Informatik B - Objektorientierte Programmierung in Java

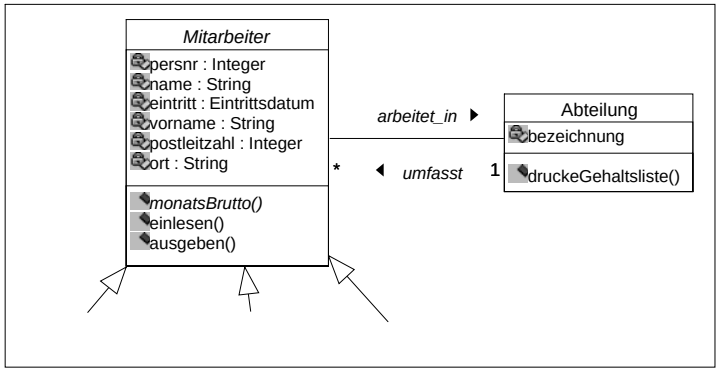
Vorlesung 05:
Assoziationen und Pakete

Inhalt

- Assoziationen
  - und ihre Umsetzung in Java
- Pakete

Assoziationen

- Beispiel:
  - Ein Mitarbeiter in einem Unternehmen gehört zu einer Abteilung.



Assoziationen

- Die Assoziation „arbeitet_in“ hat folgende Bedeutung:
  - Ein Mitarbeiter arbeitet immer genau in einer Abteilung.
  - In einer Abteilung können beliebig viele Mitarbeiter (auch keine) arbeiten.
- Die folgenden Programmstücke zeigen eine einfache Umsetzung der Assoziation „arbeitet_in“ in Java:

Assoziationen

```
class Mitarbeiter
{
   private Abteilung arbeitetIn;
   ...
   public void link(Abteilung abt)
   {
     arbeitetIn = abt;
   }
   public void unlink()
   {
     arbeitetIn = null;
   }
   public Abteilung getlink()
   {
       return arbeitetIn;
   }
}
```


Assoziationen

- Die Klasse Mitarbeiter benötigt ein Attribut vom Typ Abteilung.
  - Damit wird jedem Objekt der Klasse Mitarbeiter genau ein Objekt der Klasse Abteilung zugeordnet.
- Die Klasse Mitarbeiter erhält Operationen, um genau diese Zuordnung durchzuführen:
  - **`link`**: Das Attribut **`arbeitetIn`** vom Typ Abteilung wird auf das als Parameter übergebene Objekt **`abt`** gesetzt.
  - **`unlink`**: Hier wird diese Verbindung wieder gelöscht.
  - **`getlink`**: Liefert die gerade aktuelle Verbindung.

Assoziationen

- Da in einer Abteilung mehrere Mitarbeiter arbeiten können, müssen in einem Objekt der Klasse Abteilung mehrere Verbindungen zu Objekten der Klasse Mitarbeiter verwaltet werden.
- Als Datenstruktur für solche Fälle könnte ein Array geeignet sein.
- Da aber durch das Löschen und Hinzufügen Lücken in einem Array entstehen können, deren Verwaltung aufwendiger zu programmieren ist, ist die Datenstruktur **`Vector`** besser geeignet.

Assoziationen

- Ein **`Vector`** ist ein abstrakter Datentyp, der eine Liste von Referenzen verwalten kann.
  - Bildlich ist ein **`Vector`** ein „Array ohne Lücken“.
- Ein Objekt der Klasse **`Vector`** besitzt u.a. folgende Operationen:
  - **`void addElement(Object o)`** fügt Element hinzu
  - **`boolean removeElement(Object o)`** löscht Element
  - **`Object elementAt(int index)`** liefert Element
  - **`int size()`** liefert die Anzahl der Elemente im Vector
- Der unten verwendete **`Vector`** verwaltet also Objekte der Klasse Mitarbeiter. Er setzt die Assoziation "umfasst" in Java um.

Assoziationen

```
class Abteilung
{
   protected Vector ma;     // verwaltet eine Liste
                            // von Referenzen
   String bezeichnung;

   Abteilung(String bezeichnung)
   {
       ma = new Vector(50);
       this.bezeichnung = bezeichnung
   }

   int anzahlMitarbeiter()
   {
       return ma.size();
   } // Bitte umblättern
```

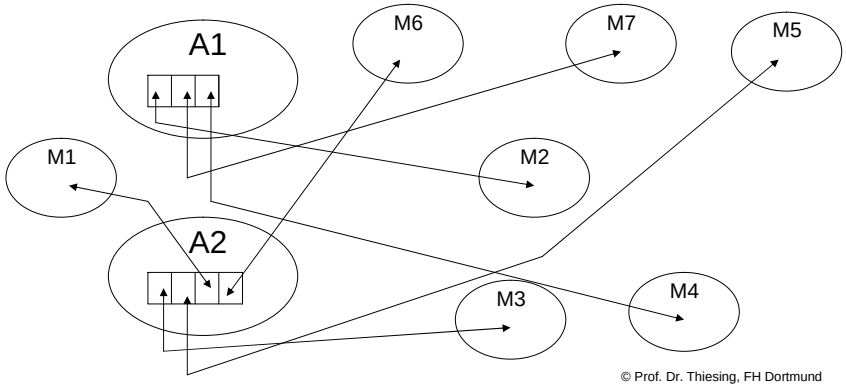

Assoziationen

```
   public void link(Mitarbeiter mit)
   {
       ma.addElement(mit);
   }
   public void unlink(Mitarbeiter mit)
   {
       ma.removeElement(mit);
   }
   Mitarbeiter getlink(int pos)
   {
       Mitarbeiter mit;

       mit = (Mitarbeiter) ma.elementAt(pos);
       return mit;
   } ...
}
```


Assoziationen

- Schematisch könnte die Verwaltung von Mitarbeiter-Objekten in Abteilungs-Objekten folgendermaßen aussehen:



Inhalt

- Pakete
  - Verwendung von Paketen
  - Erstellen eigener Pakete

Verwendung von Paketen

- Jede Klasse in Java ist Bestandteil eines Pakets.
- Der vollständige Name einer Klasse besteht aus dem Namen des Pakets, gefolgt von einem Punkt, der vom eigentlichen Namen der Klasse gefolgt wird. Der Name des Pakets selbst kann ebenfalls einen oder mehrere Punkte enthalten.
- Um eine Klasse verwenden zu können, muss angegeben werden, in welchem Paket sie liegt.

Verwendung von Paketen

- Hierzu gibt es zwei unterschiedliche Möglichkeiten:
  - Die Klasse wird über ihren vollen (qualifizierten) Namen angesprochen:
    - `java.util.Date d = new java.util.Date();`
  - Am Anfang des Programms werden mit Hilfe einer **`import`**-Anweisung die Namen der gewünschten Klassen vereinbart, damit diese unqualifiziert verwendet werden können:
    - ```
      import java.util.Date;
      ...
      Date d = new Date();
      ```


Verwendung von Paketen

- Die Verwendung voll qualifizierter Namen hat den Nachteil, dass die Klassennamen sehr lang und unhandlich werden.
- Bequemer ist daher die Anwendung der zweiten Variante, bei der die benötigten Klassen mit Hilfe einer **`import`**-Anweisung dem Compiler bekannt gemacht werden.

Verwendung von Paketen

- Die **`import`**-Anweisung gibt es in zwei unterschiedlichen Ausprägungen:
  - Mit der Syntax **`import paket.Klasse;`** wird genau eine Klasse importiert:
    ```
    import java.util.Date;
    …
    Date d = new Date();
    java.util.Vector = new java.util.Vector();
    ```
  - Unter Verwendung der Syntax **`import paket.*;`** können alle Klassen des angegebenen Pakets auf einmal importiert werden:
    ```
    import java.util.*;
    …
    Date d = new Date();
    Vector v = new Vector();
    ```
    ```

Verwendung von Paketen

- Im Gegensatz zu ähnlichen Konstrukten in anderen Sprachen ist die Verwendung der zweiten Variante der **`import`**-Anweisung nicht zwangsläufig wesentlich ineffizienter als die der ersten.
- In der Sprachspezifikation wird sie als *type import on demand* bezeichnet, was bedeutet, dass die Klasse erst dann in den angegebenen Paketen gesucht wird, wenn das Programm sie wirklich benötigt.

Verwendung von Paketen

- Keinesfalls muss der Compiler beim Analysieren der **`import`**-Anweisung zwangsläufig alle Klassendateien des angegebenen Pakets in den Hauptspeicher laden oder ähnlich zeit- und speicheraufwendige Dinge machen.
- Es ist also nicht ineffizienter, die zweite Variante (mit **`*`**) der **`import`**-Anweisung zu verwenden.

Verwendung von Paketen

- Vorsicht bei der **`import`**-Variante mit **`*`**:
  - Gleiche Klassennamen in 2 importierten Paketen führen zu einem Compilerfehler.
  - Es kann sein, dass Programme nach einem Javaversionswechsel oder einer Neustrukturierung fremder Pakete nicht ohne weiteres compiliert/gestartet werden können, da z.B. eine Klasse in einem Paket hinzukommen kann, die schon mit gleichem Namen in einem anderen Paket existiert. Hat man vor dem Versionswechsel das Programm erstellt, läuft es nach dem Hinzufügen der Klasse allerdings nicht mehr, da diese durch import xyz.* mit importiert wird.
- Besser: Klassen explizit importierten (ohne **`*`**)
  - dadurch bleibt das Programm compilierbar, auch wenn Klassen in importierten Paketen hinzukommen.

Verwendung von Paketen

- Der Import von **`java.lang`**
  - In vielen verschiedenen Beispielen haben wir Klassennamen (wie beispielsweise **`String`** oder **`Object`**) verwendet, ohne dass eine zugehörige **`import`**-Anweisung zu erkennen wäre.
  - In diesem Fall entstammen die Klassen dem Paket **`java.lang`**.
  - Dieses Paket wurde von den Entwicklern der Sprache als so wichtig angesehen, dass es von jeder Java-Datei automatisch importiert wird.
  - Am Anfang jeder Quelldatei steht demnach implizit die folgende Anweisung: **`import java.lang.*;`**

Verwendung von Paketen

- Ein expliziter Import von **`java.lang`** ist daher niemals nötig.
- Alle anderen Klassennamen müssen jedoch vor ihrer Verwendung importiert werden, wenn auf die Anwendung voll qualifizierter Klassennamen verzichtet werden soll.
- Es müssen nur Klassen fremder Pakete importiert werden.
- Paketnamen sollten stets durchgängig klein geschrieben werden.

Verwendung von Paketen

- In der Version 1.4 der Java 2 Standard Edition ist die Zahl der mitgelieferten Pakete auf über 100 gestiegen, was ein Blick in die Online-Dokumentation zeigt.

Verwendung von Paketen

- Die Bedeutung der Paketnamen
  - Wie bereits mehrfach zu sehen war, bestehen Paketnamen in Java aus mehreren Komponenten, die jeweils durch einen Punkt voneinander getrennt sind.
  - Neben der Aufgabe, die Paketnamen visuell zu strukturieren, hat die Unterteilung aber noch eine andere, sehr viel wichtigere Bedeutung:

Verwendung von Paketen

- Paketnamen entsprechen Unterverzeichnisse
  - Jeder Teil eines mehrstufigen Paketnamens bezeichnet ein Unterverzeichnis auf dem Weg zu der gewünschten Klassendatei.
  - Soll beispielsweise das Paket **`java.awt.image`** eingebunden werden, sucht der Java-Compiler die Klassen des Pakets im Unterverzeichnis **`java/awt/image`**.
  - Interessant ist in diesem Zusammenhang natürlich die Frage, in welchem Verzeichnis der Compiler mit der Suche nach vollqualifizierten Klassennamen beginnt:
    - Dies ist durch die Systemeigenschaft **`sun.boot.class.path`** festgelegt, die bei der Installation gesetzt wird.
    - Für benutzerspezifische Klassen kann die Umgebungsvariable **`CLASSPATH`** benutzt werden.
    - Auch möglich: javac –cp <Classpath> …, java –cp <Classpath> …

Verwendung von Paketen

- Die Datei **`rt.jar`**
  - Im Java-Installationsverzeichnis befindet sich unter **`jre/lib`** eine Datei **`rt.jar`** anstelle der erwähnten Unterverzeichnisse mit den Klassendateien.
  - Aus Gründen der Performance beim Übersetzen haben sich die Entwickler entschlossen, alle Standardklassendateien in diesem Archiv abzulegen, um einen schnelleren Lesezugriff zu erhalten.
  - Die Datei **`rt.jar`** sollte keinesfalls ausgepackt werden, denn der Compiler kann sie in archivierter Form verwenden.

Verwendung von Paketen

- Die Archivierung bietet den Vorteil, mehrere unterschiedliche Dateien in einer einzigen großen zusammenzufassen. Dadurch entfallen zeitaufwendige Verzeichniszugriffe, wenn der Compiler nach Klassennamen suchen muss oder den Inhalt einer Klassendatei laden will.

Verwendung von Paketen

- Umgekehrte Domain-Namen
  - Die Entwickler von Java haben sich einen Mechanismus ausgedacht, um auch bei sehr großen Projekten, an denen beispielsweise sehr viele Entwickler beteiligt sind, mögliche Namenskollisionen zwischen den beteiligten Klassen und Paketen zu vermeiden.
  - Auch die Verwendung einer großen Anzahl unterschiedlicher Klassenbibliotheken von verschiedenen Herstellern sollte möglich sein, ohne dass Namensüberschneidungen dies verhindern.

Verwendung von Paketen

- Um diese Probleme zu lösen, hat sich das Java-Team eine Vorgehensweise zur Vergabe von Paketnamen überlegt, die an das Domain-Namen-System bei Internet-Adressen angelehnt ist.
- Danach sollte jeder Anbieter seine Pakete entsprechend dem eigenen Domain-Namen benennen, dabei allerdings die Namensbestandteile in umgekehrter Reihenfolge verwenden.
- Einzige Ausnahme zu diesem Namensschema sind die Pakete, die im Standardumfang einer Java-Installation enthalten sind, z.B.: **`java.*, javax.*`**

Verwendung von Paketen

- So sollten beispielsweise die Klassen der Firma Sun, deren Domain-Name **`sun.com`** ist, in einem Paket **`com.sun`** oder in darunter befindlichen Subpaketen liegen.
- Da die Domain-Namen weltweit eindeutig sind, werden Namenskollisionen zwischen Paketen unterschiedlicher Hersteller auf diese Weise von vornherein vermieden.
- Unterhalb des Basispakets können Unterpakete beliebig geschachtelt werden. Die Namensvergabe liegt dabei in der Entscheidung des Unternehmens.

Verwendung von Paketen

- Gibt es beispielsweise die Abteilungen **`FE, KIS`** und **`RIS`** in einem Unternehmen mit der Domain **`prompt.de`**, kann es sinnvoll sein, diese Abteilungsnamen auch in den Paketnamen zu verwenden.
- Die von diesen Abteilungen erstellten Klassen würden dann in den Paketen **`de.prompt.fe`**, **`de.prompt.kis`** und **`de.prompt.ris`** liegen.
- Der Vollständigkeit halber sollte man anmerken, dass das hier beschriebene System sich in der Praxis noch nicht vollständig durchgesetzt hat und insbesondere die Klassen der **`java.`**-Hierarchie eine Ausnahme bilden. Es wird mit der zunehmenden Anzahl allgemein verfügbarer Pakete jedoch an Bedeutung gewinnen.

Erstellen eigener Pakete

- Bisher wurde nur gezeigt, wie Klassen aus fremden Paketen verwendet werden können, aber nicht, wie Pakete erstellt werden.
- Um eine Klasse einem ganz bestimmten Paket zuzuordnen:
  - Muss am Anfang des Quelltextes eine geeignete **`package`**-Anweisung verwendet werden.
  - Diese besteht (analog zur **`import`**-Anweisung), aus dem Schlüsselwort **`package`** und dem Namen des Pakets, dem die nachfolgenden Klassen zugeordnet werden sollen.
  - Die **`package`**-Anweisung muss als erste Anweisung in einer Quelldatei stehen, so dass der Compiler sie noch vor den **`import`**-Anweisungen findet.

Erstellen eigener Pakete

- Der Aufbau und die Bedeutung der Paketnamen in der **`package`**-Anweisung entspricht exakt dem der **`import`**-Anweisung.
- Der Compiler löst ebenso wie beim **`import`** den dort angegebenen hierarchischen Namen in eine Kette von Unterverzeichnissen auf, an deren Ende die Quelldatei angefügt wird.
- Neben der Quelldatei wird auch die Klassendatei in diesem Unterverzeichnis erstellt.

Erstellen eigener Pakete

- Beispiel:
  - Es werden im folgenden Beispiel zwei Pakete **demo** und **demo.tools** angelegt und die darin enthaltenen Klassen **A** und **B** bzw. **C** in einer Klasse **Example** verwendet.
  - Im aktuellen Verzeichnis werden die Unterverzeichnisse **demo** und **demo/tools** angelegt.

Erstellen eigener Pakete

- Zunächst wird im Unterverzeichnis **demo** die Datei **A.java** angelegt:

```
package demo;
public class A
{
   public void hello()
   {
      System.out.println("Hier ist A");
   }
}
```

  - Sie enthält die Anweisung **package demo**, um anzuzeigen, dass die Klasse **A** zum Paket **demo** gehört.

Erstellen eigener Pakete

- Im Unterverzeichnis **demo** wird weiterhin die Datei **B.java** angelegt:

```
package demo;

public class B
{
   public void hello()
   {
      System.out.println("Hier ist B");
   }
}
```

  - Auch diese Quelldatei enthält die Anweisung **package demo**, um anzuzeigen, dass die Klasse zum Paket **demo** gehört.

Erstellen eigener Pakete

- Im Unterverzeichnis **demo/tools** wird die Datei **C.java** angelegt:

```
package demo.tools;

public class C
{
   public void hello()
   {
      System.out.println("Hier ist C");
   }
}
```

  - Diese Quelldatei enthält die Anweisung **package demo.tools**, um anzuzeigen, dass die Klasse zum Paket **demo.tools** gehört.

Erstellen eigener Pakete

- Nun wird im Stammverzeichnis die Datei **`Example.java`** angelegt:

```
import demo.A;
import demo.B;
import demo.tools.C;

public class Example {
   public static void main(String[] args)
   {
      A a = new A();
      B b = new B();
      C c = new C();
      a.hello();
      b.hello();
      c.hello();
   }
}
```


Erstellen eigener Pakete

- Ohne vorher die Klassen **`A, B`** oder **`C`** separat übersetzen zu müssen, kann nun einfach **`Example.java`** kompiliert werden.
- Der Compiler erkennt die Verwendung von **`A, B`** und **`C`**, findet die Paketverzeichnisse **`demo`** und **`demo/tools`** und erkennt, dass die Quellen noch nicht übersetzt sind.
- Er erzeugt dann aus den **`.java`**-Dateien die zugehörigen **`.class`**-Dateien, bindet sie ein und übersetzt schließlich die Klasse **`Example`**.

Erstellen eigener Pakete

- Das Default-Paket
  - Würde nur dann ein eigenes Paket erzeugt werden, wenn die Quelldatei eine **`package`**-Anweisung enthält, müsste man sich fragen, zu welchem Paket die Klassen gehören, in deren Quelldatei die **`package`**-Anweisung fehlt (was ja erlaubt ist).
  - Die Antwort auf diese Frage liegt darin, dass es in Java ein Default-Paket gibt, das genau dann verwendet wird, wenn keine andere Zuordnung getroffen wurde.
  - Das Default-Paket ist ein Zugeständnis an kleinere Programme oder einfache Programmierprojekte, bei denen es sich nicht lohnt, eigene Pakete anzulegen.

Erstellen eigener Pakete

- Ohne Teile des Projektes in Unterverzeichnissen abzulegen und durch **`import`**- und **`package`**-Anweisungen Aufwand zu treiben, ist es auf diese Weise möglich, Quelldateien einfach im aktuellen Verzeichnis abzulegen, dort zu kompilieren und automatisch einzubinden. Klassen des Default-Pakets können ohne explizite **`import`**-Anweisung verwendet werden.
- Ein Java-Compiler braucht laut Spezifikation nur ein einziges Default-Paket zur Verfügung zu stellen. Typischerweise wird dieses Konzept aber so realisiert, dass jedes unterschiedliche Unterverzeichnis die Rolle eines Default-Paketes übernimmt. Auf diese Weise lassen sich beliebig viele Default-Pakete erzeugen, indem bei Bedarf einfach ein neues Unterverzeichnis angelegt wird und die Quelldateien eines Java-Projektes dort abgelegt werden.

Erstellen eigener Pakete

- Der **`public`**-Modifikator
  - Es gibt noch eine wichtige Besonderheit bei der Deklaration von Klassen, die von anderen Klassen verwendet werden sollen. Damit eine Klasse **`A`** eine andere Klasse **`B`** einbinden darf, muss eine der beiden folgenden Bedingungen erfüllt sein:
    - entweder gehören **`A`** und **`B`** zu demselben Paket oder
    - die Klasse **`B`** wurde als **`public`** deklariert und befindet sich nicht im Default-Paket.

Erstellen eigener Pakete

- Wenn nur das Default-Paket verwendet wird, spielt es keine Rolle, ob eine Klasse vom Typ **`public`** ist, denn alle Klassen liegen in demselben Paket. Werden aber Klassen aus externen Paketen eingebunden, so gelingt dies nur, wenn die einzubindende Klasse vom Typ **`public`** ist. Andernfalls verweigert der Compiler deren Einbindung und bricht die Übersetzung mit einer Fehlermeldung ab.