# Unterabschnitt 5.3.9

## Kapselung und Strukturieren von Klassen

# Kapselung und Strukturieren von Klassen

Zwei Aspekte zur weiteren Strukturierung objektorientierter Programme:

- Schnittstellenbildung und Kapselung
- Schachtelung von Klassen und Pakete

# Schnittstellenbildung und Kapselung

Objekte stellen eine bestimmte Funktionalität/Dienste zur Verfügung:

- Anwendersicht: Nachrichten schicken, Ergebnisse empfangen.
- Implementierungssicht: Realisierung der Zustände und Funktionalität durch
  - → objektlokale Attribute,
  - → Referenzen auf andere Objekte,
  - → Implementierung von Methoden.

**Ziel:**

- Lege die Anwendungsschnittstelle genau fest.
- Verhindere Zugriff ”von außen” auf Implementierungsteile, die nur für internen Gebrauch bestimmt sind.

# Begriffsklärung: (Anwendungsschnittstelle)

Die ***Anwendungsschnittstelle*** eines Objekts bzw. eines Referenztyps besteht aus

- den Nachrichten, die es für Anwender zur Verfügung stellt;
- den Attributen, die für den direkten Zugriff von Anwendern bestimmt sind.

**Bemerkung:**

- Die Festlegung von Anwendungsschnittstellen ist eine Entwurfsentscheidung.
- Direkter Zugriff auf Attribute muss nicht gewährt werden, sondern kann mit Nachrichten/Methoden realisiert werden.

# Beispiel:

```
class MitDirektemAttributZugriff {
    public int attr;
}

class AttributZugriffueberMethoden {
    private int attr;
    int getAttr() { return attr; }
    void setAttr( int a ) { attr = a; }
}
```

# Begriffsklärung: (Information Hiding)

Das Prinzip des ***Information Hiding*** (deutsch meist ***Geheimnisprinzip***) besagt, dass

- Anwendern nur die Informationen zur Verfügung stehen sollen, die zur Anwendungsschnittstelle gehören,
- alle anderen Informationen und Implementierungsdetails für ihn verborgen und möglichst nicht zugreifbar sind.

**Gründe für Information Hiding:**

- Vermeiden unsachgemäßer Anwendung
- Vereinfachen von Software durch Reduktion der Abhängigkeiten zwischen ihren Teilen
- Austausch von Implementierungsteilen

# Javas Sprachmittel für Information Hiding:

Java ermöglicht es, für Programmelemente sogenannte *Zugriffsbereiche* zu deklarieren.

Vereinfachend gesagt, kann ein Programmelement nur innerhalb seines Zugriffsbereichs verwendet werden.

Java unterscheidet vier Arten von Zugriffsbereichen:

- nur innerhalb der eigenen Klasse (Modifikator `private`)
- nur innerhalb der eigenen Pakets (ohne Modifikator)
- nur innerhalb des eigenen Pakets und in Subklassen (Modifikator `protected`)
- ohne Zugriffsbeschränkung (Modifikator `public`)

Generell können Klassen, Attribute, Methoden und Konstruktoren mit den Zugriffsmodifikatoren deklariert werden.

# Beispiel: (Vermeiden falscher Anwendung)

```
public class SLinkedList {
  private SEntry entries = null;
  private int size = 0;

  public int getFirst(){ ... }
  ...
}
```

Außerhalb der Klasse SLinkedList kann das Attribut size nicht mehr verändert werden, so dass inkonsistente Zustände, in denen size nicht die Anzahl der Elemente enthält, vermieden werden können.

# Beispiel: (Austausch von Implementierungen)

```
public class W3Seite {
   private String titel;
   private String inhalt;
   public W3Seite ( String t, String i ) {
      titel  = t;
      inhalt = i;
   }
   public String getTitel() { return titel; }
   public String getInhalt(){ return inhalt;}
}
```

# Beispiel: (Austausch von Implementierungen) (2)

Die obige Klasse kann ersetzt werden durch die folgende, ohne dass Anwender der Klasse davon tangiert werden:

```
public class W3Seite {
  private String seite;
  public W3Seite( String t, String i ) {
    seite =  "<TITLE>" + t + "</TITLE>" + i ;
  }
  public String getTitel(){
    int ix = seite.indexOf("</TITLE>")-7;
    return new String(seite.toCharArray(),7,ix);
  }
  public String getInhalt(){
    int ix = seite.indexOf("</TITLE>") + 8;
    return new String( seite.toCharArray(), ix,
                       seite.length() - ix  );
  }}
```

# Bemerkung:

- Information Hiding erlaubt insbesondere:
  - konsistente Namensänderungen in versteckten Implementierungsteilen.
  - Verändern versteckter Implementierungsteile, soweit sie keine Auswirkungen auf die öffentliche Funktionalität haben (kritisch).

    **Beispiel:**
    Die zweite Implementierung von W3Seite kann nur dann anstelle der ersten benutzt werden, wenn Titel die Zeichenreihe ”</TITLE>” nicht enthalten.

- Attribute sollten tendenziell privat sein und nur in Ausnahmefällen öffentlich.

# Begriffsklärung: (Kapselung)

Wir verstehen unter ***Kapselung*** eine verschärfte Form des Information Hiding, die es dem Anwender unmöglich macht, auf interne Eigenschaften gekapselter Objekte zuzugreifen.

**Bemerkung:**

- Die Verwendung von „private“ führt nicht automatisch zur Kapselung.
- Um Kapselung zu erreichen, bedarf es im Allg. eines Zusammenwirkens unterschiedlicher Sprachmittel und Programmiertechniken

# Beispiel: (zur Kapselung)

1. Der private Zugriffsbereich bezieht sich auf die Klasse und im Allg. nicht auf einzelne Objekte:

```
public class FamilienMitglied {
    private int a;
    private FamilienMitglied geschwister;

    public void incrA(){
        a++;
        geschwister.a++;
    }
}
```

Die Methode incrA modifiziert nicht nur das Objekt, auf dem sie aufgerufen wurde, sondern auch das private Attribut a des Geschwisterobjekts.

# Beispiel: (zur Kapselung) (2)

2. Wenn Objekte andere Objekte zur Realisierung ihrer Funktionalität nutzen, müssen sie die Kontrolle über diese Objekte behalten:

```
public class MyFaultyString {
  private char[] value;

  public MyFaultyString( String s ) {
    value = s.toCharArray();
  }
  //...
  public char[] toCharArray() {
    return value;
  }
}
```

# Beispiel: (zur Kapselung) (3)

```
public class  Main {
  public static void main( String[] args ) {

    MyFaultyString mfs =
      new MyFaultyString("Staatssicherheit");
    char[] inhalt = mfs.toCharArray();

    // ... spaeter im Programm:
    inhalt[3] = 't';
    inhalt[5] = ' ';
    inhalt[6] = 'S';

    // mfs noch unveraendert?
    System.out.println( mfs.toCharArray() );
  }
}
```

# Beispiel: (zur Kapselung) (4)

Es reicht also im Allg. nicht, die Attribute als privat zu deklarieren, um Kapselung zu erreichen.

Es muss auch verhindert werden, dass Referenzen auf interne Objekte herausgegeben werden.

# Schachtelung von Klassen und Pakete

Java-Quellprogramme bestehen aus ***Typdeklarationen***, d.h. Klassen- und Schnittstellendeklarationen (s.u.). Es gibt drei Sprachkonstrukte zur Strukturierung der Typdeklarationen:

- Schachtelung von Klassen
- Gruppierung von Klassen zu Übersetzungseinheiten
- Gruppierung von Übersetzungseinheiten zu Paketen

***Übersetzungseinheiten*** entsprechen einer Datei und können mehrere Typdeklarationen zusammenfassen, wobei maximal eine öffentlich sein darf.

# Übersetzungseinheiten, Programme

**Syntax:**

`package` <Paketname>;

<Liste von import-Anweisungen>

<Liste von Typdeklarationen>

***Ausführbare*** Java-Programme sind Ansammlungen von übersetzten Typdeklarationen. Die Gruppierung in Übersetzungseinheiten und die Schachtelung von Klassendeklarationen existiert auf der Ebene nicht mehr.

# Schachtelung von Klassen

Klassen integrieren zwei Aufgaben:

- Sie beschreiben Objekte (Typaspekt).
- Sie gruppieren Daten und Methoden, bilden einen Gültigkeitsbereich und unterstützen Information Hiding (Modulaspekt).

Mehrere objektorientierte Sprachen verstärken den zweiten Aspekt, indem sie die Schachtelung von Klassen unterstützen.

**Vorteile:**

- Mehr Flexibilität bzgl. Sichtbarkeit und Information Hiding
- Klassendeklaration dichter bei ihren Anwendungen
- Erlaubt syntaktische Abkürzungen

# Begriffsklärung: (geschachtelte Klassen)

Eine Klasse heißt in Java ***geschachtelt*** (engl. ***nested***), wenn sie innerhalb der Deklaration einer anderen Klasse deklariert ist:

- als Komponente (ähnlich einem Attribut),
- als ***lokale*** Klassen (ähnlich einer lokalen Variablen)
- als ***anonyme*** Klassen bei der Objekterzeugung.

Ist eine Klasse nicht geschachtelt, nennen wir sie ***global*** (engl. ***top-level***).

Java unterscheidet statische und innere Klassen.

# Begriffsklärung: (geschachtelte Klassen) (2)

Geschachtelte Klassen heißen ***statisch***, wenn sie mit dem Modifikator `static` deklariert sind.

Statische Klasse haben die gleiche Bedeutung wie globale Klassen. Allerdings ergeben sich andere Sichtbarkeits- und Zugriffsbereiche.

Insbesondere:

- Geschachtelte Klassen können als privat deklariert werden, so dass sie außerhalb der umfassenden Klasse nicht verwendet werden können.
- Geschachtelte Klassen können auf private Attribute und Methoden der umfassenden Klasse zugreifen.

# Beispiel: (statische Klassen)

```
public class SLinkedList2 {
   private SEntry entries = null;
   private int size = 0;
   private static class SEntry {
      int head;
      SEntry tail;
   }
   int getFirst() { ... }
   ...
}
```

Der Typ SEntry ist nur innerhalb von SLinkedList2 sichtbar und zugreifbar.

# Bemerkung:

- Die Zugriffs- und Sichtbarkeitsregeln im Zusammenhang mit geschachtelten Klassen sind recht komplex (siehe folgendes Beispiel).
- Zugreifbare geschachtelte Klassen können über zusammengesetzte Namen außerhalb der umfassenden Klasse angesprochen werden (z.B. SLinkedList2.SEntry).

# Beispiel: (mehrfach geschachtelte Klassen)

```
public class MyClassIsMyCastle {

   private static int streetno = 169;
   private static class FirstFloor {
      private static class DiningRoom {
         private static int size = 36;
         private static void mymessage(){
            System.out.print("Access to streetno");
            System.out.println(": "+ streetno );
         }
      }
   }
   ...
```

# Beispiel: (mehrfach geschachtelte Klassen) (2)

```
    private static class SecondFloor {
        private static class BathRoom {
          private static int size = 16;
          private static void mymess(){
            System.out.print("I can access the ");
            System.out.print("dining room size: ");
            System.out.println(FirstFloor.DiningRoom.size);
          }
        }
    }

    public static void main( String[] argv ) {
        FirstFloor.DiningRoom.mymessage();
        SecondFloor.BathRoom.mymess();
    }
}
```

# Begriffsklärung: (innere Klassen)

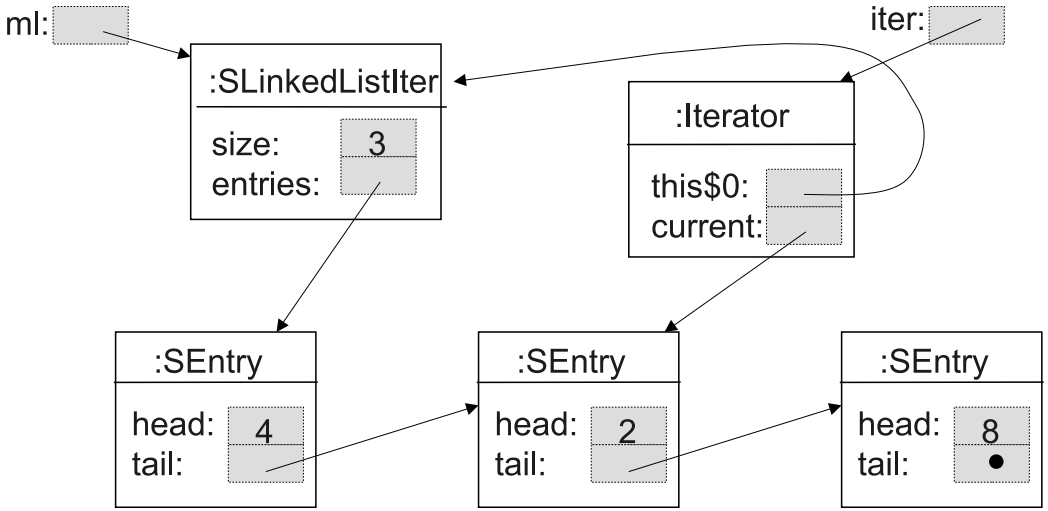
Geschachtelte Klasse, die nicht statisch sind, heißen ***innere*** Klassen.

Sei K eine Klasse und INN eine direkte innere Klasse von K. Dann bekommt jedes INN-Objekt eine Referenz auf das K-Objekt, in deren *Kontext* es erzeugt wurde.

# Beispiel: (innere Klassen)

Wir betrachten einfach verkettete Listen:

- SLinkedListIter: Listen mit Iteratoren
- SEntry: die Objekte für die Verkettung
- Iterator: die Iterator-Objekte als innere Objekte zu SLinkedListIter-Objekten



# Beispiel: (innere Klassen) (2)

```
import java.util.NoSuchElementException;

class SLinkedListIter {
   private int size = 0;
   private SEntry entries = null;

   private static class SEntry { ... }

   int getFirst(){ ... }
   ...
   public Iterator createIterator() {
      return new Iterator();
   }
   ...
```

# Beispiel: (innere Klassen) (3)

```
  public class Iterator { // innere Klasse
     private SEntry current;
     Iterator() { current = entries;  }
     boolean hasNext() {  return current != null;  }
     int next() {
        if( current == null ) {
           throw new NoSuchElementException();
        }
        int res = current.head;
        current = current.tail;
        return res;
     }
  }
} // Ende von SLinkedListIter
```

# Beispiel: (innere Klassen) (4)

```
class SLinkedListIterMain  {
  public static void main( String[] argf ){
     SLinkedListIter ml = new SLinkedListIter();
     ml.addFirst(8);
     ml.addFirst(2);
     ml.addFirst(4);
     SLinkedListIter.Iterator iter = ml.createIterator();
     while( iter.hasNext() ) {
        System.out.println( iter.next() );
      }
   }
}
```

# Programmmodule

Große Programme bestehen aus hunderten und tausenden von Typen/Klassen, die von unterschiedlichen EntwicklerInnen und Institutionen realisiert und verwaltet werden.

Es ist von großer Bedeutung, diese Klassen geeignet zu gruppieren und einzuteilen. Wichtige Aspekte:

- Auffinden, Verstehen des Zusammenhangs
- Übersetzungs-, Installationsorganisation
- Zugriffsrechte, Namensräume
- Pflege, Wartung

Für diese Aufgabenbereiche gibt es Modulkonzepte.

# Pakete in Java

Javas ***Pakete*** bieten ein relativ einfaches Modulkonzept:

- Ein Paket hat einen Namen P. Der Name besteht ggf. aus mehreren Teilen.
- Ein Paket P ist üblicherweise in einem Dateiverzeichnis mit Namen P abgelegt.

  **Beispiel:**

  Paket java.lang ist abgelegt in .../java/lang
- Ein Paket ist eine endl. Menge von Übersetzungseinheiten; der Paketname erscheint am Anfang der Übersetzungseinheiten.

# Pakete in Java (2)

- Pakete stellen einen Zugriffsbereich dar: Alle Programmelemente ohne Zugriffsmodifikator sind paketweit zugreifbar.
- Pakete bilden Namensräume: Ist P ein Paket, dann können alle Klassen K in P mittels P.K angesprochen werden (vollständiger Name).

# Beispiel: (Anwendung von Paketen)

Wir benutzen die Listenklasse LinkedList aus dem Paket java.util in unserem Paket kalau.

```
package kalau;

public class AufEinWort {
  public static void main( String[] argf ){
    java.util.LinkedList ml = new java.util.LinkedList();
    ml.addLast("Hasen");
    ml.addLast("hopsen");

    java.util.Iterator iter = ml.iterator();
    while( iter.hasNext() ) {
       System.out.print( iter.next() );
    }
    System.out.println();
} }
```

# Beispiel: (Anwendung von Paketen) (2)

Variante mit Import der angegebenen Typnamen:

```
package kalau;

import java.util.LinkedList;
import java.util.Iterator;

public class AufEinWort {
  public static void main( String[] argf ){
    LinkedList ml = new LinkedList();
    ml.addLast("bunny");
    ml.addLast("rabbits");
    Iterator iter = ml.iterator();
    ...
  }
}
```

# Beispiel: (Anwendung von Paketen) (3)

Variante mit Import aller Typnamen des Pakets java.util:

```
package kalau;

import java.util.*;

public class AufEinWort {
  ... // wie auf letzter Folie
}
```