# 5. Grundkonzepte der Objektorientierung (3):

# Vererbung
# Polymorphismus
# dynamisches Binden

Java-Beispiel:
Time2.java



## Schwerpunkte

- ***Vererbung***
  - Unterklassen (Subklassen)
  - Oberklassen (Superklassen)
  - Hierarchiebildung
- **Einfachvererbung** (Java) - **Mehrfachvererbung** (C++)
- Noch einmal Sichtbarkeit: private, public, **protected**
- Vererbungshierarchie und die Klasse **'Object'**
- **Polymorphismus:** Überschreiben von Methoden (vgl.: Überladen/Overloading)
- **Dynamisches Binden:** dynamische Methodensuche
- **Final**-Methoden, **Final**-Klassen
- **Abstrakte Klassen**



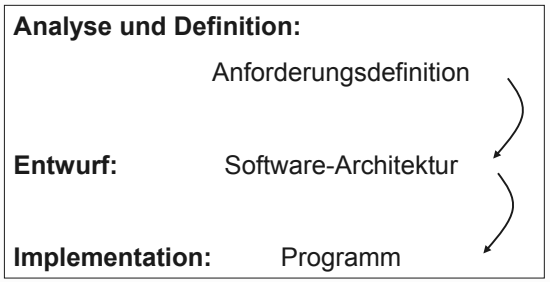
## Vererbung:

- *eine* Beziehung (unter vielen) zwischen Klassen
- Modellierungselement in Software-Architekturen
- also: Vererbung nicht nur für Programmierung wesentlich, sondern auch besonders in anderen Phasen der SW-Entwicklung



## Klassen: Wann werden sie entwickelt?

**Software-Entwicklung (Phasen):**

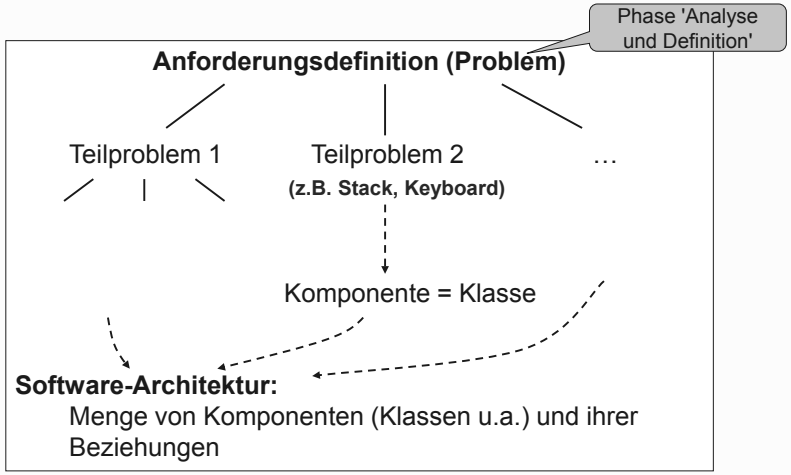



Spätestens: in der *Entwurfsphase* durch Analyse der Anforderungsdefinition

Ziel: bereits bei der *'Analyse und Definition'* Klassen identifizieren (Klassen entsprechen *Objekten des Problembereichs*, z.B. Keyboard, Stack, Time)

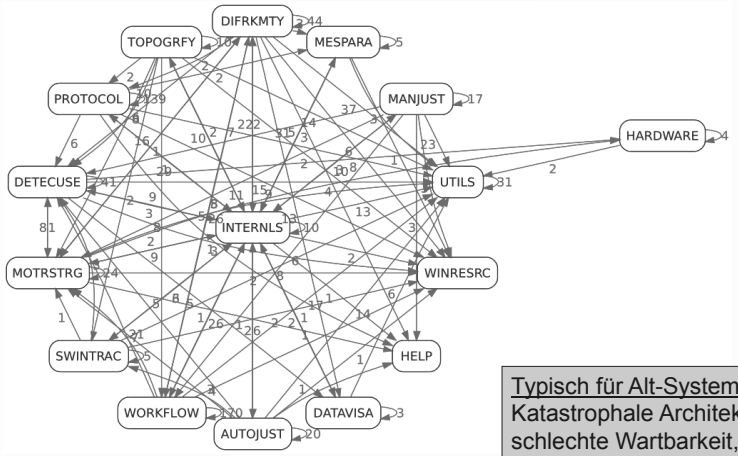
# Wie gewinnt man Klassen (Komponenten)?

↳ **durch Dekomposition des Problems**






# Softwarearchitekturen müssen vor der Programmierung geplant werden, sonst erhält man soetwas …

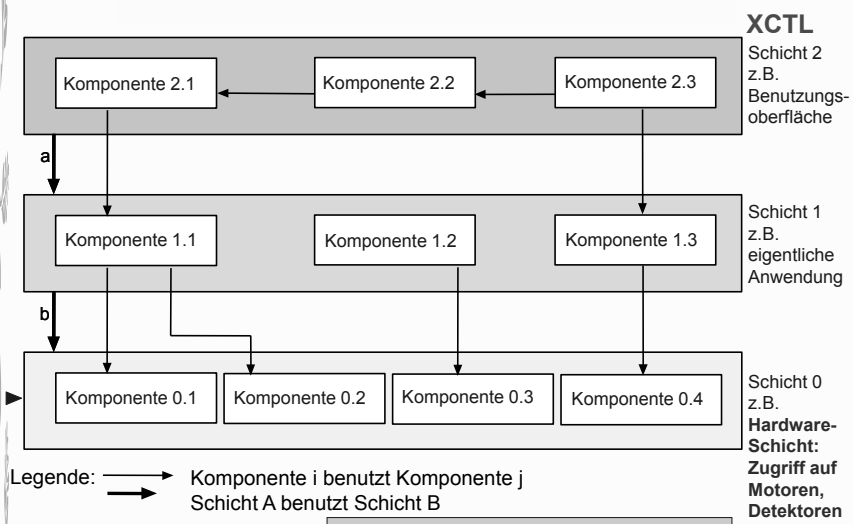



**Reale Architektur des System XCTL**
**(Projekt für das Institut für Physik)**

Typisch für Alt-Systeme:
Katastrophale Architektur, schlechte Wartbarkeit, Verständlichkeit
→ zyklische Abhängigkeiten



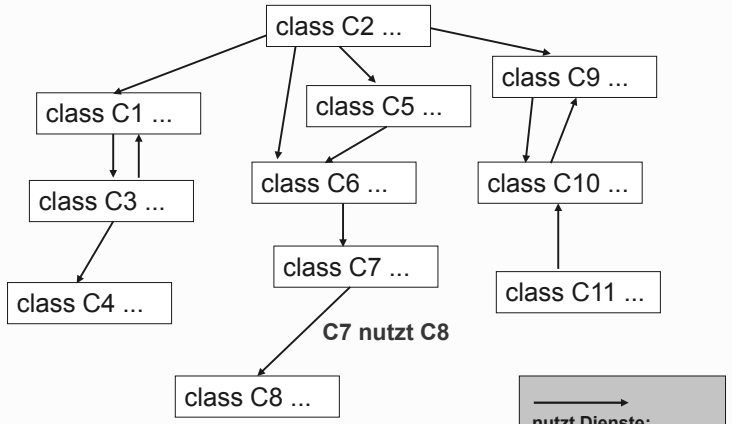
# … und benötigt eigentlich eine Drei-Schichten-Architektur



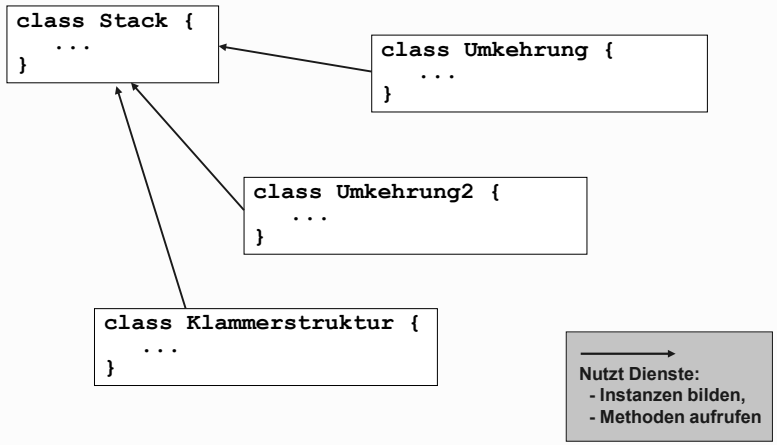

Bessere Wartbarkeit, Verständlichkeit
→ einfache Abhängigkeiten



# Art der Beziehungen zwischen den aus dem Problem heraus modellierten Klassen?



# Beziehungen zwischen Klassen: Beispiel






# Beziehungen zwischen den aus dem Problem heraus modellierten Klassen

***Nutzung von Diensten,*** **Vererbung,** Assoziation, Aggregation …

Klasse 1: Daten, Methoden — **allgemeiner**

**Vererbung**

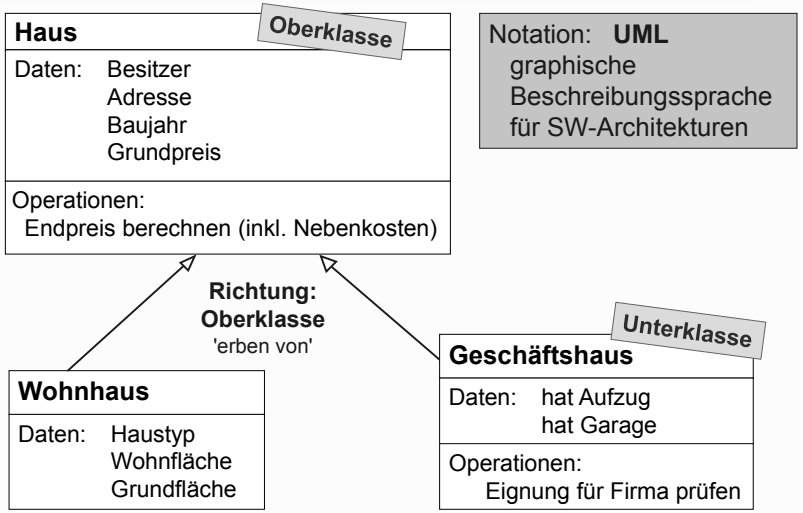
Klasse 2: alle Daten und Methoden der Klasse 1 + weitere … — **spezieller**

Klasse 1:
- vererbt ihre Eigenschaften auf Klasse 2
- ist allgemeiner als Klasse 2
- ist ***Oberklasse*** von Klasse 2

Klasse 2:
- erbt …
- ist spezieller …
- ist ***Unterklasse*** …

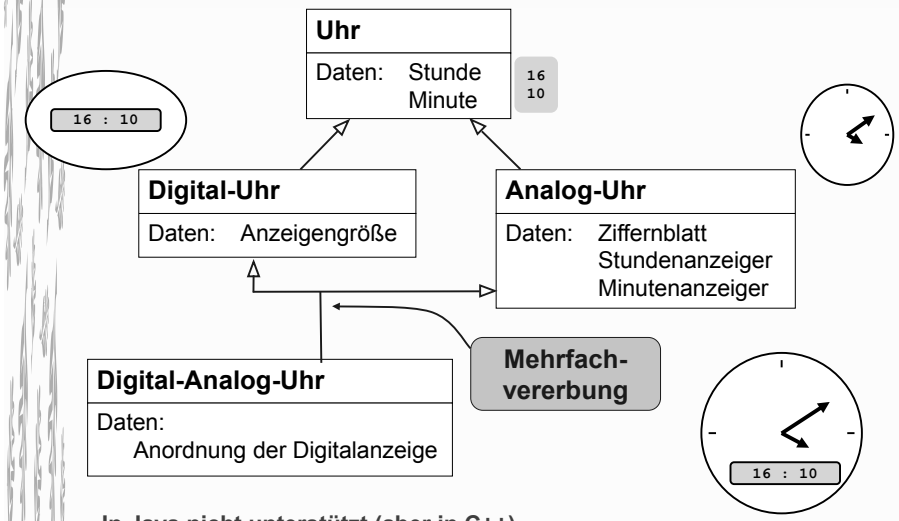


# Beispiel für Vererbung: Immobilienverwaltung (Maklerbüro)



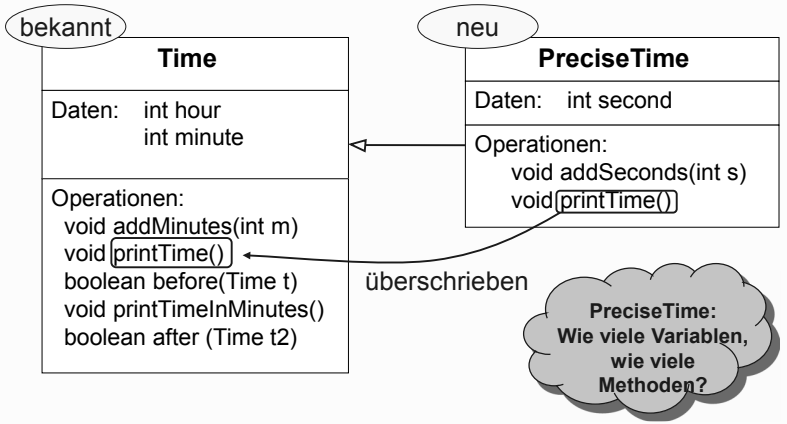



# Beispiel für (Mehrfach-)Vererbung: Uhr




**In Java nicht unterstützt (aber in C++)**

# Beispiel für Vererbung: präzise Zeit






# Vererbung in Java:

- extends
- protected
- super

**Bis jetzt: unabhängig von Programmiersprache, nur Architektur des Systems in UML beschrieben**



# Vererbung in Java

Basisklasse (unverändert)

Time2.java

```
class Time {
   private int hour, minute;
   public Time() ...
   public void addMinutes(...)
   public void printTime()
}
```

?

Vererbung

```
class PreciseTime extends Time {
   private int second;*)
   public PreciseTime(...) ...
   public void addSeconds(int s)*)
   public void printTime()
}
```

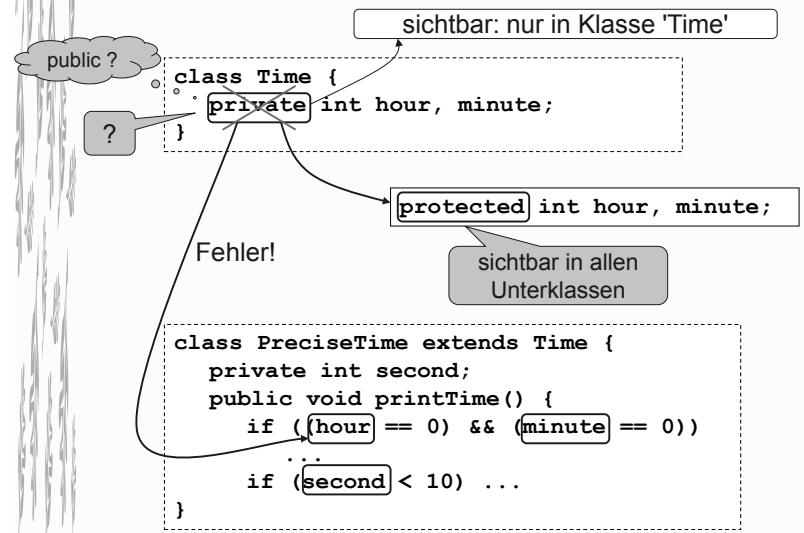
neuer Konstruktor

zum Überschreiben

*) neu



# Noch einmal: Sichtbarkeit



# Anwendung: abgeleitete Unterklasse

Time2.java

```
class Time2 {

  public static void main(String[] args) {
    PreciseTime lunchtime = new PreciseTime(12, 1, 0);

    lunchtime.addMinutes(1);
    lunchtime.printTime();
    lunchtime.addSeconds(-61);
    lunchtime.printTime();
    lunchtime.addSeconds(1);
    lunchtime.printTime();
  }
}
```

geerbt von Oberklasse

neu

überschrieben

```
% java Time2
12:02:00PM
noon
12:01:00PM
```



# super (1)

**Initialisierung mit Hilfe des Konstruktors der Oberklasse**

Time2.java

```
public PreciseTime(int h, int m, int s) {
  super(h, m); *)
  second = s;
}
```

Aufruf des Konstruktors der direkten Oberklasse: `Time (h, m)`

```
hour = h;
minute = m;
```

ohne *): (implizit) immer existierender 0-stelliger Konstruktor Time() der Oberklasse aufgerufen



# super (2)

**Verweis auf Methoden der direkten Oberklasse**

```
class Time {
  protected int hour, minute;
   . . .
  public void printTime () {…}
}
```

```
class PreciseTime extends Time {
  private int second;

  public void printTime () { …}

  public void printTimeShort () {
    super.printTime();
  }
}
```

Vollständige Ausgabe: ‘hour‘, ‘minute‘, ‘second‘

Ausgabe: nur ‘hour‘ und ‘minute‘

kein super.super …



# Ohne Vererbung?

Kommt man ohne Vererbung aus?

```
class Time { /* wie bisher */ }
```

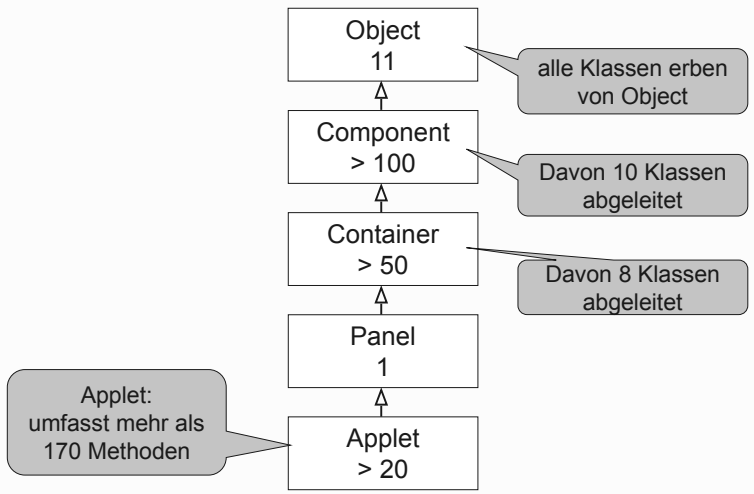
```
class PreciseTime {
  private int hour, minute;
  private int second;
  public PreciseTime(...) ...
  public addMinutes(...)
  public void addSeconds(...)
  public void printTime()
}
```

Konsequenzen?

- Beide Versionen äquivalent! (man kommt ohne Vererbung aus)
- Aber:
  - Doppelter Code:
  - Modifikationen komplizierter, falls zu ändernder Code doppelt (z.B. addminutes() in Oberklasse + abgeleiteten Klassen)
  - Keine logischen Beziehungen zwischen Klassen widergespiegelt

# API: erster Eindruck von Vererbungsbeziehungen

(Anzahl der Methoden)





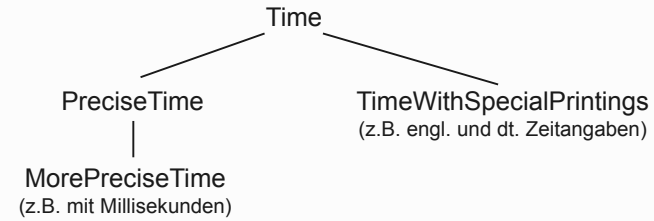

# Klassenhierarchie:

- Hierarchiebildung durch Vererbung
- Rolle von 'Object'



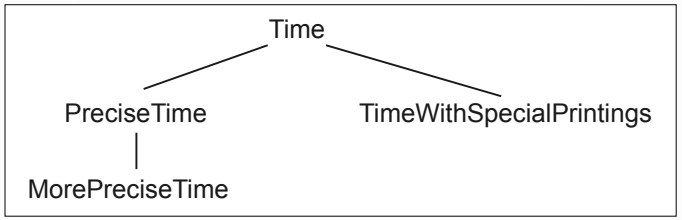
# Vererbungshierarchie

```
class Time ...

class PreciseTime extends Time ...

class MorePreciseTime extends PreciseTime ...

class TimeWithSpecialPrintings extends Time ...
```





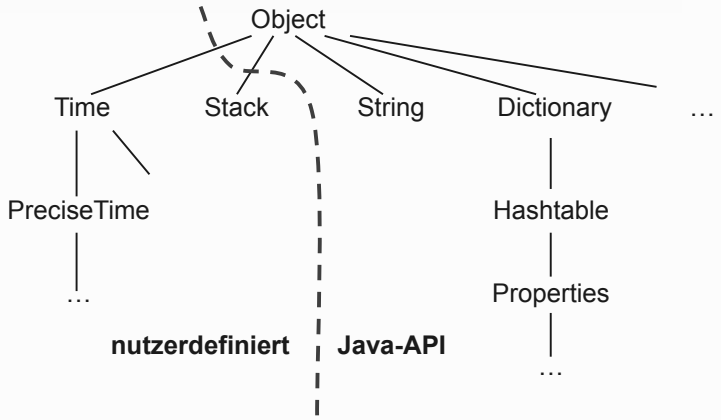

# Vererbungshierarchie: Ober- u. Unterklassen




**Oberklassen** von MorePreciseTime:
PreciseTime (***direkte*** Oberklasse)
Time

**Unterklassen** von Time:
PreciseTime (***direkte*** Unterklasse)
MorePreciseTime
TimeWithSpecialPrintings (direkte Unterklasse)

# Klasse 'Object': Oberklasse aller anderen Klassen




Object: Oberklasse aller anderen Klassen

→ Java-API



# API-Klasse java.lang.Object

java.lang
**Class Object**
java.lang.Object

public class **Object**

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

**Since:**
JDK1.0

**See Also:**
Class

**Constructor Summary**

| |
|---|
| Object() |

**Method Summary**

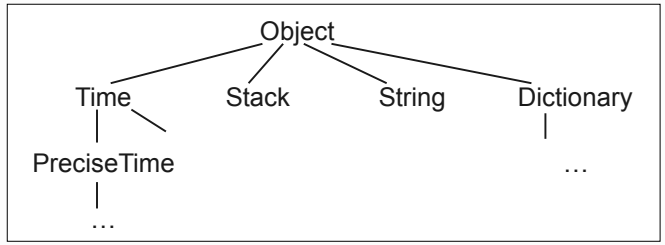
| | |
|---|---|
| protected Object | clone() Creates and returns a copy of this object. |
| boolean | equals(Object obj) Indicates whether some other object is "equal to" this one. |
| protected void | finalize() Called by the garbage collector on an object when garbage co |
| Class | getClass() Returns the runtime class of an object. |
| int | hashCode() Returns a hash code value for the object. |
| void | notify() Wakes up a single thread that is waiting on this object's moni |
| void | notifyAll() Wakes up all threads that are waiting on this object's monitor |
| String | toString() Returns a string representation of the object. |
| void | wait() Causes current thread to wait until another thread invokes the |
| void | wait(long timeout) Causes current thread to wait until either another thread invok |
| void | wait(long timeout, int nanos) Causes current thread to wait until another thread invokes the notify |

**Class** Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Schlussfolgerung:
Jede beliebige Klasse umfasst (erbt) auch diese 11 Methoden von Object:
- Standardimplementation (z.B. equals testet auf Identität der Objekte; clone kopiert ein Objekt) oder
- Überschreiben mit eigenem Algorithmus (z.B. equals testet of identischen Inhalt der Objekte)



# Vererbungshierarchie: Kompatibilitätsregeln



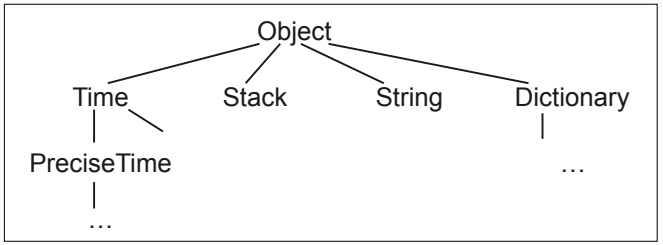

**Kompatibilitätsregel:**
**Objekte einer Unterklasse dürfen dort stehen, wo Objekte von Oberklassen zugelassen sind.**

```
Time t;
t = new PreciseTime(12, 10, 1);
t.printTime();
```

printTime() von PreciseTime



# Kompatibilität mit „Object"




**Konsequenz:**
**Variablen vom Typ 'Object' können Instanzen beliebiger Klassen als Wert besitzen**

```
Object o1, o2;
o1 = new PreciseTime(12, 10, 1);
o2 = new Stack(100);
```

Jede Klasse ist Unterklasse von Object

# Polymorphismus:

- Ähnliche Operationen
- Dynamisches Binden
- Final
- Abstrakte Klassen



# Polymorphismus: Grundidee

(Polymorphismus: "viele Erscheinungsformen")

```
Time t1;
PreciseTime t2;

t1 = new Time(12, 10);
t2 = new PreciseTime(0, 10, 1);
t1.printTime();
t2.printTime();
```

**Eine Methode ist polymorph,**
falls sie in verschiedenen Klassen die gleiche Signatur hat, jedoch jeweils neu implementiert ist.

**Literatur: unterschiedliche Definitionen:**

**Polymorphismus**:
Ähnliche Methoden mit derselben Signatur (Name, Parameterzahl, -typen)

**Polymorphismus**:
Methoden können in einer Vererbungshierarchie überschrieben werden

Vergleiche: *Overloading* = derselbe Name bei Methoden mit unterschiedlichen Parameterlisten, z.B. Time(), Time(hour, minute)



# Polymorphismus: Auswahl der richtigen Methode?

```
Time t1;
PreciseTime t2;

t1 = new Time(12, 10);
t2 = new PreciseTime(0, 10, 1);
t1.printTime();
t2.printTime();
```

Welches printtime()?

Auswahl der Methode abhängig vom Zielobjekt (z.B. t1, t2)

Wer wählt aus: Compiler o. Interpreter?

hier: Compiler

Ist das immer möglich?

nein: Klasse steht zur Compilationszeit nicht immer fest



# Dynamisches Binden: Methodensuche zur Laufzeit

```
Time t;
char ch = Keyboard.readChar();
if (ch == 'T')
   t = new Time(12, 10);
else
   t = new PreciseTime(12, 10, 1);
t.printTime();
```

Aus Klasse Time oder PreciseTime?

Wer wählt aus: Compiler o. Interpreter?

Compiler: kann es nicht entscheiden!

also: Auswahl der Methode erst zur Laufzeit – dynamisches Binden (kostet extra Laufzeit)

# Final: Einschränkung des Polymorphismus

- Methode darf in Unterklasse nicht überdefiniert werden

  → Idee: für stabile Semantik sorgen (Sicherheit)

```
class Time {
   public final void addMinutes(...) {
      ...
   }
   public void printTime()
}
```

- Klasse darf nicht erweitert werden
  z.B. Java-API

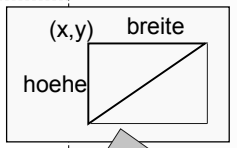
```
public final class String;
```

Klasse außerhalb des Paketes sichtbar, in dem sie definiert ist



# Abstrakte Klassen: enthalten Methoden ohne Implementation

```
abstract class Figur {
   protected int x, y, breite, hoehe;
   public void setzex(int xNeu) {
      x = xNeu;
   }
   // setzeY, setzeBreite, setzeHoehe ebenso

   public abstract float berechneFlaeche();
}
```




Offen gelassen: Dreieck, Rechteck, Ellipse

damit: Fläche hier unbekannt

‚berechne Flaeche': Methode ohne Implementation

→ Implementation muss in Unterklassen erfolgen

Figur → Rechteck, Dreieck, Ellipse …

→ keine Instanzenbildung der Klasse ‚Figur' erlaubt

→ Sinn: logische Strukturierung von Klassen

**Mögliche Aufgabe**: Man implementiere Dreieck, Rechteck, Ellipse ...