Objektorientierte Programmierung

Kapitel 12: Subklassen/Vererbung

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2018/19

http://www.informatik.uni-halle.de/~brass/oop18/

Inhalt

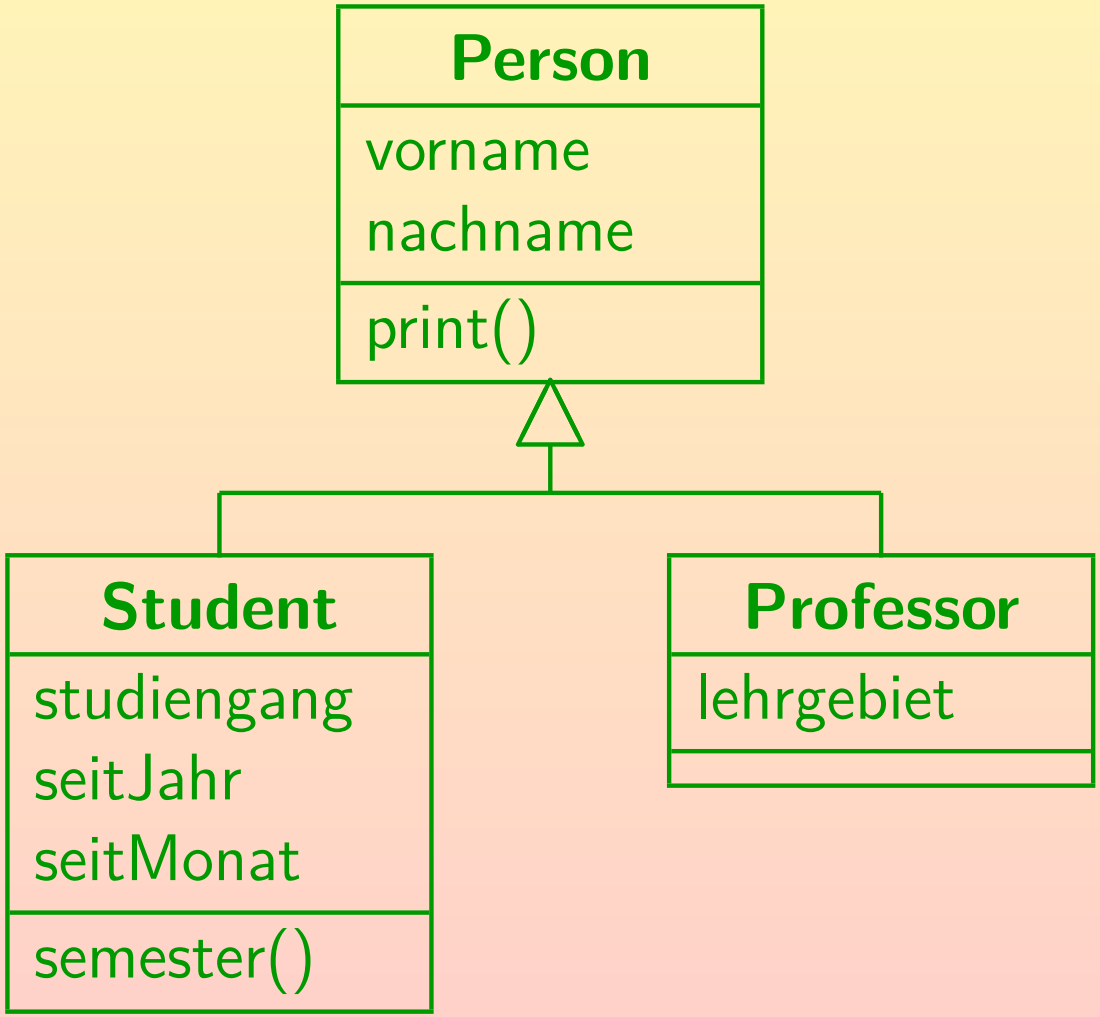
Subklassen (1)

- Oft ist ein Typ von Objekten eine spezielle Art (Spezialfall) eines anderen Typs von Objekten.
- Z.B. könnte man eine Klasse `Person` für Personen an der Universität deklarieren.
  - Diese haben u.a. die Attribute `vorname`, `nachname`.
- Studenten (Klasse `Student`) sind spezielle Personen.
  - Sie haben alle Attribute von `Person`,
  - aber zusätzlich auch Attribute `studiengang` und `seitJahr`, `seitMonat` (Einschreibe-Datum, Studienbeginn).

Subklassen (2)

- Professoren (Klasse `Professor`) sind eine andere Art von Personen an der Universität.
- Die Klasse `Person` könnte eine Methode `print()` haben zur Ausgabe von Vorname und Nachname.
  - Diese Methode ist genauso auf Objekte der Klassen `Student` und `Professor` anwendbar.
- Die Klasse `Student` könnte eine Methode `semester()` haben, die das Semester aus dem Studienbeginn berechnet.
  - Diese Methode ist nicht auf Professoren anwendbar (sie haben keinen Studienbeginn).

Subklassen (3)



Subklassen (4)

- `Student` und `Professor` sind Unterklassen / Subklassen der Oberklasse / Superklasse `Person`.
- Die Klassen `Student` und `Professor` erweitern (engl. “extend”) die Klasse `Person`.

  Sie führen zusätzliche Attribute und Methoden ein.

  Bei C++ nennt man die Unterklasse (wie `Student`) auch abgeleitete Klasse (engl. “derived class”) der Basisklasse (engl. “base class”) `Person`.
- Zwischen Unterklasse und Oberklasse besteht eine “ist-ein” (engl. “is a”) Beziehung: Ein Student ist eine Person.

  Entsprechend für Professoren.
- D.h.: Die Menge der Studenten (Unterklasse) ist eine Teilmenge der Personen (Oberklasse).

Subklassen (5)

- Man nennt diese Beziehung auch eine
  - Spezialisierung (von `Person` zu `Student`, ein Student ist eine spezielle Person), bzw.
  - Generalisierung (von `Student` zu `Person`).
- Man betrachtet hier die gleichen Objekte der realen Welt, z.B. die Studentin "Lisa Weiss" auf verschiedenen Abstraktionsebenen:
  - Mal als Studentin (mit Studiengang etc.),
  - mal nur als beliebige Person: Hier sind Details wie der Studiengang weggelassen.

Subklassen (6)

- Eine Subklasse "erbt" (engl. "inherits") alle Komponenten (Attribute und Methoden) von der Oberklasse.

  Das ist etwas vereinfacht: Manches wird überschrieben oder ist in der Unterklasse nicht direkt zugreifbar (siehe genauere Erklärung unten).

- Man braucht für Studenten also nicht explizit zu deklarieren,
  - dass sie einen Vornamen und einen Nachnamen haben,
  - und wie man den vollständigen Namen ausdruckt.
- Man kann Objekte der Unterklasse auch überall verwenden, wo ein Objekt der Oberklasse verlangt wird (Substitutionsprinzip):
  - Die Unterklasse ist ja eine Teilmenge der Oberklasse.

Subklassen (7)

- Z.B. könnte es eine Klasse `Login` für Accounts (Benutzerkonten) auf Rechnern geben.
- Die Klasse enthält ein Attribut `Inhaber` vom Typ `Person`.

  Studenten, Professoren u.a. Personen an der Universität haben Logins.
- Die in diesem Attribut gespeicherte Referenz kann insbesondere auch auf ein Objekt der Klasse `Student` zeigen.
- Man unterscheidet hier
  - den statischen Typ des Attributes (im Programm deklariert, zur Compilezeit bekannt): `Person`, und
  - den dynamischen Typ des zur Laufzeit tatsächlich gespeicherten Wertes: `Student`.

    Der dynamische Typ muss immer ein Untertyp des statischen Typs sein.

Subklassen (8)

- Eine Unterklasse kann selbst wieder Oberklasse für eine andere Unterklasse sein.
- Z.B. ist ein Gastprofessor ein spezieller Professor.

  Als zusätzliche Eigenschaft hat er z.B. das Datum, bis zu dem er an der Uni ist.
- So kann eine ganze Hierarchie von Subklassen entstehen.

  Auch `Gastprofessor` ist eine Subklasse von `Person`.

  Wenn man es besonders klar machen will, kann man von indirekter Subklasse sprechen, während `Professor` eine direkte Subklasse ist.
- Zyklen in der "ist Unterklasse von"-Beziehung sind verboten.

  Eine Klasse kann nicht ihre eigene direkte oder indirekte Subklasse sein.
- In Java steht ganz oben in der Hierarchie (Wurzel des Baums) die vordefinierte Klasse `Object` (s.u.).

Motivation für Subklassen (1)

- Es ist immer gut, mit der Typstruktur des Programms dicht an der realen Welt zu bleiben, und keine künstlichen Verrenkungen machen zu müssen.

  Leichteres Verständnis, weniger Überraschungen.

- Man spart Tippaufwand: `vorname`, `nachname` und `print()` hätte man sonst für `Student`, `Professor`, und weitere Arten von Personen einzeln definieren müssen.

- Es wäre auch grundsätzlich schlecht, die gleichen Dinge mehrfach aufschreiben zu müssen.

  Man muss nicht nur den Tippaufwand rechnen (den man sich mit Copy&Paste vereinfachen kann), sondern auch den Aufwand, das Programm zu lesen und zu verstehen (z.B. für neue Team-Mitglieder und auch den ursprünglichen Programmierer nach einem Jahr).

Motivation für Subklassen (2)

- Falls man etwas für alle Personen ändern möchte (z.B. Ausgabe-Format "Nachname, Vorname" statt "Vorname Nachname"), muss man diese Änderung nur an einer Stelle durchführen.

  Sonst größerer Aufwand, Gefahr von Inkonsistenzen.

- Ohne Subklassen würde man bei den Logins mehrere Attribute brauchen (eins für jeden Typ), um auf Studenten und Professoren als Inhaber verweisen zu können.

  Plus entsprechende Fallunterscheidungen in der Verarbeitung.

- Wenn eine neue Art von Personen hinzukommt, müsste die Klasse `Login` wieder geändert werden.

Inhalt

Subklassen: Beispiel (1)

```
(1)  class Person {
(2)
(3)      // Attribute:
(4)      private String vorname;
(5)      private String nachname;
(6)
(7)      // Konstruktor:
(8)      Person(String v, String n) {
(9)          vorname = v;
(10)         nachname = n;
(11)     }
(12)
(13)     // Zugriffs-Methoden:
(14)     String vorname()  { return vorname;  }
(15)     String nachname() { return nachname; }
(16)
```

Subklassen: Beispiel (2)

```
(17)      // Methode zum Drucken des Namens:
(18)      void print() {
(19)          System.out.print(
(20)              vorname + " " + nachname);
(21)      }
(22)
(23)  } // Ende der Klasse Person
(24)
```

Hier wurde für die von außen zugreifbaren Methoden wieder die Standard-Sichtbarkeit (nur aus dem Paket) gewählt. Selbstverständlich könnte man Konstruktor, Methoden und die Klasse selbst auch mit "`public`" kennzeichnen und so noch mehr Zugriffe erlauben. Dies wird genauer in Kapitel 15 diskutiert.

Subklassen: Beispiel (3)

```
(25)  class Student extends Person {
(26)
(27)      // Zusaetzliche Attribute:
(28)      private String studiengang;
(29)      private int seitJahr; // Studienbeginn
(30)      private int seitMonat;
(31)
(32)      // Konstruktor:
(33)      Student(String vname, String nname,
(34)          String studiert, int jahr, int monat) {
(35)          super(vname, nname);
(36)          this.studiengang = studiert;
(37)          this.seitJahr = jahr;
(38)          this.seitMonat = monat;
(39)      }
(40)
```

Subklassen: Beispiel (4)

```
(41)      // Zugriffsfunktion fuer Attribut:
(42)      String studiengang() {
(43)          return studiengang;
(44)      }
(45)
(46)      // Methode zur Berechnung des Semesters:
(47)      int semester() {
(48)
(49)          // Zuerst aktuelles Datum beschaffen:
(50)          java.util.Calendar heute =
(51)              java.util.Calendar.getInstance();
(52)          int jahr  = heute.get(
(53)                  java.util.Calendar.YEAR);
(54)          int monat = heute.get(
(55)                  java.util.Calendar.MONTH) + 1;
(56)
```

Subklassen: Beispiel (5)

```
(57)            int sem = (jahr - seitJahr) * 2 - 1;
(58)            if(monat >= 4)
(59)                sem++;
(60)            if(monat >= 10)
(61)                sem++;
(62)            if(seitMonat < 10)
(63)                sem++;
(64)            if(seitMonat < 4)
(65)                sem++;
(66)            return sem;
(67)        }
(68)    } // Ende der Klasse Student
```

Alternativ: Man könnte das WS 1969/70 als das 0-te Semester definieren.

Das aktuelle Semester bekommt man dann durch `(jahr-1970) * 2`, plus 1, falls `monat >= 4`, bzw. plus 2, falls `monat >= 10`. Nun subtrahiert man das so berechnete Semester für das aktuelle Datum von dem für den Studienbeginn, plus 1.

Subklassen: Beispiel (6)

```
(70)  class Test {
(71)      public static void main(String[] args) {
(72)          Person daniel =
(73)                  new Person("Daniel", "Sommer");
(74)          daniel.print();
(75)          System.out.println();
(76)
(77)          Student lisa =
(78)                  new Student("Lisa", "Weiss",
(79)                      "Informatik", 2012, 10);
(80)          lisa.print();
(81)          System.out.println();
(82)          System.out.println(lisa.vorname());
(83)          System.out.println(lisa.semester());
(84)      }
(85)  }
```

Subklassen: Beispiel (7)

Vererbung von Methoden und Attributen:

- Man beachte, dass die in der Oberklasse `Person` definierten Methoden `vorname()`, `nachname()`, `print()` auch für Objekte der Unterklasse `Student` aufrufbar sind.

  Die Methoden der Oberklasse werden automatisch an die Unterklasse vererbt. Objekte der Unterklasse haben eigentlich auch die Attribute der Oberklasse (sonst würden die ererbten Methoden ja nicht funktionieren). Die Attribute sind in der Unterklasse durch die `private`-Deklaration nicht zugreifbar (s.u.).

- Dies entspricht dem Substitutionsprinzip:
  Objekte der Unterklasse können überall verwendet werden, wo auch Objekte der Oberklasse erlaubt wären.

  Da man die Methoden für Objekte der Oberklasse aufrufen kann, muss man sie auch für Methoden der Unterklasse aufrufen können (dies muss zumindest für Aufrufe von außen gelten, innerhalb wirkt auch der Zugriffsschutz, s.u.).

Aufruf des Oberklassen-Konstruktors (1)

- Der Konstruktor der Unterklasse muss den Konstruktor der Oberklasse/Superklasse aufrufen (wenn es keinen parameterlosen Konstruktor der Oberklasse gibt):

```
super(vname, nname);
```

  Die Attribute der Oberklasse müssen ja auch initialisiert werden.

- Die Anweisung für den Aufruf des Konstruktors der Oberklasse muss die erste Anweisung im Rumpf des Konstruktors sein.

- Erst wird also zuerst der Konstruktor für die Oberklasse aufgerufen, und dann der Rumpf des Konstruktors für die Unterklasse ausgeführt.

  Allgemein werden die Konstruktoren in der Klassenhierarchie von oben nach unten ausgeführt. Falls die Oberklasse selbst noch eine Oberklasse hat, wird deren Konstruktor zuerst ausgeführt.

Aufruf des Oberklassen-Konstruktors (2)

- Ohne expliziten Aufruf des Konstruktors für die Oberklasse fügt Java automatisch einen solchen Aufruf ein, allerdings für einen Konstruktor ohne Parameter.

  Der Compiler versteht ja von der Anwendung nichts und kann sich keine Parameterwerte ausdenken.

- Läßt man im Beispiel den Aufruf `super(...)` weg, so erhält man eine Fehlermeldung (parameterloser Konstruktor fehlt):

```
Person.java:34: cannot find symbol
symbol : constructor Person()
location: class Person
        String studiert, int jahr, int monat) {
                                               ^
```

  Eine entsprechende Fehlermeldung erhält man auch, wenn man in der Unterklasse gar keinen Konstruktor aufgeschrieben hat. Dann fügt der Compiler einen ein mit genau diesem Aufruf.

Typ-Umwandlungen und Typ-Tests (1)

Up-Cast (Umwandlung von Unterklasse in Oberklasse):

- Objekte der Unterklasse können an Variablen der Oberklasse zugewiesen werden:

```
Person p = lisa; // lisa ist ein Student
```

  Auch dies entspricht dem Substitutionsprinzip: Auf der rechten Seite der Zuweisung könnte eine `Person` stehen, also muss dort auch ein `Student` erlaubt sein.

  Da die Parameter-Übergabe wie eine Zuweisung behandelt wird, gilt das dort entsprechend. Man kann `lisa` (ein `Student`-Objekt) an eine Methode übergeben, die einen Parameter vom Typ `Person` hat.

- Auf `p` können jetzt nur noch die Methoden der Klasse `Person` angewendet werden.

  Der Compiler weiss nicht mehr, dass in `p` tatsächlich eine Referenz auf ein `Student`-Objekt steht. Der statische (deklarierte) Typ von `p` ist ja `Person`.

Typ-Umwandlungen und Typ-Tests (2)

Down-Cast (Umwandlung von Oberklasse in Unterklasse):

- Die umgekehrte Zuweisung (Objekt der Oberklasse an Variable der Unterklasse) wird vom Compiler zurückgewiesen:

```
Student s = daniel; // daniel ist Person
```

- Man bekommt folgende Fehlermeldung:

```
Person.java:92: incompatible types
found : Person
required: Student
        Student s = daniel; // daniel ist Person
                    ^
```

Würde der Compiler die Zuweisung erlauben, könnte man anschließend Methoden wie `semester()` für Personen aufrufen, die keine Studenten sind. Diese würden dann auf Attribute zugreifen, die die Objekte gar nicht haben.

Typ-Umwandlungen und Typ-Tests (3)

Down-Cast mit expliziter Typ-Umwandlung:

- Da eine Variable vom Typ `Person` (Oberklasse) aber eventuell doch ein Objekt vom Typ `Student` (Unterklasse) enthält, ist eine Zuweisung mit Typ-Cast möglich:

```
Student s = (Student) p;
```

- Man teilt dem Compiler so mit, dass man sich sicher ist, dass `p` ein Objekt vom Typ `Student` enthält.

  Oder möglicherweise eine Unterklasse von `Student`. `null` wäre auch ok.

- Der Compiler akzeptiert die Zuweisung dann, fügt aber einen Laufzeit-Test in das Programm ein. Falls `p` doch kein `Student` war, erhält man eine `ClassCastException`.

  Mit der Fehlermeldung "`Person cannot be cast to Student`".

Typ-Umwandlungen und Typ-Tests (4)

Typ-Test (`instanceOf`):

- Wenn man sich nicht sicher ist, welcher Typ tatsächlich in der Variablen `p` vom Typ `Person` gespeichert ist, kann man es folgendermaßen testen:

```
if(p instanceof Student) { ... }
```

  Auf der linken Seite kann ein beliebiger Ausdruck von einem Referenz-Typ stehen, rechts muss der Name eines Referenz-Typs stehen. Es muss grundsätzlich möglich sein, dass das links berechnete Objekt zu der Klasse rechts gehört, sonst meldet der Compiler einen Fehler (die Bedingung wäre ja immer falsch). Der `instanceof`-Operator liefert einen booleschen Wert.

- Die Bedingung ist wahr, wenn `p` eine Referenz auf ein Objekt der Klasse `Student` (oder einer Subklasse von `Student`) enthält. Die Bedingung ist falsch, wenn `p` `null` ist.

Zugriffsschutz und Subklassen (1)

- Attribute und Methoden, die in der Oberklasse als `private` deklariert sind, sind in der Unterklasse nicht zugreifbar.
- Das ist auch vernünftig, denn sonst könnte der Zugriffsschutz jederzeit durch Deklaration einer Subklasse umgangen werden.

  Der Kreis der Methoden, die auf ein Datenelement zugreifen können, soll ja möglichst klein gehalten werden.
- Genauer müssen zwei Bedingungen erfüllt sein, damit auf ein `private`-Attribut der Oberklasse zugegriffen werden kann:
  - Der Zugriff muss innerhalb der Oberklasse stehen, und
  - der statische Typ auf der linken Seite vom "`.`" muss die Oberklasse sein (Beispiel auf der nächsten Folie).

Zugriffsschutz und Subklassen (2)

```
(1)  class Ober {
(2)      private int a = 1;
(3)      void mOber() {
(4)          Unter u = new Unter();
(5)          System.out.println(u.a); // Fehler
(6)          System.out.println(((Ober)u).a); // Ok
(7)      }
(8)  }
(9)
(10) class Unter extends Ober {
(11)     void mUnter() {
(12)         Ober o = new Ober();
(13)         System.out.println(o.a); // Fehler
(14)     }
(15) }
```

Inhalt

Überschreiben von Methoden (1)

- Man kann in einer Subklasse eine Methode neu definieren, die in der Oberklasse bereits definiert ist.
- In diesem Fall wird die ererbte Methode überschrieben ("Overriding", "Redefinition").

  Manche Autoren sprechen auch von "überlagern".
- Z.B. könnte man in der Subklasse `Professor` die Methode `print()` so definieren, dass sie nach dem Namen "(Professor für ⟨Lehrgebiet⟩)" mit ausgibt.
- Für die Objekte der Subklasse wird jetzt also eine andere Implementierung der Methode `print()` verwendet als für die Objekte der Oberklasse.

Überschreiben von Methoden (2)

```
(1)  class Professor extends Person {
(2)
(3)      // Attribut:
(4)      private String lehrgebiet;
(5)
(6)      // Konstruktor:
(7)      Professor(String vorname, String nachname,
(8)                String lehrgebiet) {
(9)          super(vorname, nachname);
(10)         this.lehrgebiet = lehrgebiet;
(11)     }
(12)
(13)     // Zugriffsfunktion fuer Attribut:
(14)     String lehrgebiet() {
(15)         return this.lehrgebiet;
(16)     }
(17)
```

Überschreiben von Methoden (3)

```
(18)      // Methode zur Ausgabe:
(19)      void print() {
(20)          super.print();
(21)          System.out.print(' (Professor fuer '
(22)                  + this.lehrgebiet + ')');
(23)      }
(24)
(25)  } // Ende der Klasse Professor
```

- Im Beispiel wird mit `super.print()` zunächst die Methode der Oberklasse ausgeführt.
- Im Unterschied zu Konstruktoren ist das bei (normalen) Methoden keine Vorschrift.

  Man hätte z.B. auch direkt `vorname` und `nachname` ausgeben können.

Überschreiben von Methoden (4)

- Eine Methode der Oberklasse wird nur dann an die Unterklasse vererbt, wenn sie dort nicht überschrieben wird.
- Sie wird überschrieben, wenn in der Unterklasse eine Methode mit gleichem Namen und gleicher Anzahl sowie gleichen Typen der Argumente definiert wird.

  Die Namen der Argumente spielen keine Rolle, aber die Argument-Typen müssen genau gleich sein, sonst handelt es sich um Überladen (siehe Kap. 14), nicht Überschreiben, d.h. es gibt beide Methoden in der Unterklasse.
- Der Ergebnis-Typ der Methode in der Unterklasse darf selbst ein Untertyp des Ergebnis-Typs in der Oberklasse sein.

  Dies wird auf Folie 1 näher erläutert, dort wird auch auf Exceptions und Zugriffsschutz eingegangen. Der wesentliche Gedanke ist, dass ein speziellerer Ergebnistyp nicht verhindert, dass die Methode aus der Unterklasse verwendet wird, selbst wenn der Compiler nur den Typ aus der Oberklasse kennt.

Überschreiben von Methoden (5)

- Java verwendet den tatsächlichen (dynamischen) Typ eines Objektes, um die Methoden-Implementierung zu wählen.
- Beispiel:

```
Professor sb =
  new Professor("Stefan", "Brass", "Informatik");
Person p = sb;
p.print();
```

- Obwohl der statische (deklarierte) Typ der Variablen `p` die Oberklasse `Person` ist, wird das Lehrgebiet mit ausgegeben, d.h. es wird die für das in `p` gespeicherte Objekt der Unterklasse zuständige Methode verwendet.

  In C++ müsste man die Methode dazu "`virtual`" deklarieren. In Java sind alle Methoden automatisch "`virtual`" (und es gibt dieses Schlüsselwort nicht).

Überschreiben und Überladen (1)

- Eine Methode wird nur überschrieben (d.h. die Implementierung wird für Objekte der Subklasse ersetzt), wenn die Argumentliste genau übereinstimmt.

  Nur die Typen müssen gleich sein, die Namen spielen keine Rolle.

- Falls man andere Argument-Typen angibt, wird die Methode überladen: Es gibt in der Subklasse dann zwei verschiedene Methoden mit dem gleichen Namen:
  - Die geerbte Methode aus der Oberklasse, und
  - die neu definierte Methode der Unterklasse.
- Der Compiler wählt die Methode basierend auf den Typen der beim Aufruf angegebenen Argument-Werte.

Überschreiben und Überladen (2)

- Wenn man eine Methode überschreiben will, ist empfohlen, die "Annotation" `@Override` zu verwenden:

```
class Professor extends Person {
    ...
    @Override
    void print() { ... }
}
```

Formal ist eine Annotation ein Modifier (wie `public` oder `static`).

Es ist üblich, Annotationen an den Anfang der Modifier-Liste zu schreiben.

- Dies hat zwei Vorteile:
  - Für den Leser des Programms ist klarer, was passiert.
  - Der Compiler gibt eine Warnung bzw. Fehlermeldung aus, wenn doch kein Überschreiben vorliegt.

Hinweis zu private-Methoden (1)

```
(1)  class C {
(2)      private int m() { return 1; }
(3)      void test() {
(4)          System.out.println(m());
(5)      }
(6)  }
(7)
(8)  class D extends C {
(9)      int m() { return 2; }
(10) }
(11)
(12) class Test {
(13)     public static void main(String[] args) {
(14)         D d = new D();
(15)         d.test();
(16)     }
(17) }
```

Hinweis zu `private`-Methoden (2)

- `private`-Methoden werden nicht vererbt und können deswegen nicht überschrieben werden.
- Das obige Beispiel-Programm druckt `1` aus, obwohl `m()` hier für ein Objekt der Klasse `D` aufgerufen wird (dynamischer Typ).
- Wenn man das `private` dagegen weglässt, druckt das Programm `2` aus.

  Nun wird das `m()` von der Oberklasse in der Unterklasse überschreiben, und da es für ein Objekt der Unterklasse aufgerufen wird, wird die Version aus der Unterklasse genommen.

- Wenn man also sicher sein will, dass eine aufgerufene Hilfsmethode auch wirklich die selbst programmierte Methode ist, deklariere man sie als `private`.

final

- Mit dem Schlüsselwort `final` vor einer Methode verbietet man das Überschreiben dieser Methode.

  Es gibt eine Fehlermeldung, wenn man versucht, sie in einer Subklasse neu zu definieren. "`final`" ist ein Modifier wie `public` und `static`.
  Üblicherweise wird `final` gegen Ende der Liste angegeben (vor `synchronized`).

- Die Aufrufe von `final`-Methoden gehen vermutlich etwas schneller als die von normalen Methoden.

  Der Compiler weiss ja sicher, dass sie nicht in einer Unterklasse überschrieben werden können. Siehe die "Virtual Methode Table/Dispatch Table" unten.

- `final` vor einer Klasse verbietet, dass Subklassen dieser Klasse definiert werden.

  Dann können die Methoden natürlich nicht überschrieben werden. Es wird aber empfohlen, eher alle Methoden einzeln als `final` zu kennzeichnen (und die Attribute als `private`), dann wären Erweiterungen der Klasse noch möglich.

Hinweis zu Konstruktoren (1)

- Konstruktoren werden nicht vererbt:

```
(1)  class C {
(2)      int a;
(3)      C()      { this.a = 1; }
(4)      C(int a) { this.a = a; }
(5)  }
(6)
(7)  class D extends C {
(8)  }
(9)
(10) class Test {
(11)     public static void main(String[] args){
(12)         C c = new C(2); // Ok
(13)         D d = new D(3); // Fehler!
(14)     }
(15) }
```

Hinweis zu Konstruktoren (2)

- Konstruktoren sind formal keine Methoden. Obwohl sie in vielem ähnlich sind, unterscheiden sie sich bei der Vererbung.

  Weil sie nicht vererbt werden, können sie auch nicht überschrieben werden. Das wäre aber auch uninteressant, da man wenigstens bei der Konstruktion die genaue Subklasse kennt.

- Im Beispiel legt der Compiler für die "leere" Subklasse `D` automatisch einen Konstruktor ohne Parameter an, der seinerseits den parameterlosen Konstruktor der Oberklasse aufruft.

- Es ist sehr sinnvoll, dass Konstruktoren nicht vererbt werden: Die Subklasse führt ja normalerweise zusätzliche Attribute ein, die auch initialisiert werden müssen.

  Der Compiler stellt auch sicher, dass ein Konstruktor der Oberklasse aufgerufen wird. Wenn man will, kann man dies als sehr spezielle Form der Vererbung sehen.

Gleich benannte Attribute in Subklassen

- Angenommen, es gibt in einer Klasse `C` ein Attribut `A`.
- Deklariert man nun in einer Subklasse `D` von `C` auch ein Attribut `A`, so gibt es zwei verschiedene Attribute `A`.
- Das Attribut `A` aus der Oberklasse ist in der Unterklasse nicht direkt zugreifbar (es ist "versteckt", engl. "hidden").
- Das ist aber kein Problem, denn wenn der Autor der Unterklasse gewusst hätte, dass es bereits ein Attribut `A` in der Oberklasse gibt, hätte er sein Attribut anders genannt.

  Er braucht es also offenbar nicht. Tatsächlich wurde diese Regel eingeführt, um dem Autor der Oberklasse zu erlauben, nachträglich ein zusätzliches Attribut `A` einzuführen. Der Autor der Oberklasse kann nicht immer alle Unterklassen seiner Klasse kennen. Würde der Compiler die Situation als Fehler behandeln, wären Schwierigkeiten bei solchen Änderungen vorprogrammiert.

Inhalt

Abstrakte Klassen (1)

- Eine abstrakte Klasse ist eine Klasse, die selbst keine direkten Instanzen haben kann.

  Eine nicht-abstrakte Klasse wird auch "konkrete Klasse" genannt.

- Wenn z.B. `Person` eine abstrakte Klasse wäre, so wären alle Objekte dieser Klasse eigentlich Objekte der Unterklassen `Student` oder `Professor`.

  Objekte der Unterklasse gehören ja immer automatisch mit zur Oberklasse, zumindest findet bei Bedarf eine implizite Typ-Umwandlung statt.

- Abstrakte Klassen werden mit dem entsprechenden "Modifier" markiert:

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

- Der Compiler verbietet dann die Erzeugung von Objekten dieser Klasse, d.h. `new Person(...)` gibt einen Fehler.

Abstrakte Klassen (2)

- Abstrakte Klassen sind nützlich als gemeinsame Schnittstelle für ihre Unterklassen.
- Manchmal möchte man in der abstrakten Klasse eine Methode deklarieren (so dass sie Bestandteil der Schnittstelle wird), kann sie aber noch nicht sinnvoll definieren (implementieren).
- Wenn man z.B. eine abstrakte Klasse für geometrische Objekte in der Ebene definiert (Kreise, Rechtecke, etc.), so würde es Sinn machen, eine Methode zur Berechnung des Flächeninhaltes vorzusehen.
- Man kann die Flächenberechnung aber für allgemeine geometrische Objekte nicht sinnvoll implementieren (das geht nur für die Subklassen, d.h. konkrete Formen).

Abstrakte Klassen (3)

- Methoden ohne Implementierung werden in der Klasse auch als abstrakt gekennzeichnet:

```
abstract class GeoObject {
    abstract double flaeche();
    ...
}
```

- Der Methoden-Rumpf wird dann weggelassen (es gibt ja gerade keine Implementierung der Methode in dieser Klasse).

  Anstelle eines Methoden-Rumpfes wird ein Semikolon angegeben.

- Abstrakte Methoden können nur in abstrakten Klassen angegeben werden.

  Man bekommt eine Fehlermeldung, wenn man eine Methode als "`abstract`" kennzeichnet, ohne das die Klasse "`abstract`" ist. Umgekehrt dürfen abstrakte Klassen aber nicht-abstrakte Methoden haben (s.u.).

Abstrakte Klassen (4)

- Eine abstrakte Klasse muss nicht unbedingt nur abstrakte Methoden haben: Manche Methoden lassen sich vielleicht schon auf der allgemeinen Ebene implementieren.

  Falls eine abstrakte Klasse nur abstrakte Methoden und keine Attribute hat, sollte man eventuell ein "Interface" zu verwenden (siehe Kapitel 13).

- Damit man mit der abstrakten Klasse etwas anfangen kann, muss es eine Unterklasse geben, in der die abstrakten Methoden überschrieben (tatsächlich implementiert) sind.

  Das stimmt nicht ganz: Eine abstrakte Klasse nur mit statischen Methoden wäre auch sinnvoll (klassisches Modul).

- Wenn eine Subklasse nicht alle ererbten abstrakten Methoden überschreibt, muss sie selbst "`abstract`" sein.

  Die abstrakte Klasse verspricht, dass es die in ihr deklarierten Methoden gibt. Objekte kann man nur von Unterklassen anlegen, die das Versprechen einlösen.

Abstrakte Klassen (5)

- Beispiel für fehlende Implementierung:

```
(1)  abstract class GeoObject {
(2)      abstract double flaeche();
(3)  }
(4)
(5)  class Rechteck extends GeoObject {
(6)  }
```

- Dies liefert folgende Fehlermeldung:

```
GeoObject.java:5: Rechteck is not abstract
                  and does not override abstract
                  method flaeche() in GeoObject
class Rechteck extends GeoObject {
^
1 error
```

Abstrakte Klassen (6)

- Eine abstrakte Methode kann nicht
  - `private` sein,

    Eine private Methode könnte ja nicht in einer Unterklasse überschrieben werden. Der ganze Sinn einer abstrakten Methode ist aber, dass sie später überschrieben wird.
  - `static` sein.

    Statische Methoden werden über Angabe der Klasse aufgerufen, oder ersatzweise über den statischen Typ eines Objektes. Wenn man nur den Namen der Oberklasse hat, wüßte man nicht, welche in einer Unterklasse definierte statische Methode benutzt werden soll.
- Es ist grundsätzlich möglich, eine abstrakte Klasse ohne abstrakte Methoden zu deklarieren.

  Macht aber vermutlich wenig Sinn.

Inhalt

Die Klasse `Object` (1)

- Wenn bei einer Klassendeklaration nicht explizit eine Oberklasse angegeben ist, verwendet Java die vordefinierte Klasse `Object` als Oberklasse.

  Auch Arrays haben `Object` als Oberklasse.

- Damit ist `Object` indirekte Oberklasse von jeder Klasse, d.h. jede Klasse erbt die in `Object` definierten Methoden:

  [http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html]

- Zum Teil sollten diese Methoden überschrieben werden, da die allgemeine Definition aus `Object` zu unspezifisch ist.

  Dies betrifft insbesondere die Methode `toString()`, aber eventuell auch `equals()`, `hashCode()` und `clone()`, siehe unten.

Die Klasse `Object` (2)

- Zusammen mit dem Autoboxing kann man für einen "`Object`"-Parameter jeden Wert in Java übergeben.

  Autoboxing: automatische Umwandlung von einem primitiven Typ in die entsprechende Klasse, z.B. `int` nach `Integer`.

- `public String toString()`

  Diese Methode soll eine Repräsentation des Objektes als String liefern, die knapp, aber informativ ist.

  Das Ergebnis wird z.B. zum Ausdrucken des Objektes mit `System.out.print()` verwendet, oder auch bei der String-Konkatenation mit +, wenn ein Operand die Objekt-Referenz ist. Die Standard-Implementierung aus `Object` gibt den Namen der Klasse, ein "`@`, und dann hexadezimal eine relativ eindeutige Nummer aus (z.B. Hauptspeicheradresse). Es wird empfohlen, dies zu überschreiben. Es ist auch für die Fehlersuche günstig, wenn man die wesentlichen Daten des Objektes schnell sehen kann (z.B. in Test-Ausgaben, auch manche Debugger verwenden `toString()` zum Anzeigen der Objekte).

Die Klasse `Object` (3)

- `public boolean equals(Object obj)`

  Diese Methode testet, ob das aktuelle Objekt gleich dem als Parameter übergebenen Objekt ist.

  Die Standard-Implementierung liefert das gleiche Ergebnis wie `this == obj` (d.h. die Objekte sind identisch). Es wäre aber jede Äquivalenzrelation möglich. Man könnte also auch die Attributewerte vergleichen (der Parameter hat den Typ `Object`, man muss also zuerst mit `instanceof` testen, ob es sich um den richtigen Typ handelt, und dann einen Down-Cast ausführen, damit man die Attributwerte abfragen kann). Wenn man `equals()` ändert, muss man zwingend auch `hashCode()` ändern (siehe nächste Folie).

Die Klasse `Object` (4)

- `public int hashCode()`

  Diese Methode liefert eine Zahl, die

  - für `equals()`-gleiche Objekte sicher gleich ist, und
  - für verschiedene Objekte meistens verschieden ist.

  Mit Hashtabellen kann man eine Abbildung von Objekten dieser Klasse auf irgendwelche andere Daten realisieren (ähnlich einem Array, dass statt mit Zahlen mit Objekten indiziert ist). Solche Hashtabellen sind z.B. in `java.util.HashTable` realisiert, aber sie benötigen eine Zahl-Repräsentation für die Objekte. Es wäre formal möglich, z.B. immer `1` zu liefern, aber dann werden die Hashtabellen sehr ineffizient. Die Standard-Implementierung aus `Object` liefert vermutlich die Hauptspeicher-Adresse des Objektes. Das passt aber nur mit `==` für `equals()` zusammen.

Beispiel: Datum (1)

- Als Beispiel sei wieder die Klasse `Datum` mit den drei Komponenten Tag, Monat, Jahr betrachtet (siehe Kap. 11):

```
(1)  class Datum {
(2)      int tag;   // 1..31
(3)      int monat; // 1..12
(4)      int jahr;  // 1600..
```

- Die Methode `toString()` wird u.a. zur Ausgabe des Objektes mit `System.out.println(...)` verwendet:

```
(91)     @Override public String toString() {
(92)         return String.format(
(93)             "%02d.%02d.%04d",
(94)             tag, monat, jahr);
(95)     }
```

Beispiel: Datum (2)

- Außerdem kann man die Methode `equals` definieren, um einen Vergleich zu ermöglichen, der "in die Objekte hineinschaut", statt nur die Referenzen zu vergleichen.

  Wenn man die Methode `equals` nicht definiert, gibt es schon eine vordefinierte ("ererbte") Methode, aber diese vergleicht nur die Referenzen.

- Im Beispiel sind zwei Datumswerte gleich, wenn sie in den drei Komponenten Tag, Monat und Jahr übereinstimmen.

- Beispiel-Aufruf:

```
if(d1.equals(d2)) { ... }
```

  Das funktioniert genau so, wie wir es schon von Strings kennen.

- Der Parameter von `equals` muss als beliebiges Objekt (Klasse `Object`, Obertyp aller Referenzen) deklariert werden.

  Weil es die Methode schon für beliebige Objekte gibt.

Beispiel: Datum (3)

- Definition der Methode `equals`:

```
(96)      @Override public
(97)            boolean equals(Object o) {
(98)         // Ist o ueberhaupt ein Datum?
(99)         if(!(o instanceof Datum))
(100)           return false;
(101)
(102)        // o als Datum-Objekt betrachten:
(103)        Datum d = (Datum) o;
(104)
(105)        // Eigentlicher Vergleich:
(106)        return this.tag == d.tag &&
(107)           this.monat == d.monat &&
(108)           this.jahr == d.jahr;
(109)     }
```

Beispiel: Datum (4)

- Wenn man `equals(...)` umdefiniert, muss man auch die Methode `hashCode()` ändern:
  - Wenn `o1.equals(o2)` gilt, muss auch `o1.hashCode() == o2.hashCode()` gelten.
  - Die umgekehrte Bedingung ist dagegen nicht gefordert. Es wäre möglich, dass `hashCode()` z.B. immer `0` liefert.
  - Dann wäre aber ungünstig für bestimmte Datenstrukturen, die Objekte in einer Menge schnell finden sollen (Hashtabellen). Diese würden langsam laufen.
  - Die in `Object` vordefinierte Methode `hashCode()` liefert meist die Hauptspeicheradresse des Objektes.

    Das ist nicht garantiert. Es funktioniert aber mit der vordefinierten `equals(...)`-Methode, die die Referenzen vergleicht. Es funktioniert nicht mit unserer modifizierten `equals(...)`-Methode.

Beispiel: Datum (5)

- Definition der Methode `hashCode`:

```
(110)     @Override public
(111)           int hashCode() {
(112)       return tag + 31*monat +
(113)              365*(jahr-1600);
(114)     }
```

- Es ist nicht verlangt, dass jeder Wert auch angenommen wird (d.h. die gelieferten Hashcodes dürfen "Lücken" haben).