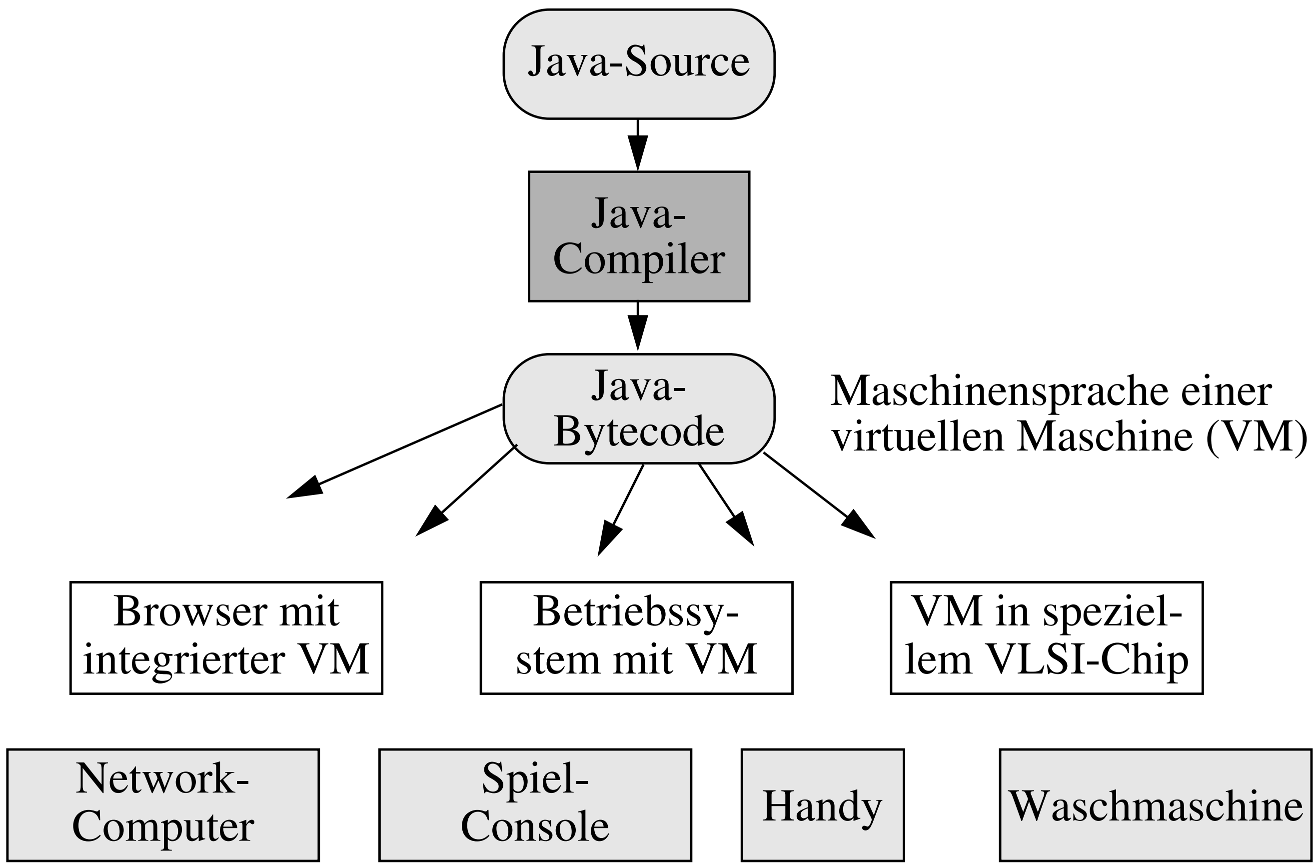
Einführung in das Programmieren mit Java

Praktikum zur Vorlesung Verteilte Systeme
ETH Zürich, SS2001

Plattformunabhängigkeit

- Ein Applet läuft auf allen gängigen Rechnern und Betriebssystemen (PC, Workstation, UNIX, Windows...)

- Entsprechend auch: Graphische Interfaces (GUI)
  - Zeit- / Kostenersparnis in der Softwareindustrie
  - Hier aber noch gewisse Probleme mit den APIs



- VM ist ein Bytecode-Interpreter
  - programmierter Simulator der virtuellen Maschine
  - relativ einfach für verschiedene Plattformen realisierbar

- Effizienzverlust durch Interpretation?
  - ggf. in Zielsprache (weiter-) übersetzen

- Prinzip an sich ist nicht neu (vgl. Pascal-P-Maschine)

Hello World

- Das "Programm" (eine Klasse!):

```
class Hallo {
  //  Mein erstes Java-Programm!!
  public static void main (String args[]) {
    System.out.println("Hello Java-World!");
  }
}
```

ein ';' beendet jede Anweisung und jede Deklaration

- Unix Shell:

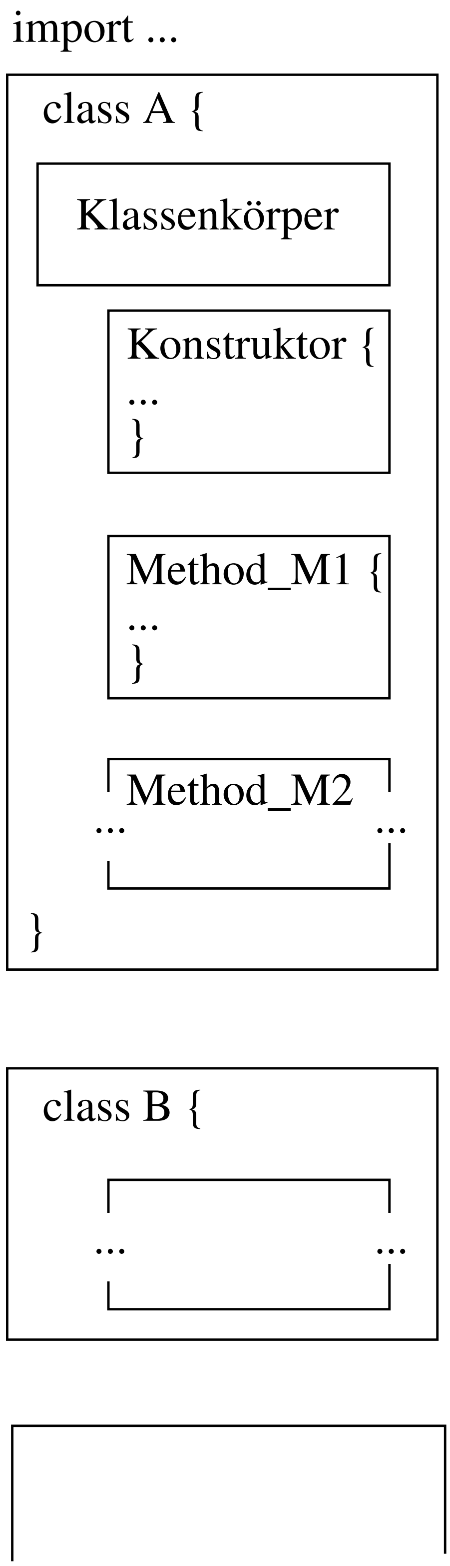
```
%ls
Hallo.java

%javac Hallo.java

%ls
Hallo.java  Hallo.class

%java Hallo
Hello Java-World!
```

- Datei sollte gleich wie die Klasse heißen

- Kommentare:

```
// bis Zeilenende
/* oder
so über mehrere
Zeilen */
```

Java-Programmstruktur



- Mit *import* werden anderweitig vorhandene Pakete von Klassen verfügbar gemacht

- Der *Klassenkörper* enthält
  - statische Klassenvariablen ("Attribute")
  - benannte Konstanten
  - klassenbezogene Methoden

- *Konstruktoren* sind spezielle Methoden

- *Methoden* können bestehen aus
  - Parametern
  - lokalen Variablen
  - Anweisungen

- Bei *eigenständigen Programmen* (=/= Applets) muß es eine "main"-Methode geben, die so aussieht:

```
public static void
main (String args[]){...
```

- Jede Klasse kann eine (einzige) solche main-Methode enthalten; sie wird ausgeführt, wenn der entsprechende Klassenname beim "java"-Kommando genannt wird

- Klassen können getrennt übersetzt werden

- Es gibt keine globalen Variablen etc.

"Removed from C/C++"

> You know you've achieved perfection in design,
> Not when you have nothing more to add,
> But when you have nothing more to take away.
>
> *Antoine de Saint Exupery*

- Strukturen, union (statt dessen: Klassen)
- Zeigerarithmetik, malloc (aber: arrays und new)
- Funktionen (statt dessen: Methoden)
- Mehrfachvererbung (statt dessen: interfaces)
- Präprozessor: #define, #include, #ifdef, ...
- #DEFINE (statt dessen: final static)
- Überladen von Operatoren
- goto (statt dessen: break/continue, exceptions)
- .h-Dateien (aber: Pakete)
- Destruktoren, free (aber: Garbage-Collecor; finalize)
- Implizite Typkonvertierung

Außerdem neu gegenüber C++

- Threads in der Sprache (als Objekte)
- Datentyp "boolean"
- Character im 16-Bit-Unicode ("Internationalisierung")

Einfache Datentypen

Integer (Zahlen im 2er-Komplement):

`byte` (8 Bits)

`short` (16 Bits)

`int` (32 Bits)

`long` (64 Bits)

Floating Point, IEEE 754-Standard

`float` (32 Bits)

`double` (64 Bits)

Zeichen (Unicode! --> "Internationalisierung")

`char` (16 Bits)

Wahrheitswerte (nicht mit Integer kompatibel!)

`boolean (true/false)`

Referenzen auf Objekte ("Zeiger")

- Es gibt natürlich auch arrays und strings (--> später)
- Komplexere Datentypen lassen sich mit Klassen bilden
    - alle nicht-einfachen Datentypen sind Objekte (als Instanzen von vorgegebenen oder eigenen Klassen)
    - auch einfache Datentypen können in Klassen gepackt werden

Variablen, Bezeichner, Deklaration

- Bezeichner müssen sich von keywords unterscheiden
  - es gibt ca. 50 keywords (int, class, while,...)
  - es gibt Standardklassen und -methoden (z.B. String, File, Stack...)
  - Sonderzeichen (Umlaute etc.) sind in Namen zulässig

- Konvention:
  - Variablen und Methoden beginnen mit einem Kleinbuchstaben
  - Klassennamen beginnen mit einem Großbuchstaben
  - benannte Konstanten ganz mit Großbuchstaben

- Beispiele für Deklarationen:

```
int j;
int i = 1; // mit Initialisierung
float x_koordinate, y_koordinate;
String s = "Hallo";
Person p = new Person ("Hans Dampf", 1974);
float [][] matrix;  // 2-dimensionales array
```

- Namensräume; einige "grobe" Regeln:
  - zwei im gleichen Namensraum deklarierte Variablen müssen verschieden heißen
  - typische Namensräume: Methoden und Klassenrümpfe
  - mit {...} wird *kein* neuer Namensraum festgelegt (aber: for-Schleife etc!)
  - lokal deklarierte Bezeichner können andere Bezeichner *verdecken* (z.B. ererbte Attribute, importierte Typen)
  - Deklarationen müssen nicht am Anfang stehen, sondern können mit Anweisungen "gemischt" werden
  - es gibt keine (absolut) globalen Variablen etc.
  - "voll qualifizierte" Namen: Paketname.Klassenname.Attributname

Typkonvertierung

- Java ist eine *streng typisiert*e Sprache
  --> Compiler kann viele Typfehler entdecken

- Gelegentlich muß dies jedoch durchbrochen werden
  --> *Typecast* (to cast --> hier: "formen"; "in Form bringen")

- So geht es nicht (-->Fehlermeldung durch Compiler):

```
int myInt;
double myFloat = 3.14159;

myInt = myFloat;
```

- Statt dessen explizite Typumwandlung:

```
int myInt;
double myFloat = 3.14159;

myInt = (int)myFloat;
```

- Umwandlung hin zu einem größeren Wertebereich
  (z.B. int --> float) geht auch implizit

- Typumwandlung ist gelegentlich bei Referenzen sinnvoll:

```
Hund h; Tier fiffi;
...
if (fiffi instanceof Hund)
   h = (Hund)fiffi;
```

Operatoren

- Binäre arithmetische Operatoren

```
+       op1 + op2
-       op1 - op2
*       op1 * op2
/       op1 / op2
%       op1 % op2
```

`+` (auch für Stringkonkatenation!)

`%` (Rest bei Division)

- Shortcut ++ und -- (Inkrementieren / Dekrementieren)

`op++` (Wert *vor* Inkrementierung auswerten)

`++op` (..*nach*...)

`op--`

`--op`

- weitere Abkürzungen: `i = i+7` ersetzen durch `i += 7` etc.

- Relationale Operatoren

`>`	`op1 > op2`	("true", wenn op1 größer als op2)
`>=`	`op1 >= op2`	
`<`	`op1 < op2`	
`<=`	`op1 <= op2`	
`==`	`op1 == op2`	(gleich)
`!=`	`op1 != op2`	(ungleich)

- Logische Operatoren

`&&`	`op1 && op2`	("und")
`||`	`op1 || op2`	("oder")
`!`	`!op`	(Negation)

- Bit-Operatoren: `&  |  ^  ~  >>  <<  >>>`

Priorität der Operatoren

<table>
<tr><td>postfix operators</td><td><code>[] . (params) expr++ expr--</code></td></tr>
<tr><td>unary operators</td><td><code>++expr --expr +expr -expr ~ !</code></td></tr>
<tr><td>creation or cast</td><td><code>new (type)expr</code></td></tr>
<tr><td>multiplicative</td><td><code>* / %</code></td></tr>
<tr><td>additive</td><td><code>+ -</code></td></tr>
<tr><td>shift</td><td><code><< >> >>></code></td></tr>
<tr><td>relational</td><td><code>< > <= >= instanceof</code></td></tr>
<tr><td>equality</td><td><code>== !=</code></td></tr>
<tr><td>bitwise AND</td><td><code>&</code></td></tr>
<tr><td>bitwise exclusive OR</td><td><code>^</code></td></tr>
<tr><td>bitwise inclusive OR</td><td><code>|</code></td></tr>
<tr><td>logical AND</td><td><code>&&</code></td></tr>
<tr><td>logical OR</td><td><code>||</code></td></tr>
<tr><td>conditional</td><td><code>? :</code></td></tr>
<tr><td>assignment</td><td><code>= += -= *= /= %= ^= &= ...</code></td></tr>
</table>

- Ansonsten (und in Zweifelsfällen) Klammern verwenden!

- Auswertungsreihenfolge von links nach rechts
- es werden alle Operanden vor Ausführung der Operation ausgewertet
- Ausnahme bei && und || :

```
if((count > NUM_ENTRIES) &&
   (System.in.read() != -1))...
```

hier wird der zweite Operator von && ggf. nicht ausgewertet!

Kontrollstrukturen

```
class CompNum {
static final int last=100;
//...
/* Zusammengesetzte
   Zahlen in 1..last*/
for(int i=2; i <= last; i++)
{ int j=2;
  Test:
  while (j*j <= i) {
    if (i%j == 0) {
      System.out.println(i);
      break Test;
    }
    else
      j++;
  }
}
// Hier sind i und j nicht mehr sichtbar
//...
```

so werden benannte Konstanten definiert

for-Kontrolltripel: Initialisierung, Terminierung, Inkrement

i und j dürfen außerhalb des for-Blockes nicht (schon) deklariert sein

Anweisungsblöcke in {...}

break ohne label beendet den innersten Kontrollblock (for, while, switch...)

else-Teil ist natürlich optional

- Weitere Kontrollkonstrukte:
  - switch
  - return (Methode beenden; ggf. mitRückgabewert)
  - continue (Sprung an das Ende einer ggf. benannten Schleife)
  - exception-handling
  - do-while (Auswertung erst am Ende der Schleife; Schleifenkörper wird mindestens ein Mal ausgeführt)

```
do {
   statements
} while (booleanExpression);
```

Switch

```
int monat;
. . .
switch (monat) {
case 1:  System.out.print("Jan"); break;
case 2:  System.out.print("Feb"); break;
case 3:  System.out.print("März"); break;
. . .
default: System.out.print("Fehler"); break;
}
```

```
. . .
switch (monat) {
case 1:
case 3:
case 5:
case 7:
case 8:
case 10:
case 12:
   numDays = 31;
   break;
case 4:
. . .
}
```

ohne break wird mit dem nächsten case-Zweig fortgefahren

Ein- und Ausgabe

```
int count = 0;
while (System.in.read() != -1)
    count++;
System.out.println("Eingabe hat " +
    count + " Zeichen.");
```

- System.in:

  - System ist eine Klasse mit Schnittstellenmethoden zum ausführenden System (Betriebssystem, Rechner)
  - System.in ist der Standard-Eingabestrom (vom Typ InputStream)
  - read liest ein einzelnes Zeichen; liefert -1 bei Dateiende, ansonsten einen Wert zwischen 0 und 255 für das Zeichen
  - es gibt noch einige weitere Methoden (skip, close...)
  - erst abgeleitete Typen von InputStream enthalten Methoden, um ganze Zeilen etc. zu lesen (z.B.Klasse DataInputStream)

- System.out: Standard-Ausgabestrom

  - print gibt das übergebene Argument aus
  - println erzeugt zusätzlich noch ein newline
  - es können u.a. int, float, string, boolean... ausgegeben werden

Einlesen von Zahlen

```
import java.io.* ;

class X {
  public static void main (String args[])
  throws java.io.IOException
  {
  int i = 0;  String Zeile;
  DataInputStream ein =
     new DataInputStream(System.in);

  while(true)
  {
     Zeile = ein.readLine();
     i += Integer.valueOf(Zeile).intValue();
     System.out.println(i);
  }
...
```

in diesem Paket stehen die Ein-Ausgabe-Methoden (innerhalb von "Dateiklassen")

die auftretbaren exceptions müssen nach throws am Anfang einer Methode genannt werden

Der InputStream muß beim Aufruf des Konstruktors angegeben werden

man könnte hier auch "readLine()" für "Zeile" substituieren

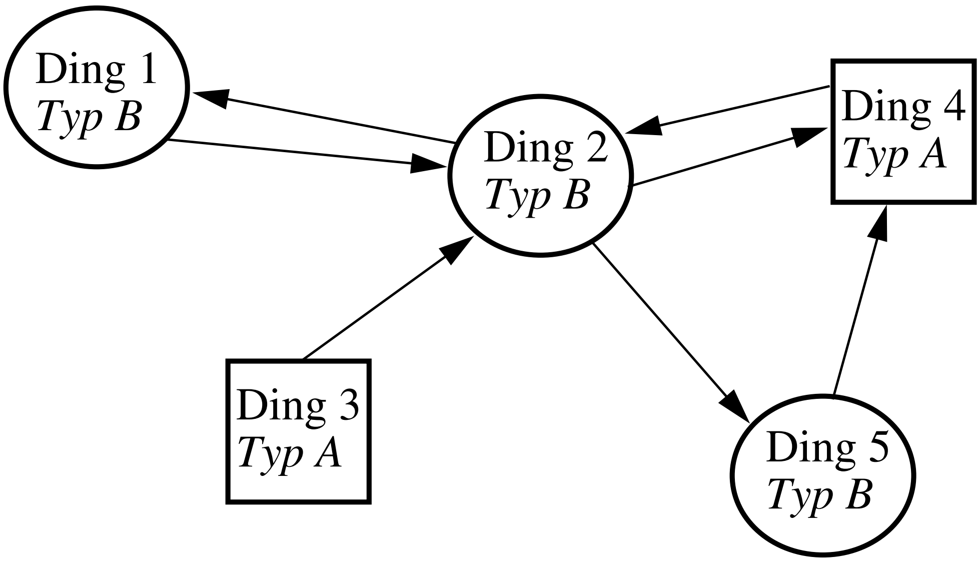
- Die Klasse DataInputStream enthält die Methode readLine, welche alle Zeichen bis Zeilenende liest und daraus einen String konstruiert

- "Integer" ist eine Wrapper-Klasse, welche Konvertierungsroutinen etc. enthält
    - "valueOf" ist eine Methode von Integer, die einen String nach Integer konvertiert
    - "intValue" ist eine Methode, die aus einem Integer-Objekt den int-Wert "herausholt"

- Entsprechend kann man floating-point-Zahlen etc. einlesen

Objektorientiertes Programmieren

<u>Weltsicht:</u> Die Welt besteht aus verschiedenen *inter-agierenden "Dingen"*, welche sich *klassifizieren* lassen.



<u>Ziel:</u> Betrachteten Weltausschnitt strukturkonsistent mit *kommunizierenden Objekten* abbilden und modellieren.

Simulationssprache SIMULA war die erste objektorientierte Programmiersprache (1967)

Objekte:

- sind autonome *gekapselte Einheiten* (= Module)
- haben einen eigenen *Zustand* (= lokale Variablen)
- besitzen ein *Verhalten* (wenn sie aktiviert werden)
- bieten anderen Objekten *Dienstleistungen* an

  - Durchführung von Berechnungen
  - Änderungen des lokalen Zustandes
  - Zurückliefern von Variablenwerten oder Berechnungsergebnissen
  - Allgemein: "Reaktion" auf Aufruf einer "Methode"

- besitzen eine *Identität*
- sind von einem bestimmten *Typ*
  (= "Klasse" gleichartiger Objekte)

Klassen

- können zu Objekten "instanziiert" werden
- sind daher Schablonen, Prototypen, Muster, templates...
- stellen den Typ der daraus erzeugten Objekte dar
- realisieren abstrakte Datentypen
- enthalten Variablen ("Attribute")
  - ► machen den Zustand der zugehörigen Objekte aus
  - ► sichtbare ("public") / verborgene ("private") Variablen
- enthalten Methoden als Anweisungskomponenten
  - ► realisieren die Funktionalität der Objekte
  - ► sichtbare / verborgene Methoden
- sind hierarchisch (= baumartig) organisiert
  Spezialisierung, Verallgemeinerung, Vererbung ==> Klassenhierarchie

---

Objektorientiertes Programmieren =

- Strukturierung der Problemlösung in eine Menge kooperierender Objekte
- Entwurf der Objekttypen (= Klassen)
- Herausfaktorisierung gemeinsamer Aspekte verschiedener Klassen ==> Hierarchie, Klassenbibliothek
- Festlegung der einzelnen Dienstleistungen
- Entwurf der Objektbeziehungen ("Protokoll")
- Feinplanung der einzelnen Methoden, Festlegung der Klassenattribute etc.
- Strukturierung und Implementierung der Methoden

Eine Beispiel-Klasse in Java

```
class Datum
{ private int Tag, Monat, Jahr;
```

Der Name der Klasse

Diese 3 *Attribute* sind von außerhalb nicht sichtbar

Eine Metode mit dem gleichen Namen wie die Klasse selbst stellt einen "*Konstruktor*" dar. Er wird bei Erzeugen eines Objektes automatisch aufgerufen; man kann (nur) damit die neuen Objekte (deren Variablen) initialisieren.

```
  public Datum()
  { System.out.println("Datum mit
    Wert 0.0.0. gegründet");
  };
```

Diese Klasse hat einen *zweiten Konstruktor* mit einer unterschiedlichen *Signatur*. Welcher Konstruktor genommen wird, richtet sich nach der Signatur beim new-Aufruf.

```
  public Datum(int T, int M, int J)
  { Tag = T; Monat = M; Jahr = J; };

  public void Drucken()
  { System.out.println(Tag + "." +
    Monat + "." + Jahr); }
```

Dies sind zwei Methoden

```
  public void Setzen (int T, int M, int J)
  { Tag = T; Monat = M; Jahr = J; };
};
```

Verwendung von Klassen

- Zugriff auf Methoden und Variablen ("Attribute") eines Objektes mit *Punktnotation*

hier wird der erste Konstruktor aufgerufen

```
class Beispiel
{ public static void main (String args[])
  { Datum Ostermontag = new Datum();
    /* Datum mit Wert 0.0.0. gegründet */
    Ostermontag.Drucken();
```

liefert 0.0.0

```
    Ostermontag.Setzen(31,03,97);
    Ostermontag.Drucken();
```

liefert 31.3.97

```
  }
}
```

- "Ostermontag" ist eine Variable vom Typ "Datum".
    - genauer: eine Referenz, die auf Datum-Objekte zeigen kann
    - alle Referenzen haben den Defaultwert **`null`**
    - Objekte werden mit **`new`** erzeugt, dabei wird eine Referenz auf das neu erzeugte Objekt zurückgeliefert

- Eigentlich sollte "Setzen" zumindest einen Plausibilitätstest machen (Monat ≤ 12, Tag ≤ 31 etc.).

- Datenstrukturen mit zugehörigen Operationen
  --> abstrakte Datentypen
    - Klassen können also abstrakte Datentypen implementieren

Statische Klassenvariablen

```
class Datum {
  public static int Zahl =0;
  private int Tag, Monat, Jahr;
  public Datum() {... Zahl++;}
...
class Beispiel { ...
  Datum Geburtstag = new Datum(23,03,56);
  Geburtstag.Drucken();
  Datum Glueckstag;
  Glueckstag = Geburtstag;
  Glueckstag.Drucken();
  Datum[] Januar = new Datum[32];
  for (int i=1; i<=31; i++)
  {  Januar[i] = new Datum(i,01,97);
     Januar[i].Drucken();
  }
  System.out.println("Es gibt " +
    Datum.Zahl + " Datum-Objekte");
...
```

static-Variablen existieren für alle Instanzen einer Klasse nur ein einziges Mal!

wie vorhin

hier wird der andere Konstruktor verwendet

liefert 23.3.56

Zuweisung von Referenzen

liefert 23.3.56

liefert 32

Statt Datum hätte man auch "Geburtstag" oder "Glueckstag" schreiben können

- Statische Variablen in Klassen wirken also ähnlich wie "globale Variablen" in anderen Sprachen. Alle Objekte einer Klasse sehen immer den gleichen Wert ==> Kann zur Kommunikation zwischen diesen Objekten benutzt werden.

Methoden mit Parameter

- Bei Aufruf von Methoden kan man Parameter übergeben
  - Wertübergabesemantik ("by value")
  - Auch Referenzen "by value" - referenzierte Objekte damit "by reference"

d ist ein formaler Parameter vom Typ "Datum"

```
class Datum
{ ...
  boolean frueher_als(Datum d)
  { return Jahr<d.Jahr ||
           Jahr==d.Jahr && Monat<d.Monat ||
           Jahr==d.Jahr && Monat==d.Monat
                        && Tag<d.Tag;
  };
}
```

Methode liefert einen Boole'schen Wert zurück; ist also nicht void

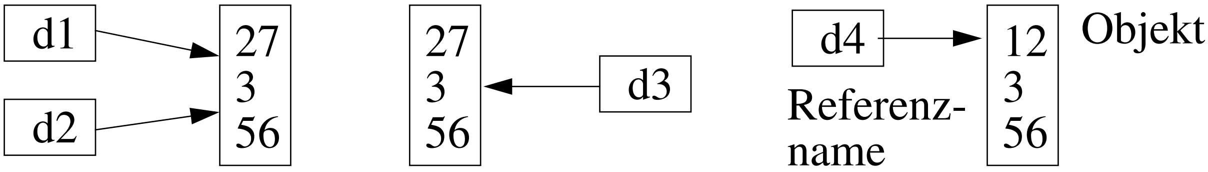
```
class Beispiel
{ ...
  Datum d1 = new Datum(23,03,56);
  Datum d2 = new Datum(27,03,56);
  System.out.println(d1.frueher_als(d2));
  /* true */
  System.out.println(d2.frueher_als(d1));
  /* false */
}
```

Hier wird die Funktion "frueher_als" aus dem Objekt d2 aufgerufen, und zwar mit Objekt d1 als Parameter.

Durch diese in der Klasse "Datum" definierten Operationen kann man nun also Datum-Objekte bezüglich früher / später vergleichen - ganz analog, wie man beispielsweise ganze Zahlen ("int-Objekte") mit dem Operator '<' vergleichen kann!

Gleichheit und "this"

- *Gleichheit* ist keine "natürlich gegebene" Eigenschaft. Sie muß erst geeignet definiert werden (Abstraktion!)



- welche der Paare von Referenzen sollen als gleich gelten?
- was würde ein Vergleich mit dem Operator "==" bringen?

```
class Datum...
    boolean frueher_als(Datum d)...
    boolean gleich(Datum d)
    { return !frueher_als(d) &&
             !d.frueher_als(this) ; };
...
```

Hier wird die Funktion "frueher_als" aus dem Objekt d aufgerufen, und zwar mit "einem selbst" als Parameter!

```
class Beispiel ...
  Datum d1 = new Datum(23,03,56);
  Datum d2 = new Datum(27,03,56);
  System.out.println(d1.gleich(d2));/*false*/
  Datum d3 = new Datum(23,03,56);
  System.out.println(d1.gleich(d3));/*true*/
```

- *Beachte:* "**this**" ist ein Schlüsselwort, mit dem stets ein *Zeiger auf das aktuelle Objekt* zurückgeliefert wird

- "gleich" ließe sich natürlich auch direkter, ohne Rückgriff auf "frueher_als" realisieren

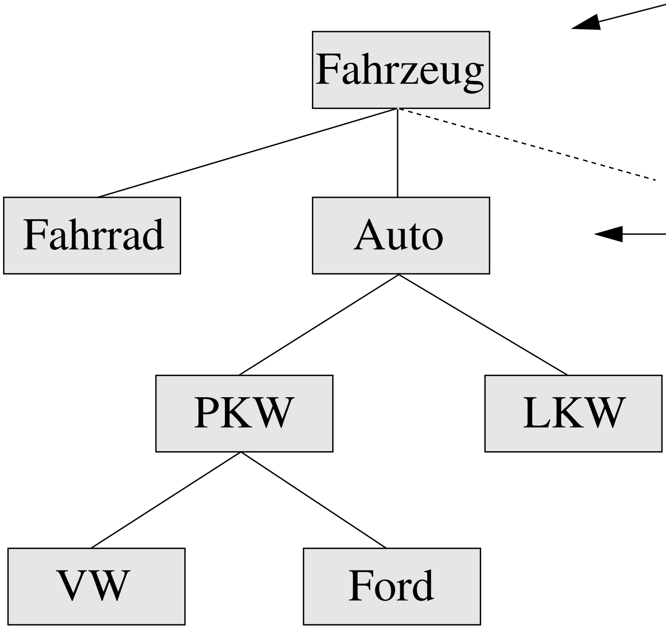
Vererbung ("inheritance")

- Idee: Vorhandene Klasse zur Definition einer neuen "ähnlichen" Klasse verwenden

Erweitert oder angepaßt an spezifische Bedürfnisse ("Spezialisierung")

Klassenhierarchie:

Basisklasse (auch: “Oberklasse”)



abgeleitete Klassen von "Fahrzeug" (auch: “Unterklassen” oder “erweiterte Klassen”)

Entwurf eines Klassenbaumes ist wichtige Aufgabe der Entwurfsphase beim objektorientierten Programmieren!

"is-a"-Relation

- Ein VW *ist ein* PKW *ist ein* Auto *ist ein* Fahrzeug
- Eine Trompete *ist ein* Blasinstrument *ist ein* Musikinstrument

- Ein Fahrzeug hat Räder ==> ein PKW hat Räder

Eigenschaften werden ("von oben nach unten") an abgeleitete Klassen *vererbt*!

Abgeleitete Klassen

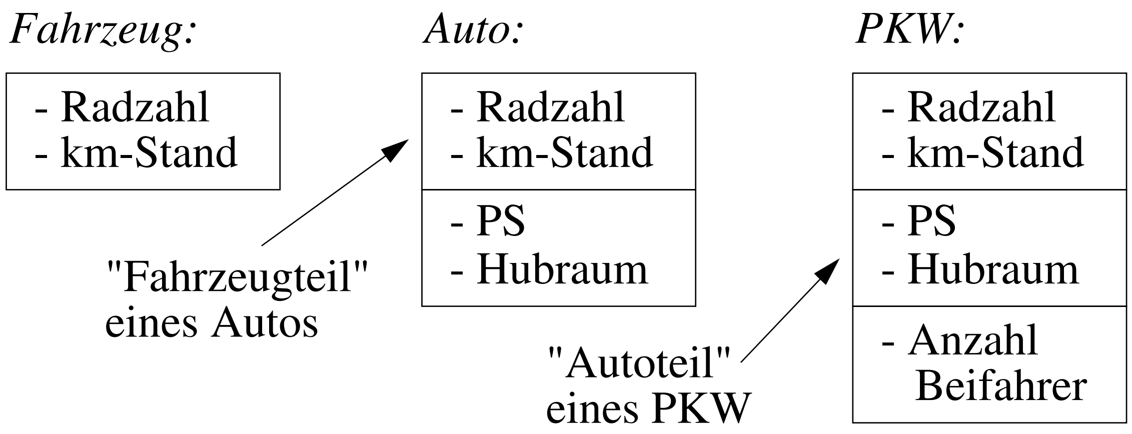
- Eine abgeleitete Klasse besitzt automatisch alle Eigenschaften der zugehörigen Basisklasse(n).

- Konkret: Sie besitzt alle "Attribute" und alle "Methoden" der Basisklassen.

- Außer: Es werden einige davon *unsichtbar gemacht* oder einige Methoden *redefiniert*.

Heißen noch genauso, tun aber etwas anderes!

- Eine abgeleitete Klasse kann *zusätzliche* Attribute und Methoden definieren.



- Eine Methode "Berechne_KFZ_Steuer" läßt sich nicht für alle Fahrzeuge gleichermaßen definieren.

==> Man würde z.B. in "Auto" eine Standardmethode vorsehen (Benutzung von "Hubraum"), jedoch für *spezielle* Fahrzeuge (z.B. Elektroautos) diese Methode *anders* definieren.

Ein Beispiel in Java

```
class Fahrzeug
{ public int Radzahl };

class Auto extends Fahrzeug
{ public int PS;
  public float Hubraum;};

class PKW extends Auto
{ public int Beifahrerzahl;
  void print()
  {System.out.println("Radzahl: "+ Radzahl
   + Beifahrerzahl: " + Beifahrerzahl);}
};

class LKW extends Auto
{ public float Zuladung; };

class Beispiel {
  public static void main (String args[]) {
  Fahrzeug f = new Fahrzeug();
  Auto a = new Auto();
  PKW p = new PKW();
  LKW l = new LKW();

  p.Beifahrerzahl = 5;
  p.PS = 70;
  p.Hubraum = 1794;
  p.Radzahl = 4;
  p.print();
```

Erweiterung der Klasse "Fahrzeug": Alles, was in "Fahrzeug" deklariert ist, gehört damit auch zu "Auto" (sowohl Attribute als auch Methoden) - mit gewissen Einschränkungen (--> später)

Auf "weiter oben" definierte Attribute kann ohne weiteres zugegriffen werden - diese sind Teil der abgeleiteten Klasse!

Hier werden Instanzen (also Objekte) der verschiedensten Hierarchiestufen erzeugt.

Zugriff auf Variablen und Methoden des mit 'p' bezeichneten PKW-Objektes.

p.Zuladung geht natürlich nicht!

Idee: Gemeinsame Aspekte heraus-*faktorisieren* und in eine übergeordnete Klasse einbringen.

Zuweisungskompatibilität

- Objekte von abgeleiteten Klassen können an Variablen vom Typ der Basisklasse zugewiesen werden.

> - `Fahrzeug f; Auto a; ... f = a;`
> - Variable `f` kann Fahrzeugobjekte speichern.
> - Ein Auto ist ein Fahrzeug.
> - Daher kann `f` auch Autoobjekte speichern.

- Die Umkehrung gilt jedoch nicht!

> - `a = f;` ist verboten!
> - Variable `a` kann Autoobjekte speichern.
> - Ein Fahrzeug ist aber kein Auto!

"Gleichnis" zur Zuweisungskompatibilität: Auf einem Parkplatz für Fahrzeuge dürfen Autos, PKWs, Fahrräder... abgestellt werden. Auf einem Parkplatz für Fahrräder jedoch keine beliebigen Fahrzeuge!

- Merke also:

genauer: *Zeiger* auf Basisklasse

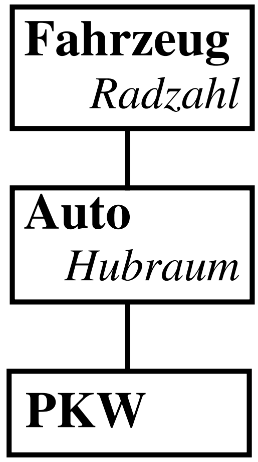
! *Eine Variable vom Typ "Basisklasse" darf auch auf ein Objekt der abgeleiteten Klasse zeigen!*

Man nennt diese Eigenschaft auch *Polymorphismus,* da ein Zeiger auf Objekte *verschiedenen Typs* zeigen kann. (Bzw. eine Variable Werte untersch. Typs haben kann.)

Beispiel: Eine Variable vom Typ "Zeiger auf Fahrzeug" kann zur Laufzeit sowohl zeitweise auf *PKW-Objekte*, als auch zeitweise auf *LKW-Objekte* zeigen.

Ein Java-Beispiel

```
class Fahrzeug  {... int Radzahl;}
class Auto extends Fahrzeug {...float Hubraum;}
class PKW extends Auto  ...
```




```
Fahrzeug f;  Auto a;  PKW p;
... new ...
p.Hubraum = 1702;
p.Radzahl = 4;
```

Ein PKW ist ein Auto und ein Fahrzeug

```
a = p;
f = p;
f = a;
```

Eine Fahrzeug-Variable darf PKW-Objekte und Auto-Objekte speichern

```
/* a = f; */
/* incompatible types for =... */
```

Andersherum geht es nicht!

```
a.Radzahl = 3;
a.Hubraum = 1100;
f = a;
System.out.println
        (f.Radzahl);
/* System.out.println(f.Hubraum); */;
/* No variable Hubraum defined in Fahrzeug */
```

Es wurde zwar Radzahl und Hubraum zugewiesen; auf Hubraum ist aber über f nicht zugreifbar!

- f.Hubraum ist aus gutem Grund verboten: Auf f könnte ja zufällig ein Fahrrad (ohne Hubraum!) "parken"!
  - Durch Umtypen kommt man aber notfalls auch über f an den Hubraum des Auto-Objektes: `System.out.println(((Auto)f).Hubraum);`
  - Aber wenn dort "gerade" kein Auto (sondern ein Fahrrad) parkt? Dann gibt es einen *Laufzeitfehler* "ClassCastException"!
  - Dem kann man wie folgt vorbeugen:

```
if (f instanceof Auto)
     System.out.println(((Auto)f).Hubraum);
else System.out.println("kein Auto, kein Hubraum!");
```

PKW/LKW-Beispiel (1)

```
class AUTO {
   int i, j;
   AUTO(int ii) {
     i = ii; j = ii+1;
   }
   void Meldung() {
     System.out.println("AUTO: "
     + i + "  " + j);
   }
}
```

Zuweisung von den "temporären" Variablen des Konstruktors an "permanente" Variablen

die Variable "i" existiert auf allen Ebenen!

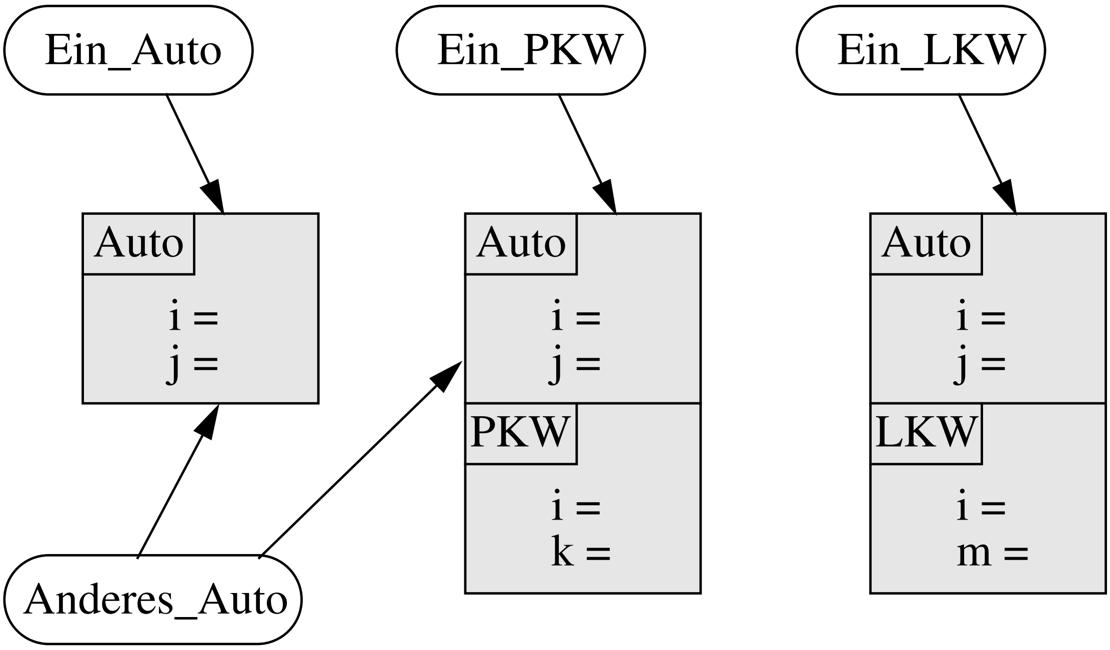
```
class PKW extends AUTO {
   int i, k;
   PKW(int ii) {
     super(ii*2);
     i = ii; k = ii+1;
   }
   void Meldung() {
     System.out.println("PKW:  "
     + i + "  " + j + "  " + k);
   }
}
```

hier wird der Konstruktor der Oberklasse aufgerufen

```
class LKW extends AUTO {
   int i, m;
   LKW(int ii) {
     super(ii*2);
     i = ii; m = ii+1;
   }
   void Meldung() {
     System.out.println("LKW: "
      + i + "  " + j + "  " + m);
   }
}
```

LKW ganz analog zu PKW (Variable "m" statt "k")

PKW/LKW-Beispiel (2)



Für Methoden, die in einer Unterklasse (hier: PKW bzw. LKW) definiert sind, ist eine überdefinierte Variable (hier: i) *verdeckt* und nicht direkt zugreifbar.

Allerdings kann mit "super.i" die obere Variable aus PKW und LKW erreicht werden

PKW/LKW-Beispiel (3)

```
AUTO Ein_AUTO, Anderes_AUTO;
PKW Ein_PKW, Anderer_PKW;

Ein_AUTO =  new AUTO(5);
System.out.println(Ein_AUTO.i); /* 5 */
System.out.println(Ein_AUTO.j); /* 6 */
/* System.out.println(Ein_AUTO.k); */
/* No variable k defined in class AUTO */
Ein_AUTO.Meldung(); /* AUTO: 5 6 */

Ein_PKW =  new PKW(22);
System.out.println(Ein_PKW.i); /* 22 */
System.out.println(Ein_PKW.j); /* 45 */
System.out.println(Ein_PKW.k); /* 23 */
Ein_PKW.Meldung(); /* PKW: 22 45  23 */
```

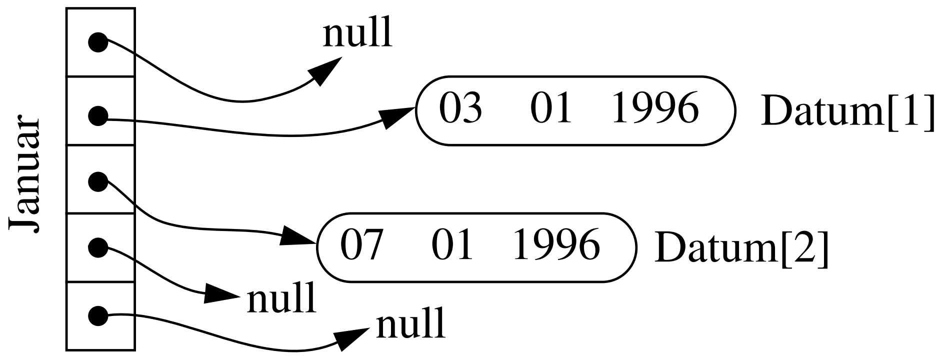
durch "casting" ("Typkonversion") kann auf die verdeckte Variable "i" dennoch zugegriffen werden

```
System.out.println(((AUTO)Ein_PKW).i); /* 44*/
System.out.println(((AUTO)Ein_PKW).j); /* 45*/

Anderes_AUTO = Ein_AUTO;
Ein_AUTO.i = 2;
System.out.println(Anderes_AUTO.i); /* 2 */
Anderes_AUTO.Meldung(); /* AUTO: 2  6 */
```

```
LKW Ein_LKW =  new LKW(333);
Ein_LKW.Meldung(); /* LKW: 333  667  334 */
/* Ein_LKW = Ein_PKW ; */
/* Incompatible type. Can't convert PKW to LKW */
```

Felder (arrays)

```
int [] x; // array of int
x = new int[7]; // Länge 7 (Indexbereich 0..6)

for (int i=0; i < x.length; i++) x[i]=17;

int [] w = new int[7]; // so geht es auch

y = x;      // y zeigt auf das gleiche Objekt
y[3] = 9;   // x[3] ist daher jetzt auch 9
```




```
Datum[] Januar = new Datum[32];

// Damit wird ein array mit 32 Verweisen
// auf potentielle Datum-Objekte angelegt.
// Die Verweise sind zunächst null.
// Erst so zeigen sie auf ein Datum-Objekt:

Januar[1] = new Datum(01,01,1996);
Januar[2] = new Datum(02,01,1996);
   ...
Januar[31]= new Datum(31,01,1996);
// Zugriff: ... Januar[27].Jahr ...
```

Zeichenketten (Strings)

- Zeichenketten sind durch 2 Standardklassen realisiert:
  - `String` : Zeichenkette selbst kann nicht verändert werden
  - `StringBuffer` : veränderbare Zeichenketten

```
String msg = "Die"; // String-Objekt wird
int i = 7;          // automatisch erzeugt
msg = new String("Die"); // So ginge es auch
msg = msg + " " + i;  // Konkatenation
msg = msg + " Zwerge";
System.out.println(msg); // Die 7 Zwerge
System.out.println(msg.length()); // 12

String b = msg;
msg = null;
System.out.println(b); // Die 7 Zwerge

class ReverseString {
String reverseIt(String source) {
   int i, len = source.length();
   StringBuffer dest = new StringBuffer(len);

   for (i = (len - 1); i >= 0; i--) {
       dest.append(source.charAt(i));
   }
   return dest.toString();
}
}
```

es gibt eine Vielzahl
von hübschen Methoden

Noch mehr Strings...

- Vergleich von Strings:

  - Vergleich mit == ist oft nicht sinnvoll (Referenzvergleich)
  - Statt dessen Wertevergleich: `s1.equals(s2)`
  - Lexikographische Anordnung mit `s1.compareTo(s2)`
    (liefert einen int <0, =0, oder >0)

- Es gibt eine Vielzahl von Methoden und Konstruktoren

  - Teilstrings
  - Umwandlung von Zeichen (z.B. Groß- / Kleinschreibung)
  - Umwandlung von anderen Datentypen in Strings (und umgekehrt)
  - Umwandlung von char- und byte-Felder in Strings
  - ...

- Mehr dazu in der API-Beschreibung zu java.lang

  - Gibt es online an verschiedenen Stellen und in Büchern, z.B.:

```
http://java.sun.com/products/jdk/1.2/
docs/api/overview-summary.html
```

Auszug aus der API-Beschreibung (1)

Class String

```
public final class java.lang.String
   extends java.lang.Object
{
      // Constructors
public String();
public String(byte ascii[], int hibyte);
public String(byte ascii[], int hibyte,
                int offset, int count);
public String(char value[]);
public String(char value[], int offset, int count);
public String(String value);
public String(StringBuffer buffer);

      // Methods
public char charAt(int index);
public int compareTo(String anotherString);
public String concat(String str);
public static String copyValueOf(char data[]);
public static String
      copyValueOf(char data[], int offset, int count);
public boolean endsWith(String suffix);
public boolean equals(Object anObject);
public boolean equalsIgnoreCase(String anotherString);
public void getBytes(int srcBegin, int srcEnd,
                        byte dst[], int dstBegin);
public void getChars(int srcBegin, int srcEnd,
                        char dst[], int dstBegin);
public int hashCode();
public int indexOf(int ch);
public int indexOf(int ch, int fromIndex);
public int indexOf(String str);
public int indexOf(String str, int fromIndex);
public String intern();
public int lastIndexOf(int ch);
public int lastIndexOf(int ch, int fromIndex);
public int lastIndexOf(String str);
public int lastIndexOf(String str, int fromIndex);
public int length();
...
```

Auszug aus der API-Beschreibung (2)

compareTo

```
public int compareTo(String anotherString)
```

Compares two strings lexicographically.

Parameters:
anotherString - the String to be compared

Returns:
The value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

concat

```
public String concat(String str)
```

Concatenates the string argument to the end of this string. If the length of the argument string is zero, then this object is returned.

Parameters:
str - the String which is concatenated to the end of this String

Returns:
A string that represents the concatenation of this object's characters followed by the string argument's characters.

copyValueOf

```
public static String copyValueOf(char data[])
```

Returns:
a String that contains the characters of the character array.

Parameters:
data - the character array

Abstrakte Methoden und Klassen

```
abstract class Sort {

abstract boolean kleiner (Sort y);

static void sort(Sort[] Tab) {
  for (int i=0; i<Tab.length; i++)
    for (int j=i+1; j<Tab.length; j++)
      if (Tab[i].kleiner(Tab[j])) {
        Sort swap = Tab[i];
        Tab[i] = Tab[j];
        Tab[j] = swap;
      }
}
}
```

wieso static?

Achtung: Es wird *absteigend* sortiert!

Das einfache Sortierverfahren ("deletion sort") ist ineffizient!

- Wir fordern, daß die zu sortierenden Objekte vom Typ einer von Sort abgeleiteten Klasse sind.
- In der abgeleiteten Klasse muß außerdem die Funktion "kleiner" realisiert werden.

Als *totale Ordnungsrelation* auf den Objekten!

- Unabhängig davon, wie die Relation "kleiner" definiert ist, funktioniert unser Sortierverfahren!
- Das Sortierverfahren kann also bereits implementiert (und getestet) werde, bevor überhaupt die Daten selbst bekannt sind!
- Einmal entwickelt, kann man den Algorithmus auch zum *Sortieren anderer Datentypen* verwenden! (int, float, Brüche als rationale Zahlen, Zeichenketten...)
- Sort ist eine *abstrakte Klasse* (von solchen können keine Objekte erzeugt werden, sie dienen nur dazu, hiervon abgeleitete Klassen zu definieren).

Überladen von Methoden

- Gleicher Methodenname bei unterschiedlicher *Signatur* (=Typ und Anzahl der Parameter) möglich
  - wenn sich mehrere Methoden qualifizieren, wird die speziellste genommen
  - auch Konstruktoren können überladen werden

```
class Auto {...
class Bus extends Auto {...
class Zug {...

class Test {
  int f(int x){ return 1; }
  int f(double y) { return 2;  }
  int f(char z) { return 3;  }
  // char f(char z) { return 4;  }
  // ERROR: Methods can't be redefined
  // with a different return type
  int f(Object x) {return 5; }
  int f(Auto x) { return 6; }
  short f(int x, double y){ return 7; };
...
class Beispiel {...
   int i; int j = 0;
   Test t = ...
   Object o = ...; Auto a = ...;
   Bus b = ...; Zug z = ...;

   i=t.f(i);        // 1
   i=t.f(3.14);     // 2
   i=t.f('c');      // 3
   i=t.f(o);        // 5
   i=t.f(a);        // 6
   i=t.f(b);        // 6
   i=t.f(z);        // 5
   i=t.f(5, 3.14);  // 7
...
```

Alle Klassen erweitern (direkt oder indirekt) die Standardklasse Object

2, wenn es kein f(int x) gibt!

Fehler, wenn es kein f(double x) gibt!

1, wenn es kein f(char x) gibt!

Schnittstellen ("Interfaces")

- Interface = (abstrakte) Klasse, die alle Methoden nur deklariert, aber nicht implementiert
  - enthält also nur (implizit) abstrakte Methoden (und Konstanten)
  - Bsp:

```
interface Menge {
    int cardinal();
    void insert (Object x);
    void remove (Object x);
}
```

- Interface muß von anderen Klassen implementiert werden
  - Bsp:

```
class S implements Menge {
...
public int cardinal(); {
    ...
    while ... i++ ...
    return i;
}
```

- Der Typ des Interfaces (hier: "Menge") kann mit seinen Methoden anderswo benutzt werden
  - Bsp:

```
Menge M;
M.insert(...);
```

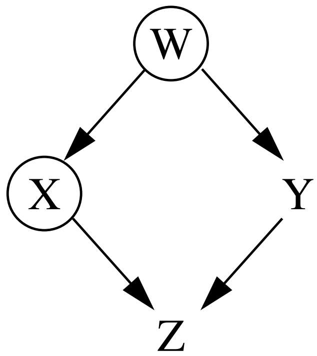
- Interfaces können *mehrere* andere erweitern
  - Bsp:

```
interface I extends A, B {
    int M();
}
```

  - I "enthält" alle Methoden von A und B (und zusätzlich M)
  - eine Klasse dagegen kann nur eine einzige Klasse erweitern

Mehrfachvererbung

- Klassen können *nicht* von mehreren Eltern erben
- Interfaces dienen als (teilweise!) Ersatz dafür
  - das sogen. "diamond inheritance problem" so manchmal lösbar:




```
interface W {...}
interface X extends W {...}
class Y implements W {...}
class Z extends Y implements X {...}
```

  - die von Z ererbten Attribute und Methodenimplementierungen können nur aus Y (und nicht indirekt doppelt aus W stammen)

  - Namenskonflikte beim Erben von mehreren Eltern müssen allerdings gelöst werden...

Pakete

- Paket = (zusammengehörige) Menge von Klassen
  - und Interfaces und Unterpakete

- Hierarchischer Aufbau
  - "lang" im Paket "java" --> "java.lang"
  - spiegelt sich in der Verzeichnishierarchie wieder

- Wichtig für Strukturierung und Zugriffskontrolle
  - Klassen und Attribute von Klassen (z.B. Methoden) sind ohne weitere Angaben nur im eigenen Paket sichtbar / zugreifbar

- Klassen befinden sich immer in Paketen
  - Paketdeklaration direkt am Anfang einer Quelldatei, z.B.
    `package` `abc;`
  - Falls package-Deklaration fehlt: "unnamed package"

- Attribute / Methoden von Klassen können vollqualifiziert (d.h. mit dem Paketnamen) benannt werden
  - z.B.: $\underbrace{\texttt{java.lang}}_{\text{Paket}}\texttt{.}\underbrace{\texttt{String}}_{\text{Klasse}}\texttt{.}\underbrace{\texttt{substring}}_{\text{Methode}}$

- Importieren von Klassen (als Namensabkürzung)
  - z.B. **`import`** `java.util.Random`
    (es wird diese Klasse importiert und kann als "Random" benutzt werden)
  - oder **`import`** `java.util.*`
    (es wird alles aus diesem Paket importiert)

Die Java-Umgebung (API, Standard-Pakete)

java.lang

Package that contains essential Java classes, including numerics, strings, objects, compiler, runtime, security, and threads. This is the only package that is automatically imported into every Java program.

java.io

Package that provides classes to manage input and output streams to read data from and write data to files, strings, and other sources.

java.util

Package that contains miscellaneous utility classes, including generic data structures, bit sets, time, date, string manipulation, random number generation, system properties, notification, and enumeration of data structures.

java.net

Package that provides classes for network support, including URLs, TCP sockets, UDP sockets, IP addresses...

java.awt

Package that provides an integrated set of classes to manage user interface components such as windows, dialog boxes, buttons, checkboxes, lists, menus, scrollbars, and text fields. (AWT = Abstract Window Toolkit.)

java.awt.image

Package that provides classes for managing image data, including color models, cropping, color filtering, setting pixel values, and grabbing snapshots.

java.applet

Package that enables the creation of applets through the Applet class.

java.util

- Das Paket "java.util" enthält einige interessante Klassen zur Verwaltung von Daten:

Class BitSet

This class implements a vector of bits that grows as needed. Individual bits can be examined, set, or cleared.

Class Hashtable

This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value. (Methods: put, get, remove, contains, size...)

Class Properties   (extends Hashtable)

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and and its corresponding value in the property list is a string.

Class Stack

Methods: push, pop, peek, empty...

Class Vector

Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

- Einige Standardpakete (z.B. java.awt*) sind komplex
  - Vielzahl von Methoden etc.
  - nicht einfach zu benutzen

Zugriffsmodifikatoren

- Reihenfolge der einzelnen Modifikatoren beliebig
- Durch Modifikatoren wird i.w. der Zugriff (d.h. Sichtbarkeit des Namens) geregelt
  - leider etwas verwirrend
  - es gibt Tabellen, in denen man die anzugebenden Modifikatoren je nach gewünschter Situation nachsehen kann

---

Class Modifiers:

final: no sub-classes
public: usable from other packages
abstract: no instances, only sub-classes

Variable Modifiers:

final: constant
static: class variable
private: use only inside class
(no modifier): + in Package
protected: + sub-classes
public: anywhere

Method Modifiers:

final: no overriding
static: class method
abstract: implement in subclass
native: implemented in C
private, public, protected: like variables.

- "private protected": *nur* in Unterklassen (und eigener Klasse) sichtbar
- es gibt weitere Modifikatoren: **synchronized**, **volatile** (beide bei Threads) und **transient** (bei persistenten Objekten, derzeit noch nicht benutzt)

Ausnahmen (Exceptions)

- Ausnahmen sind Fehlerereignisse
  - werden oft vom System ausgelöst ("throw")
  - können aber auch explizit im Programm ausgelöst werden
  - können abgefangen und behandelt werden ("catch")

- Bessere Strukturierung durch "try" und "catch":

```
void readFile() {
  try {
    // open the file;
    // determine its size;
    // allocate that much memory;
    // read the file into memory;
    // close the file;
  } catch (fileOpenFailed) {
    // doSomething;
  } catch (sizeDeterminationFailed) {
    // doSomething;
  } catch (memoryAllocationFailed) {
    // doSomething;
  } catch (readFailed) {
    // doSomething;
  } catch (fileCloseFailed) {
    // doSomething;
  }
}
```

  - Fehlerbehandlung muß auf diese Weise nicht mit dem "normalen" Programmcode verwoben werden

Ausnahmen - ein E/A-Beispiel

```
import java.io.*;

public class EA_Beispiel
//  Prints "Hello World" to a file
//  specified by the first parameter.
{
   public static void main(String args[])
```

Da wir Fehler selbst abfangen, können wir auf "throws..." verzichten!

```
   {
      FileOutputStream out = null;

      //  Attempt to open the file, if we
      //  can't display error and quit
      try
      {
         out = new FileOutputStream(args[0]);
      }
      catch (Throwable e)
```

Diese Fehlerklasse ganz oben in der Hierarchie und fängt damit alles ab

```
      {
         System.out.println("Error in opening file");
         System.exit(1);
      }
```

Z.B.: Zugriffsrechte "falsch"

```
      PrintStream ps = new PrintStream(out);
```

Fehler hierbei würden nicht abgefangen!

```
      try
      {
         ps.println("Hello World");
         out.close();
      }
      catch(IOException e)
```

Über diese Variable kann man mehr über den Fehler erfahren

```
      {
         System.out.println("I/O Error");
         System.exit(1);
      }
   }
}
```

Shortcut zum Verlassen des Programms

Fehlerarten

- Typische Situationen, in denen Ausnahmen auftreten können:
    - Ein- / Ausgabe (IOException)
    - Sockets, URL-Verbindungen (z.B.MalformedURLException)
    - Erzeugen von Objekten mit "new"
    - Typkonvertierung (z.B. NumberFormatException)

- Wichtige Fehlerklasse: Laufzeitfehler
    - können, müssen aber nicht abgefangen werden
    - Beispiele: Zugriff über Null-Referenz; int / 0; Indexfehler bei arrays

```
try {
   value = value / x;
}
catch(ArithmeticException e){
   System.out.println("Division durch 0?");
}
```

- Alle anderen Fehlerarten müssen behandelt werden
    - entweder durch try / catch in der Methode
    - oder durch Angabe, daß die Methode diese Ausnahme selbst wieder auslöst (und damit weiterreicht), z.B.:

```
import java.io.*;
public Eine_Methode (...)
   throws java.io.IOException
{... read ...}
```

Fehlerbehandlung

- Fehlerbehandlung ist nicht immer einfach

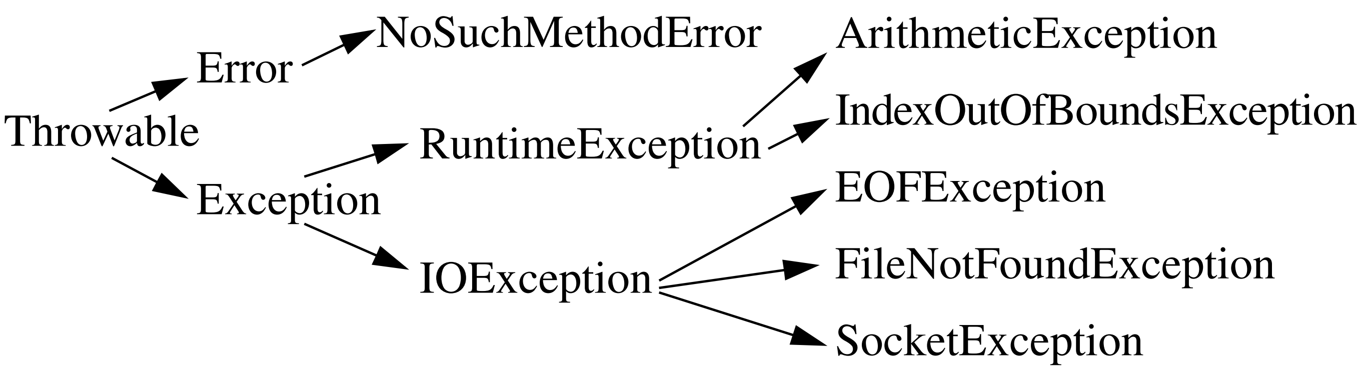
- Meldung an den Benutzer (wenn es einen gibt...), aber was dann?
- Oft ist es dann ratsam, reservierte Ressourcen wieder freizugeben
- Die oberste Fehlerklasse "Throwable" stellt einige sinnvolle Methoden zur Verfügung, z.B. printStackTrace oder getMessage (letzteres für genauere Fehlermeldungen):

```
try { ... }
catch (Exception e) {
    System.err.println("Fehler: "
    + e.getMessage());
```

- Suche eines passenden "error handlers"

- Laufzeitsystem durchsucht den Laufzeitkeller (rückwärts)
- sucht den nächsten passenden handler
- "passend" entsprechend der Hierarchie der Fehlerklassen

- Hierarchie der Fehlerklassen (Auszug):



Definieren eigener Ausnahmen

- Ausnahmen sind Objekte!
  - was auch sonst...

- Eigener Ausnahmetyp muß von java.lang.Throwable (indirekt) abgeleitet sein

- Kann dann mit "throw" ausgelöst werden

```
class IllegalesDatum extends Throwable {
  IllegalesDatum (int Tag,int Monat,int Jahr)
  {
    super ("Fehlerhaftes Datum ist:" +
           Tag + "." + Monat + "." + Jahr);
    // und E-mail an den Boss schicken...
  }
}
```

Der Konstruktor von Throwable erwartet einen String, der als Fehlermeldung (mit dem Stack-Trace) ausgegeben wird

```
class Datum
...
    void Setzen (int T, int M, int J)
      throws IllegalesDatum {
      Tag = T; Monat = M; Jahr = J; ;
      if (Tag > 31) throw new
         IllegalesDatum (Tag, Monat, Jahr);
    }

class Beispiel ...
d.Setzen(47,03,97); // Zeile 49 in der Datei
...
```

```
Fehlerhaftes Datum ist:47.3.97
        at Datum.Setzen(Datum.java:27)
        at Beispiel.main(Datum.java:49)
```