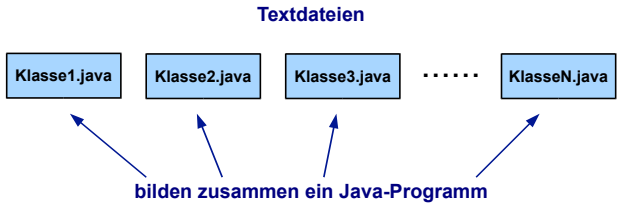
Zur Struktur von Java-Programmen

- der Text von Java-Programmen wird – bis auf triviale Ausnahmen – immer auf mehrere Textdateien verteilt
  - jede dieser Dateien muss die Extension `.java` haben
- anfangs sollten sie sich alle im selben Verzeichnis befinden
  - jedes Programm in einem eigenen Verzeichnis
- große Java-Programme können aus Hunderten oder gar Tausenden von Dateien bestehen, die dann auf mehrere Verzeichnisse verteilt werden



- den Überblick darüber zu bewahren, welche Datei zu welchem Programm gehört, ist Aufgabe der Programmiererinnen und Programmierer

- in Java wird formal zwischen Klassen und Interfaces unterschieden
- wir empfehlen, dass jede Datei eines Java-Programms die Deklaration genau einer Klasse oder genau eines Interfaces enthält
  - die aber jeweils innere Klassen oder innere Interfaces haben können

Hinweis: aus Sicht der Sprache ist diese Einschränkung nicht zwingend, d.h. eine Datei kann auch die Deklaration mehrerer Klassen oder Interfaces enthalten

  - dann sind aber einige Dinge zusätzlich zu beachten

- der Name der Datei sollte identisch sein mit dem Namen der (einzigen) darin enthaltenen Klasse oder des (einzigen) darin enthaltenen Interfaces
  - weil auch in Java Groß- und Kleinschreibung eine Rolle spielt und es üblich ist, dass die Namen von Klassen und Interfaces immer mit einem großen Buchstaben beginnen, beginnen die Dateinamen – im Gegensatz zum in C/C++ Üblichen – immer mit einem großen Buchstaben
- wie in C/C++ können die Dateien eines Java-Programms mit einem einfachen Texteditor erstellt werden
  - weit verbreitet ist aber der Einsatz der IDE Eclipse

- als erstes Beispiel betrachten wir ein einfaches Programm, das aus zwei Dateien besteht

ErstesProgrammMain.java

```
public class ErstesProgrammMain
{
  public static void main(String[] args)
  {
    schreibe("\n");
    MeinDatum datum = new MeinDatum();
    schreibe("Heute ist der " + datum.liefereDatum() + ".");
    schreibe("\n\n");
    schreibe("Mein erstes Java-Programm funktioniert!");
    schreibe("\n\n");
  }

  private static void schreibe(String text)
  {
    System.out.print(text);
  }
}
```


MeinDatum.java

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MeinDatum
{
  private int tag;
  private int monat;
  private int jahr;

  public MeinDatum()
  {
    GregorianCalendar kalender = new GregorianCalendar();
    tag = kalender.get(Calendar.DAY_OF_MONTH);
    monat = kalender.get(Calendar.MONTH) + 1;
    jahr = kalender.get(Calendar.YEAR);
  }

  public String liefereDatum()
  {
    return tag + ". " + monat + ". " + jahr;
  }
}
```


- wie in C/C++ kann jede Datei eines Java-Programms einzeln für sich übersetzt werden
- wie man Java-Dateien übersetzt, hängt von der Entwicklungsumgebung ab
  - wir geben an, wie man in der Kommandozeile einer (Betriebssystem-)Shell vorgeht
- im ersten Schritt wird die Datei mit dem Compiler in den Maschinencode einer abstrakten Maschine überführt und zwischengespeichert
  - der Compiler heißt **javac**, die abstrakte Maschine wird Java Virtual Machine (JVM) genannt, der entstehende Maschinencode ist der Java Byte Code

- dazu wechselt man (der Einfachheit halber) in das Verzeichnis, in dem sich die zu übersetzende Datei befindet (z.B. **`ErstesProgrammMain.java`**), und gibt folgendes Kommando ein:

  `javac ErstesProgrammMain.java`

- wurde die Entwicklungsumgebung korrekt installiert und ist der Inhalt der Datei syntaktisch korrekt, wird vom Compiler im selben Verzeichnis eine Datei namens

  `ErstesProgrammMain.class`

  erzeugt, die den zugehörigen Java Byte Code enthält

- dieser Byte Code ist auf allen Rechnern gleich !
- was passiert beim Übersetzen einer Datei eines Java-Programms, wenn Dinge benötigt werden, die in einer anderen Datei deklariert wurden?
  - es gibt keine Header-Dateien und keinen Präprozessor, mit dem sie eingebunden werden könnten
- in der Regel ist das kein Problem:
  - der Compiler **`javac`** ist so gestaltet, dass er die benötigten Dinge (in Zusammenarbeit mit dem Betriebssystem) selbst sucht

- dabei ist es hilfreich, wenn sich die anderen Dateien des Programms im selben Verzeichnis befinden
- auf den Inhalt einer Datei schließt er über deren Namen
  - das ist der Grund dafür, dass der Dateiname und der Name der sich darin befindenden (einzigen!) Klasse oder des (einzigen!) Interfaces identisch sein sollen
- wenn er allerdings etwas Benötigtes nicht findet, übersetzt er die Datei nicht

- die Dateien **`ErstesProgrammMain.java`** und **`MeinDatum.java`** unseres Beispiels können also in beliebiger Reihenfolge einzeln übersetzt werden
- ausgeführt wird ein Java-Programm mit Hilfe eines Interpreters
  - er heißt **`java`** und wird auch als Java Virtual Machine (JVM) bezeichnet

- für unser Beispiel wechselt man (der Einfachheit halber) in das Verzeichnis, in dem sich die Datei **`ErstesProgrammMain.class`** mit dem Java Byte Code befindet, und gibt folgendes Kommando ein:

```
java ErstesProgrammMain
```

Achtung: ohne Extension !

auf dem Bildschirm wird ausgegeben:

bei Ihnen sollte hier das aktuelle Datum stehen !

```
Heute ist der 2. 11. 2010.

Mein erstes Java-Programm funktioniert!
```


- die JVM beginnt die Ausführung des Byte Codes immer mit der Ausführung der Methode **`main`**

Achtung: hat eine Klasse keine Methode namens **`main`** (wie z.B. **`MeinDatum`**), kann sie nicht auf diese Weise ausgeführt werden, d.h.

```
java MeinDatum
```

würde zu einer Fehlermeldung folgender Art führen:

```
Exception in thread "main"
java.lang.NoSuchMethodError: main
```


- der Byte-Code derartiger Klassen kann nur im Rahmen der Ausführung anderer Klassen ausgeführt werden
  - wenn die JVM feststellt, dass er benötigt wird, wird er vom so genannten Class Loader gesucht, geladen und dann von der JVM ausgeführt
- im Prinzip kann jede Java-Klasse mit einer (geeignet programmierten) Methode namens **`main`** versehen werden
  - das empfehlen wir aber nicht

- es ist übersichtlicher, wenn jedes Programm nur genau einen Startpunkt hat
  - daher sollte es in jedem Programm nur genau eine Klasse mit Methode **`main`** geben
- damit man diese Klasse leicht identifizieren kann, sollte sie stets **`XyzMain`** heißen, wobei **`Xyz`** einen Hinweis darauf gibt, wozu das Programm dient (**`ErstesProgrammMain`**)

- alle Java-Dateien haben dieselbe Struktur:
  - es gibt keine oder genau eine **`package`**-Anweisung
  - danach kommen keine, eine oder mehrere **`import`**-Anweisungen
  - danach folgen eine oder mehrere Klassen- oder Interfacedeklarationen
    - wir empfehlen, dass nur genau eine Klassen- oder Interfacedeklaration folgt
    - statt einer Klassen- oder Interfacedeklaration kann als seltener Sonderfall auch nur ein `;` stehen
  - außerdem kann es Kommentare geben
    - für Kommentare gilt in Java dasselbe wie in C/C++

`MusterKlasse.java`

```
package-Anweisung   // kann fehlen

import-Anweisung1   // kann fehlen
......
import-AnweisungN   // kann fehlen

Klassendeklaration
```


wenn die `package`- und die `import`-Anweisungen fehlen, sieht das konkret zum Beispiel so aus:

`ErstesProgrammMain.java`

```
public class ErstesProgrammMain
{
  public static void main(String[] args)
  {
    schreibe("\n");
    MeinDatum datum = new MeinDatum();
    schreibe("Heute ist der " + datum.liefereDatum() + ".");
    schreibe("\n\n");
    schreibe("Mein erstes Java-Programm funktioniert!");
    schreibe("\n\n");
  }

  private static void schreibe(String text)
  {
    System.out.print(text);
  }
}
```


bei der zweiten Datei des Programms fehlt die `package`- Anweisung, es gibt aber `import`-Anweisungen:

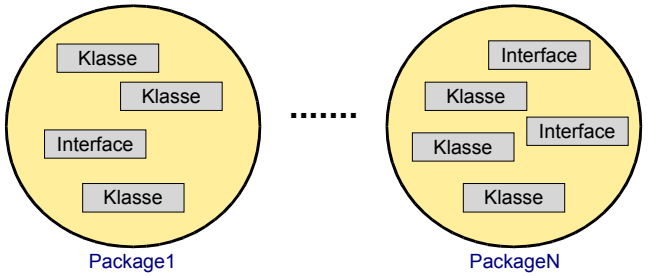
`MeinDatum.java`

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MeinDatum
{
  private int tag;
  private int monat;
  private int jahr;

  public MeinDatum()
  {
    GregorianCalendar kalender = new GregorianCalendar();
    tag = kalender.get(Calendar.DAY_OF_MONTH);
    monat = kalender.get(Calendar.MONTH) + 1;
    jahr = kalender.get(Calendar.YEAR);
  }
```

```
  public String liefereDatum()
  {
    return tag + ". " + monat + ". " + jahr;
  }
}
```

- zu beachten ist, dass
  - Klassendeklarationen in Java nicht mit einem Semikolon abgeschlossen werden
  - der Code der Konstruktoren und Methoden direkt bei der Deklaration angegeben wird

- wozu dient die **`package`**-Anweisung?
  - inhaltlich zusammenhängende Klassen und Interfaces können in Java zu so genannten Packages zusammengefasst werden



- Packages dienen zur Strukturierung großer Programme oder Klassenbibliotheken
  - die Java-Klassenbibliothek der Java SE 6 besteht zum Beispiel aus mehr als 160 Packages, die zusammen mehr als 3.200 Klassen und Interfaces enthalten
- sie dienen auch zur Vermeidung von Namenskonflikten (und erfüllen damit dieselbe Funktion wie Namespaces bei C++)
  - innerhalb eines Packages müssen alle Klassen und Interfaces unterschiedliche Namen haben
  - dagegen dürfen Klassen und Interfaces aus verschiedenen Packages denselben Namen haben

- erreicht wird das dadurch, dass dem Namen von Klassen und Interfaces beim Übersetzen automatisch der Name des Packages vorangestellt wird
  - die Java Klassenbibliothek enthält z.B. drei Klassen namens **`Timer`**, die sich in den Packages **`java.util`**, **`javax.management.timer`** und **`javax.swing`** befinden
    - deren vollständiger Name lautet daher
      - **`java.util.Timer`**
      - **`javax.management.timer.Timer`**
      - **`javax.swing.Timer`**
    - bei Bedarf kann er auch so benutzt werden

- **Achtung:** obwohl es bei Packages in Java und Namespaces in C++ einige Gemeinsamkeiten gibt, handelt es sich um unterschiedliche Konzepte, auf die wir aber nicht näher eingehen
- mit der `package`-Anweisung wird festgelegt, zu welchem Package die in der Datei deklarierte Klasse oder das Interface gehört
  - fehlt sie, wird die Klasse oder das Interface automatisch einem Default-Package zugeordnet, das keinen Namen hat

- wozu dienen `import`-Anweisungen?
  - mit ihnen wird dem Compiler mitgeteilt, dass in der anschließend deklarierten Klasse oder dem Interface bereits vorhandene Klassen oder Interfaces aus anderen Packages wiederverwendet werden
- es gibt vier Varianten der `import`-Anweisung, von denen wir zunächst nur zwei betrachten
- bei der ersten Variante werden wiederzuverwendende Klassen oder Interfaces jeweils einzeln mit ihrem vollständigen (internen) Namen angegeben:


```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MeinDatum
{
  private int tag;
  private int monat;
  private int jahr;

  public MeinDatum()
  {
    GregorianCalendar kalender = new GregorianCalendar();
    tag = kalender.get(Calendar.DAY_OF_MONTH);
```

- bei der zweiten Variante wird lediglich das Package angegeben, aus dem Klassen oder Interfaces wiederverwendet werden sollen:


```
import java.util.*;

public class MeinDatum
{
  private int tag;
  private int monat;
  private int jahr;

  public MeinDatum()
  {
    GregorianCalendar kalender = new GregorianCalendar();
    tag = kalender.get(Calendar.DAY_OF_MONTH);
```

  - in diesem Fall ermittelt die JVM selbst, welche Klassen oder Interfaces tatsächlich wiederverwendet werden
- die zweite Variante ist kürzer, hat aber Nachteile

- werden mehrere Packages importiert, ist beim Lesen des Programmtextes nicht klar, aus welchem Package welche der wiederverwendeten Klassen oder Interfaces stammen
  - da das vor allem bei Nichtstandard-Packages zu Problemen führt, müssen dann entsprechende Kommentare eingefügt werden
- werden Packages importiert, die gleichnamige Klassen enthalten, führt deren (unvorsichtige) Verwendung zu einem Fehler

- wir empfehlen daher, stets die erste Variante zu benutzen

- werden mehrere Klassen oder Interfaces importiert, empfehlen wir außerdem, die **`import`**-Anweisungen zu sortieren
  - und zwar alphabetisch nach den Package-Namen
    - und innerhalb der Packages nach den Namen

```
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
```


- bei beiden Varianten muss dafür gesorgt werden, dass der Class Loader die wiederzuverwendenden Klassen und Interfaces findet
  - für Packages der Java-Klassenbibliothek ist dies kein Problem
  - bei selbst erstellten Packages müssen deren Namen aber gewissen Regeln genügen, auf die wir hier nicht eingehen
- eine Sonderrolle hat das Package **`java.lang`** der Java-Klassenbibliothek
  - es wird automatisch importiert, da dessen Inhalt in jedem Java-Programm gebraucht wird

- zusammenfassend ergibt sich, dass ein typisches Java-Programm aus mehreren Dateien besteht, die (bei uns) jeweils die Deklaration genau einer Klasse oder genau eines Interfaces enthalten
  - (globale) Funktionen wie in C/C++ gibt es nicht !

- ein Java-Programm zu schreiben bedeutet also, eine Reihe von Bauplänen zu verfassen
  - und damit nicht, die einzelnen Schritte zur Lösung eines Problems zu beschreiben, wie in der prozeduralen Programmierung

- es ist somit nur noch zu klären, wie in Java Klassen und Interfaces zu deklarieren sind
- wir beginnen zunächst mit einem Überblick
  - für die Details werden werden wir viele Wochen brauchen
- dabei beschränken wir uns vorerst auf die Deklaration gewöhnlicher Klassen
  - seit der J2SE 5.0 gibt es in Java Klassen vom Typ `enum` und so genannte generische Klassen
  - für sie gibt es besondere Regeln, die wir erst zu gegebener Zeit betrachten

- formal wird durch eine Klassendeklaration ein neuer Referenzdatentyp eingeführt
- von der Idee her sind Klassen aber Baupläne, nach denen bei Bedarf beliebig viele Objekte „gebaut“ werden können
  - im selben Programm oder – bei Wiederverwendung – später
- wie hat eine gewöhnliche Klassendeklaration auszusehen?
  - in der Java Language Specification nimmt die Antwort auf diese Frage 75 Seiten ein

    siehe: http://docs.oracle.com/javase/specs/

- eine gewöhnliche Klassendeklaration hat immer folgende Gestalt:

```
KlassenModifizierer class KlassenName
{
  KlassenrumpfDeklaration1
  ....
  KlassenrumpfDeklarationN
}
```

  - an Hand des Schlüsselwortes `class` erkennt der Compiler, dass es sich um die Deklaration einer Klasse handelt
  - jede Klasse erhält einen Namen, dessen erster Buchstabe immer groß geschrieben wird (Konvention)
  - die Klassenmodifizierer können fehlen
    - in der Regel gibt es aber mindestens einen

  - der Klassenrumpf enthält ausschließlich Deklarationen (keine, eine oder in der Regel mehrere)
- die Klassendeklaration kann auch so angeordnet werden:

```
KlassenModifizierer class KlassenName {
  KlassenrumpfDeklaration1
  ....
  KlassenrumpfDeklarationN
}
```

  - es bleibt Ihnen überlassen, welchen der beiden Stile Sie bevorzugen
    - aber bitte keinen anderen
    - und immer denselben
  - wir bevorzugen den klassischen C/C++-Stil

- die Klassenmodifizierer dienen dazu, die deklarierte Klasse mit speziellen Eigenschaften zu versehen
- es gibt drei Arten von Klassenmodifizierern:
  - Annotations
  - die Zugriffsmodifizierer **`public`**, **`protected`** und **`private`**
  - die sonstigen Klassenmodifizierer **`abstract`**, **`static`**, **`final`** und **`strictfp`**
- die Zugriffsmodifizierer legen fest, von wo aus auf die Klasse zugegriffen werden kann
  - d.h. wo überall der „Bauplan“ benutzt werden kann

- werden Klassen als **`public`** deklariert, können sie überall benutzt werden

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MeinDatum
{
  private int tag;
  private int monat;
  private int jahr;

  public MeinDatum()
  {
    GregorianCalendar kalender = new GregorianCalendar();
    tag = kalender.get(Calendar.DAY_OF_MONTH);
```

  - das ist sinnvoll, wenn sie – ganz im Sinn der Objektorientierung – zur allgemeinen Wiederverwendung zur Verfügung stehen sollen

  - fast alle Klassen (und Interfaces) der Java-Klassenbibliothek sind **`public`**
- die Zugriffsmodifizierer **`protected`** und **`private`** dürfen nur bei der Deklaration innerer Klassen eingesetzt werden
  - darauf gehen wir erst zu gegebener Zeit ein
- wird in der Deklaration einer Klasse kein Zugriffsmodifizierer angegeben, kann sie nur innerhalb des Packages benutzt werden, zu dem sie gehört
  - da die Klassen des eigenen Packages als „Freunde“ betrachtet werden, wird dieser Zustand auch als „friendly“ bezeichnet

  - **Achtung:** wird einfach nur vergessen, eine Klasse mit einem Zugriffsmodifizierer zu versehen, handelt es sich in der Regel um einen Programmierfehler!

- der sonstige Klassenmodifizierer **`static`** darf ebenfalls nur bei der Deklaration innerer Klassen benutzt werden
- welche Auswirkungen die sonstigen Klassenmodifizierer **`abstract`**, **`final`**, und **`strictfp`** haben, klären wir zu gegebener Zeit

- Annotations wurden erst mit der J2SE 5.0 in Java eingeführt
  - sie sind vor allem für die Gestaltung von Frameworks wichtig, worauf wir erst zu gegebener Zeit eingehen
- **Fazit:** bei uns werden Klassen zunächst immer (nur) `public` sein

- in Java besteht der Rumpf von Klassen immer aus einer Folge von Deklarationen, wobei sieben Arten von Deklarationen zulässig sind:
  - Attributdeklarationen
  - Konstruktordeklarationen
  - Methodendeklarationen

  sowie die Deklaration
  - innerer Klassen
  - innerer Interfaces
  - statischer Initialisierungsblöcke
  - so genannter instance initializer

- wozu dienen Attributdeklarationen?
  - Objekte, die nach dem Bauplan einer Klasse „gebaut“ werden, bestehen in der Regel aus „Einzelteilen“
  - diese Einzelteile werden Attribute (fields) genannt
    - durch die Werte der Attribute wird der aktuelle Zustand der Objekte beschrieben
  - die Attributdeklarationen sind der Teil des Bauplans, in dem festgelegt wird, welche Attribute es gibt und von welcher Art sie sind
    - sie legen also die Gestalt der Objekte fest
  - eine Klasse kann beliebig viele Attribute haben, die beliebig komplex sein können

- wozu dienen Konstruktordeklarationen?
  - wann immer ein Objekt nach dem Bauplan einer Klasse „gebaut“ wird, wird ein Konstruktor benutzt
  - der Konstruktor ist dazu da, das Objekt in einen sinnvollen Initialzustand zu bringen (und nur dazu!)
    - vergleichbar der Werkseinstellung von Geräten
  - die Konstruktordeklarationen sind der Teil des Bauplans, in dem festgelegt wird, welche Konstruktoren es gibt und was sie leisten
  - eine Klasse kann mehrere Konstruktoren haben
    - wird keiner deklariert, erzeugt der Compiler automatisch einen Default-Konstruktor

- wozu dienen Methodendeklarationen?
  - Objekte, die nach dem Bauplan einer Klasse „gebaut“ werden, sind deswegen wichtig, weil sie über Fähigkeiten verfügen, die sie zur Nutzung zur Verfügung stellen
  - jede derartige Fähigkeit wird durch eine Methode beschrieben
  - die Methodendeklarationen sind der Teil des Bauplans, in dem festgelegt wird, welche Methoden es gibt und was sie leisten
    - sie legen also das Verhalten der Objekte fest
  - eine Klasse kann beliebig viele Methoden haben

konkret sieht das zum Beispiel so aus:

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class MeinDatum
{
  private int tag;                    <- drei Attributdeklarationen
  private int monat;
  private int jahr;

  public MeinDatum()                  <- eine Konstruktordeklaration
  {
    GregorianCalendar kalender = new GregorianCalendar();
    tag = kalender.get(Calendar.DAY_OF_MONTH);
    monat = kalender.get(Calendar.MONTH) + 1;
    jahr = kalender.get(Calendar.YEAR);
  }

  public String liefereDatum()        <- eine Methodendeklaration
  {
    return tag + ". " + monat + ". " + jahr;
  }
}
```


- im Beispiel haben wir zuerst alle Attribute deklariert, dann den Konstruktor und zuletzt die Methoden
- wir empfehlen, diese Reihenfolge immer einzuhalten:
  - erst alle Attributdeklarationen
  - dann alle Konstruktordeklarationen
  - dann alle Methodendeklarationen
- aus Sicht von Java wäre das nicht nötig
  - die Deklarationen könnten beliebig gemischt werden

- wie Attribut-, Konstruktor- und Methodendeklarationen genau auszusehen haben, müssen wir noch ausführlich studieren
- unseren Überblick beschließen wir mit Bemerkungen zur Kommentierung von Java-Programmen
- alle Klassen und Interfaces der Java-Klassenbibliothek sind in vorbildlicher Weise dokumentiert
  - als Beispiel sollten Sie sich die Dokumentation der Klasse `java.util.GregorianCalendar` ansehen

siehe: http://download.oracle.com/javase/6/docs/api/java/util/GregorianCalendar.html

- die Programmierer der Klassenbibliothek benutzen dafür das Werkzeug **javadoc**
  - es wird bei der Java-Installation in der Regel automatisch mit installiert
- **javadoc** arbeitet ähnlich wie das aus der C/C++-Programmierung bekannte **Doxygen**
  - das Programm war das Vorbild für **Doxygen**
- in die Deklaration von Klassen (und Interfaces) sind Dokumentationskommentare einzufügen
  - sie müssen immer direkt vor den Dingen stehen, die kommentiert werden, d.h.
    - unmittelbar vor der Klassendeklaration

    - unmittelbar vor jeder Attributdeklaration
    - unmittelbar vor jeder Konstruktordeklaration
    - unmittelbar vor jeder Methodendeklaration
- **javadoc** erzeugt aus ihnen die externe Dokumentation der Klasse oder des Interfaces in Form einer HTML-Datei mit standardisierter Struktur
  - das Ergebnis sieht genau so aus, wie die Dokumentation der Klassen und Interfaces der Java-Klassenbibliothek
- dazu ist (bei uns) in der Shell folgender Befehl einzugeben:

```
javadoc -private *.java
```


- zur Illustration haben wir die Klasse **MeinDatum** mit Dokumentationskommentaren versehen

```
import java.util.Calendar;
import java.util.GregorianCalendar;

/**
    Repraesentiert ein Datum, das aus Tag (1 - 31),
    Monat (1 – 12) und Jahr besteht (ganze Zahl).
    @author H. Brandenburg
 */
public class MeinDatum
{
  /** Der Tag des Datums. */
  private int tag;

  /** Der Monat des Datums. */
  private int monat;

  /** Das Jahr des Datums. */
  private int jahr;
```



```
  /**
     Erzeugt das aktuelle Datum, d.h. das aktuell im
     Rechner vorhandene Datum.
   */
  public MeinDatum()
  {
    GregorianCalendar kalender = new GregorianCalendar();
    tag = kalender.get(Calendar.DAY_OF_MONTH);
    monat = kalender.get(Calendar.MONTH) + 1;
    jahr = kalender.get(Calendar.YEAR);
  }

  /**
     Liefert das Datum in der Form <code>tt. mm. jjjj</code>.

     @return Das Datum in der Form <code>tt. mm. jjjj</code>.
   */
  public String liefereDatum()
  {
    return tag + ". " + monat + ". " + jahr;
  }
}
```

- wir wollen lernen, Klassen und Interfaces genauso gut zu kommentieren, wie die Programmierer der Java-Klassenbibliothek
- über die Details im Umgang mit **javadoc** sollten Sie sich an Hand der in der Literaturliste aufgeführten Bücher informieren
- eine ausführliche Anleitung, wie man gute Dokumentationskommentare verfasst, finden Sie hier:

How to Write Doc Comments

for the Javadoc Tool

http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html