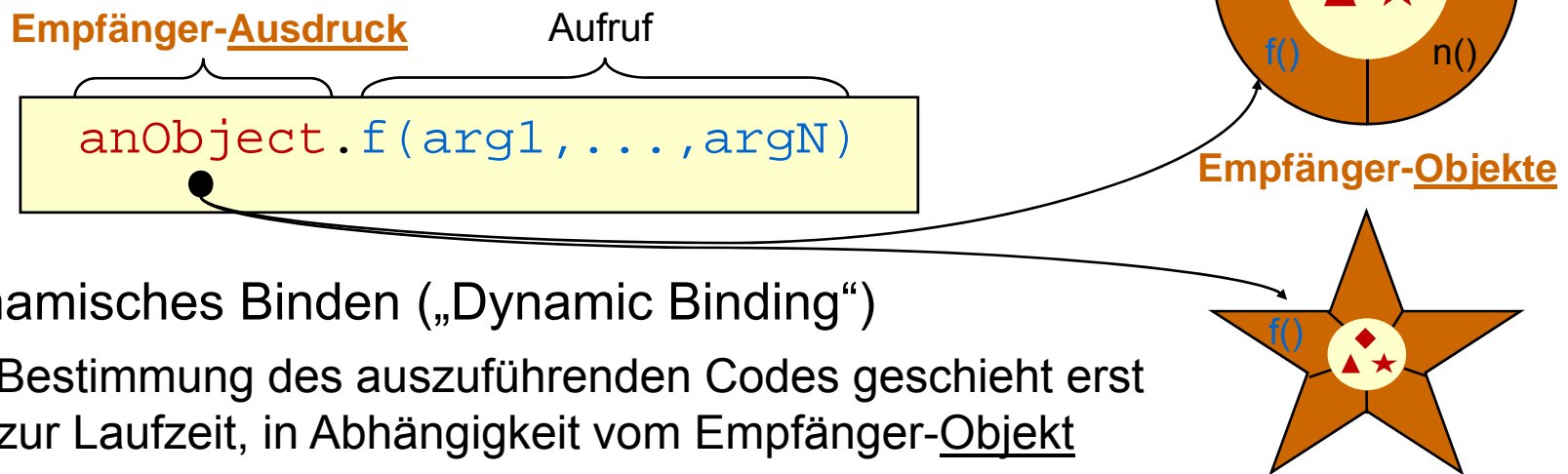
Benutzung

```
Point p;           // Variablendeklaration
MyPoint mp;        // Variablendeklaration
...
p = mp;           // ok: Point ist Obertyp von MyPoint
...
p.getX();         // ok: getX() ist in Point enthalten
```

- Statisches Binden: Führt die Methode aus dem statischen Typ des Empfänger-Ausdrucks aus
  - ◆ Leider hat **Point**, der statische Typ von **p**, keine Implementierung von `getX()` die man ausführen könnte ☹

# Nachrichten und „Dynamisches Binden“

- Nachricht
  - ◆ Aufforderung an ein **Objekt**, eine seiner **Methoden** auszuführen
  - ◆ Java-Syntax: `<empfängerausdruck>.<aufruf>`



- Dynamisches Binden („Dynamic Binding“)
  - ◆ Bestimmung des auszuführenden Codes geschieht erst zur Laufzeit, in Abhängigkeit vom Empfänger-Objekt
  - ◆ Die Methode des Empfänger-Objekts wird ausgeführt!
- Nachricht ≠ Prozeduraufruf
  - ◆ Eine Nachricht → Verschiedene Effekte, je nach Empfängerobjekt!

# Obiges Beispiel und „Dynamic Binding“

Typ- und Untertyp-Deklarationen

```
interface Point { int getX(); int getY() }  
  
class MyPoint implements Point {  
    private int x; public int getX(){...}  
    private int y; public int getY(){...}  
}
```

Benutzung

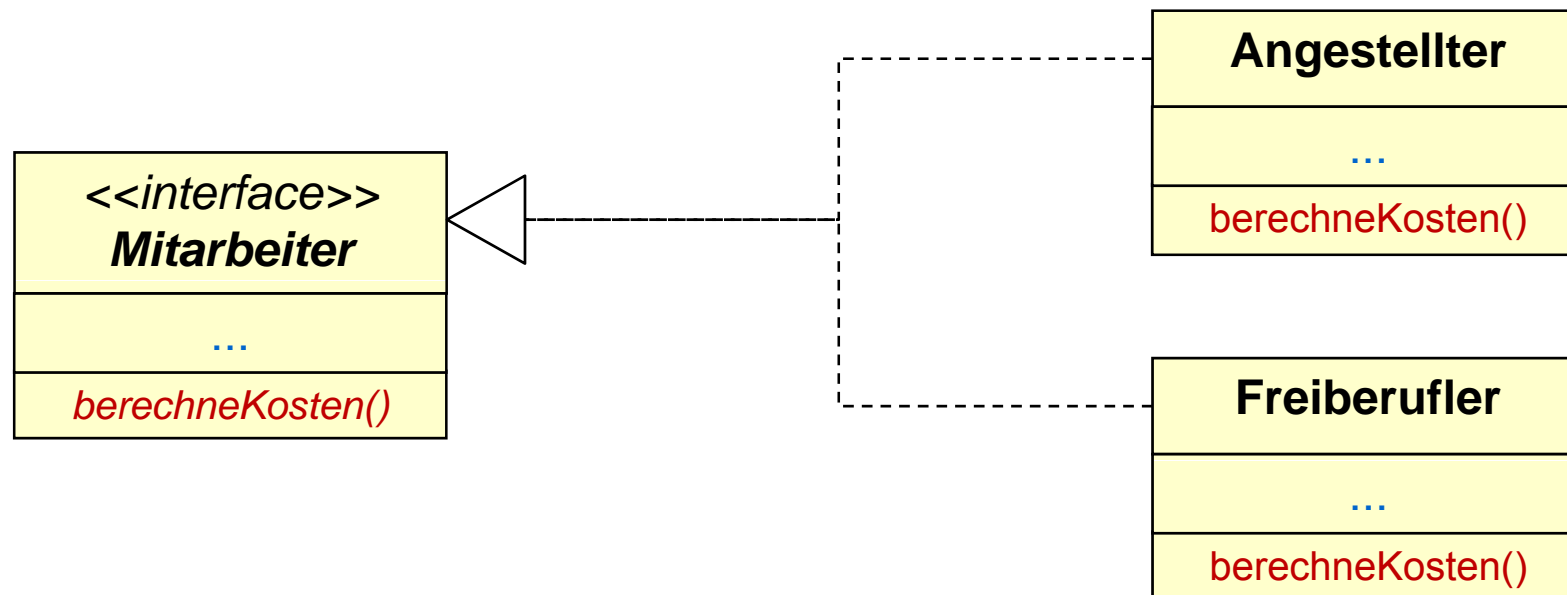
```
Point p;           // Variablendeklaration  
MyPoint mp;       // Variablendeklaration  
...  
p = mp;           // ok: Point ist Obertyp von MyPoint  
...  
p.getX();         // ok: getX() ist in Point enthalten
```

Hier wird die Methode aus dem von **p** aktuell referenzierten **MyPoint**-Objekt aufgerufen!

Hier wird die Ersetzbarkeit ausgenutzt

# Nachrichten und dynamisches Binden: Beispiel

- Angenommen die Klassen „Angestellter“ und „Freiberufler“ implementieren das Interface „Mitarbeiter“



# Nachrichten und dynamisches Binden: Beispiel (Fortsetzung)

