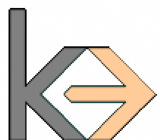
Kapitel 12

Vererbung



Fachgebiet Knowledge Engineering
Prof. Dr. Johannes Fürnkranz

TECHNISCHE
UNIVERSITÄT
DARMSTADT

12. Vererbung

1. Vererbung
2. Überschreiben und Überladen von Methoden
3. Polymorphie
4. Interne Realisierung
5. Vererbung und Zugriffsrechte
6. Die Klasse `java.lang.Object`

Erinnerung an KarelJ

- Eine Klasse kann eine andere erweitern.
- Die erstere Klasse kann dann die Funktionalität der zweiten Klasse quasi ”gratis“ mitbenutzen.
- Dass eine Klasse eine Erweiterung darstellt, wird mit Schlüsselwort **`extends`** angezeigt:

```
public class MeinRobot extends Robot  {
        ...
  }
```

(To extend = erweitern)

Vererbung von Datenkomponenten

Terminologie:

- Der ”Stamm“ der Erweiterungen heißt meist Basisklasse.
- Von einer ”Erweiterung“ einer Basisklasse sagen wir, sie sei von der Basisklasse abgeleitet.
- Wenn eine Klasse von einer anderen (Basis-)Klasse abgeleitet wird, dann ”erbt“ sie
  - deren Datenkomponenten.
  - deren Methoden

Eine abgeleitete Klasse darf natürlich ihrerseits als Basisklasse für weitere abgeleitete Klassen fungieren.

→ Die Klassen in der Java–Standardbibliothek sind in vielen, durchaus langen, sich verzweigenden Ketten von gegenseitigen Vererbungsbeziehungen organisiert.

Beispiel

```
public class BasisKlasse {
    public int n1;
}

class AbgeleiteteKlasse extends BasisKlasse {
    public int n2;
}

...

BasisKlasse x1 = new BasisKlasse();
AbgeleiteteKlasse x2 = new AbgeleiteteKlasse();

System.out.println ( x1.n1 );
System.out.println ( x2.n2 );
System.out.println ( x2.n1 );
```

`AbgeleiteteKlasse` erbt die Komponente `n1` von `Basisklasse`

die Komponente `n1` kann daher auch von Objekten der abgeleiteten Klasse verwendet werden.

Ketten von Vererbungsbeziehungen

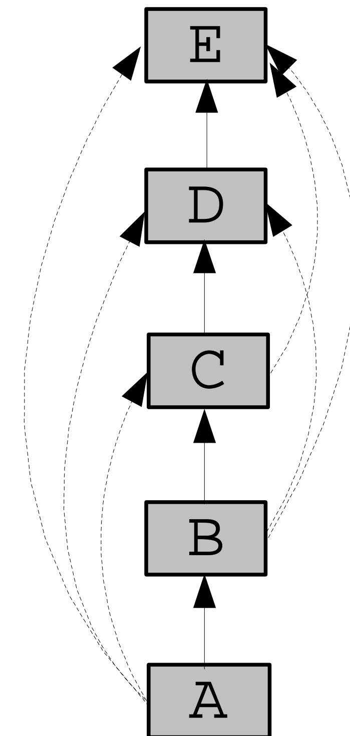
Damit ist gemeint, dass eine Klasse A von einer Klasse B abgeleitet ist, B wiederum von einer Klasse C, C von einer Klasse D usw.

Wir sagen, A ist von B, B von C usw. *direkt* abgeleitet.

Daraus ergeben sich auch indirekte Vererbungsbeziehungen:

- A ist von C,
- A ist von D,
- A ist von E,
- B ist von D,
- B ist von E,
- C ist von E

indirekt abgeleitet.



Vererbung von Methoden

```
public class BasisKlasse
{
    public void f1 () {
        System.out.println ( "f1" );
     }
}

class AbgeleiteteKlasse extends BasisKlasse
{
    public void f2 () {
        System.out.println ( "f2" );
    }
}

BasisKlasse x1 = new BasisKlasse();
AbgeleiteteKlasse x2 = new AbgeleiteteKlasse();

x1.f1 ();
x2.f2 ();
x2.f1 ();
```

Die Methode `f1` wurde von der Basis-Klasse ererbt und kann daher auch von Objekten der abgeleiteten Klasse verwendet werden.

Regeln zur Vererbung von Methoden

Jede Methode der Basisklasse ist automatisch auch eine Methode der abgeleiteten Klasse:

- exakt dieselbe Signatur (natürlich abgesehen vom Klassennamen),
- exakt dieselbe Implementation.

Einzige Ausnahme: Konstruktoren werden nicht in diesem Sinne vererbt.

→ Sind von ihrer Logik her zu eng mit ”ihrer“ Klasse verbunden.

12. Vererbung

1. Vererbung
2. Überschreiben und Überladen von Methoden
3. Polymorphie
4. Interne Realisierung
5. Vererbung und Zugriffsrechte
6. Die Klasse `java.lang.Object`

Überschreiben von Methoden

Wenn eine Methode in der Basisklasse vorhanden ist, ist sie auf jeden Fall auch in der abgeleiteten Klasse vorhanden.

Es gibt aber zwei Möglichkeiten, wie sie in der abgeleiteten Klasse vorhanden sein kann:

- von der Basisklasse vererbt (wie bisher betrachtet);
- in der abgeleiteten Klasse überschrieben.

Was heißt eine Methode überschreiben:

- Eine Methode mit exakt identischer Signatur(!) wie in der Basisklasse wird in der abgeleiteten Klasse noch einmal definiert.
- Beispiele:
  - hatten wir zahlreich in der KarelJ-Programmierung (z.B. Überschreiben von `move()`)

Überschreiben vs. Überladen

Überschrieben werden Methoden nur, wenn in der abgeleiteten Klasse die exakt gleiche Signatur verwendet wird.

Es ist durchaus auch möglich, den Namen einer Methode, die in der Basisklasse schon definiert wurde, in der abgeleiteten Klasse für eine weitere Methode *mit anderer Signatur* zu verwenden.

- Das ist dann aber keine Vererbung, sondern Überladung:
  - Eine neue Methode, die genauso heißt wie die alte

Einfache allgemeine Regel:

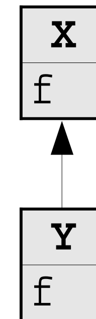
- Wenn zwei Methoden gleichen Namens unterschiedliche Signatur haben, handelt es sich *immer* um Überladung.

  →Die beiden Parameterlisten müssen wie immer unterschiedlich sein.
  (sonst wären die Signaturen identisch und die Methoden könnten nicht voneinander unterschieden werden)

Beispiel

```
public class BasisKlasse {
    public void f ( int i ) {        ← Originalmethode
        System.out.println ( "1" );
    }
}

public class AbgeleiteteKlasse extends BasisKlasse {
    public void f ( int j ) {        ← Überschreibung
        System.out.println ( "2" );
    }

    public void f ( double d ) {     ← Überlagerung
        System.out.println ( "3" );
    }
}

BasisKlasse x = new BasisKlasse();
AbgeleiteteKlasse y = new AbgeleiteteKlasse();

x.f (1);      // -> "1"
y.f (1);      // -> "2"
y.f (3.14);   // -> "3"
x.f (3.14);   // FEHLER: gibt keine
              // BasisKlasse.f(double)!
```

Schlüsselwort `super`



Betrachte wieder die allgemeine Situation, dass eine Methode `f` in einer Basis-Klasse `X` definiert und in einer von `X` abgeleiteten Klasse `Y` überschrieben ist.

- In dieser Methode von `Y` oder auch in anderen Methoden von `Y` möchte man aber vielleicht auch gerne `X.f` benutzen.
- Aber durch die Überschreibung ist in den Methoden von `Y` zunächst einmal nur `Y.f` ansprechbar.

  → Ähnlich wie bei `this`

Mit Schlüsselwort `super` kann man die Methoden der Basisklasse in den Methoden der abgeleiteten Klasse aufrufen.

→ Egal ob in der abgeleiteten Klasse überschrieben oder nicht.

Syntax

- wie bei `this`.
- Geschachtelte Aufrufe von Methoden indirekter Basisklassen über `super.super`, `super.super.super` usw. sind nicht erlaubt.

Beispiel

```
public class X
 {
   public void f () { ... }
 }

public class Y extends X
 {
   public void f ()
    {
      super.f();
    }

   public void g ()
    {
      super.f();
      f();
    }
 }
```

Konstruktoren der Basisklasse

Erinnerung: Konstruktoren werden nicht vererbt.

- Trotzdem möchte man im Konstruktor der abgeleiteten Klasse oft auch den Konstruktor der Basisklasse aufrufen, um den Anteil der Basisklasse am Gesamtobjekt mit einem dafür schon vorgesehenen Konstruktor zu initialisieren.
- Mit Schlüsselwort `super` kann man einen Konstruktor der Basisklasse in einem Konstruktor der abgeleiteten Klasse aufrufen.
- Verwendung analog zur Verwendung von `this`

Verwendung von `super`

Nur die allererste Zeile in einem Kontruktor darf der Aufruf eines anderen Konstruktors mit `this` oder `super` sein.

→ Insbesondere darf es in jedem Konstruktor nur einen einzigen Aufruf eines anderen Konstruktors geben.

Dadurch wird dafür gesorgt, dass immer ein Konstruktor aufgerufen wird

- Wenn die direkte Basisklasse keinen Konstruktor mit leerer Parameterliste hat (weder selbst geschrieben noch vom Compiler hinzugedacht), dann muss ein Konstruktor mit `super` aufgerufen werden (mit entprechenden Parametern).
- Wenn die direkte Basisklasse einer Klasse `X` einen Konstruktor mit leerer Parameterliste hat und die erste Zeile eines Konstruktors von `X` kein Konstruktoraufruf ist, wird der leere Konstruktor der Basisklasse implizit am Anfang aufgerufen.

Beispiel

Der Konstruktor für Cabrio ist nicht zulässig

- es gibt eine Compiler-Fehler-Meldung

Diese Version würde aber auch keinen Sinn machen:

- Zuerst wird die Anzahl der Türen auf auf 2 gesetzt
- danach wieder durch den Konstruktor der Überklasse auf 4 gesetzt

→ Konstruktoren der Überklasse müssen *immer* vor den Konstruktoren der Unterklasse aufgerufen werden (d.h. in die erste Zeile geschrieben werden)

```
public class Auto {
   int anzahlTueren;

    public Auto() {
       anzahlTueren = 4;
    }
  }

public class Kombi extends Auto {
    public Kombi()
      {
         super(); // OK!
         anzahlTueren = 5;
      }
  }

public class Cabrio extends Auto {
    public Cabrio()
      {
         anzahlTueren = 2;
         super(); // -> Fehler!
      }
  }
```

12. Vererbung

1. Vererbung
2. Überschreiben und Überladen von Methoden
3. Polymorphie
4. Interne Realisierung
5. Vererbung und Zugriffsrechte
6. Die Klasse `java.lang.Object`

Polymorphie

- Eine Variable vom Typ der Basisklasse kann also auch auf ein Objekt vom Typ einer davon abgeleiteten Klasse verweisen:

```
AbgeleiteteKlasse x = new AbgeleiteteKlasse();
BasisKlasse y = x;
```

- Diese beiden Anweisungen lassen sich selbstverständlich auch zu einer Anweisung zusammenfassen:

```
BasisKlasse x = new AbgeleiteteKlasse();
```

- Das macht Sinn, da Objekte der abgeleiteten Klasse alle Fähigkeiten haben, die die Objekte der Basisklasse haben.

- Das heißt: alle Methodenaufrufe, die für Objekte der Basis-Klasse definiert sind, sind auch für Objekte der abgeleiteten Klasse definiert.

Beispiel

```
public class BasisKlasse {
    public void f () {
        System.out.println ( "Basisklasse" );
    }
}

public class AbgeleiteteKlasse extends BasisKlasse {
    public void f () {
        System.out.println ( "Abgeleitete Klasse" );
    }
}
....
// Verwendungsbeispiel
BasisKlasse x1          = new BasisKlasse();
AbgeleiteteKlasse x2 = new AbgeleiteteKlasse();
BasisKlasse x3          = new AbgeleiteteKlasse();

x1.f (); // -> "BasisKlasse.f()"
x2.f (); // -> "AbgeleiteteKlasse.f()"
x3.f (); // -> "AbgeleiteteKlasse.f()" !!!
```

Statischer (Formaler) und Dynamischer (Aktualer) Typ

Um Zuweisungen der Form

```
X x = new Y (...);
```

zu ermöglichen, hat das Objekt, auf das die Referenz `x` zeigt, also nicht unbedingt denselben Typ bei der Deklaration angegeben.

Bei einer Variablen wie `x`, die von einem Klassentyp ist, müssen wir daher sorgfältig zwischen zwei Typen unterscheiden:

- *statischer (formaler)* Typ:
  - der Typ der *Variablen* (also `X`)
- *dynamischer (aktualer)* Typ:
  - dem Typ des *Objekts* (also `Y`).

```
X x = new Y (...);
```

Bei einer Zeile der Form

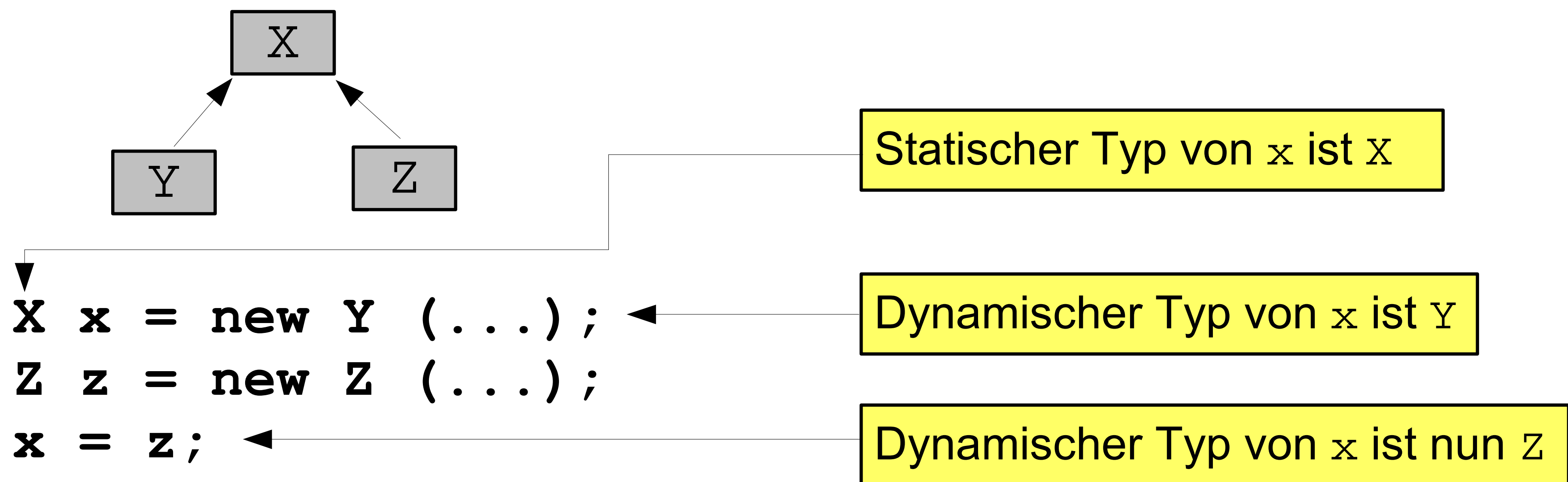
```
X x = new X (...);
```

sind dynamischer Typ und statischer Typ von `x` identisch.

Statisch und Dynamisch

- Der statische Typ einer Variablen bleibt immer gleich.
- Der dynamische Typ kann sich jederzeit ändern.

→ Einfach durch Zuweisung eines neuen Objekts an die Variable wird der Typ des neuen Objekts der neue aktuale Typ:



Beziehung Statischer und Dynamischer Typ

Es gibt nur zwei Möglichkeiten für eine Variable von einem Klassentyp:

- Der statische und der dynamische Typ sind identisch.
- Der dynamische Typ eine direkte oder indirekte Unterklasse des statischen Typs

Warum diese Einschränkung:

- Der statische Typ fungiert als ”Fassade“, hinter dem Objekte von allen möglichen dynamischen Typen stecken können.
- Dazu ist es natürlich notwendig, dass der dynamische Typ alles kann, was der statische Typ ”verspricht“.
- Das ist am einfachsten und elegantesten eben durch diese Einschränkung gewährleistet.

Rollenverteilung

Welche Methoden mit einer Variablen eines Klassentyps aufgerufen werden dürfen, hängt von seinem *statischen* Typ ab.

- Falls eine Methode für Basisklasse und abgeleitete Klasse(n) implementiert ist:
  → Dann wird die Implementation des dynamischen Typs aufgerufen.

- Falls eine Methode nur für die Basisklasse, aber nicht für die abgeleitete Klasse implementiert ist:
  → Dann wird die Implementation der Basisklasse auf die abgeleitete Klasse vererbt.
  → Dann ist die Implementation der Basisklasse in jedem Fall auch die Implementation des dynamischen Typs.

Beispiel

```
public class BasisKlasse {
   public void f1 ()
     {
        System.out.println ( "B.f1" );
     }
 }

public class AbgeleiteteKlasse extends BasisKlasse {
   public void f1 () {      // Ueberschrieben
        System.out.println ( "A.f1" );
     }
   public void f2 () {      // Neu in abgeleiteter Klasse
        System.out.println ( "A.f2" );
     }
 }
...

BasisKlasse x = new AbgeleiteteKlasse();

x.f1();
x.f2();
```

Verboten: `f2` ist zwar für diesen speziellen dynamischen Typ (`AbgeleiteteKlasse`) definiert, aber nicht für den statischen Typ (`BasisKlasse`)

Kann der Compiler nicht den dynamischen Typ erkennen?

Warum dürfen denn eigentlich nur die Methoden aufgerufen werden, die für den *statischen* Typ schon definiert sind?

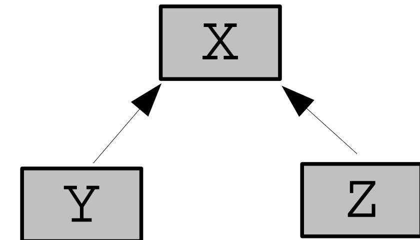
- Der Compiler kann doch den dynamischen Typ ”erkennen“ und dementsprechend auch alle Methoden erlauben, die erst für den *dynamischen* Typ definiert sind, oder?

Einfache Antwort: <u>Nein!</u>

- Es stimmt leider nicht immer, dass der Compiler den dynamischen Typ einer Variablen erkennen kann.
- Der dynamische Typ einer Variablen kann von Informationen abhängen, die erst zur Laufzeit des Programms zur Verfügung stehen (und sich auch von Programmlauf zu Programmlauf ändern können)

Beispiel

```
public class X { ... }

public class Y extends X { ... }

public class Z extends X { ... }
...

boolean b;

b = ...

X x;
if ( b ) {
    x = new Y ();
}
else {
    x = new Z ();
}
```



Der Wert von b kann von verschieden Faktoren abhängen, z.B. von Benutzereingaben

Der dynamische Typ von `x` kann daher von Programmlauf zu Programmlauf variieren und daher prinzipiell nicht vom Compiler erkannt werden.

Down-Cast

- *Erinnerung*:
  - Unsichere Konversionen zwischen eingebauten Typen sind möglich.
  - Man muss den Zieltyp dann in runden Klammern vor den zu konvertierenden Ausdruck schreiben.
    z.B. `float pi = (float) 3.14159;`
- Man kann auch von einer Basisklasse auf eine abgeleitete Klasse "herunter" konvertieren (engl. *down–cast*).
- Dies ist ebenfalls eine unsichere Konversion:
  - Wenn der dynamische Typ des Objekts gleich dem Zieltyp oder vom Zieltyp abgeleitet ist, geht alles gut.
  - Wenn nicht: Programmabsturz, da das Objekt nicht die Anforderungen erfüllt, die der Zieltyp haben muß.
- Da diese Art von Konversion generell unsicher ist, muss man eben wieder den Zieltyp in runden Klammern vor die zu konvertierende Variable schreiben.

Beispiel

```
public class BasisKlasse { ... }

public class AbgeleiteteKlasse1 extends BasisKlasse
 {
   public void f () { ... }
 }

public class AbgeleiteteKlasse2 extends BasisKlasse { ... }

...

BasisKlasse x = new AbgeleiteteKlasse1 ();
x.f();

AbgeleiteteKlasse1 y = (AbgeleiteteKlasse1) x;
y.f();

AbgeleiteteKlasse2 z = (AbgeleiteteKlasse2) x;
```

Statischer Typ: `BasisKlasse`
Dynamischer Typ: `AbgeleiteteKlasse1`

nicht o.k., da `x` vom Typ `BasisKlasse`

Down-Cast

o.k., da `y` vom Typ `Abgeleitetelasse1`

Fehler: Down-Cast auf anderen Typ nicht möglich

Erläuterungen

- Der Quelltext lässt sich nach Entfernen von `x.f();` problemlos kompilieren.
- Aber beim Laufenlassen stürzt das Programm in der letzten Zeile, in der `z` eingerichtet wird, ab mit der Fehlermeldung:

```
java.lang.ClassCastException
```

- Die Zeile, in der `y` eingerichtet wird, ist absolut korrekt:
  - ◊ Die Variable `y` verweist hinterher auf dasselbe Objekt wie `x`.
  - ◊ Der dynamische und der statische Typ von `y` sind dann gleich.

Allgemeine Regel für Down-Casts

Ein Down–Cast

```
AbgeleiteteKlasse x = (AbgeleiteteKlasse) y;
```

ist genau dann ok, wenn der dynamische Typ von `y`

- entweder `AbgeleiteteKlasse` selbst oder
- direkt oder indirekt von `AbgeleiteteKlasse` abgeleitet

ist.

Schlüsselwort `instanceof`

Bevor man einen unsicheren Down–Cast macht — und dadurch einen Programmabsturz riskiert — kann man in Java abprüfen, ob der Down–Cast ok ist.

- Dafür gibt es in Java das Schlüsselwort `instanceof`.

Verwendung:

- Der logische Ausdruck

  `x instanceof Y`

  liefert `true` genau dann, wenn der dynamische Typ von `x` entweder gleich `Y` ist oder direkt oder indirekt von `Y` abgeleitet ist.
- Mit anderen Worten: genau dann, wenn der Down–Cast ok ist.

```
Y y;
if ( x instanceof Y )
  y = (Y) x;
else
  System.out.println ( "Falscher Typ!" );
```

Was ist nun Polymorphie?

Name:

- ”Poly“ ≈ viel
- ”Morphe“ ≈ Gestalt

Idee:

- Eine Variable der Basisklasse (z.B. `Robot`) fungiert als eine einheitliche Fassade.
- Dahinter können sich Objekte von unterschiedlichen (abgeleiteten) Klassentypen verbergen
  - zum Beispiel `MeinRobot1, MeinRobot2,` etc.
- Dieselbe Methode kann nun in den verschiedenen abgeleiteten Klassen unterschiedlich implementiert sein
  - z.B. `MeinRobot1.move(), MeinRobot2.move(),` etc.

→ Ein Methodenaufruf vermittels dieser Fassade kann je nach Typ des dahinterstehenden Objekts völlig unterschiedliche Effekte erzeugen.

12. Vererbung

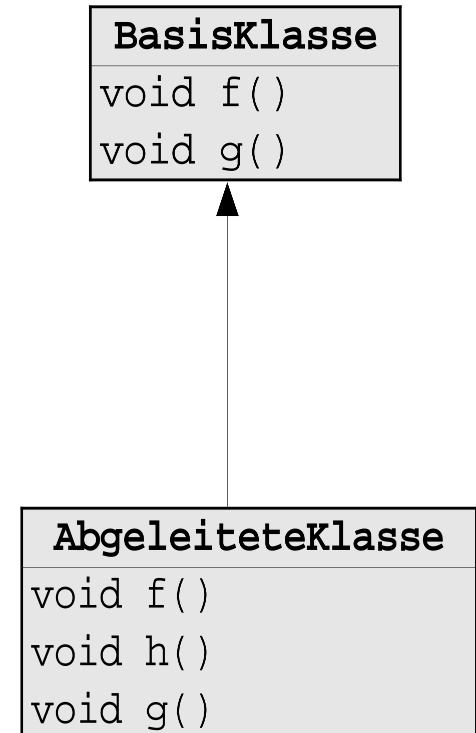
1. Vererbung
2. Überschreiben und Überladen von Methoden
3. Polymorphie
4. Interne Realisierung
5. Vererbung und Zugriffsrechte
6. Die Klasse `java.lang.Object`

Interne Realisierung der Vererbung

- Jedes Objekt einer Klasse enthält eine weitere, ”unsichtbare“ Datenkomponente.
- In dieser Datenkomponente ist der Typ des Objekts kodiert.
- Zu jeder Klasse wird separat irgendwo im Speicher einmal eine Tabelle angelegt, in der zu jeder Objektmethode der Klasse die Startadresse gespeichert ist.
- Falls eine Objektmethode in der Klasse nicht selbst implementiert ist, sondern von der Basisklasse vererbt wird, dann wird die Startadresse der ererbten Methode dort überschrieben.

Beispiel

```
public class BasisKlasse {
    public int i;

    public void f () {
        System.out.println ( "1" );
    }

    public void g () {
        System.out.println ( "2" );
    }
}

public class AbgeleiteteKlasse extends BasisKlasse {
    public double d;

    public void f () {
        System.out.println ( "3" );
    }

    public void h () {
        System.out.println ( "4" );
    }
}
    ...
BasisKlasse          x = new BasisKlasse();
BasisKlasse          y = new AbgeleiteteKlasse();
AbgeleiteteKlasse    z = new AbgeleiteteKlasse();
```



Schematische Darstellung

Aufruf einer Methode

Jeder Aufruf einer Objektmethode im Java–Quelltext wird vom Compiler in Code übersetzt, der

- zuerst die zusätzliche Datenkomponente mit der Typinformation liest,
- damit die Speicheradresse der zu diesem Klassentyp zugehörigen Tabelle von Methodenadressen heraussucht,
- in dieser Tabelle dann den Eintrag der aufgerufenen Methode nachschlägt und
- einen Methodenaufruf mit der dort gefundenen Startadresse ausführt.

→ Erst im letzten dieser vier Schritte werden die Argumente des und die Rücksprungadresse auf den Run-Time-Stack gelegt

12. Vererbung

1. Vererbung
2. Überschreiben und Überladen von Methoden
3. Polymorphie
4. Interne Realisierung
5. Vererbung und Zugriffsrechte
6. Die Klasse `java.lang.Object`

Vererbung und Zugriffsrechte

Bisher konnten die Methoden einer Klasse alle Datenkomponenten und Methoden dieser Klasse benutzen.

- Insbesondere konnte eine Klasse auf alle ererbten Datenkomponenten zugreifen
- Manchmal ist das aber nicht sinnvoll
  - z.B. für eine Klasse `Konto` ist es eventuell nicht sinnvoll, abgeleiteten Klassen einen direkten Zugriff auf den Kontostand zu ermöglichen, sondern nur über Methoden wie `einzahlung` und `auszahlung`, die auch den entsprechenden Geldverkehr sicherstellen.

→ Solche Datenkomponenten (oder Methoden) kann man mit dem Schlüsselwort `private` vor Zugriff in den Unterklassen schützen

Die Methoden einer abgeleiteten Klasse dürfen `private`–deklarierte Datenkomponenten und Methoden, die sie von einer Basisklasse ererbt haben, nicht verwenden.

- Zugriff wird nur für die Klasse, in der die Datenkomponente bzw. Methode eingeführt wurde erlaubt.

Beispiel

```
public class MeineKlasse
 {
   public  int n1;
   private int n2;

   public  void f1 () { ... }
   private void f2 () { ... }
 }


// Verwendung in einer anderen Klasse:
MeineKlasse meinObjekt = new MeineKlasse();

meinObjekt.n1 = 1;      // Erlaubt  (n1 ist public)
meinObjekt.n2 = 1;      // Verboten (n2 ist private)
meinObjekt.f1();        // Erlaubt  (f1 ist public)
meinObjekt.f2();        // Verboten (f2 ist private)
```

Weitere Zugriffsrechte

Eine Datenkomponente oder Methode

- darf in den Methoden jeder beliebigen Klasse angesprochen werden, wenn sie durch **`public`** gekennzeichnet ist,
- nur in den Methoden derselben Klasse, wenn sie durch **`private`** gekennzeichnet ist, und
- in den Methoden aller Klassen desselben Packages, wenn sie gar nicht gekennzeichnet ist.
- nur von den Methoden von `X` selbst, und auch von den Methoden aller direkt und indirekt von `X` abgeleiteten Klassen, aber nicht von den Methoden irgendeiner anderen Klasse, wenn sie durch **`protected`** gekennzeichnet ist

Übersicht:

- Methode/Datenkomponente... ...deklariert als

...sichtbar in

	`public`	`protected`	Standard	`private`
Selbe Klasse	ja	ja	ja	ja
Selbes Paket	ja	ja	ja	nein
Unterklasse	ja	ja	nein	nein
Beliebige Klasse	ja	nein	nein	nein

Beispiel

```
public class A {
        private   int i;
        protected int j;
        public    int k;
}

public class B {
    public void f () {
        A x = new A();
        System.out.print(x.i);     // Verboten (private)
        System.out.print(x.j);     // Verboten (protected, B keine Unterklasse)
        System.out.print(x.k);     // Erlaubt  (public)
    }
}

public class C extends A {
    public void f () {
        System.out.print(i);       // Verboten (private)
        System.out.print(j);       // Erlaubt  (protected, C ist Unterklasse)
        System.out.print(k);       // Erlaubt  (public)
    }
}
```

12. Vererbung

1. Vererbung
2. Überschreiben und Überladen von Methoden
3. Polymorphie
4. Interne Realisierung
5. Vererbung und Zugriffsrechte
6. Die Klasse `java.lang.Object`

Klasse `Object`

Regel:

Wenn eine Klasse

- nicht mit `extends` *ex*plizit von einer anderen Klasse abgeleitet ist,
- ist sie *im*plizit von der Klasse `java.lang.Object` abgeleitet.

Einzige Ausnahme:

natürlich `java.lang.Object` selbst.

Methoden der Klasse `Object`

Die Klasse `java.lang.Object` enthält eine ganze Reihe von Methoden, zum Beispiel:

- `boolean equals (Object obj)`
  - stellt fest ob das Object gleich einem anderen Object obj ist
  - die ererbte Gleichheit kontrolliert, ob auf den selben Speicherbereich gezeigt wird
- `String toString ()`
  - verwandelt das Object in einen String
  - der Name der Klasse und eine technische Zusatzinformation wird in einem einzigen String zusammengefasst.

Da jede Klasse von Object erbt, stehen diese Methoden in jeder Klasse zur Verfügung

- mit genau dieser Semantik
- Man kann sie jedoch überschreiben, um eine andere Bedeutung zu realisieren

Beispiel

```
public class ImaginaerZahl
 {
    private double real;
    private double imag;

   public boolean equals ( Object obj )
    {
      ImaginaerZahl x = (ImaginaerZahl) obj;
      return real == x.real && imag == x.imag;
    }

   public String toString () {
      return new String(real + " + " + imag + "i");
    }
  }
```

explizites Type-Cast notwendig, da die Signatur der Methode ja eine Vergleich mit `Object` verlangt!

Zwei Objekte vom Typ `ImaginaerZahl` sind gleich, wenn ihre Datenkomponenten `real` und `imag` gleich sind.

Ein `ImaginaerZahl` Objekt soll durch “real + i imag” beschrieben werden

Beispiel: `java.util.Vector`

Die Klasse `java.util.Vector` realisiert im Prinzip ein Array, in dem man auch einfügen und löschen kann.

- Ist aber eine ganz normale Klasse.

Dilemma:

- In Methoden wie `add` und `elementAt` muss man den Elementen des Vektors ja irgendeinen Typ geben.
- Aber eigentlich wollen wir ja wie bei Arrays Vektoren für alle möglichen Typen haben.

Lösung in Java: Verwenden von `java.lang.Object`.

- Objekte werden zwar mit ihrem dynamischen Typ abgespeichert
- Deklaration der Methoden verwendet nur den statischen Typ `Object`

Anmerkung: Java 1.5 bietet eine einfachere Lösung (→ Generics)

Beispiel auf der nächsten Folie zeigt auch, dass Wrapper–Klassen wie `Integer` durchaus wichtig sind.

- `Integer` ist eine Klasse für int-Zahlen (analog gibt es `Double`, `Char`, etc.)

Beispiel für Verwendung von `Object`

```
import java.util.*;

Vector v = new Vector();

for ( int i=0; i<4; i++ )
 {
    Integer x = new Integer ( i * i );
    v.add (x);
 }

for ( int i=0; i<v.size(); i++ )
 {
    Integer x = (Integer)(v.elementAt(i));
    System.out.print ( x.intValue() );
 } // Gesamtausgabe der Schleife: "0149"

v.remove(2);

for ( int i=0; i<v.size(); i++ )
 {
    Integer x = (Integer) (v.elementAt(i));
    System.out.print ( x.intValue() );
 } // Gesamtausgabe der Schleife: "019"
```

Importieren von `java.util.Vector`

definiert eine neues `Vector` Objekt (es muß keine Größe angegeben werden)

Die Zahlen `0, 1, 4, 9` werden nacheinander in den `Vector v` eingefügt

`elementAt` retourniert ein Objekt vom generischen Typ `Object`. Daher muß eine explizite Rück-Konvertierung zu `Integer` erfolgen!

Löschen des Elements mit dem Index 2