

Objektorientierte Programmierung

Kapitel 11: Klassen

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2014/15

<http://www.informatik.uni-halle.de/~brass/oop14/>

Inhalt

- 1 Einfache Klassen, Referenzen
 - Strukturen, Attribute, Referenzen, Objekt-Erzeugung
- 2 Zugriffsschutz und Methoden
 - Von Strukturen zu Klassen, Methoden, Zugriffsschutz
 - Beispiel: Klasse für Datumswerte
- 3 Konstruktoren
 - Konstruktoren und Initialisierung
- 4 Statische Komponenten
 - Statische Variablen und Methoden
 - Konstanten
 - Beispiel: Verkettete Liste
- 5 Syntax
 - Syntax von Klassen-Deklarationen



Einfache Klassen: Attribute (1)

- Historisch sind Klassen aus Strukturen (Records) entstanden, die es schon in nicht-objektorientierten Sprachen gab.
- Zunächst enthielten Variablen nur Werte primitiver Typen.
- Mit Arrays gab es die Möglichkeit, mehrere Variablen zu einer Einheit zusammenzufassen. Dabei gilt:
 - Alle Variablen haben den gleichen Typ.
 - Die einzelnen Variablen werden über Zahlen identifiziert.
- Strukturen sind eine andere Möglichkeit, Variablen zu einer Einheit zusammenzufassen:
 - Die Variablen können unterschiedlichen Typ haben.
 - Die einzelnen Variablen werden über Namen identifiziert.

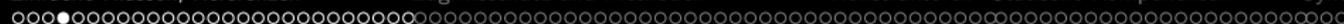


Einfache Klassen: Attribute (2)

- Im letzten Kapitel wurden schon Variablen in Klassen deklariert, das waren aber statische Variablen.
- Sie existieren nur ein einziges Mal in der Klasse.
Und entsprechen "globalen Variablen".
- Ohne das Schlüsselwort "static" hat jedes Objekt der Klasse dagegen seine eigene Kopie der Variablen:

```
class Datum {  
    int tag;  
    int monat;  
    int jahr;  
}
```

Natürlich haben Klassen normalerweise auch Methoden, aber wenn man historisch von Strukturen kommt, ist das erst der zweite Schritt.



Einfache Klassen: Attribute (4)

- Der neu definierte Klassentyp läßt sich wie ein primitiver Typ verwenden, z.B. kann man Variablen dieses Typs deklarieren:

```
Datum d;
```

- Allgemein können mit Klassen neue Datentypen definiert werden, die über die eingebauten Typen wie `int`, `double` u.s.w. hinausgehen.

“A program that provides types that closely match the concepts of the application tends to be easier to understand and easier to modify than a program that does not.” [Stroustrup: The C++ Prog. Lang., 2000]

- Klassen sind ein Mittel zur Abstraktion:
 - Man kann auf einer höheren Programmebene von den einzelnen Attributen (Komponenten) abstrahieren und
 - das Objekt als Ganzes behandeln.



Referenzen (1)

- Es gibt aber einen wichtigen Unterschied von Klassen zu primitiven Typen:
 - Wenn man eine Variable von einem primitiven Typ, z.B. `int`, deklariert, reserviert der Compiler Speicherplatz, in dem man eine Zahl speichern kann.
 - Wenn man eine Variable von einem Klassentyp deklariert, wird nur Speicherplatz für eine Referenz auf ein Objekt der Klasse reserviert, aber nicht für das Objekt selbst.
- Man kann es auch so ausdrücken:
 - Werte von einem Klassentyp sind Referenzen auf Objekte (Hauptspeicher-Adressen von Objekten, Zeiger auf Objekte),
 - und nicht die Objekte selber.



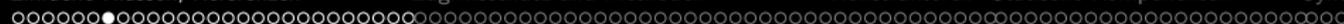
Objekterzeugung: `new`

- Um mit einer Variablen vom Typ `Datum` etwas anfangen zu können, muss man eine Referenz auf ein Objekt eintragen.
- Man erzeugt ein neues Objekt mit dem Schlüsselwort `new`, dem Namen der Klasse, und einer Parameterliste für den Konstruktor (s.u.):

```
d = new Datum();
```

Hierbei wird der Speicherplatz für die drei Variablen `tag`, `monat`, `jahr` reserviert (und ggf. weitere Verwaltungs-Information).

- Dies ist die einzige Möglichkeit, wie man in Java ein neues Objekt anlegen kann (Ausnahme: Reflection).
- Das Objekt heißt auch Instanz der Klasse.
Man erzeugt Objekte durch Instanziierung der Klasse.

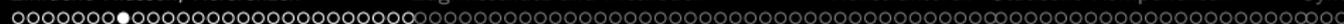


Attribut-Zugriff (1)

- Nun kann man auf die Komponenten des Objektes (Variablen im Objekt, Attribute) zugreifen, z.B.

```
d.tag = 24;
```

- Der Zugriff auf ein Attribut (“field access expression”) besteht also aus:
 - einer Expression, die einen Wert eines Klassentyps liefert,
Im Beispiel ist das einfach die Variable `d`, es könnte aber z.B. auch ein Methoden-Aufruf sein, oder selbst ein Attribut-Zugriff.
 - dem Zeichen “.”,
 - und dem Namen einer Variablen, die in der Klasse deklariert ist (Attribut, Instanzvariable).
Da es sich um drei einzelne Token (Wortsymbole) handelt, kann man links und rechts vom “.” Leerzeichen/Zeilenumbrüche einfügen.



Attribut-Zugriff (2)

- Der Zugriff auf ein Attribut ist selbst wieder eine Expression, die eine Variable des entsprechenden Typs liefert.

Daher kann der Attribut-Zugriff als Teil eines komplexeren Wertausdrucks verwendet werden, im Beispiel in der Zuweisung.

- Selbstverständlich sind auch lesende Zugriffe möglich:

```
System.out.println(d.tag);
```

Dies würde 24 ausgeben.

- Java initialisiert die Attribute eines Objektes automatisch (Instanzvariablen und statische Variablen). Z.B. wäre

```
System.out.println(d.jahr);
```

möglich (ohne Fehler) und würde 0 drucken.

Unterschied zu lokaler Variable: Dort würde der Compiler den Zugriff nicht erlauben. In beiden Fällen stellt Java aber sicher, dass nicht auf uninitialisierte Variablen zugegriffen wird.

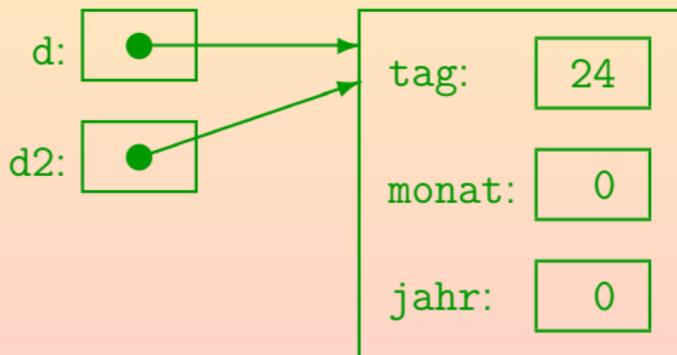


Referenz vs. Objekt (1)

- Wenn man ein Objekt erzeugt hat, kann man eine Referenz darauf auch in weitere Variablen speichern:

```
Datum d2 = d;
```

- Die Situation ist jetzt:



- Beide Variablen verweisen also auf das gleiche Objekt.



Referenz vs. Objekt (2)

- Wenn man das Objekt über eine Variable ändert, kann man die Änderung auch über die andere Variable bemerken:

```
d2.monat = 12;  
System.out.println(d.monat); // druckt 12
```

- Die beiden Ausdrücke "d.monat" und "d2.monat" bezeichnen also tatsächlich die gleiche Variable.

Nämlich das Attribut `monat` im gleichen Objekt.

- Die Variablen `d` und `d2` sind dagegen nicht gleich.

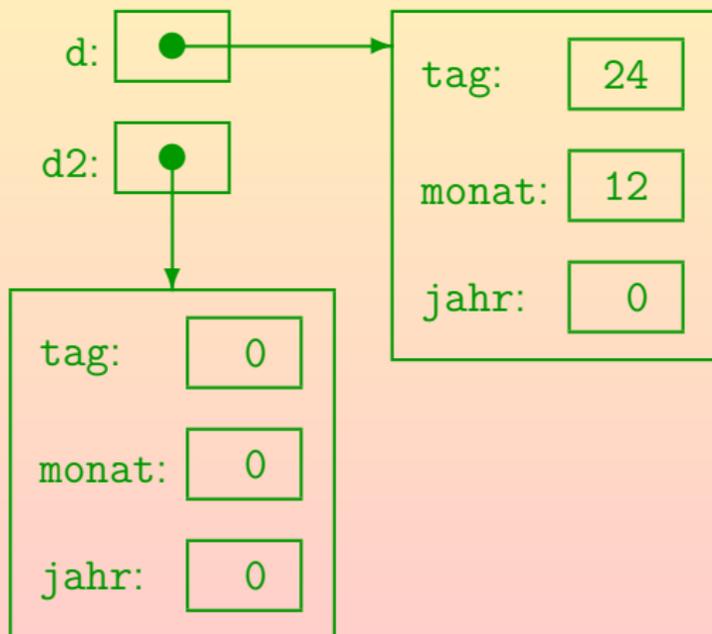
Nur ihr Inhalt ist gleich (die Hauptspeicher-Adresse des bisher einzigen Objektes der Klasse `Datum`).

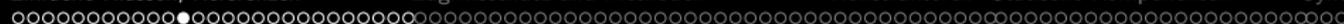
- Man kann daher die Variablen `d` und `d2` unabhängig von einander ändern.



Referenz vs. Objekt (3)

- Man kann nun z.B. ein zweites Objekt erzeugen, und dieses Objekt `d2` zuweisen: `d2 = new Datum();`





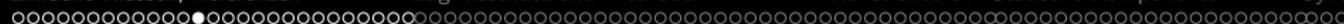
Referenz vs. Objekt (4)

- Nun betreffen Änderungen über die beiden Variablen unterschiedliche Objekte:

```
d.jahr = 2014;
d2.tag = 1;
d2.monat = 1;
d2.jahr = 2015;
System.out.println(d.tag + "." + d.monat
                   + "." + d.jahr);
    // Druckt: 24.12.2014
System.out.println(d2.tag + "." + d2.monat
                   + "." + d2.jahr);
    // Druckt: 1.1.2015
```

- **Aufgabe:** Was würde folgende Anweisung drucken?

```
System.out.println(d2.tag + d2.monat + d2.jahr);
```



Methoden und Referenzen (1)

- Die Parameterübergabe funktioniert wie eine Zuweisung.
- Daher bekommt trotz “Call by Value” die aufgerufene Methode die Möglichkeit, das übergebene Objekt zu ändern:

```
static void silvester(Datum x, int jahr) {  
    x.tag = 31;  
    x.monat = 12;  
    x.jahr = jahr;  
}
```

- Nach Aufruf von

```
silvester(d, 2014);
```

steht z.B. in `d.tag` der Wert 31.

Es passiert das Gleiche wie wenn man `x = d`; ausführen würde und dann den Rumpf der Methode.



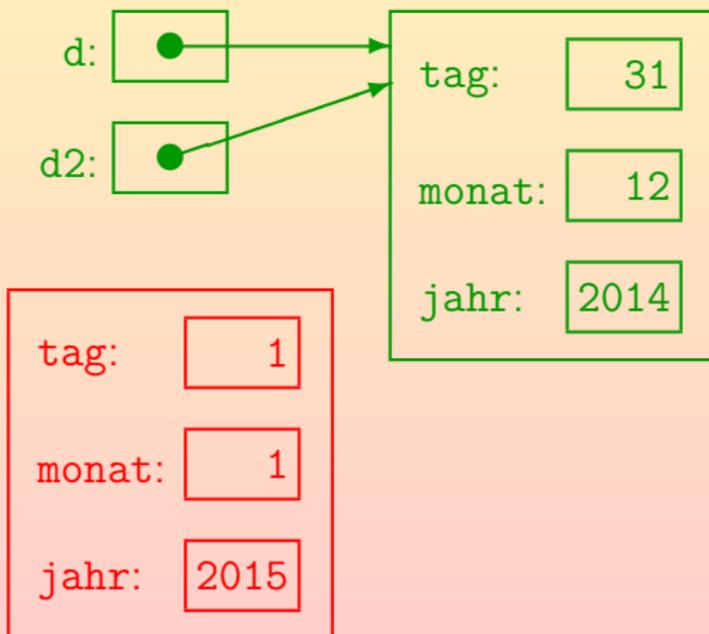
Methoden und Referenzen (2)

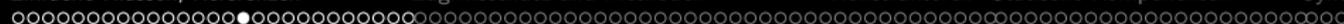
- Ein Methodenaufruf ist eine sehr häufige Operation und muss schnell gehen.
 - Es ist guter Programmierstil, viele eher kleine Methoden zu verwenden, und nicht das ganze Programm in eine riesige `main`-Methode zu stecken. Dafür soll der Programmierer nicht mit einem Leistungsverlust bestraft werden.
- Ganze Objekte beim Aufruf zu kopieren ist mindestens für größere Objekte zu aufwendig.
 - In C++ hat man die Wahl, aber entscheidet sich meist gegen das Kopieren.
- Oft will man ja auch, dass eine aufgerufene Methode die Möglichkeit hat, den Objekt-Zustand zu ändern.
- Wenn man das nicht will, kann man das nicht beim Aufruf sicher stellen, nur beim Entwurf der Klasse des übergebenen Objektes (nach Initialisierung nicht änderbar, s.u.).



Garbage Collection (1)

- Nach der Zuweisung `d2 = d`; gibt es ein Objekt, das nicht mehr zugreifbar ist:





Garbage Collection (2)

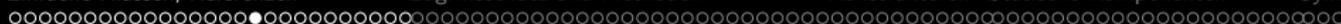
- Die Laufzeit-Umgebung von Java enthält einen “Garbage-Collector” (“Müll-Einsammler”).
- Dieser bemerkt nach einiger Zeit, dass das Objekt von den Programm-Variablen aus nicht mehr zugreifbar ist.

Die Analyse ist recht aufwendig. Daher wird sie ausgeführt, wenn der Rechner sonst nichts zu tun hat, oder der Hauptspeicher langsam knapp wird.

- Da das Objekt sowieso nicht mehr angesprochen werden kann, kann es gelöscht werden, d.h. der von ihm belegte Hauptspeicher freigegeben werden.

Der Hauptspeicher kann dann für neue Objekte wiederverwendet werden.

In C++ ist dagegen der Programmierer dafür verantwortlich, dass mit `new` angeforderter Speicher wieder freigegeben wird (mit `delete`). Das ist effizienter als ein Garbage Collector, aber auch fehleranfälliger.



Null-Referenz (1)

- Mit dem Schlüsselwort “`null`” wird die sogenannte Null-Referenz bezeichnet.

Oft ist es eine Adresse mit lauter Null-Bits, d.h. die Hauptspeicher-Adresse 0, daher der Name. In Pascal heißt es “`nil`”.

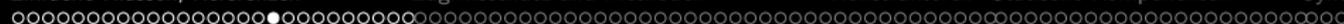
- Sie ist verschieden von jeder Referenz auf ein tatsächlich existierendes Objekt.

Falls man die Hauptspeicher-Adresse 0 nimmt, muss also garantiert sein, dass dort kein Objekt stehen kann. Viele CPUs speichern dort Interrupt-Vektoren, so dass der Speicherbereich tatsächlich belegt ist.

- Man kann `null` an jede Variable von einem Referenztyp zuweisen:

```
Datum d = null;
```

An Variablen von einem primitiven Typ kann man `null` dagegen nicht zuweisen.



Null-Referenz (2)

- Wenn `d` die Null-Referenz enthält, führt jeder Versuch zum Zugriff auf einer Komponente, z.B.

```
System.out.println(d.tag);
```

zur `NullPointerException` (Laufzeit-Fehler).

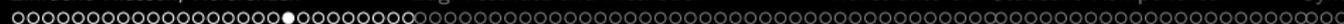
Die Variable `d` zeigt dann ja gerade auf kein Objekt, deswegen kann man dann auch auf keine Komponente zugreifen. Den Wert `null` kann man dagegen drucken: `System.out.println(d);` gibt keinen Fehler.

- Man kann aber selbstverständlich vor dem Zugriff testen, ob `d` die Null-Referenz enthält:

```
if(d != null) System.out.println(d.tag);
```

- Wenn eine Klasse selbst Attribute von einem Referenztyp enthält, so setzt die automatische Initialisierung sie auf `null`.

Das gilt auch für `String` (Zeichenketten). D.h. nicht der leere `String`!



Objekt-Identität (1)

- Zu den objektorientierten Basiskonzepten gehört die Objektidentität: Zwei Objekte können auch dann verschieden sein, wenn sie in allen Komponenten übereinstimmen.
- Dem gegenüber sind Datenwerte gleich, wenn sie in allen Komponenten übereinstimmen.

In Java gibt es Datenwerte mit mehreren Komponenten zunächst höchstens bei den Gleitkommazahlen (`float`, `double`), die Mantisse und Exponent enthalten. Es gibt aber auch Klassen, bei denen man sich eher ein Verhalten wie bei Datenwerten wünscht, dies würde z.B. für `Datum` zutreffen (s.u.).

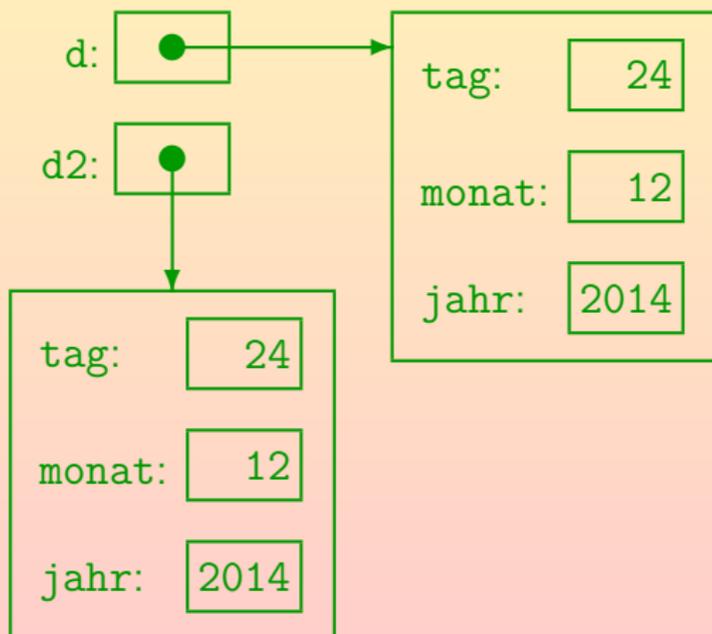
- Wenn ein Datenwert geändert wird, ist es ein neuer Wert. Ein Objekt kann dagegen geändert werden, aber das gleiche Objekt bleiben.

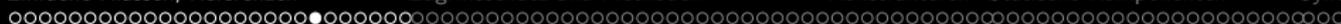
Der Zustand eines Objektes ist die Belegung seiner Attribute mit konkreten Werten. Ein Objekt verhält sich hier wie eine Variable.



Objekt-Identität (2)

- Hier gilt $d \neq d2$ (es sind unterschiedliche Objekte, wenn auch alle Komponenten gleich sind):

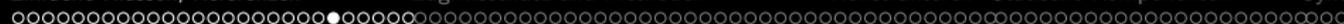




Objekt-Identität (3)

- **Datum** ist hier ein schlechtes Beispiel. Man wünscht eher ein Verhalten wie bei Werten, und kann dies auch mit einer Klasse realisieren:
 - Man kann eine Klasse so definieren, dass die Objekte nach der Initialisierung nicht mehr geändert werden können (s.u.).
 - Die Methode `equals` kann so definiert werden, dass sie `true` liefert, wenn die Objekte in allen Komponenten übereinstimmen.

Der Operator `==` testet dagegen, ob es das gleiche Objekt ist (gleiche Referenz/Hauptspeicher-Adresse). Wenn man Objekte nicht einfach mit `new`, sondern mit einer Klassenmethode erzeugt, kann man dafür sorgen, dass es nicht zwei Objekte gibt, die in allen Komponenten übereinstimmen (→ Beispiel am Ende des Kapitels). Dann funktioniert `==`.



Objekt-Identität (4)

- Anders wäre es mit **Termin**-Objekten (Eintrag in einem Terminkalender):
 - Hier könnten alle Komponenten geändert werden, aber es dennoch der gleiche Termin bleiben.

Dass es nicht ein neues **Termin**-Objekt wird, ist z.B. wichtig, wenn es an anderer Stelle Referenzen auf das Objekt gibt (z.B. beim Projekt). Bei einer Terminverschiebung möchte man diese Referenzen vermutlich nicht ändern. Dies sind natürlich Entwurfsentscheidungen für die Software, die man bewusst fällen muss (und gut dokumentieren).
 - Ob es zwei **Termin**-Objekte möglich sein sollen, die in allen Komponenten übereinstimmen, ist dagegen fraglich.

Es würde allerdings zusätzlichen Programmcode erfordern, das zu verhindern. Außerdem ist nicht klar, ob die Klasse so einfacher zu benutzen ist. In jedem Fall muss man sich der Entscheidung wieder bewusst sein.



Typ-Gleichheit

- Auch wenn zwei Klassen die gleichen Komponenten in der gleichen Reihenfolge haben (also einen identischen {...}-Teil), gelten Sie als unterschiedlich:

```
class DatumX {  
    int tag;  
    int monat;  
    int jahr;  
}
```

Java verwendet (wie C/C++) Namensgleichheit und keine Strukturgleichheit.

- Die Zuweisung `DatumX x = d;` gibt eine Fehlermeldung:

```
TypeTest.java:17: incompatible types  
found   : Datum  
required: DatumX  
DatumX x = d;  
      ^
```



Gleich benannte Komponenten

- Das Beispiel zeigt auch, dass Komponenten (Attribute, Methoden) in unterschiedlichen Klassen gleiche Namen haben dürfen.

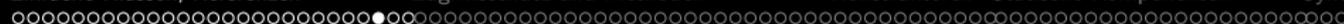
Die Klassendeklaration von `DatumX` ist ja in Ordnung, nur die Zuweisung nicht.

- Nur innerhalb einer Klasse darf es nicht zwei Attribute mit gleichem Namen geben.

Es darf ein Attribut und eine Methode mit gleichem Namen geben, weil Methoden anhand der folgenden "(" unterschieden werden können: Variablen und Methoden liegen in unterschiedlichen Namensräumen.

- Auch wenn Attribute in unterschiedlichen Klassen gleich heißen, haben sie nichts mit einander zu tun.

Sie können z.B. ganz unterschiedliche Datentypen haben. Die Namen der Komponenten sind nur lokal innerhalb einer Klasse relevant.



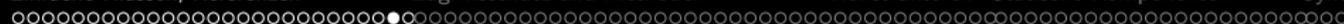
Quelldateien: Beispiel (1)

- Es ist möglich, mehrere Klassen-Deklarationen in eine Java-Quelldatei (.java) zu schreiben.
- Allerdings kann nur eine dieser Klassen als “public” deklariert sein, denn eine public-Klasse C muss in der Datei **C.java** definiert sein.

“public” für Klassen wird erst im Zusammenhang mit Paketen interessant, die in Kapitel 15 eingeführt werden. Im Moment ist es nicht nötig, Klassen als “public” zu deklarieren.

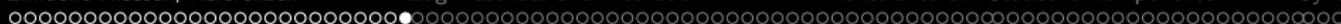
- Wenn man den Compiler für die Quelldatei aufruft, erzeugt er für jede Klasse C eine eigene Datei **C.class**.

Wenn mehrere Klassen in einer Quelldatei deklariert sind, wird diese Datei in mehrere class-Dateien übersetzt.



Quelldateien: Beispiel (2)

```
(1) class Datum {
(2)     int tag;    // 1..31
(3)     int monat; // 1..12
(4)     int jahr;  // vierstellig
(5) }
(6)
(7) class DatumTest {
(8)     public static void main(String[] args)
(9)         Datum d = new Datum();
(10)        d.tag = 24;
(11)        d.monat = 12;
(12)        d.jahr = 2014;
(13)        System.out.println(d.jahr);
(14)    }
(15) }
```



Quelldateien: Beispiel (3)

- Man kann das Hauptprogramm `main` zum Test auch gleich in die zu testende Klasse mit hineinschreiben.

Ich halte das stilistisch für etwas fragwürdig, das ist aber eine Geschmackssache (manche Programmierer schreiben in jede Klasse eine `main`-Methode).

Der Zugriffsschutz (s.u.) kann so allerdings nicht ausprobiert werden.

- Man kann das Programm auch auch mehrere Quelldateien verteilen:

- `Datum.java` für die Deklaration der Klasse `Datum`.
- `DatumTest.java` für das Hauptprogramm zum Test.

- Man kann den Compiler einzeln für beide Klassen aufrufen.

Wenn man ihn für `DatumTest.java` aufruft, und es gibt `Datum.class` noch nicht, aber `Datum.java`, übersetzt er die Datei automatisch mit.

Inhalt

- 1 Einfache Klassen, Referenzen
 - Strukturen, Attribute, Referenzen, Objekt-Erzeugung
- 2 Zugriffsschutz und Methoden
 - Von Strukturen zu Klassen, Methoden, Zugriffsschutz
 - Beispiel: Klasse für Datumswerte
- 3 Konstruktoren
 - Konstruktoren und Initialisierung
- 4 Statische Komponenten
 - Statische Variablen und Methoden
 - Konstanten
 - Beispiel: Verkettete Liste
- 5 Syntax
 - Syntax von Klassen-Deklarationen



Von Strukturen zu Klassen (1)

- Ein Datentyp legt nicht nur eine Wertemenge fest, sondern es gehören immer auch Operationen dazu.
- Alles, was man mit der oben gezeigten einfachen Klasse machen kann, ist, auf die Komponenten zuzugreifen.

Klassen mit öffentlichen Attributen und ohne Methoden entsprechen Strukturen/Records in nicht-objektorientierten Sprachen wie C oder Pascal.

- Zu **Datum** sollten weitere Funktionen gehören, z.B.:
 - Wochentag zu einem Datum bestimmen,
 - Datum des nächsten Tages bestimmen (oder Tage addieren),
 - zwei Datums-Objekte vergleichen.

D.h., feststellen, ob ein Datum kleiner/vorher, gleich, oder größer/später als ein anderes Datum ist.

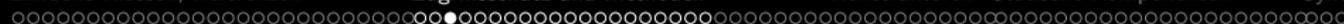


Von Strukturen zu Klassen (2)

- Früher hat man die Operationen/Funktionen/Methoden zu einem Strukturtyp getrennt deklariert.

Es gab natürlich ein Modulkonzept schon vor der objektorientierten Programmierung, bei C die getrennte Übersetzung verschiedener Quelldateien. Ohne besondere Tricks hatten aber auch die Anwender eines Struktur-Datentyps Zugriff auf die Komponenten. Sie konnten sich so neben den "mitgelieferten" leicht weitere Funktionen für Datums-Werte schreiben.

- Angenommen, es stellt sich später heraus, dass eine andere Repräsentation der Datumsangaben besser wäre, z.B. als Anzahl Tage seit einem festen Startdatum (1.1.1970).
- Dann ist es sehr schwer, das Programm entsprechend zu ändern: Die Zugriffe auf die Komponenten sind über das ganze Programm verteilt.



Von Strukturen zu Klassen (3)

- Mit Klassen ist es möglich, die Komponenten der Struktur selbst zu verstecken:
 - Man kann von außen nicht mehr direkt auf die Komponenten zugreifen,
 - sondern nur noch über die explizit definierten Zugriffsfunktionen (Methoden).
- Durch den indirekten Zugriff auf die Komponenten kann man die Implementierung des Datentyps (sein “Innenleben”) nachträglich ändern, aber die Schnittstelle (nach außen) stabil halten.

Die Trennung von Schnittstelle und Implementierung konnte man natürlich schon vorher, aber zum Teil war es Konvention, dass man auf die Struktur-Komponenten von außen nicht zugreift. Bei Klassen überwacht es der Compiler.

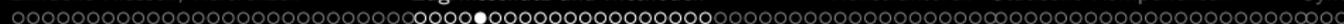


Von Strukturen zu Klassen (4)

- Natürlich wird man im Datums-Beispiel Methoden haben, die den Tag, den Monat, und das Jahr liefern.

Wenn das Datum intern mit diesen drei Komponenten realisiert ist, sind diese Methoden natürlich trivial. Wenn aber intern die Anzahl Tage seit einem Referenzdatum gespeichert sind, müssen diese Methoden schon etwas rechnen. Der Anwender der Klasse merkt davon nichts.

- Damit ist aber zunächst nur ein lesender Zugriff auf diese Komponenten möglich.
- Man wird sicher auch Methoden zum Setzen des Datums anbieten (oder zumindest zur Initialisierung).
- Diese Methoden sollten prüfen, dass die Datumsangabe korrekt ist, z.B. nicht 35.20.2014.



Von Strukturen zu Klassen (5)

- Man beachte den Unterschied:
 - Wenn mit Strukturen ein ungültiges Datum auftritt, kann der Fehler an beliebiger Stelle im Programm sein: Überall hat man Zugriff auf die Komponenten der Struktur.
 - Wenn mit der Klasse ein ungültiges Datum auftritt, dann hat eine der Methoden der Klasse es verursacht oder zumindest “durchschlüpfen lassen”.
 - Wenn die Methode mit ungültigen Eingabewerten aufgerufen wird, hätte sie das Programm normalerweise mit einer Fehlermeldung beenden sollen (mit einer Exception bzw. fehlgeschlagenen “Assertion”: später).
- Das Neue, was das Klassenkonzept bringt, ist also die Möglichkeit, Daten und Programmcode zu “verkapseln”, so dass man auf die Daten nur über die definierten Funktionen/Methoden zugreifen kann.

Syntax für Methoden-Aufruf

- Funktionen, die zu einem Struktur-Datentyp gehörten, haben einen Wert dieses Datentyps als normales Argument übergeben bekommen, z.B.

```
wochentag(d)
```

- Bei in einer Klasse definierten Methoden sieht der Funktionsaufruf dagegen ähnlich zu einem Komponentenzugriff aus:

```
d.wochentag()
```

- Dies ist aber nur eine andere Syntax: Jede Methode hat ein implizites "0-tes" Argument für das Objekt der Klasse, auf das die Methode angewendet wird.
- Das ist der wesentliche Unterschied zu statischen Methoden, die ohne Objekt aufgerufen werden.

this, Komponenten-Zugriff

- In einer normalen (nicht-statischen) Methode kann man auf das "aktuelle Objekt", für das diese Methode aufgerufen wurde, mit dem Schlüsselwort "`this`" zugreifen.

"`this`" ist ein Wertausdruck und liefert ein Objekt der aktuellen Klasse.

- Man kann aber auch einfach die Namen der Attribute (Instanzvariablen) ohne Objekt schreiben, und bekommt dann automatisch den Wert aus dem aktuellen Objekt.
- Z.B. kann man in einer Methode der Klasse `Datum` einfach "`tag`" schreiben, das ist äquivalent zu "`this.tag`".
- Ausnahme: Wenn man einen Parameter oder lokale Variable "`tag`" deklariert hat, muss man "`this.tag`" schreiben, um das Attribut anzusprechen.

Einfach "`tag`" würde sich auf die lokale Variable beziehen (Verschattung).

Beispiel: Methodendeklaration (1)

- Z.B. könnte man die Klasse `Datum` um eine Methode erweitern, die den Monatsnamen berechnet:

```
class Datum {
    int tag;
    int monat;
    int jahr;

    String MonatsName() {
        switch(monat) { // oder this.monat
            case 1: return "Januar";
            case 2: return "Februar";
            ...
            case 12: return "Dezember";
            default: return "Ungueftig!";
        }
    }
}
```

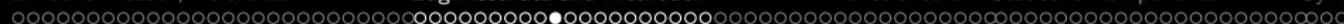
Beispiel: Methodendeklaration (2)

- Obwohl es so aussieht, als hätte `monatsName()` keine Parameter, hat die Methode doch implizit Zugriff auf das "aktuelle Objekt".

Auf die Attribute dieses Objektes kann die Methode einfach mit dem Attributnamen zugreifen, wahlweise auch mit `this.Attribut`.

- Wenn man die Methode aus einer anderen Klasse aufruft, muss man so ein Objekt im Aufruf angeben:

```
class DatumTest1 {  
    public static void main(String[] args) {  
        Datum d = new Datum();  
        d.tag = 24; d.monat = 12; d.jahr = 2014;  
        System.out.println(d.monatsName());  
    }  
}
```



Komponenten einer Klasse

- Komponenten (engl. “member”) einer Klasse sind:
 - Variablen
 - Auch Attribute, Felder, engl. “fields”, “data members” genannt.
 - Methoden
 - geschachtelte Klassen (später behandelt)
 - geschachtelte Interface-Deklarationen (später behandelt)
- Außerdem enthalten Klassen noch (formal keine “member”):
 - Konstruktoren (s.u.) und
 - Initialisierungs-Blöcke (s.u.).

Ob “member” oder nicht wird erst später im Zusammenhang mit Unterklassen (Kapitel 12) wichtig: Nur “member” werden an Unterklassen vererbt.

Zugriffsschutz/Sichtbarkeit (1)

- Für Attribute, Methoden u.s.w. (Komponenten und Konstruktoren einer Klasse) kann man Zugriffe von außerhalb der Klasse erlauben oder verbieten mit den folgenden Schlüsselworten:
 - **private**: Die Komponente/der Konstruktor ist nur innerhalb der Klasse zugreifbar/sichtbar.

D.h. nur in Programmcode, der in der Klasse steht, also Rümpfe von Methoden der Klasse und Initialisierungs-Blöcke.
 - **public**: Keine Einschränkungen (beliebig zugreifbar).
- Falls man nichts hinschreibt, bekommt man einen Zugriffsschutz ähnlich wie “public”.

D.h. fast keinen Schutz. Den Unterschied kann man erst später verstehen, wenn Pakete (engl. “Packages”: Zusammenfassungen von Klassen) besprochen wurden, siehe Kapitel 15.

Zugriffsschutz/Sichtbarkeit (2)

Zur Syntax:

- Die “Access Modifier” `private` und `public` müssen in jeder Komponenten-Deklaration einzeln angegeben werden.

Sonst bekommt man den Default “package” (ähnlich “public”, siehe Kap. 15).

- Die Angabe muss am Anfang der Deklaration erfolgen (vor dem Typ bzw. dem Konstruktornamen).
- Innerhalb der “Modifier”, zu denen z.B. auch “`static`” und “`final`” zählen, ist die Reihenfolge egal.

Es ist aber üblich, den Zugriffsschutz nach Annotationen (s.u.) und vor allen anderen Modifiern zu schreiben. Z.B. ist “`public static void main(...)`” die normale Schreibweise, aber “`static public void main(...)`” geht auch.

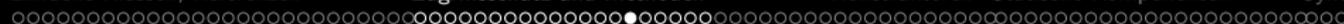
Zugriffsschutz/Sichtbarkeit (3)

```
(1) class Test {
(2)     private int a;
(3)     int b;
(4)     public int c;
(5)     public int m() { return a; } // Ok
(6) }
(7)
(8) class OtherClass {
(9)     public static void main(String[] args) {
(10)         Test x = new Test();
(11)         System.out.println(x.a); // Fehler
(12)         System.out.println(x.b); // Ok
(13)         System.out.println(x.c); // Ok
(14)         System.out.println(x.m()); // Ok
(15)     }
(16) }
```

Zugriffsschutz/Sichtbarkeit (4)

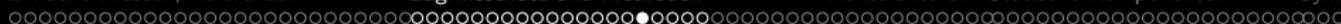
- Man bekommt hier die folgende Fehlermeldung:

```
OtherClass.java:11: a has private access in Test
    System.out.println(x.a); // Fehler
                        ^
```
- Alle anderen Zugriffe sind möglich.
Zwar waren nur lesende Zugriffe gezeigt, aber man kann `x.b` und `x.c` von der anderen Klasse aus auch ändern (mit einer Zuweisung).
- Normalerweise sollten Attribute von außerhalb der Klasse nicht zugreifbar sein, also als “`private`” deklariert werden.
- Eigentlich stellen nur schreibende Zugriffe ein Problem dar. Ist das Attribut als “`final`” deklariert (kann also nach der Initialisierung nicht mehr geändert werden), so kann man Zugriffe von außerhalb der Klasse erlauben.



Zugriffsschutz/Sichtbarkeit (5)

- Allgemein sollten nur die Attribute und Methoden von außen zugreifbar sein, für die das tatsächlich nötig ist.
- Wenn man sich z.B. eine Hilfs-Methode schreibt, um eine andere Methode zu vereinfachen, so sollte diese Hilfs-Methode als “`private`” deklariert werden.
- Je mehr von außen zugreifbar ist, desto schwieriger werden spätere Änderungen der Klasse.
 - Spätestens, wenn die Klasse über das Web verbreitet wurde, hat man keine Kontrolle mehr darüber, welche “`public`” Attribute und Methoden wirklich verwendet wurden — man muss sie für immer unterstützen.
- Man schließt auch mögliche Fehlerquellen aus, indem man Zugriffe auf Klassen-Interna explizit verbietet.



Zugriffsschutz/Sichtbarkeit (6)

Feinheit zu “private”:

- Auf “**private**” deklarierte Komponenten kann man in jeder Methode der Klasse zugreifen.

Natürlich auch in Konstruktoren und Initialisierungs-Blöcken. Allgemein aller Programmcode, der im Rumpf der `class`-Deklaration steht. Aus Subklassen kann man auf diese Komponenten z.B. nicht zugreifen (siehe Kapitel 12).

- Man kann aber auch auf “**private**”-Komponenten anderer Objekte der gleichen Klasse zugreifen, nicht nur auf Komponenten des Objektes, für das die Methode bzw. der Konstruktor aufgerufen wurde.

In dieser Hinsicht verhält sich Java wie C++ und anders als z.B. Smalltalk: Dort kann man nur auf die `private`-Komponenten des aktuellen Objektes `this` zugreifen. Die Kapselung ist dort also wirklich auf Objekt-Ebene, während sie in Java und C++ eher auf Klassenebene ist.



Übliche Methoden-Namen (1)

- Die Zugriffsfunktionen für das Attribut `“attr”` heißen normalerweise:
 - `“getAttr”`: Lesefunktion, liefert aktuellen Attributwert,
 - `“setAttr”`: Schreibfunktion, ändert den Attributwert.

Diese Funktion hat einen Parameter, nämlich den neuen Wert. Dieser wird (nach eventuellen Tests) der Attribut-Variablen zugewiesen. Möglicherweise müssen auch noch abgeleitete Daten oder Zugriffsstrukturen aktualisiert werden (d.h. dem veränderten Attributwert angepasst). Nicht alle Attribute müssen nach der Konstruktion des Objektes noch verändert werden. Falls das nicht nötig ist, deklariert man keine Funktion `“setAttr”`.
- Im Beispiel liefert das einen deutsch-englischen Mischmasch. Empfohlene Lösung: Ganz auf Englisch wechseln.



Übliche Methoden-Namen (2)

- Da Methoden Aktionen beschreiben, sind Verben oder Verphrasen besonders typisch für Methodennamen.

Das ist aber keine Vorschrift, z.B. ist `length()` eine Methode der Klasse `String`.

- Methoden, die eine Bedingung **B** testen, also einen booleschen Wert (Wahrheitswert) liefern, heißen oft `isB`.

Z.B. `isEmpty()` für `String`-Objekte.

- Methoden, die ein Objekt in einen anderen Datentyp **D** konvertieren, heißen üblicherweise `toD`.

Z.B. ist `toString()` eine nützliche Methode für alle Objekte. Wenn wir `toString()` für die Klasse `Datum` definieren (s.u.), erhalten wir eine sinnvolle Ausgabe, wenn wir ein `Datum`-Objekt mit `println` drucken.

Übliche Methoden-Namen (3)

- Es ist auch möglich, die Lesefunktion für das Attribut `attr` genauso zu nennen, wie das Attribut: `attr()`.
- Zum Beispiel könnte man die Tag-Komponente eines Datums-Objektes `d` mit `d.tag()` statt mit `d.getTag()` abfragen.
- Der gleiche Name wird hier für Variable (Attribute) und Methode verwendet: Java kann beides durch den Kontext unterscheiden, nach einer Methode kommt immer eine "(".
In C++ wäre das nicht möglich, dort kann man nicht den gleichen Namen für zwei verschiedene Komponenten der Klasse verwenden. In C++ können Funktionen auch als Werte verwendet werden, daher folgt nicht immer die "(".
- Im Folgenden soll diese Namensgebung verwendet werden (weniger Sprachwirrwarr, kein Unterschied von Attributzugriff zu berechnetem Wert, aber Nicht-Standard).



Einhaltung von Spezifikationen

- Wenn Sie die Aufgabe bekommen, eine Klasse mit einer bestimmten Schnittstelle zu entwickeln, müssen von außen genau die in der Aufgabe genannten Attribute und Methoden zugreifbar sein (alles, was nicht `private` ist, ist gegeben).

Von außerhalb der Klasse zugreifbare Attribute sind eher selten (s.o.).

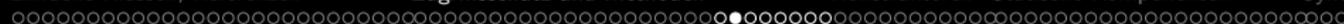
- Es steht Ihnen aber frei, weitere Hilfs-Methoden zu definieren, die `private` sind.
- Entsprechend sagt die Aufgabenstellung meist über die Attribute überhaupt nichts aus — Sie müssen selbst herausfinden, welche Daten Sie sich zwischen den Methodenaufrufen im Objekt merken müssen.

Daten, die Sie nur während eines Methodenaufrufs brauchen, müssen Sie als lokale Variable deklarieren (sonst eindeutig schlechter Stil).



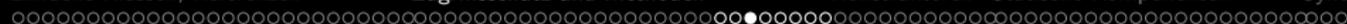
Beispiel (1)

```
(1) class Datum {
(2)
(3)     // Attribute:
(4)     private int tag;    // 1..31
(5)     private int monat; // 1..12
(6)     private int jahr;  // 1600 ..
(7)
(8)     // Lesezugriff auf Attribute:
(9)     // ("accessor" oder "getter"-Methoden,
(10)    //  üblich wären Namen wie "getTag()")
(11)    public int tag()    { return this.tag; }
(12)    public int monat() { return this.monat; }
(13)    public int jahr()  { return this.jahr; }
(14)
```



Beispiel (2)

```
(15) // Hilfsfunktion: Test auf Schaltjahr:  
(16) private boolean schaltjahr(int jahr) {  
(17)     if(jahr % 400 == 0)  
(18)         return true;  
(19)     if(jahr % 100 == 0)  
(20)         return false;  
(21)     if(jahr % 4 == 0)  
(22)         return true;  
(23)     return false;  
(24) }  
(25)
```



Beispiel (3)

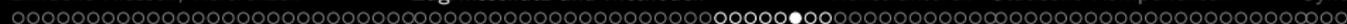
```
(26) // Hilfsfunktion: Tage im Monat
(27) private int mTage(int monat, int jahr) {
(28)     if(monat == 2) {
(29)         if(schaltjahr(jahr))
(30)             return 29;
(31)         else
(32)             return 28;
(33)     }
(34)     if(monat == 4 || monat == 6 ||
(35)         monat == 9 || monat == 11) {
(36)         return 30;
(37)     }
(38)     return 31;
(39) }
(40)
```

Beispiel (4)

```
(41) // Setzen der Attribute
(42) // (besser mit Konstruktor, s.u.)
(43) // Liefert false wenn Datum ungueltig.
(44) public boolean init(int tag, int monat,
(45)                     int jahr) {
(46)     if(jahr < 1600)
(47)         return false;
(48)     if(monat < 1 || monat > 12)
(49)         return false;
(50)     if(tag < 1 || tag > mTage(monat,jahr))
(51)         return false;
(52)     this.tag = tag;
(53)     this.monat = monat;
(54)     this.jahr = jahr;
(55)     return true;
(56) }
(57)
```

Beispiel (5)

```
(58) // Berechnung des Wochentags
(59) // (1=Montag, 2=Dienstag, ...)
(60) public int wochentag() {
(61)     int t = 1;
(62)     int m = 1;
(63)     int j = 1600;
(64)     int w = 6; // 1.1.1600 war Samstag
(65)     while(j<jahr || m<monat || t<tag) {
(66)         w++;
(67)         if(w == 8) { w = 1; }
(68)         t++;
(69)         if(t > mTage(m, j)) { t = 1; m++; }
(70)         if(m > 12) { m = 1; j++; }
(71)     }
(72)     return w;
(73) }
```



Beispiel (6)

```
(74)
(75)     // Name des Wochentags:
(76)     public String tagname() {
(77)         switch(wochentag()) {
(78)             case 1: return "Montag";
(79)             case 2: return "Dienstag";
(80)             case 3: return "Mittwoch";
(81)             case 4: return "Donnerstag";
(82)             case 5: return "Freitag";
(83)             case 6: return "Samstag";
(84)             case 7: return "Sonntag";
(85)         }
(86)         return null;
(87)     }
(88)
(89) } // Ende der Klasse Datum
```

Beispiel (7)

Anmerkungen:

- Die Gregorianische Kalenderreform war 1582, in diesem Jahr folgte der 15. Oktober direkt auf den 4. Oktober, um den bisher entstandenen Fehler zu korrigieren.

Um dieses Problem nicht behandeln zu müssen, sind in obiger Klasse Datumswerte vor 1600 ausgeschlossen. Allerdings wurde zunächst in den Schaltjahren der 24. Februar verdoppelt, und nicht der 29. Februar verwendet. Das ist in obige Klasse anders. [<http://de.wikipedia.org/wiki/Schaltjahr>]

- Es gibt schon eine Klasse "Date" im Paket "java.util", sowie eine Klasse "Calendar" im gleichen Paket.

Viel mächtiger als unsere Klasse. Datumswerte sind aber ein gutes Beispiel.

- Die Berechnung des Wochentags kann man beschleunigen, indem man zunächst ganze Jahre springt (zur Übung).



Methode toString

- Man sollte eine Methode `toString()` definieren, die eine Objektbeschreibung als Zeichenkette liefert:

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

- Wenn man ein Objekt druckt, wird diese Methode verwendet:

```
System.out.println(d);
```

Wenn man die Methode `toString()` nicht definiert, gibt es schon eine vordefinierte ("ererbte") Methode, die würde z.B. "Datum@7919298d" drucken, was nicht besonders hilfreich ist. Weil es diese Methode schon gibt, sollte man `@Override` angeben, und weil sie `public` ist, muss auch die hier neudefinierte Version `public` sein. Ausführlicher in Kapitel 12.

Inhalt

- 1 Einfache Klassen, Referenzen
 - Strukturen, Attribute, Referenzen, Objekt-Erzeugung
- 2 Zugriffsschutz und Methoden
 - Von Strukturen zu Klassen, Methoden, Zugriffsschutz
 - Beispiel: Klasse für Datumswerte
- 3 **Konstruktoren**
 - **Konstruktoren und Initialisierung**
- 4 Statische Komponenten
 - Statische Variablen und Methoden
 - Konstanten
 - Beispiel: Verkettete Liste
- 5 Syntax
 - Syntax von Klassen-Deklarationen

Konstruktoren (1)

- So wie die Klasse `Datum` bisher deklariert ist, gibt es keine Garantie, dass die Methode `init` auch aufgerufen wird.

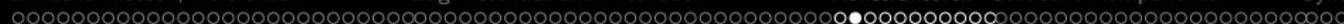
Dieses Problem wird noch dadurch verschärft, dass die Standard-Initialisierung von Java die drei Attribute `tag`, `monat`, `jahr` auf 0 setzt, und damit ungültige Werte einträgt.

- Java bietet (wie z.B. C++) die Möglichkeit, spezielle Methoden, sogenannte Konstruktoren, zu definieren.

Formal zählen Konstruktoren nicht zu den Methoden, sind aber recht ähnlich. Die Lösung mit `init` wurde oben nur aus pädagogischen Gründen gewählt.

- Es ist dann garantiert, dass bei der Objekt-Erzeugung ein Konstruktor aufgerufen wird.

Bei lokalen Variablen ist ja auch empfohlen, sie gleich bei der Deklaration zu initialisieren. Die oben deklarierte Methode `init` kann aber nicht gleich bei der Objekt-Erzeugung mit aufgerufen werden.



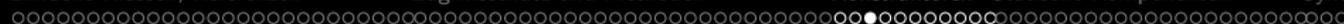
Konstruktoren (2)

- Konstruktoren sind dadurch gekennzeichnet, dass
 - sie den gleichen Namen wie die Klasse haben, und
 - kein Rückgabe-Typ angegeben wird.

Konstruktoren werden bei der Ausführung von `new` aufgerufen.

Sie sorgen für die anwendungs-spezifische Initialisierung des von `new` erzeugten Objektes, das `new` dann auch zurückliefert.

```
class Datum {  
    public Datum(int tag, int monat, int jahr) {  
        if(!init(tag, monat, jahr)) {  
            System.err.println("Datum ungültig!");  
            System.exit(1);  
        }  
    }  
}  
...  
...
```



Konstruktoren (3)

- Im Beispiel ist stilistisch etwas fragwürdig, dass im Konstruktor eine andere Methode der Klasse aufgerufen wird.
Diese wird dann ja für ein nicht vollständig initialisiertes Objekt aufgerufen.
Zwar stellt Java sicher, dass alle Attribute auf die Standardwerte initialisiert sind. Aber bei allen späteren Methoden-Aufrufen kann man sicher sein, ein gültiges Datum zu haben, während das hier nicht gilt.
- Wenn man das Setzen der Attribute nur im Konstruktor erlauben will, kann man die Methode `init()` auch als `private` deklarieren.
Wenn man keine anderen Methoden zum Ändern anbietet, könnte ein einmal erzeugtes `Datum`-Objekt nicht mehr verändert werden.
- **Hinweis:** Java erlaubt Methoden, die so heißen wie die Klasse. Ein Konstruktor ist es nur, wenn kein Rückgabe-Typ angegeben ist.

Konstruktoren (4)

- Konstruktoren können Parameter haben (wie im Beispiel).
- Dann müssen bei der Objekt-Erzeugung mit `new` entsprechende Werte angegeben werden:

```
Datum d = new Datum(24, 12, 2014);
```

- Wenn man das nicht tut, bekommt man folgende Fehlermeldung:

```
sb/Datum.java:103: cannot find symbol
symbol : constructor Datum()
location: class sb.Datum
    Datum d = new Datum();
                  ^
```

- Somit ist sichergestellt, dass `Datum`-Objekte immer mit sinnvollen Werten initialisiert werden.



Konstruktoren (5)

- Es ist möglich, mehrere Konstruktoren zu definieren, die sich in Anzahl oder Typen der Parameter unterscheiden.

Der Compiler kann dann anhand der Liste von Werten beim Aufruf von `new` entscheiden, welcher der Konstruktoren ausgeführt wird. Dies gilt entsprechend auch für Methoden (“überladene Methoden”: später).
- Wenn man will, könnte man z.B. einen zweiten Konstruktor ohne Parameter definieren.

Dieser könnte das `Datum`-Objekt auf das aktuelle Datum setzen. So ist es bei `java.util.GregorianCalendar` gelöst. Dort geschieht die Abfrage eines Attributes in der Form `“g.get(Calendar.DAY_OF_MONTH)”`.
Objekte dieses Typs enthalten auch Uhrzeiten und Zeitzonen-Information.
- Man kann aus einem Konstruktor heraus einen anderen aufrufen, z.B.: `“this(24,12,2014);”`.

Dies muss aber das erste Statement im Konstruktor sein.



Konstruktoren (6)

- Wenn man keinen Konstruktor deklariert, legt der Compiler automatisch einen “Default-Konstruktor” an, der nichts anderes tut, als den Konstruktor der Oberklasse aufzurufen.

Das ist insofern wichtig, weil es bei diesem Aufruf Schwierigkeiten geben kann (z.B. könnte er `private` sein oder Parameter haben). Siehe Kap. 12.

- Dies geschieht aber nur, wenn man überhaupt keinen Konstruktor deklariert hat.

Sobald man einen Konstruktor selbst deklariert hat, auch mit Parametern, wird nicht mehr automatisch ein Konstruktor ohne Parameter angelegt.

- Es ist möglich, Konstruktoren z.B. als “`private`” zu deklarieren, und so das direkte Anlegen von Objekten außerhalb der Klasse zu verhindern.

Es könnte statische Methoden geben, die ihrerseits Objekte anlegen (wenn man z.B. zu jedem Datum nur ein Objekt haben will).



Konstruktoren: Bitte beachten!

- Konstruktoren müssen genau so heißen, wie die Klasse.
Auch nicht in anderer Groß-/Kleinschreibung. Schon gar nicht "Konstruktor".
- Bei Konstruktoren darf man keinen Rückgabe-/Ergebnistyp hinschreiben, auch nicht `void`.
Sonst ist es eine Methode, die zufällig so heißt wie die Klasse.
- Konstruktoren können nicht `static` sein.
Sie müssen natürlich auf das zu initialisierende Objekt zugreifen.
- Lokale Variablen im Konstruktor und Attribute (Instanzvariablen) müssen klar unterschieden werden.
Lokale Variablen im Konstruktor werden gelöscht, wenn seine Ausführung endet.
- Man kann Konstruktoren nicht wie eine Methode aufrufen (nur implizit über `new`).

Weitere Initialisierungen (1)

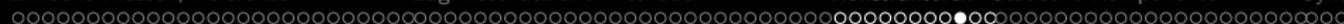
- Es ist auch möglich, Attribute direkt bei der Deklaration zu initialisieren, z.B.

```
public class Datum {  
    private int tag = 1;  
    private int monat = 1;  
    private int jahr = 1970;  
    ...  
}
```

Dies ist ja bei lokalen Variablen auch möglich.

- Außerdem kann man Blöcke mit Initialisierungscode (“instance initializer”) in die Klasse schreiben:

```
public class Datum {  
    private int tag, monat, jahr;  
    { tag = 1; monat = 1; jahr = 1970; }  
}
```



Weitere Initialisierungen (2)

- Wenn es außerdem einen Konstruktor gibt, erfolgt erst diese Initialisierung, und anschließend wird der Konstruktor ausgeführt.

Ganz am Anfang geschieht die Standard-Initialisierung aller Attribute (auf 0 bzw. `null`). Wenn der Compiler erkennen kann, dass die Attribute vorher nicht abgefragt werden, braucht er die entsprechenden Befehle natürlich nicht zu erzeugen. Dann wird ggf. der Konstruktor der Oberklasse aufgerufen. Anschließend werden die obigen Initialisierungen und Initialisierungs-Blöcke (s.u.) ausgeführt (in der Reihenfolge, in der sie in der Klasse angegeben sind). Dann der Konstruktor.

Das Programm auf der nächsten Seite gibt Folgendes aus, wenn ein Objekt vom Typ `InitTest` erzeugt wird: `A, B0, C, E, F5, D5`. Das Attribut `b` hat am Ende den Wert `7`. Natürlich sollte man so unübersichtliche Programme nicht schreiben, das ist sehr schlechter Stil. Grundsätzlich ist es aber gut, zu wissen, dass präzise definiert ist, was passiert (und man es mit solchen Testprogrammen ausprobieren kann).



Weitere Initialisierungen (3)

```
(1) class InitTest {
(2)     { System.out.println("A"); }
(3)     int a = m("B");
(4)     int b = 5;
(5)     int m(String s) {
(6)         System.out.println(s + b);
(7)         return 1;
(8)     }
(9)     { System.out.println("C"); }
(10)    InitTest() {
(11)        System.out.println("D" + b);
(12)        b = 7;
(13)    }
(14)    { System.out.println("E"); }
(15)    int c = m("F");
(16) } // Ausgabe bei new(): A, B0, C, E, F5, D5.
```



Destruktoren

- Man kann auch eine Methode

```
protected void finalize() { ... }
```

definieren.

- Diese Methode wird vom Garbage Collector aufgerufen bevor er das Objekt löscht (d.h. den vom Objekt belegten Hauptspeicher dem Recycling zuführt).
- Die Methode wird daher auch als Destruktor bezeichnet.
- Allerdings ist bei Java nicht vorhersehbar, wann ein Objekt vom Garbage Collector eingesammelt wird, oder ob der Garbage Collector überhaupt aktiv wird.

Es ist aber möglich, ihn explizit aufzurufen. Ansonsten sind Destruktoren in Java kaum nützlich. In C++, wo die Speicherverwaltung manuell geschieht, sind Destruktoren wichtig.

Inhalt

- 1 Einfache Klassen, Referenzen
 - Strukturen, Attribute, Referenzen, Objekt-Erzeugung
- 2 Zugriffsschutz und Methoden
 - Von Strukturen zu Klassen, Methoden, Zugriffsschutz
 - Beispiel: Klasse für Datumswerte
- 3 Konstruktoren
 - Konstruktoren und Initialisierung
- 4 **Statische Komponenten**
 - Statische Variablen und Methoden
 - Konstanten
 - Beispiel: Verkettete Liste
- 5 Syntax
 - Syntax von Klassen-Deklarationen

Statische Variablen: Grundlagen (1)

- Die bisher in der Klasse `Datum` deklarierten Variablen `tag`, `monat`, `jahr` existieren für jedes Objekt einzeln.

Bei fünf Objekten der Klasse `Datum` gibt es also entsprechend fünf `int`-Variablen `tag`, `monat`, `jahr` — jeweils eine pro Objekt. Deswegen heißen sie auch Instanzvariablen (Objekte entstehen durch Instanziierung / sind Instanzen einer Klasse).

- Die im letzten Kapitel schon eingeführten statischen Variablen (gekennzeichnet durch das Schlüsselwort `static`) existieren dagegen nur ein einziges Mal für die ganze Klasse.
- Sie heißen daher auch Klassenvariablen/Klassen-Attribute.
- Statische Variablen existiert auch dann, wenn noch kein einziges Objekt der Klasse angelegt wurde.

Deswegen können sie bei klassischer Programmierung ohne Objekte (nur mit statischen Methoden und Variablen) wie globale Variablen verwendet werden.

Statische Variablen: Grundlagen (2)

- Da die Variable nur einmal für die ganze Klasse existiert, ändert sich ihr Wert natürlich für alle Objekte, wenn in einem Objekt ein neuer Wert zugewiesen wird.

Der Zugriff in einer normalen Methode der Klasse sieht aus wie ein Zugriff auf eine Instanzvariable (normales Attribut), insofern ist es erwähnenswert, dass das Verhalten anders ist.

- Syntaktisch ist “**static**” ein Modifier wie “**public**” u.s.w. und muss in der Deklaration vor dem Typ angegeben werden:

```
private static int anzObjekte;
```

Die Reihenfolge innerhalb der Modifier ist egal. Es ist aber üblich, den Zugriffsschutz zuerst anzugeben (s.o.).

- Beispiel-Anwendung: Aktuelle Objekt-Anzahl der Klasse. Die statische Variable wird im Konstruktor inkrementiert (+1). Im Destruktor dekrementiert (-1). Zählt nicht mehr zugreifbare Objekte mit.



Statische Variablen: Anwendungen

- Konstanten sind statische und unveränderliche Variablen.
 - Mit konstanter Initialisierung. Konstanten werden unten genauer behandelt.
- Wählbare Parameter/Optionen für den Programmcode der Klasse.
 - Z.B. Größe eines Arrays (bestimmt maximal mögliche Anzahl Einträge).
- Wenn jedes Objekt der Klasse mit einer eindeutigen Nummer versehen werden soll, merkt man sich die jeweils nächste Nummer in einer statischen Variablen.
- Es ist manchmal nötig, sich die Menge aller erzeugten Objekte der Klasse zu speichern. Die dafür nötigen Variablen sind auch der Klasse zugeordnet.

Statische Variablen: Zugriff

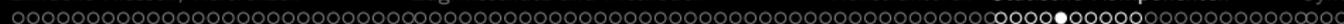
- Aus Programmcode der Klasse kann man auf statische Variablen einfach mit dem Namen der Variablen zugreifen.

Vorausgesetzt natürlich, sie wird nicht durch eine lokale Variable oder einen Parameter mit gleichem Namen verschattet.

- Von außerhalb stellt man den Klassennamen voran, also z.B. `Datum.anzObjekte`.

Ein Ausdruck mit dieser Klasse als Ergebnistyp würde auch gehen, ist aber verwirrend, weil es dann wie eine Instanzvariable aussieht.

(Bei C++ ist die Syntax `Datum::anzObjekte`, also anders als bei einem normalen Attributzugriff. Das Symbol `::` dient in C++ dazu, einen Bezeichner im Namensraum einer Klasse anzusprechen, wenn man ihn außerhalb der Klasse benutzt. Beim normalen Attributzugriff oder Methodenaufruf ist das natürlich nicht nötig, dort legt das Objekt ja schon die Klasse fest.)



Statische Variablen: Initialisierung

- Selbstverständlich sind auch statische Variablen automatisch auf `0/null` initialisiert.
- Man kann aber auch in der Deklaration einen Wert explizit angeben:

```
private static int anzObjekte = 0;
```

Es ist eine Stilfrage, ob man auch den voreingestellten Wert `0` explizit angibt. Ich würde es tun, da es klar macht, dass man von diesem Wert ausgeht.

- Kompliziertere Initialisierungen kann man in einem statischen Initialisierungsblock machen ("static initializer"):

```
static {  
    anzObjekte = 0;  
}
```



Statische Methoden (1)

- In Kapitel 8 wurden statische Methoden bereits eingeführt.
Entsprechend klassischer, nicht-objektorientierter Programmierung. Neu ist jetzt, dass eine Klasse beides, statische und nicht-statische Methoden haben kann.

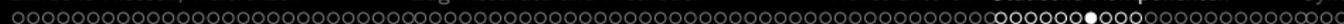
- Statische Methoden werden durch "static" gekennzeichnet:

```
public static int anzahl() {  
    return anzObjekte;  
}
```

- Beim Aufruf gibt man kein Objekt der Klasse an, sondern den Klassen-Namen:

```
System.out.println(Datum.anzahl());
```

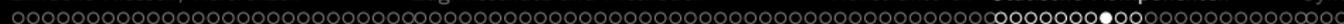
Verschiedene Klassen können Methoden gleichen Namens enthalten, diese Mehrdeutigkeit wird durch den Klassennamen aufgelöst. Der Compiler braucht den Klassennamen, um die Implementierung der Methode zu finden.



Statische Methoden (2)

- Innerhalb der Klasse kann man natürlich alle statischen Komponenten dieser Klasse ohne den Klassennamen ansprechen, also z.B. einfach "`anzahl()`".
- Weil beim Aufruf kein Objekt angegeben wurde, gibt es in statischen Methoden kein aktuelles Objekt "`this`".
- Daher kann man nicht-statische Komponenten (Instanzvariablen und normale Methoden) nicht einfach durch ihren Namen ansprechen.
- Wenn man aber ein Objekt `x` der Klasse hat (z.B. als Parameter der Methode oder aus einer statischen Variable), kann man in der Form `x.A` auf eine Instanzvariable zugreifen.

Dann ist es auch kein Problem, wenn `A` `private` ist. Nur `A` alleine geht nicht, weil Abkürzung für `this.A`, und es in statischen Methoden kein `this` gibt.



Statische Methoden (3)

- Der Aufruf einer statischen Methode aus einer normalen (nicht-statischen) Methode ist kein Problem.

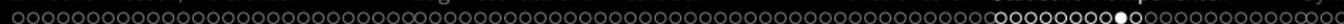
Der implizite Parameter `this` für das aktuelle Objekt wird dann einfach nicht weitergegeben. Umgekehrt muss man mit "`x.m(...)`" wieder explizit ein Objekt angeben, wenn man aus einer statischen Methode eine normale Methode aufrufen will.

- Statische Methoden kann man natürlich auch aufrufen, wenn noch kein einziges Objekt der Klasse erzeugt wurde.

Oder die Klasse sogar so definiert ist, dass man keine Objekte erzeugen kann.

- Statische Methoden entsprechen klassischen Funktionen in nicht-objektorientierten Sprachen, und Klassen nur mit statischen Komponenten entsprechen klassischen Modulen.

Z.B. hat die vordefinierte Klasse `Math` nur statische Methoden (wie `sin`, `cos`, `log`, `sqrt`) und Konstanten (`PI`, `E`). Man kann keine `Math`-Objekte erzeugen.

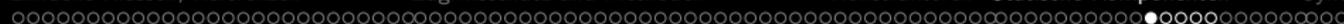


Statische Methoden (4)

- Wenn eine Methode nicht auf Instanzvariablen zugreift, und keine nicht-statischen Methoden aufruft, dann sollte man sie auch als **“static”** deklarieren:
 - Man kann diese Methode dann auch aufrufen, wenn man kein Objekt der Klasse hat (z.B. in statischen Methoden).
 - Man dokumentiert klar, dass ein Objektzustand weder abgefragt noch verändert wird.
 - Der Aufruf ist möglicherweise effizienter.
 - Es muss kein **“this”**-Objekt übergeben werden.
- Für den Aufrufer ändert sich (meist) nichts, wenn man eine Methode nachträglich als **“static”** deklariert.
 - Er darf vor dem **“.”** auch ein Objekt der Klasse angeben, er muss nicht unbedingt die Klasse schreiben (Wird die Methode in einer Subklasse überschreiben, entscheidet bei statischen Methoden der Compilezeit-Typ).

Übersicht

Komponente der Klasse C	In normaler Methode der Klasse C	In statischer Methode der Klasse C
Zugriff auf Attribut a (Instanzvariable)	<code>a</code> oder <code>this.a</code> oder <code>x.a</code> (x vom Typ C)	— — <code>x.a</code> (x vom Typ C)
Zugriff auf Klassenattribut a (statische Var.)	<code>a</code> oder <code>C.a</code> (oder <code>x.a</code> mit x vom Typ C)	<code>a</code> oder <code>C.a</code> (oder <code>x.a</code> mit x vom Typ C)
Aufruf von Methode m (normal/nicht statisch)	<code>m(...)</code> oder <code>this.m(...)</code> oder <code>x.m(...)</code> (x vom Typ C)	— — <code>x.m(...)</code> (x vom Typ C)
Aufruf von statischer Methode m	<code>m(...)</code> oder <code>C.m(...)</code> (oder <code>x.m(...)</code> , x vom Typ C)	<code>m(...)</code> oder <code>C.m(...)</code> (oder <code>x.m(...)</code> , x vom Typ C)



Konstanten: Grundlagen (1)

- Wenn man eine Variable als “**final**” deklariert, kann man ihr nur ein Mal einen Wert zuweisen.

Das kann gleich in der Deklaration geschehen, in einem Initialisierungs-Block, oder bei nicht-statischen Variablen im Konstruktor. Der Compiler muss verifizieren können, dass der Variablen vorher noch kein Wert zugewiesen wurde.

- Konstanten sind statische “**final**” Variablen, die gleich in der Deklaration initialisiert werden, und von denen der Compiler den Wert schon berechnen kann:

```
public static final int MIN_JAHR = 1600;
```

Die Initialisierung der Variablen muss mit einer “constant expression” geschehen. Dies sind in erster Linie Datentyp-Literale primitiver Typen und String-Literale, aber man kann auch andere solche Konstanten verwenden, und z.B. mit den vier Grundrechenarten rechnen (auch möglich: u.a. String-Konkatenation, Typ-Cast in primitive Typen und String, bedingte Ausdrücke).



Konstanten: Grundlagen (2)

- Es ist üblich, dass Konstanten in Großbuchstaben geschrieben werden (mit “_” zwischen Worten).
- Die Werte von Konstanten werden vom Compiler direkt in die Klassen eingebaut, die die Konstanten verwenden.

Wenn man den Wert einer Konstanten nachträglich ändert, müssen daher auch die Klassen neu übersetzt werden, die die Konstante verwenden. Es reicht nicht, nur die Klasse neu zu compilieren, in der die Konstante definiert ist.
- Für statische “final” Variablen, die nicht in der Deklaration initialisiert werden (sondern per “static initializer”), gilt das nicht. Diese würde man nicht “Konstanten” nennen.

Hier erzeugt der Compiler einen Zugriff auf die unveränderbare Variable zur Laufzeit. Damit läuft das Programm etwas langsamer.
- Selbstverständlich können Konstanten auch “private” sein.

Konstanten: Anwendungen (1)

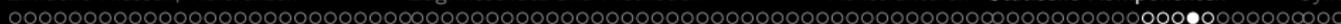
- Grenzen der Implementierung

Im Beispiel kann der Benutzer der Klasse das minimale Jahr abfragen.
Das kann sich eventuell in zukünftigen Versionen der Klasse ändern.

- Werte eines Typs mit einer kleinen Anzahl möglicher Werte, z.B. Wochentag:

```
public static final int MONTAG = 1;  
...  
public static final int SONNTAG = 7;
```

Es ist ja nicht eindeutig geregelt, ob Sonntag oder Montag der erste Tag der Woche ist. Daher sind Zahlencodes 1–7 möglicherweise missverständlich und symbolische Konstanten besser. Dies würde einem Aufzählungstyp in Sprachen wie Pascal, C und C++ entsprechen (Schlüsselwort `enum`).
In J2SE 5.0 wurden spezielle `enum`-Klassen eingeführt (siehe späteres Kapitel).



Konstanten: Anwendungen (2)

- Spezielle Argumentwerte für Methoden

Z.B. könnte eine Methode Noten (1 bis 5) entgegennehmen, und zusätzlich spezielle Werte für “nicht erschienen”, “bestanden”, die intern z.B. als negative Zahlen codiert sind. Der Anwender der Klasse sollte aber nicht mit diesen willkürlichen Zahlen arbeiten, sondern mit den symbolischen Konstanten.

- Texte für leichte Änderung der Sprache (z.B. Deutsch/Englisch).

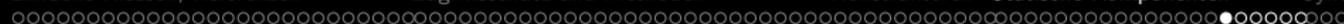
Man könnte z.B. eine Klasse anlegen, in der Konstanten für alle Texte definiert sind, die in dem Programm ausgegeben werden. Diese Klasse muss dann ausgetauscht werden, wenn das Programm an eine andere Sprache angepasst werden soll. Bei vielen verschiedenen Texten wird dies aber mühsam und unübersichtlich. Dann könnte es besser sein, die Texte in einer Standard-Sprache in das Programm zu schreiben, und sie vor der Ausgabe in eine “Übersetzungsmethode” zu schicken. Siehe auch “`java.util.ResourceBundle`”.



Konstanten: Guter Programmierstil

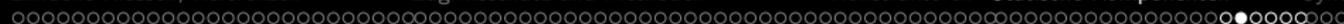
- Eine Empfehlung für guten Programmierstil besagt, dass Programme direkt nur die Werte 0 und 1 enthalten sollten, und alle anderen Zahlwerte nur über symbolische Konstanten.
- Wenn Sie einen Zahlwert direkt in Ihr Programm schreiben wollen, sollten Sie sich folgende Fragen stellen:
 - Wäre es möglich, dass ich diesen Wert später einmal ändern möchte, und tritt er an mehreren Stellen in meinem Programm auf, die dann alle geändert werden müssen?

Auch bei nur einer Verwendung könnte es von Vorteil sein, alle änderbaren Werte an einer Stelle in der Klasse zu versammeln.
 - Könnte die Bedeutung des Wertes durch einen symbolischen Namen klarer werden?



Beispiel (1)

- Es soll eine Klasse “Date” definiert werden, die garantiert, dass es zu einem Datum auch nur ein Objekt gibt.
- Dazu wird der Konstruktor als “private” deklariert, so dass er von außen nicht aufrufbar ist.
- Stattdessen wird eine statische Methode “get” angeboten, mit der man sich ein “Date”-Objekt zu gegebener Kombination von Tag, Monat, Jahr beschaffen kann.
- Alle bisher erzeugten “Date”-Objekte werden in einer verketteten Liste gehalten (siehe nächste Folie).
- Die Methode “get” kann daher prüfen, ob es das gewünschte Datum schon als Objekt gibt, und nur bei Bedarf ein neues Objekt erzeugen.



Beispiel (2)

- Dieses Programm enthält eine “verkettete Liste” zur Speicherung der Menge der bisher erzeugten “Date”-Objekte.
 - Effizienter wäre eine Hashtabelle, weil man das gewünschte Datum dann schneller finden kann. → “Datenstrukturen und effiziente Algorithmen I”.
- Dazu gibt es zwei statische Variablen:
 - `first` für das zuerst erzeugte Date-Objekt,
 - `last` für das bisher letzte erzeugte Objekt.
 - Am Anfang (wenn noch kein Objekt erzeugt wurde) sind beide `null`.
- Jedes Date-Objekt hat ein Attribut `next`, das auf das nächste Objekt verweist (also das direkt danach erzeugte).
 - Bzw. `null` für das momentan letzte. Der Name “next” ist so üblich, dass ich dieses Beispiel in Englisch machen musste, um Deutsch-Englisches-Mischmasch zu vermeiden (Empfehlung: Programmcode ganz in Englisch).

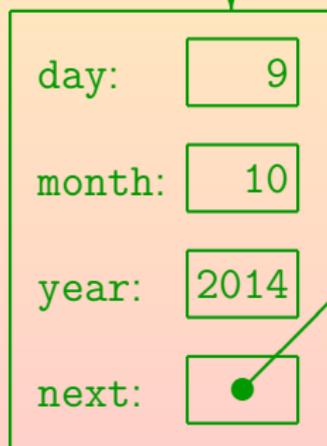
Beispiel (3)

Statische Variablen:

first: 

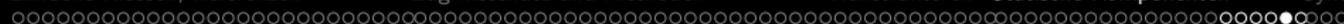
last: 

Objekte:



Beispiel (4)

```
(1) class Date {
(2)
(3)     // Attribute (Instanzvariablen):
(4)     private int day;
(5)     private int month;
(6)     private int year;
(7)     private Date next;
(8)
(9)     // Klassen-Attribute (statische Variablen):
(10)    private static Date first = null;
(11)    private static Date last  = null;
(12)
(13)    // Lesezugriff fuer Attribute:
(14)    public int day()    { return this.day; }
(15)    public int month() { return this.month; }
(16)    public int year()  { return this.year; }
(17)
```



Beispiel (5)

```
(18)    // Konstruktor (zur Vereinfachung
(19)    //         ohne Korrektheitspruefung):
(20)    private Date(int d, int m, int y) {
(21)        this.day    = d;
(22)        this.month = m;
(23)        this.year   = y;
(24)    }
(25)
(26)    // Suchen / Erzeugen von Objekten:
(27)    public static Date get(int d, int m, int y)
(28)    {
(29)        // Falls noch gar keine Objekte:
(30)        if(first == null) {
(31)            first = new Date(d, m, y);
(32)            last  = first;
(33)            return first;
(34)        }
```

Beispiel (6)

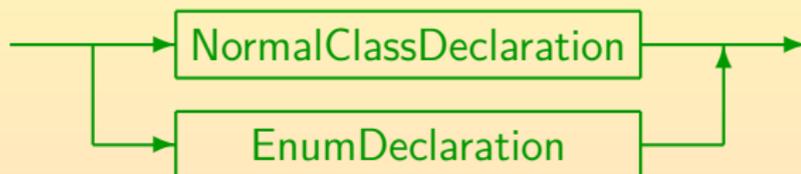
```
(35)         // Sonst suche Datum in Liste:
(36)         for(Date o = first; o != null;
(37)             o = o.next) {
(38)             if(o.day == d && o.month == m
(39)                 && o.year == y)
(40)                 return o;
(41)         }
(42)
(43)         // Nicht gefunden, neues Objekt erzeugen
(44)         // und hinten an Liste anhaengen:
(45)         Date n = new Date(d, m, y);
(46)         last.next = n;
(47)         last = n;
(48)         return n;
(49)     } // Ende der statischen Methode get()
(50)
(51)     ... // Weitere Methoden, s.o. Klasse Datum
```

Inhalt

- 1 Einfache Klassen, Referenzen
 - Strukturen, Attribute, Referenzen, Objekt-Erzeugung
- 2 Zugriffsschutz und Methoden
 - Von Strukturen zu Klassen, Methoden, Zugriffsschutz
 - Beispiel: Klasse für Datumswerte
- 3 Konstruktoren
 - Konstruktoren und Initialisierung
- 4 Statische Komponenten
 - Statische Variablen und Methoden
 - Konstanten
 - Beispiel: Verkettete Liste
- 5 **Syntax**
 - **Syntax von Klassen-Deklarationen**

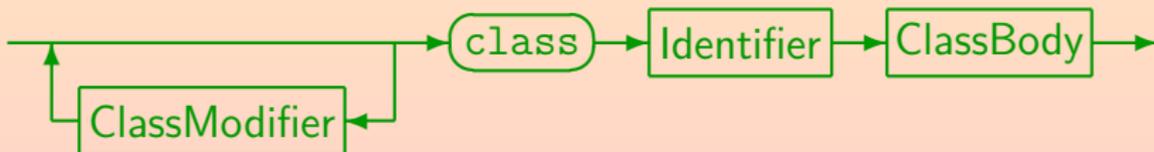
Klassen-Deklarationen: Syntax (1)

- **ClassDeclaration:**



Aufzählungs-Typen (enumeration types) werden in Kapitel 22 behandelt.

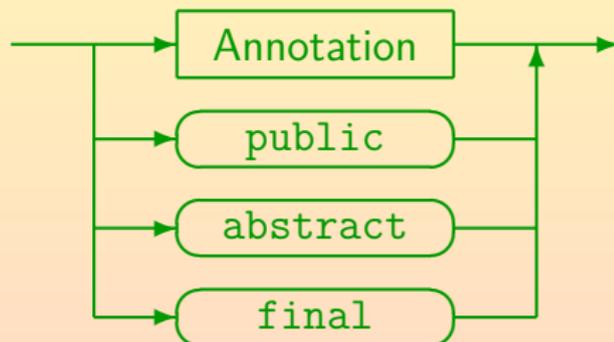
- **NormalClassDeclaration (vereinfacht):**



Hier fehlen noch Angaben zur Oberklasse (siehe Kapitel 12), zu implementierten Interfaces (Kapitel 13) und zu Typ-Parametern (Kapitel 20).

Klassen-Deklarationen: Syntax (2)

- **ClassModifier:**



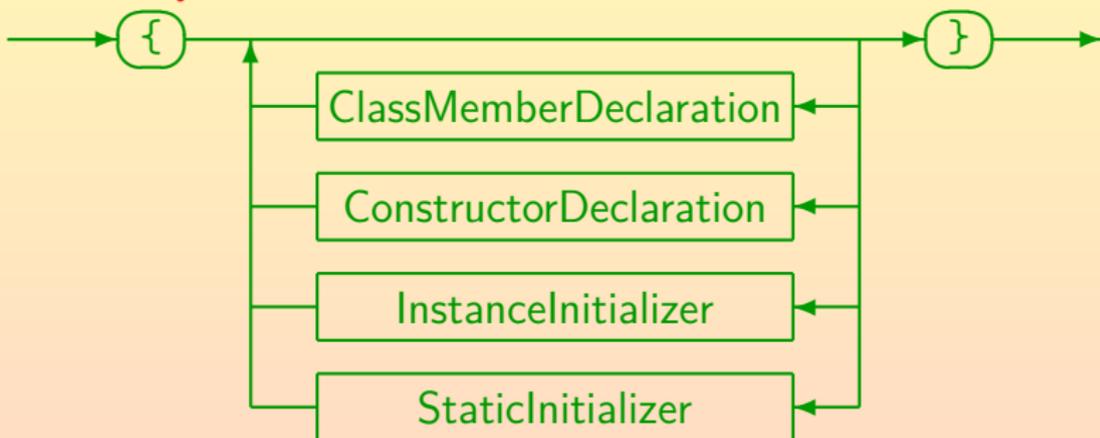
Der Modifier `public` wird in Kapitel 15 (Pakete) besprochen, `abstract` und `final` in Kapitel 12 (Subklassen).

Für geschachtelte Klassen (in dieser Vorlesung nicht behandelt) könnten auch die Modifier `protected`, `private`, `static` verwendet werden.

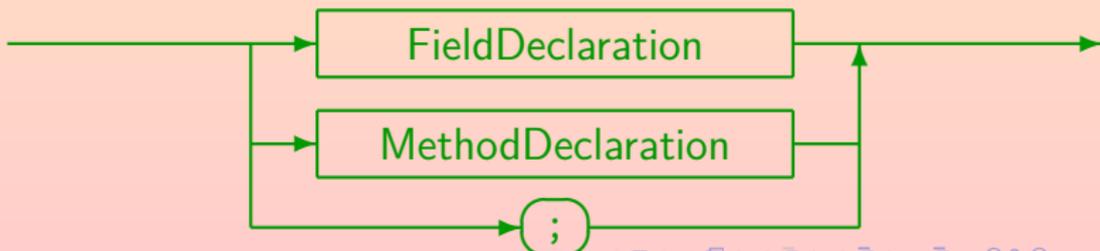
Außerdem gibt es noch den Modifier `strictfp` (ebenfalls hier nicht behandelt).

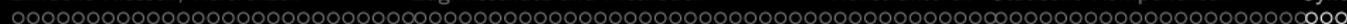
Klassen-Deklarationen: Syntax (3)

- **ClassBody:**



- **ClassMemberDeclaration:**





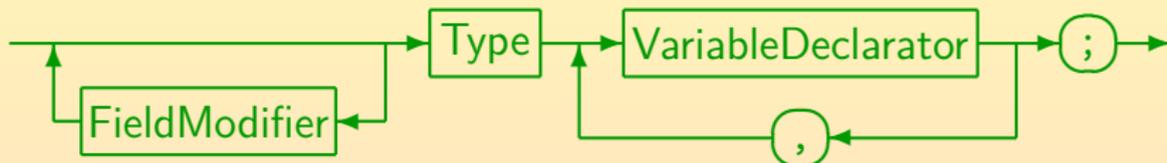
Klassen-Deklarationen: Syntax (4)

- In den geschweiften Klammern `{ ... }` kann man also Folgendes schreiben:
 - eine “Feld-Deklaration”, das ist eine Deklaration einer Instanzvariable (Attribut) oder einer statischen Variable,
 - eine Methoden-Deklaration (siehe Kapitel 8),
 - eine Deklaration eines Konstruktors,
 - Formal sind Konstruktoren und Initialisierungs-Blöcke keine “Member”, sie werden deswegen nicht an Subklassen vererbt (siehe Kapitel 12).
 - ein Initialisierungs-Block, sowohl statisch als auch für Objekte (“InstanceInitializer”),
- Außerdem (in dieser Vorlesung nicht behandelt), sind auch geschachtelte Klassen-/Interface-Deklarationen möglich.

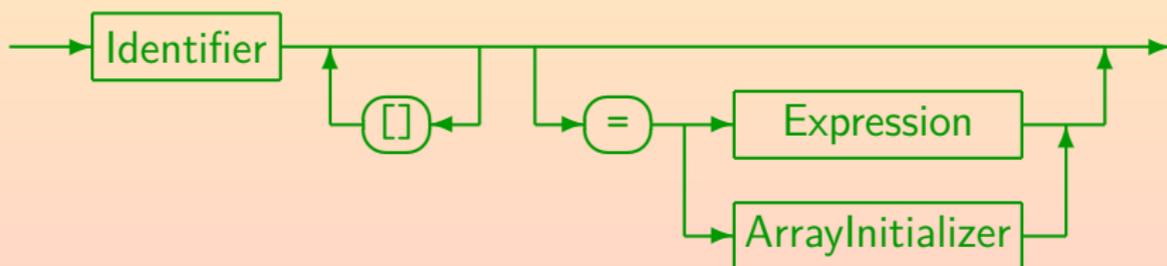
Dies zählt auch als “Member” der Klasse.

Feld-Deklarationen: Syntax (1)

- **FieldDeclaration:**



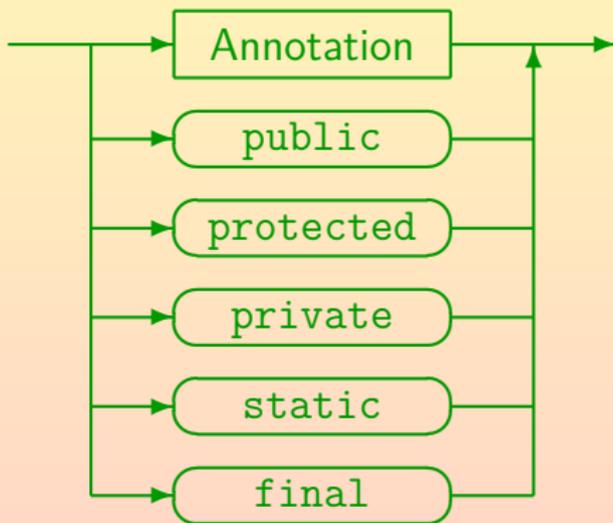
- **VariableDeclarator:**



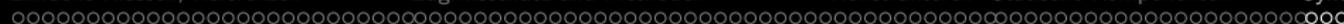
Die “[]” nach dem Variablennamen sind nur für Umsteiger von C++ gedacht.
Reine Java-Programmierer brauchen sie nicht (sie schreiben “[]” im Typ).
Initialisierungen von Arrays (“ArrayInitializer”) werden in Kapitel 16 besprochen.

Feld-Deklarationen: Syntax (2)

- FieldModifier:



Annotationen werden in dieser Vorlesung nicht behandelt (außer `@Override` für Methoden, s. Kap. 12). Die Zugriffsmodifizierer `public`, `protected`, `private` werden in Kap. 15 noch weiter diskutiert. Es gibt noch folgende FieldModifier, die in der dieser Vorlesung nicht behandelt werden: `transient`, `volatile`.



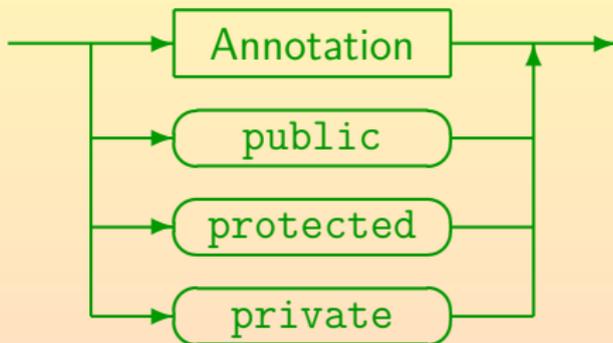
Konstruktor-Deklarationen: Syntax (2)

- Eine Konstruktor-Deklaration besteht also aus:
 - Modifizierern (nur `public`, `protected`, `private`, Kap. 15),
 - optionalen Typ-Parametern (siehe Kapitel 20),
 - dem Namen der Klasse (ein Identifier/Bezeichner),
 - eine "("
 - der Liste formaler Parameter (kann auch leer sein),
Die Syntaxdiagramme dafür finden sich in Kapitel 8.
 - eine ")"
 - optional einer `throws`-Klausel (siehe Kapitel 9), und
 - dem Rumpf des Konstruktors.

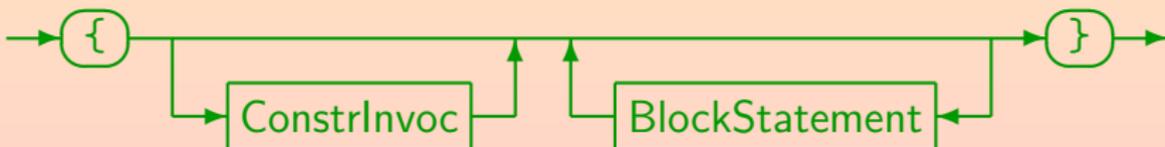
Dies ist im Prinzip wieder ein Block, aber das erste Statement kann ein Konstruktor-Aufruf sein, der speziell behandelt wird (s.u.).

Konstruktor-Deklarationen: Syntax (3)

- **ConstructorModifier:**



- **ConstructorBody:**



Bei Unterklassen (siehe Kap. 12) muss der Konstruktor der Oberklasse aufgerufen werden. Dies geschieht automatisch, wenn man nicht eine "ExplicitConstructorInvocation" als erstes Statement schreibt.

Initialisierungs-Blöcke: Syntax

- **InstanceInitializer:**



Initialisierungs-Blöcke dürfen kein `return`-Statement enthalten (und müssen normal enden können, d.h. nicht jeder Ausführungspfad darf mit `throw` enden). Im Instanz-Initialisierungsblock kann man mit `this` auf das aktuelle Objekt zugreifen. Instanz-Initialisierungsblöcke (sowie Initialisierungs-Ausdrücke in der Deklaration von Instanzvariablen) dürfen checked Exceptions nur erzeugen, wenn diese in der `throws`-Klausel von jedem Konstruktor deklariert ist. D.h. die Initialisierung wird implizit als Teil des Konstruktors behandelt. Zugriffe auf später deklarierte Felder (Vorwärts-Referenzen) sind eingeschränkt.

- **StaticInitializer:**



Statische Initialisierungsblöcke (und Initialisierungs-Ausdrücke für statische Variablen) dürfen keine checked Exceptions erzeugen.