KLASSEN ENTWERFEN

Christian Schirmer

Klassen entwerfen

- Import und Packages
- Überladen von Methoden
- Initialisierungsblöcke
- Kapselung und Sichtbarkeiten
- JavaBeans

IMPORT UND PACKAGES

Wo gibt es was?
Wo bin ich?

Packages

- Pakete helfen uns Projekte zu Strukturieren.
- Wir können Klassen gleicher Namen in unterschiedlichen Paketen haben.
- Pakete sollten Thematisch zusammengehöriger Typen darstellen.
  - *Beispiel*
    - java.awt – Enthält alle Komponenten die zum Abstraction Window Toolkit gehören
    - java.awt.event – Enthält Eventtype zugehörig zu den AWT
    - javax.swing – Enthält somit alles was zu Swing gehört
      - *Warum heißt Swing eigentlich Swing und nicht KFC?* *https://blogs.oracle.com/thejavatutorials/why-is-swing-called-swing*
    - Oder auf gerne im Sinne vom MVC-Pattern
      - *model – Alle Datenklassen*
      - *view – Hier gehören die GUI–Komponenten hin*
      - *control – Alles was Programmlogik (Business-Logik) entspricht findet sich hier*

Imports

- Ein Nebeneffekt der Paketierung ist der, dass Klassen nur auf Klassen im selben Paket zugreifen können. Wenn auf Klassen außerhalb des eigenen Paketes zugegriffen werden soll, müssen diese Importiert werden, dies gilt auch für Klassen in sogenannten SubPackages.
- Über Import können **Statische** Methoden einer anderen Klassen direkt angesprochen werden, so als wären diese in der Importierenden Klasse selbst implementiert.
- Das Paket **java.lang** stellt die Kernkomponenten von Java dar. Dieses Paket muss nicht importiert werden, damit auf Klassen von dort zugriffen werden kann.
- Pakete stellen im Dateisystem eine Ordner Struktur dar.

ÜBERLADEN

Überladen - Overload

- Überladene Methoden sind Methoden die sich in der Signatur ändern.
  - *Zur Signatur einer Methode gehören:*
    - Der Name der Methode
    - Der Rückgabetyp
    - Die übergebenen Parameter an die Methode
- Überladene Methoden müssen sich immer nach A – R – A Unterscheiden.
  - *Anzahl – Reihenfolge – Art der Parameter.*

- Konstruktoren werden besonders häufig Überladen, jedoch niemals Überschrieben (Override). – Konstruktoren werden nicht vererbt, daher kein Override.

Überladen - Overload

- Beim Überladen dürfen keine Uneindeutigkeiten enstehen

```
public void methodOne(String string) {
        doSomeThing();
}
```

- Die Zweite Methode stellt ein Uneindeutigkeit dar. Beim Ausführen wäre nicht eindeutig klar, welche Methode zum Einsatz kommen soll.

```
public String methodOne(String string) {
        doSomeThing();
}
```

INIT BLÖCKE

Initialisierungsblöcke

Initialisierungsblöcke

- Initialisierungsblöcke stellen eine weitere Methode dar, um Code während der Erzeugung eines Objekts oder beim ersten Benutzen einer Klasse auszuführen.

- Neben den Konstruktoren, welche Explizit aufgerufen werden müssen, werden die Initialisierungsblöcke implizit ausgeführt.

„By convention, init blocks usually appear near the top of the class file, somewhere around the constructors. However, this is the OCA exam we're talking about. Don't be surprised if you find an init block tucked in between a couple of methods, looking for all the world like a compiler error waiting to happen! “

Initialisierungsblöcke

Statische Initialisierungsblöcke

- Klassengebunden – static
- werden nur einmal aufgerufen und ausgeführt.
- werden beim ersten nutzen einer Klasse ausgeführt.
- haben zugriff auf ausschließlich Statische Attribute und Methoden

Nicht - Statische Initialisierungsblöcke

- Instanz gebunden – ohne static
- werden mehrmals verwendet
- werden jedes Mal, wenn eine Instanz erzeugt wird aufgerufen
- haben zugriff auf Statische wie nicht Statische Attribute und Methode

Initialisierungsblöcke

- Regeln zu den Init-blöcken.
  - *init-Blöcke werden in der Reihenfolge abgearbeitet, wie sie erstellt wurden.*
  - *Static init-blöcke, werden nur einmal ausgeführt, wenn die Klasse das erste mal geladen wird.*
  - *Instanz init-Blöcke werden jedes Mal ausgeführt, wenn eine Instanz erzeugt wird*
  - *Instanz init-Blöcke werden nach dem der Konstruktor den „super()*“ Aufruf durchgeführt hat, abgearbeitet.*
- Daraus abgeleitete Reihenfolge der Abarbeitung:

  Static Init-Blöcke -> „super()“ Vom Kontruktor -> Instanz init-Blöcke -> restlicher Quellcode vom Konstruktor.*

** super() ist der Aufruf einen Konstruktor der Elternklasse. Wenn wir nicht explizit einen anderen Aufruf angeben, steht super() implizit immer da. Auch wenn wir Garnichts angeben.*

Steuerung anfordern Originalgröße

clipse IDE

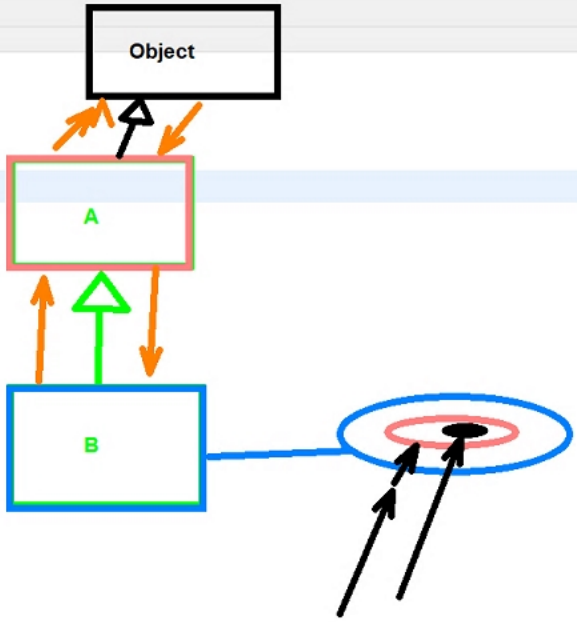
Run Window Help

Init_Blocks.java

```
        System.out.println( 2. A Init Block );
    }
}

class B extends A {
    {
        System.out.println("1. B Init Block");
    }
    public B() {
        System.out.println("B Konstruktor");
    }
}

public class Init_Blocks {
    public static void main(String[] args) {
        new B();
    }
}
```



Problems Javadoc Declaration Console Git Staging

minated> Init_Blocks [Java Application] C:\Java\AdoptJDKSE8\bin\javaw.exe (02.03.2021 09:29:01 – 09:29:01)

```
. A Init Block
. A Init Block
 Konstruktor
. B Init Block
 Konstruktor
```

DATENKAPSELUNG

Information Hiding

Datenkapselung – Information hiding

- Datenkapselung bedeutet, wir geben die Attribute(Eigenschaften) einer Klasse nach außen nicht direkt weiter.
- Weder ein direkter Lesender noch Schreibender zugriff auf die Attribute wird gewährt.
- Um die Attribute von außen Manipulierbar zu gestalten, werden sogenannte Getter und Setter Methoden verwenden.
- Über die Getter steuern wir, wie die Daten nach außen gegeben werden. Wir müssen dabei beachten, dass eventuell nur die Referenzen statt der Werte weitergeben werden. (Mit der Referenz ist es möglich, die Instanz zu manipulieren)
- Mir den Setter haben wir die Möglichkeit, die einkommenden Daten zu prüfen und eventuell abzulehnen. (Der Setter wird im Observer Pattern benötigt.)

ACCESS MODIFIERS

Kapselung und Sichtbarkeiten

PUBLIC

Public

- deklariert öffentliche Typen und Eigenschaften
- Typen sind überall sichtbar
- jede Klasse und Unterklasse aus beliebigem Paket kann auf öffentliche Eigenschaften zugreifen
- Mit public deklarierten Methoden und Variablen sind sichtbar, wo die Klasse sichtbar ist.
- Eigenschaften einer unsichtbaren Klasse sind unsichtbar

PRIVATE

Private

- Die Klasse, die den Dateinamen bestimmt, kann nicht privat sein
- Methoden und Variablen sind nur innerhalb der eigenen Klasse sichtbar
- Innere Klassen, können auch auf private Eigenschaften der äußeren Klasse zugreifen
- Wird eine Klasse erweitert, sind die privaten Elemente für Unterklassen nicht sichtbar

PAKETSICHTBAR

Paketsichtbar

- ist die Standard-Sichtbarkeit und kommt ohne Modifizierer aus
- Paketsichtbare Typen und Eigenschaften sind nur für die Klassen aus dem gleichen Paket sichtbar, also weder für Klassen noch für Unterklassen aus anderen Paketen

PROTECTED

Protected

- **protected** hat eine Doppelfunktion:
  - *Es hat er die gleiche Bedeutung wie Paketsichtbarkeit*
  - *Es gibt die Elemente für Unterklassen frei*
    - Dabei ist es egal, ob die Unterklassen aus dem eigenen Paket stammen

GETTER / SETTER

Getter / Setter

- In der OOP gibt es den Ansatz der Datenkapselung.
- Dies bedeutet auf die Attribute einer Klasse, darf nur die Klasse selbst direkt zugreifen.
- Andere Außenstehende Klassen und damit auch Objekte, müssen sich der entsprechenden Getter / Setter Methoden bedienen.
- Durch das Nutzen von Getter und Setter Methoden entsteht die Möglichkeit, nur vom Design der Klasse vorgesehen Werte in die Attribute schreiben zu lassen.

- Konsequenterweise müssen dann alle Attribute einer Klasse mit den Sichtbarkeitsmodifikator „private“ versehen werden.

POJO -> Java Bean
Plain old Java Object

Getter / Setter

- In der OOP gibt es den Ansatz der Datenkapselung.
- Dies bedeutet auf die Attribute einer Klasse, darf nur die Klasse selbst direkt zugreifen.
- Andere Außenstehende Klassen und damit auch Objekte, müssen sich der entsprechenden Getter / Setter Methoden bedienen.
- Durch das Nutzen von Getter und Setter Methoden entsteht die Möglichkeit, nur vom Design der Klasse vorgesehen Werte in die Attribute schreiben zu lassen.

- Konsequenterweise müssen dann alle Attribute einer Klasse mit den Sichtbarkeitsmodifikator „private“ versehen werden.

Getter / Setter

- Getter Methoden
  - *get wie holen, wird dem Attributnamen vorangestellt.*
  - *Liest den wert eines Attributes und gibt diesen zurück*

Aufbau: Beispiel eines int Attributes

```
public int get<Attributname>() {
   return this.attributname;
}
```

- Setter Methoden
  - *set wie festlegen, wird dem Attributnamen vorangestellt.*
  - *Schreibt einen wert in das Attribut und liefert nichts zurück.*

Aufbau: Beispiel eines int Attributes

```
public void set<Attributname>(int attr) {
   this.attributname = attr;
}
```