Objektorientierte Programmierung

Kapitel 8: Klassen

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2012/13

http://www.informatik.uni-halle.de/~brass/oop12/

Inhalt

Einfache Klassen, Referenzen (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, Referenzen (2)

- Eine Struktur entspricht einer Java-Klasse ohne Methoden und mit von außen zugreifbaren Attributen:

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

Es gibt schon eine Klasse "`Date`" im Package `java.util`", sowie eine Klasse "`Calendar`" im gleichen Paket, die viel mächtiger ist als das, was wir hier programmieren wollen. Datumswerte sind aber ein gutes Beispiel. Natürlich sind öffentlich zugreifbare Attribute problematisch, das wird später diskutiert.

- Man kann sich jetzt Variablen vom Typ `Datum` deklarieren:

```
Datum d;
```

Einfache Klassen, Referenzen (3)

- Der neu definierte Klassentyp läßt sich also wie ein primitiver Typ verwenden.
- Allgemein können mit Klassen neue Datentypen definiert werden, die über die eingebauten Typen wie `int`, `float` u.s.w. hinausgehen.

  "A type is a concrete representation of a concept."

  "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.

Einfache Klassen, Referenzen (4)

- 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.

Einfache Klassen, Referenzen (5)

- Das war in klassischen Sprachen (und C++) anders: Deklarierte man dort eine Variable von einem Struktur-, Klassen- oder Array-Typ, so wurde gleich Speicherplatz für alle Komponenten reserviert.
- Natürlich gab es auch Referenzen (Zeiger), aber das war dann eben ein anderer Typ, und man mußte mit entsprechenden Operatoren explizit umrechnen.
- Es hat sich aber herausgestellt, dass man doch sehr häufig mit Zeigern auf Objekte gearbeitet hat.
- Deswegen gibt es in Java keinen Typ und keine Variablen für die Objekte selbst, nur für die Referenzen (Zeiger).

Man kann dann nicht mit Zeiger vs. Objekt durcheinander kommen.

Einfache Klassen, Referenzen (6)

- 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).

  In C++ gibt es dagegen zwei Möglichkeiten: Eine Variable von einem Klassentyp deklarieren und die "dynamische" Erzeugung mit `new`. Das kann zu Verwirrungen und Fehlern führen, andererseits ist die manuelle Speicherverwaltung über Variablendeklarationen effizienter.

Einfache Klassen, Referenzen (7)

- 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 der in der Klasse deklarierten Variable (Komponente, Attribut, Feld).

    Da es sich um drei einzelne Token (Wortsymbole) handelt, kann man links und rechts vom "`.`" Leerzeichen/Zeilenumbrüche einfügen.

Einfache Klassen, Referenzen (8)

- 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 Komponenten eines Objektes automatisch, 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.

Begriffe

- Ein Objekt einer Klasse heißt auch Instanz dieser Klasse.

  Eine Klasse wird häufig als Blaupause zur Herstellung von Objekten gesehen.

- Man erhält ein Objekt durch Instanziierung einer Klasse.

  Dies geschieht also mit dem Operator `new`.

- Die Komponenten heißen auch Instanz-Variablen, Attribute, Felder oder Datenelemente (der Klasse).

  Bisher hatten wir nur Daten-Komponenten (Variablen). Später werden auch Methoden sowie geschachtelte Klassen/Interfaces als Komponenten (engl. "member") gesehen (s.u.). Neben Instanz-Variablen gibt es auch Klassen-Variablen, die mit dem Schlüsselwort "`static`" markiert werden (s.u.).

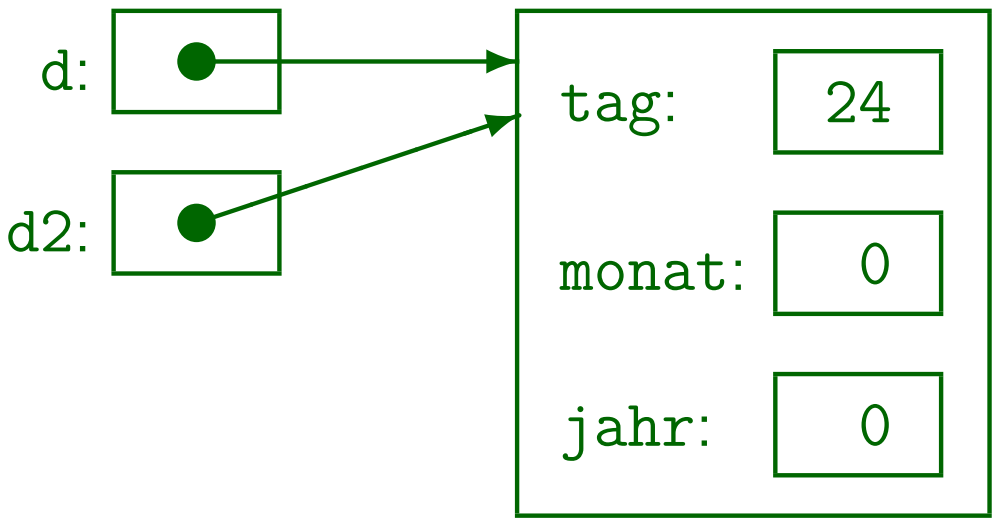
- Der Zustand eines Objektes ist die Belegung seiner Komponenten/Attribute mit konkreten Werten.

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.

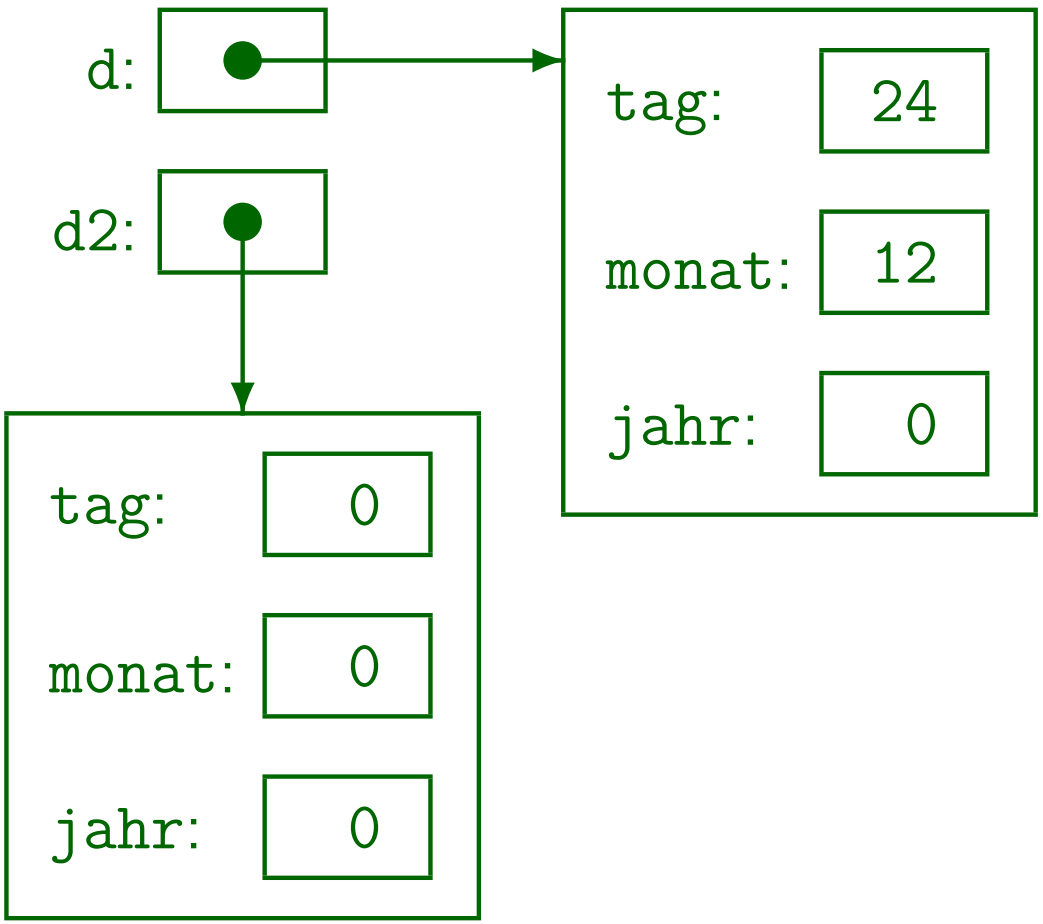
- Die Variablen `d` und `d2` sind dagegen nicht gleich.

  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 = 2012;
d2.tag = 1;
d2.monat = 1;
d2.jahr = 2013;
System.out.println(d.tag + "." + d.monat
                        + "." + d.jahr);
    // Druckt: 24.12.2012
System.out.println(d2.tag + "." + d2.monat
                        + "." + d2.jahr);
    // Druckt: 1.1.2013
```

- 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, 2012);
```

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.

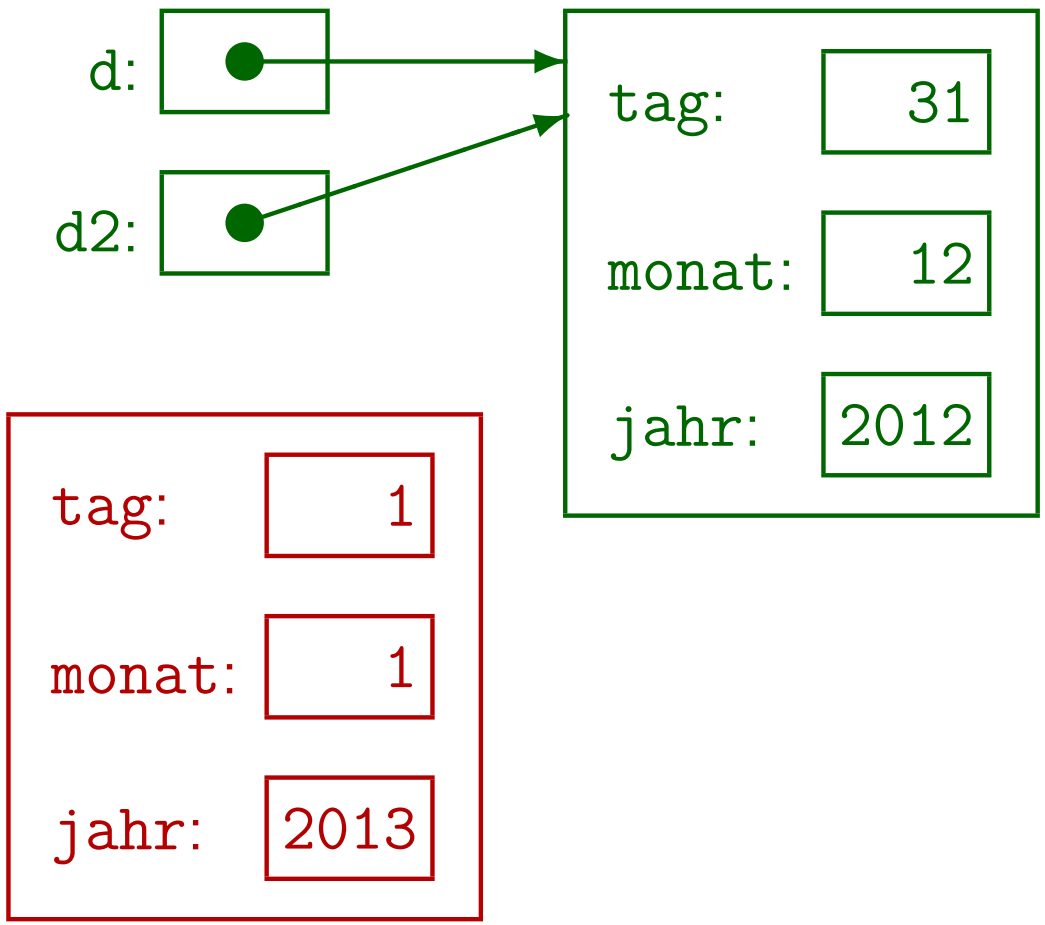
  In C++ hat man die Wahl, aber entscheidet sich meist gegen das Kopieren.

- 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 typischerweise ausgeführt, wenn der Rechner sonst nichts zu tun hat, oder wenn der Hauptspeicher langsam knapp wird.
- Daher ändert es nichts, wenn das Objekt gelöscht wird, d.h. der von ihm belegte Hauptspeicher freigegeben wird.

  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!

Null-Referenz (3)

- Wenn man ein Objekt nicht mehr braucht, auf das eine Variable `x` zeigt, kann man `x` auf `null` setzen:

```
x = null;
```

- Dadurch wird das Objekt für den Garbage Collector verfügbar.

  Falls es nicht noch andere Variablen gibt, die eine Referenz darauf enthalten.

- Das macht aber nur Sinn, wenn
  - das Objekt groß ist (oder viele andere Objekte direkt oder indirekt daran hängen), und
  - die Variable `x` noch lange existiert.

    Wenn es eine lokale Variable in einer Methode ist, und diese Methode zum Aufrufer zurückkehrt, wird `x` ohnehin gelöscht. Dann zählt die Referenz natürlich auch nicht mehr für den Garbage Collector.

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 Verhalte 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)

- Wenn `d` und `d2` beides Variablen von gleichem Referenztyp sind (z.B. Datum), kann man ihre Inhalte vergleichen:

```
if(d == d2)
    System.out.println("Gleich.");
```

- Hierfür wird geprüft, ob `d` und `d2` auf das gleiche Objekt zeigen.

- Falls `d` und `d2` auf unterschiedliche Objekte zeigen, liefert == den Wahrheitswert `false`, selbst wenn die beiden Objekte in allen Komponenten übereinstimmen.

  Also beide das gleiche Datum enthalten, d.h. `d.tag == d2.tag`, `d.monat == d2.monat` und `d.jahr == d2.jahr`.

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 natürlich so definieren, dass die Objekte nach der Initialisierung nicht mehr geändert werden können.
  - 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 (siehe Beispiel am Ende des Kapitels).

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;
}
```

- 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.

Implementierung (1)

- Der Compiler berechnet für jedes Attribut einer Klasse einen "Offset" von der Startadresse des Objektes im Hauptspeicher:

In Java ist der Typ `int` 32 Bit, also 4 Byte groß. Im Prinzip könnte man die erste Komponete des Objektes (Variable `tag`) dann direkt an der Startadresse des Objektes speichern (Offset 0). Sie belegt vier Byte, so dass die zweite Komponente (Variable `monat`) dann an der Startadresse plus 4 gespeichert wird (Offset 4). Entsprechend hat die dritte Komponente den Offset 8.

- Objekte der Klasse `Datum` wären dann 12 Byte groß.

Das ist etwas vereinfacht. Es müssen noch weitere Daten gespeichert werden, das kann man aber erst verstehen, wenn wir Subklassen und Interfaces gehabt haben.

Implementierung (2)

- Variablen vom Typ `Datum` enthalten die Startadressen dieser Struktur.

  Das ist auch eventuell vereinfacht: Bei einer Version der JVM verwiesen sie auf "Handles", von denen eine Komponente dann diese Startadresse war.
  Die andere Komponente dient zum Zugriff auf Methoden und Typ-Information.

- Falls das Objekt z.B. bei Adresse 2000 im Hauptspeicher steht, würde
  - die Komponente `tag` auch bei Adresse 2000,
  - die Komponente `monat` bei Adresse 2004, und
  - die Komponente `jahr` bei Adresse 2008 stehen.
- Ein Ausdruck wie `d.jahr` wird also ausgewertet, indem die ganze Zahl gelesen wird, die im Hauptspeicher an der Adresse "Inhalt von `d` plus 8" steht.

Inhalt

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 vorn Schnittstelle und Implementierung kannte 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 35.20.2012.

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 (oder eine "Exception auslösen", s.u.).

- 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.

  Im Rumpf der Methode kann man dann auf die Komponenten der Klasse ohne explizite Angabe eines Objektes zugreifen. Für das 0-te Argument selbst gibt es das Schlüsselwort "`this`".

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 Konstruktoren und Initialisierungs-Blöcke, die aber formal keine “member” sind.

  Weil sie keine “member” sind, werden sie nicht an Unterklassen vererbt.

  Unterklassen werden erst in Kapitel 11 behandelt.

Zugriffsschutz/Sichtbarkeit (1)

- Für Komponenten einer Klasse sowie Konstruktoren kann man Zugriffe von außerhalb erlauben oder verbieten mit den folgenden Schlüsselworten ("access modifiers"):
  - `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.
  - (keine Angabe): Zugriffe sind von allen Klassen des Paketes (s.u.) erlaubt.

    Dieser Zugriffsschutz wird "package", "package-scope" oder "default access" genannt. Es gibt kein Schlüsselwort dafür.
  - `protected`: Zugriffe sind von allen Klassen des Paketes erlaubt, und von Unterklassen (auch außerhalb des Paketes).
  - `public`: Keine Einschränkungen.

Zugriffsschutz/Sichtbarkeit (2)

Zur Syntax:

- Die "Access Modifier" `private`, `protected`, `public` müssen in jeder Komponenten-Deklaration einzeln angegeben werden.

  Wenn man nichts angibt, bekommt man den Default "package".

  Das ist anders als in C++: Dort schreibt man z.B. "`public:`", und das gilt dann für alle folgenden Komponenten, bis man den Zugriffsschutz explizit ändert.

- 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 11).

- 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.

Zugriffsschutz/Sichtbarkeit (7)

Zu "`protected`":

- Der Zugriffsschutz "`protected`" ist schwächer als wenn man gar nichts angibt: Aus dem Paket sind alle Zugriffe möglich.
- Wichtig wird "`protected`" also nur, wenn es Subklassen zur aktuellen Klasse in anderen Paketen geben kann.

  Weil man z.B. Basisfunktionalität oder eine Dummy-Implementierung schon im eigenen Paket hat, aber der Programmierer, der das Paket nutzen will, bestimmte Methoden überschreiben (selbst implementieren) muss (s. Kap. 11).
- Es sind nicht beliebige Zugriffe im Programmcode der Subklasse möglich, sondern nur für Objekte der Subklasse.

  Angenommen, die Klasse `S` ist Subklasse der Klasse `C` und in einem anderen Paket. Hat `C` ein `protected`-Attribut `a`, so ist `x.a` im Programmcode von `S` zulässig, wenn `x` den Typ `S` hat, aber nicht, wenn es den Typ `C` hat.

Einführung zu Paketen (1)

- Ein Paket (engl. “package”) ist eine Zusammenfassung von Dateien (und damit Klassen, aber ggf. auch weiterer Resourcen wie Bild-Dateien), die
  - inhaltlich zusammenhängen, also einem gemeinsamen größeren Zweck dienen,
  - ggf. als Ganzes in anderen Programmen wiederverwendet werden (wie eine Unterprogramm-Bibliothek).
- Ein Paket ist also eine Strukturierungsebene
  - oberhalb der einzelnen Klassen, aber
  - unterhalb eines größeren Anwendungsprogramms.
- Klassennamen müssen nur innerhalb eines Paketes eindeutig sein (Vermeidung von Namenskonflikten).

Einführung zu Paketen (2)

- Für Pakete ist eine hierarchische Struktur möglich (sie entsprechen Datei-Ordnern).

  Es kann also ein Paket in einem anderen Paket angelegt werden. Paketnamen sind entsprechend hierarchisch strukturiert, die Teile werden durch "`.`" getrennt.

- Ein Paket wird meist von einem einzelnen Programmierer erstellt, oder von einem Team, das eng zusammenarbeitet.

  Deswegen ist die Voreinstellung bei Java, das innerhalb eines Paketes beliebige Zugriffe möglich sind. Aber auch ein einzelner Programmierer macht Fehler, kann sich nicht mehr erinnern, oder verliert die Übersicht. Deswegen sollte man alles, was nicht von anderen Klassen aus zugreifbar sein muss, als "`private`" deklarieren. Wenn sich später herausstellt, dass man darauf doch zugreifen muss, kann man diese Einstellung ja ändern, oder entsprechende Zugriffsmethoden einführen.

Einführung zu Paketen (3)

- Eine Quelldatei wird einem Paket "P" zugeordnet durch die Deklaration

  ```
  package P;
  ```

  ganz zu Anfang der Datei.

  Also vor der oder den Klassen-Deklarationen. Kommentare wären natürlich auch vor der package-Deklaration möglich.

- Die Quelldatei muss dann auch in einem Verzeichnis "P" stehen.

  Allgemein hängt es vom "Class Loader" ab, wie er die Klassen findet.
  Es ist aber üblich, dass er Klassen, die Paketen zugeordnet sind, in einem Verzeichnis sucht, dass dem Paketnamen entspricht. Dabei werden hierarchisch strukturierte Paket-Namen entsprechend in Unterverzeichnissen gesucht, z.B. würde die Klasse "C" im Paket "P.Q" der Datei "P/Q/C.class" entsprechen.

Einführung zu Paketen (4)

- Man ruft `java` und `javac` vom übergeordneten Verzeichnis aus auf.

  Z.B. "`javac P/C.java`" und "`java P/C`" oder "`java P.C`". Der Paket-Name steht in der `class`-Datei. Selbst wenn man im Verzeichnis `P` ist, funktioniert "`java C`" nicht. Das Compilieren ist dagegen auch jeweils im Unterverzeichnis möglich. Wenn man aber auf Klassen in anderen Paketen zugreift, muss man den `CLASSPATH` entsprechend setzen, z.B. "`javac -cp ".:.." C.java`" (UNIX) bzw. "`-cp .;..`" (Windows). Damit die Klasse `C` im Paket `P` gefunden wird, muss der `CLASSPATH` ein Verzeichnis enthalten, das ein Unterverzeichnis `P` hat, in dem `C.class` steht. Wenn man nichts angibt, besteht der `CLASSPATH` aus dem aktuellen Verzeichnis "`.`". Die Standard-Pakete werden seit dem JDK 1.2 unabhängig vom `CLASSPATH` gefunden (Systemeigenschaft "`sun.boot.class.path`"). Anstatt den Pfad bei jedem Kommando anzugeben, kann man auch die Environment Variable setzen, z.B. "`setenv CLASSPATH /home/brass/oop`" bzw. "`set CLASSPATH=c:\classes`". [http://docs.oracle.com/javase/1.4.2/docs/tooldocs/findingclasses.html].

Einführung zu Paketen (5)

- Ohne `package`-Deklaration gehört die Klasse zum Default-Paket oder anonymen Paket.
- Das funktioniert gut, solange man relativ kleine Programme zum eigenen Gebrauch entwickelt.
- Will man Klassen im Internet austauschen, so sollten sie unbedingt in einem Paket "verpackt" werden.
- Um weltweit eindeutige Paketnamen zu bekommen, ist es üblich, den umgekehrten Domain-Namen der Firma als Präfix zu verwenden, z.B. "`com.sun.eng`".

  Das führt allerdings zu entsprechend tief geschachtelten Verzeichnis-Hierarchien. Solange man keinen weltweiten Austausch anstrebt, ist das nicht nötig.

Einführung zu Paketen (6)

- Eine Klasse "`C`" im Paket "`P`" kann man im Programm mit "`P.C`" ansprechen.
- Wenn man diese Klasse öfter braucht, kann man auch zu Anfang der Quelldatei folgende Deklaration schreiben:

```
import P.C;
```

  Dies muss nach der `package`-Deklaration (soweit vorhanden) stehen, und vor der Klassen-Deklaration.

- Anschließend kann man die Klasse einfach mit dem Namen "`C`" ansprechen.
- Es ist auch möglich, alle Klassen aus einem Paket unter ihrem einfachen Namen verfügbar zu machen:

```
import P.*;
```

  Der Compiler fügt automatisch "`import java.lang.*;`" zum Programm hinzu.

Einführung zu Paketen (7)

- Wenn es mehrere Klassen mit dem gleichen Namen "`C`" gibt, wird die erste in folgender Reihenfolge genommen:
  - Die aktuelle Klasse und ihre Oberklassen.
  - In der aktuellen Klasse geschachtelte Klassen.
  - Explizit importierte Klassen ("single type import").

    Man kann natürlich nicht eine Klasse importieren, die genauso heisst wie eine Klasse in der aktuellen Quelldatei.
  - Andere Klassen aus dem aktuellen Paket.
  - Mit "`*`" importierte Klassen ("import on demand").

    Falls zwei mit "`*`" importierte Pakete eine Klasse mit gleichem Namen enthalten, kann man sie nicht mit einfachem Namen ansprechen.
- Alles gilt entsprechend auch für Interfaces.

Zugriffsschutz für Klassen (1)

- Damit Klassen in einem fremden Paket verwendet werden können, müssen sie mit dem Schlüsselwort "`public`" markiert werden:

```
public class C { ... }
```

- Wenn man nichts angibt, sind sie nur innerhalb des Paketes sichtbar.

  Das ist genau wie bei Attributen und Methoden innerhalb einer Klasse.
  Die Schlüsselworte "`private`" und "`protected`" können für nicht-geschachtelte Klassen ("top-level classes") nicht verwendet werden. Geschachtelte Klassen werden später besprochen.

- Da eine "`public class`" so heißen muss wie die Quelldatei (natürlich ohne die Endung `.java`), kann pro Java-Quelldatei nur eine öffentliche Klasse definiert werden.

Zugriffsschutz für Klassen (2)

- Man auch auf die Komponenten einer Klasse nicht zugreifen, wenn die Klasse nicht zugreifbar ist.

  Insofern sehe ich keinen Sinn darin, `public`-Komponenten in einer nicht-`public` Klasse zu haben (Ausnahme ist die Methode `main`, die `public` sein muss, aber von `java` auch in einer nicht-`public` Klasse aufgerufen werden kann).

- Auch für Klassen gilt, dass man `public` nur verwenden sollte, wenn der Zugriff von außen nötig ist.

  Ein Paket hat oft relativ viele Hilfsklassen, die für die Benutzung der eigentlichen Funktion des Paketes von außen nicht sichtbar sein müssen.
  Man hat mehr Optionen für eine spätere Änderung der Paket-Implementierung, wenn man sicher sein kann, dass diese Klassen außerhalb nicht verwendet wurden.

Beispiel (1)

```
 (1)  package sb;
 (2)
 (3)  public class Datum {
 (4)
 (5)      // Attribute:
 (6)      private int tag;   // 1..31
 (7)      private int monat; // 1..12
 (8)      private int jahr;  // 1600 ..
 (9)
(10)      // Lesezugriff auf Attribute:
(11)      // ("accessor" oder "getter"-Methoden,
(12)      //  üblich wären Namen wie "getTag()")
(13)      public int tag()   { return this.tag; }
(14)      public int monat() { return this.monat; }
(15)      public int jahr()  { return this.jahr; }
(16)
```

Beispiel (2)

```
(17)      // Hilfsfunktion: Test auf Schaltjahr:
(18)      private boolean schaltjahr(int jahr) {
(19)          if(jahr % 400 == 0)
(20)              return true;
(21)          if(jahr % 100 == 0)
(22)              return false;
(23)          if(jahr % 4 == 0)
(24)              return true;
(25)          return false;
(26)      }
(27)
```

Beispiel (3)

```
(17)      // Hilfsfunktion: Tage im Monat
(18)      private int mTage(int monat, int jahr) {
(19)          if(monat == 2) {
(20)              if(schaltjahr(jahr))
(21)                  return 29;
(22)              else
(23)                  return 28;
(24)          }
(25)          if(monat == 4 || monat == 6 ||
(26)                  monat == 9 || monat == 11) {
(27)              return 30;
(28)          }
(29)          return 31;
(30)      }
(31)
```

Beispiel (4)

```
(32)      // Setzen der Attribute
(33)      // (besser mit Konstruktor, s.u.)
(34)      // Liefert false wenn Datum ungueltig.
(35)      public boolean init(int tag, int monat,
(36)                          int jahr) {
(37)          if(jahr < 1600)
(38)              return false;
(39)          if(monat < 1 || monat > 12)
(40)              return false;
(41)          if(tag < 1 || tag > mTage(monat,jahr))
(42)              return false;
(43)          this.tag = tag;
(44)          this.monat = monat;
(45)          this.jahr = jahr;
(46)          return true;
(47)      }
(48)
```

Beispiel (5)

```
(49)      // Berechnung des Wochentags
(50)      // (1=Montag, 2=Dienstag, ...)
(51)      public int wochentag() {
(52)          int t = 1;
(53)          int m = 1;
(54)          int j = 1600;
(55)          int w = 6; // 1.1.1600 war Samstag
(56)          while(j<jahr || m<monat || t<tag) {
(57)              w++;
(58)              if(w == 8) { w = 1; }
(59)              t++;
(60)              if(t > mTage(m, j)) { t = 1; m++; }
(61)              if(m > 12) { m = 1; j++; }
(62)          }
(63)          return w;
(64)      }
```

Beispiel (6)

```
(64)
(65)      // Name des Wochentags:
(66)      public String tagname() {
(67)          switch(wochentag()) {
(68)              case 1: return "Montag";
(69)              case 2: return "Dienstag";
(70)              case 3: return "Mittwoch";
(71)              case 4: return "Donnerstag";
(72)              case 5: return "Freitag";
(73)              case 6: return "Samstag";
(74)              case 7: return "Sonntag";
(75)          }
(76)          return null;
(77)      }
(78)
(79)  } // 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]

- Selbstverständlich sollte es noch eine Methode für den Monatsnamen geben (zur Übung).
- Die Berechnung des Wochentags kann man beschleunigen, indem man zunächst ganze Jahre springt (zur Übung).
- Wesentliches Problem: Initialisierung!

Inhalt

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.

```
public 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 Konstrukor 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, 2012);
```

- 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(24,12,2012);`".

  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).

- 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).

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. In C++ geht es nicht, dort muss alle Initialisierung im Konstruktor erfolgen.

- 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 auspobieren 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

Statische Variablen (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. (Man erhält ein Objekt durch Instanziierung einer Klasse, das Objekt ist dann Instanz der Klasse.)

- Daneben ist es auch möglich, Variablen zu deklarieren, die nur ein einziges Mal für die ganze Klasse existieren.

- Diese Variablen heißen Klassenvariablen/Klassen-Attribute und werden mit dem Schlüsselwort "`static`" gekennzeichnet.

  Deswegen werden sie auch statische Variablen genannt.

- Statische Variablen existiert auch dann, wenn noch kein einziges Objekt der Klasse `Datum` angelegt wurde.

Statische Variablen (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, also z.B.

  ```
  private static int anzObjekte;
  ```

  Die Reihenfolge innerhalb der Modifier ist egal. Es ist aber üblich, den Zugriffsschutz zuerst anzugeben (s.o.). Beispiel-Anwendung: Zählen, wie viele Objekte der Klasse `Datum` gerade existieren. Die statische Variable wird im Konstruktor inkrementiert ($+1$) und im Destruktor dekrementiert (-1).

Statische Variablen (3)

Typische 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 (4)

Syntax (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 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 (5)

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 Variablen (6)

Initialisierung (Feinheiten):

- In einer Initialisierung darf man nicht direkt auf später deklarierte statische Variablen zugreifen.

  Nur Hilfe für den Programmierer, um zyklische Initialisierungen auszuschließen. Man kann es umgehen, indem man eine Methode aufruft, die den Variablenwert liefert. Man bekommt dann den Defaultwert `0`/`null`.

- Die Initialisierung wird ausgeführt, wenn die Klasse zum ersten Mal verwendet wird.

  D.h. ein Konstruktor oder eine Methode aufgerufen wird, oder auf eine Variable der Klasse zugegriffen wird (außer bei zur Compilezeit bekannten Konstanten). Es wird zuerst die Oberklasse initialisiert (aber nicht implementierte Interfaces: sie werden erst beim Zugriff auf dort deklarierte Variablen initialisiert, sofern diese nicht ohnehin zur Compilezeit bekannt sind). Initialisiert wird auch nur die Klasse, in der die jeweilige Variable deklariert ist, nicht eine eventuelle Subklasse, über die darauf zugegriffen wird.

Statische Methoden (1)

- Auch Methoden-Deklarationen können mit dem Schlüsselwort "`static`" gekennzeichnet werden:

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

- Das bedeutet, dass man kein Objekt der Klasse angeben muss, wenn man die Methode aufruft:

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

- Zum Aufruf von außerhalb der Klasse stellt man also wie bei statischen Variablen den Klassennamen voran.

  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 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`).

Statische Methoden (4)

- Wenn eine Methode nicht auf Instanz-Variablen 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).

Konstanten (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 (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 (3)

Typische Anwendungen von Konstanten:

- 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 (4)

Typische Anwendungen von Konstanten, Forts.:

- 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.RecourceBundle`".

Konstanten (5)

Stil-Hinweis:

- 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 "verkette 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".

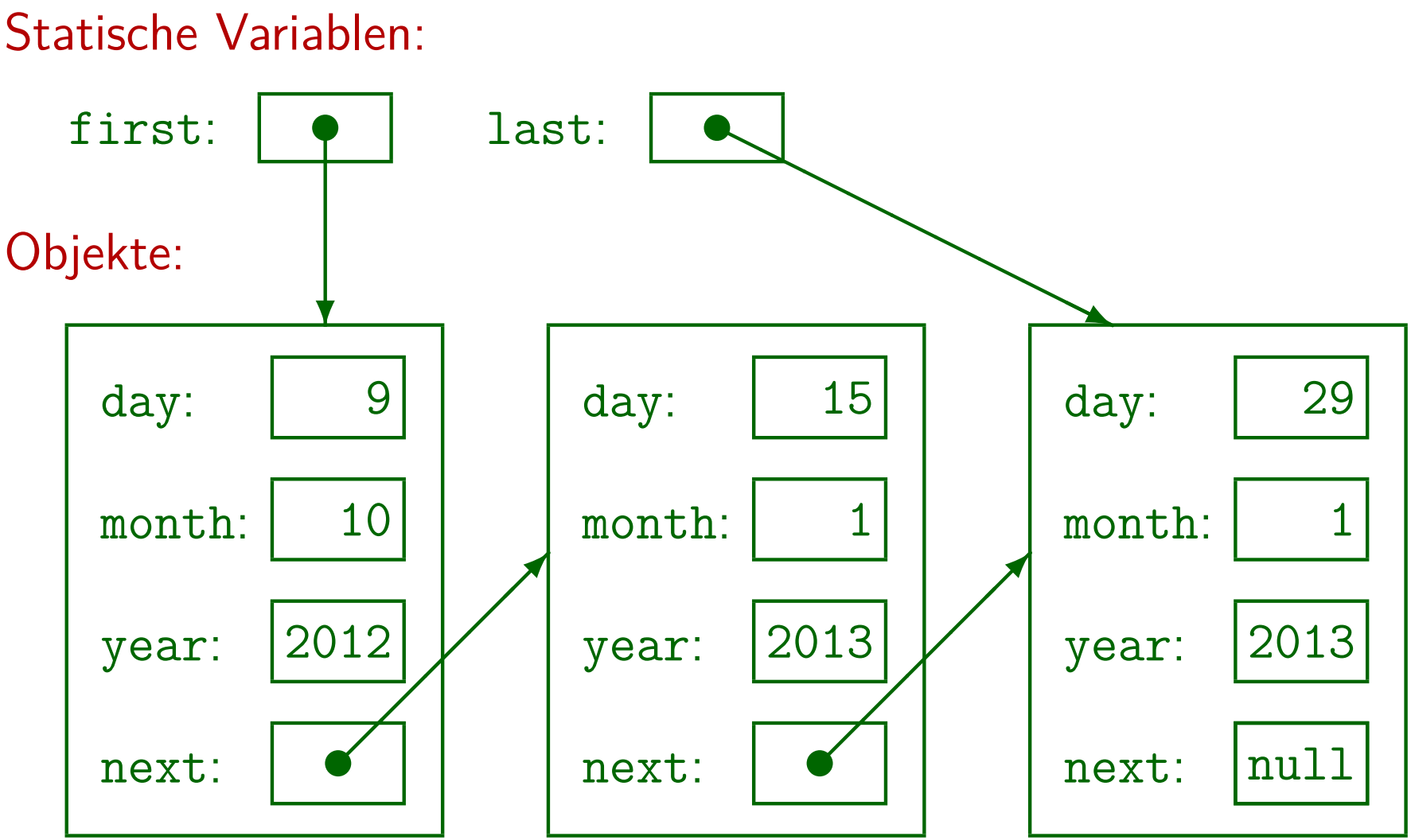
- Dazu gibt es zwei statische Variablen:
  - `first` für das zuerst erzeugte `Date`-Objekt,
  - `last` für das bisher letzte erzeugte Objekt.

    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:
day:
9
month:
10
year:
2012
next:
day:
15
month:
1
year:
2013
next:
day:
29
month:
1
year:
2013
next:
null

Beispiel (4)

```
 (1)  class Date {
 (2)
 (3)      // Attribute (Instanz-Variablen):
 (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
```