Grundlagen der Informatik für Ingenieure I

9. Programming in the Large

9 Programming in the Large

◆ ***“Programming in the Large” und “Programming in the Small”***

- ***Programming in the Large:***
  - Design eines Programmsystems
  - Konzepte
  - Globale Programm- und Datenstrukturen
  - Hierachische Organisation der Programm- und Datenstrukturen
- ***Programming in the Small:***
  - eigentliche Implementierung einzelner Aspekte.
- ***Programming in the Large*** wird in herkömmlichen Programmiersprachen kaum unterstützt;
- Objektorientierte Sprachen stellen dafür jedoch geeignete Mechanismen bzw. Konstrukte bereit; so auch Java.

9 Programming in the Large

- Bisher haben wir uns mit dem ***Programming in the Small*** beschäftigt.
- Sind die übergeordneten Mechanismen, Konstruktionen und Vorgehensweisen bekannt sind, fällt es leichter, das *Programming in the Small* durchzuführen.
- Dieses insbesondere mit Hilfe der objektorientierten Programmierung.
- Zum Entwurf komplexerer Softwaresysteme werden zunehmend Werkzeuge eingesetzt,
- Hier haben insbesondere Werkzeuge auf der Basis von **UML** (Unified Modeling Language) Verbreitung gefunden.

9.1 Abstrakte Klassen

- Beim Design von Klassenhierachien haben höhere Klassen in der Regel eher einen abstrakten Charakter.
  - Es ist nicht die Intention, aus diesen Klassen Objekte zu erzeugen.
  - Sie stellen globale Informationen zur Verfügung.
  - Es werden strukturelle Vorgaben formuliert, die durch die Vererbungsmechanismen die Gesamtstruktur eines Systems festlegen.
  - Die Implementierung findet in Klassen niedrigerer Ebenen statt.

9.1 Abstrakte Klassen

◆ Um diese Intention deutlich zu machen stellt Java den Modifier ***abstract*** bereit:

```
public abstract class VWCars {
...
}
```

- Abstrakte Klassen können **grundsätzlich** alles das enthalten, was “normale” Klassen auch enthalten können.

9.2 Abstrakte Methoden

◆ Abstrakte Methoden sind Signaturen ohne deren Rumpf (Implementation)

- Will man diese Methoden anwenden, so muss in einer abgeleiteten Klasse eine Implementierung vorgenommen worden sein (Overriding).

- Beispiel:

```
public abstract doControll();
```

◆ Abstrakte Klassen enthalten häufig auch abstrakte Methoden.

◆ In *Interfaces* (Abschnitt 9.4) können **nur** abstrakte Methoden definiert werden.

9.3 “Finale” Methoden

- Während man mit der Definition von *abstrakten* Methoden verbindlich für das Programmsystem das Aussehen von Klassen-/Objekt-Schnittstellen definiert (Implementation durch Overriding), kann man auch **verbindlich** Methoden mit Implementation vorgeben, die nicht überschreibbar (Overriding) sind.
  - Hierfür verwendet man den Modifier *final*
    - Beispiel

```
final float mostImportend(float abc){
    ....
    der methodenrumpf
    ...
}
```


9.4 Interfaces

- ***Interfaces*** enthalten ausschließlich abstrakte Methoden.
- ***Interfaces*** werden immer dann eingesetzt, wenn die strenge Klassenhierachie von Java (Single Inheritance) den gewünschten Designstrukturen nicht entspricht, bzw. diese nur sehr umständlich zu realisieren sind.
- ***Interfaces*** sind nicht Teil der Klassenhierachie.
- ***Interfaces*** unterliegen nicht dem Vererbungs-Mechanismus.
- ***Interfaces*** stellen keine konkreten Implementierungen bereit.
- ***Interfaces*** können überall dort “eingebracht” werden, wo sie benötigt werden.

9.4 Interfaces

- Klassen können mehrere ***Interfaces*** implementieren - quasi Multiple Inheritance
- ***Interfaces*** sind eine Sammlung von Methoden-Signaturen (abstrakte Methoden)
- ***Interfaces*** können keine Variablen- sondern nur Konstantenvereinbarungen enthalten (*static final*).
- Benutzt man ein ***Interface***, so müssen **sämtliche** Methoden des *Interfaces* implementiert werden; man kann nicht einfach einige ignorieren, wie es bei abstrakten Klassen möglich ist.

9.4 Interfaces

- Die Benutzung eines Interfaces drückt man mit dem *Keyword "implements"* in der Klassendefinition aus:

```
public class Neko extends java.applet.Applet
    implements Runnabel {
    .....
    }
```

  - Das *Interface "Runnable"* gibt z. B. die Strukturen für die Handhabung mehrerer Aktivitätsträger (Threads) in einem Applet oder in einer Applikation vor.
  - Dies erfolgt durch die Definition von Signaturen wie *start(), stop(), run(),* etc.
  - Näheres hierzu in Kapitel 11.

9.4 Interfaces

◆ Die Subklassen der Klasse, die ein *Interface* benutzt, erben die (implementierten) Methoden in gewohnter Weise; d. h. sie können z. B. überschrieben werden.

◆ Wie bereits angedeutet können auch mehrere *Interfaces* benutzt werden:

```
public class AnotherNeko extends java.applet.Applet
    implements Runnabel, Plausibel, Obskur {
    .....
}
```


9.4 Interfaces

◆ Kreieren eines *Interfaces:*

```
public interface Konto {
    public static final float LIMIT = 1000;

    public abstract void einzahlung( float betrag );
    public abstract void auszahlung( float betrag );
    ...
}
```

- **In Interfaces können nur Konstanten vereinbart werden.**
- **In Interfaces können nur abstakte Methoden vereinbart werden.**

◆ Mit *Interfaces* kann man auch - analog zu den Klassen - Hierachien mit *“extends”* aufbauen:

```
public interface VWCars extends
    Lupo, Polo, Golf, Passat {
....
}
```

◆ **Im Rahmen dieser Vorlesung werden *Interfaces* nur benutzt!**

9.5 Packages

◆ Mit dem ***Package***-Konzept hat man die Möglichkeit, logisch zusammenhängende Klassensysteme zusammenzufassen:

- Durch die Zusammenfassung wird die Handhabung der Softwarepakete - sei es im Programmierumfeld oder in der Verwaltung - einfacher, analog dem ***Directory***-Konzept eines Dateisystems.
- Durch die Möglichkeit der “Kapselung” des Namensraums vermeidet man Namenskonflikte, die sonst bei größeren Systemen unvermeidbar sind.
- Durch ***Packages*** wird eine weitere Hülle des Zugriffsschutzes implementiert.

9.6 Packages

- Da ***Packages*** wiederum zu *Packages* zusammengefasst werden können, ergibt sich eine hierachische Organisation.
- Man hat die Möglichkeit - weltweit - eindeutige Namen “seinem” ***Package*** zu geben. Es ist Konvention, in der obersten Hierachiestufe den Firmennamen als Prefix zu verwenden.

9.6 Packages

◆ Benutzung von *Packages:*

- Klassen des ***Package "java.lang"*** sind immer "importiert"; man referenziert sie durch den Klassennamen.
- Die Klassen des "eigenen" Programmsystems werden automatisch zu einem ***default package*** (ohne Namen) zusammengefasst. Man referenziert auch hier die Klassen durch Angabe des Klassennamens.

9.6 Packages

- Alle anderen Klasse können auf zwei Wegen referenziert werden:
  - mit vollem Namen durch die *package-Hierachie* bis hinunter zur Klasse (analog Pfadnamen im Unix-Dateisystem) z. B.

    ```
    java.awt.Font myfont = new java.awt.Font();
    ```

    Nur bei seltenen oder einmaligen Referenzen sinnvoll!

  - oder mit Hilfe des ***import commands:***

    ```
    import util.Vector;
    import java.awt.*;
    ```

    - Der * - Operator sagt nur: alle Klassen von ...;
    - Es handelt sich nicht um einen "*wildcard*"-Operator wie wir ihn z. B. bei der Angabe von Dateinamen von einigen Kommandos her kennen.
    - Es werden keine Hierachien von *Packages* importiert; jedes *Package* muß explizit importiert werden.

9.6 Packages

◆ Wie findet der Compiler die zu importierenden *Packages?*

- *Package* Namen korrespondieren mit *Directory* - Namen.
- Begonnen wird mit der Suche in der *Directory,* die in der Environmentvariablen CLASSPATH vermerkt ist.

9.6 Packages

◆ Kreieren eines *Packages:*

- Festlegung des *Package* - Namen.
- Kreieren einer *Directory*-Struktur analog zu den *Package* - Namen.
- Mit dem *package statement* wird festgelegt zu welchem *Package* eine Klasse gehören soll.
  - Es muss das erste Statement im *Sourcefile* sein.
  - Fehlt das *package statement* wird die Klasse dem *default package* zugeordnet:

```
package myclasses;
package schrank.fach.schublade;
```

9.6 Packages

◆ Zugriffsschutz:

Visibility	public	protected	package	private
From the same class	yes	yes	yes	yes
From any class in the same package	yes	yes	yes	no
From any class outside the package	yes	no	no	no
From a subclass in the same package	yes	yes	yes	no
From a subclass outside the same package	yes	yes	no	no