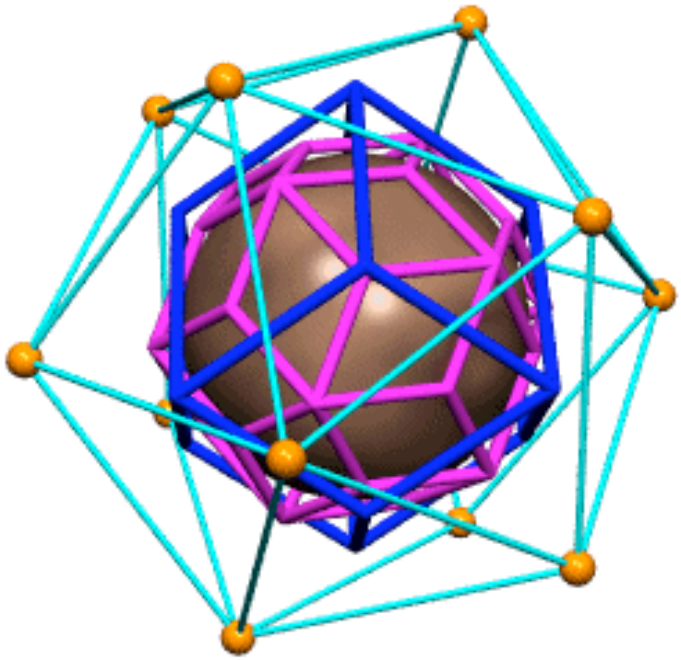
Innere Klassen in Java



SS 2012

Prof. Dr. Margarita Esponda

Innere Klassen

Klassen- oder Interfacedefinitionen können zur besseren Strukturierung von Programmen verschachtelt werden

Eine "Inner Class" wird innerhalb des Codeblocks einer anderen Klasse vereinbart.

Die bisher eingeführten Klassen werden auch Top-Level-Klassen genannt.

Einerseits erweisen sich innere Klassen als elegante, sehr nützliche Zusätze der Sprache, andererseits gibt es einige Regeln und Sonderfälle, die leicht verwirren können und deshalb vermieden werden sollten.

"Top-Level"-Klassen

```
public class TopLevelClass1 {
    ....
}
```

TopLevelClass1.java

```
public class TopLevelClass2 {
    ....
}
```

TopLevelClass2.java

Innere Klassen

```
public class OuterClass {
    ...
    public class InsideClass {

    }
}
```

OuterClass.java

Innere Klassen

Ziel:

Definition von Hilfsklassen möglichst nahe an der Stelle, wo sie gebraucht werden.

Motivation:

Geschachtelte und innere Klassen sind wesentlich motiviert durch das neue Ereignismodell im AWT von Java 1.1.

Alle Probleme lassen sich in Java mit externen Klassen lösen, aber oft verlässt man dadurch das Konzept der Sichtbarkeit.

Innere Klassen

Innere Klassen

werden insbesondere in Zusammenhang mit der Programmierung von **Benutzeroberflächen** und **Iteratoren** eingesetzt.

Innere Klassen **können auf Komponenten der sie umfassenden Klasse zugreifen**.

innerhalb von Blöcken gelten die Zugriffsregeln für Blöcke.

Eine innere Klasse ist außerhalb des sie definierenden Blockes, außer mit Hilfe des voll qualifizierenden Namens, nicht sichtbar.

Innere Klassen

Der Name einer inneren Klasse setzt sich aus dem Namen der umfassenden Klasse, dem Trennzeichen **$** sowie dem Namen der Klasse zusammen.

OuterClass$InsideClass.class

```
public class OuterClass {
    ...
    public class InsideClass {

    }
    ...
}
```

Innere Klassen

Ab Java 1.1 gibt es vier Sorten von inneren Klassen.

- **Elementklassen**

  innere Klassen, die in anderen Klassen definiert sind.

- **Geschachtelte Klassen**

  sind Top-Level-Klassen und Interfaces, die innerhalb anderer Klassen definiert sind, aber trotzdem Top-Level-Klassen sind.

- **Lokale Klassen**

  Klassen, die innerhalb einer Methode oder eines Java-Blocks definiert werden.

- **Anonyme Klassen**

  Lokale und namenlose Klassen.

Geschachtelte Top-Level-Klassen

```
class EnclosingClass{
  ...
     static class StaticNestedClass {
          . . .
      }

     class InnerClass {
       ...
     }
} // end of class
```

Geschachtelte Top-Level-Klassen und Interfaces verhalten sich wie andere Paket-Elemente auf der äußersten Ebene, außer dass ihrem Namen der Name der umgebenden Klasse vorangestellt wird.

Sie werden als **static** deklariert

Geschachtelte Top-Level-Klassen

- erlauben eine weitergehende Strukturierung des Namenraums von Klassen.

```
public class A {
        static int i = 4711;
        public static class B {
                int my_i = i;
                public static class C { ... }//end..C
        }// end of class B
}// end of class C
```

- werden genauso verwendet wie normale *Top-Level*-Klassen

```
A a = new A();
A.B ab = new A.B();
A.B.C abc = new A.B.C();
```

Geschachtelte Top-Level-Klassen

```
public class A {

    static String a = "A";
    String b = "B";

    public static class B {
        void m() {
            System.out.println( a );
        }
    } // end of class B
} // end of class A
```

Zugriff nur auf **statische** Variablen der umrahmenden Klassen.

Elementklassen

Elementklassen **echte innere Klassen** im Gegensatz zu den geschachtelten Top-Level-Klassen, die nur zur Strukturierung dienen.

Eine Elementklasse hat Zugriff auf alle Variablen und Methoden ihrer umgebenden Klasse.

Elementklassen werden analog gebildet und benutzt wie normale Klassen.

Der Compiler erzeugt beim Übersetzen einer Klassendatei für alle verschachtelte Klassen und Interfaces eigene **.class**-Dateien.

Elementklassen

Sie tragen den Namen der übergeordneten Klasse(n) und den eigenen, wobei diese mit **$** unterteilt werden:

```
public class A {
      . . .
    public class B {
          . . .
        public class C {
            . . .
        }
    }
}
```

javac A.java

A.class **A$B.class** **A$B$C.class**

Auf diese Art wird vor der "Java Virtuelle Maschine" verborgen, dass diese Klassen eigentlich ineinander geschachtelt sind.

Elementklassen

Objekte von **Elementklassen** sind immer mit einem Objekt der umgebenden Klasse verbunden.

```
public class A {
        public static int i = 30;
        public class B {
                int j = 4;
                public class C {
                        int k = i;
                }
        }
}
```

Objekterzeugung:

```
A a = new A();
A.B b = a.new B();
A.B.C c = b.new C();
```

Elementklassen

Jeder Instanz einer Elementklasse ist ein Objekt der umgebenden Klassen zugeordnet.

Damit kann das Objekt der Elementklasse implizit auf die Instanzvariablen der umgebenden Klasse zugreifen (**auch auf private Instanzvariablen**).

Elementklassen dürfen keine statischen Elemente (Attribute, Methoden, Klassen, Interfaces) besitzen.

Elementklassen

```
public class H {

    static String  t = "text";
    String  at = "another text";

    public class B {
        public  void print() {
            System.out.println( at );
            System.out.println( t );
        }
    }// end of class B
}// end of class H
```

Zugriff auf alle Variablen der umgebenden Klassen.

Lokale Klassen

Lokale Klassen sind innere Klassen, die nicht auf oberer Ebene in anderen Klassen verwendbar sind, sondern nur lokal innerhalb von Anweisungsblöcken von Methoden.

Lokale Klassen

Eine **Lokale Klasse** kann innerhalb einer Methode definiert und auch nur dort verwendet werden.

```
public class C {
    ...
     public void doSomething() {
        int i = 0;

       class X implements Runnable {
              public X() {...}
              public void run() {...}
        }
        new X().run();
    } // end of doSomething
}
```

Verwendung

Lokale Klassen

- Lokale Klassen dürfen folglich nicht als **public, protected**, **private** oder **static** deklariert werden.

- Lokale Klassen dürfen keine statischen Elemente haben (Felder, Methoden oder Klassen).

  Ebenso dürfen sie nicht den gleichen Namen tragen wie eine der sie umgebenden Klassen.

- Eine Lokale Klasse kann im **umgebenden Codeblock** nur die mit **final** markierten Variablen und Parameter benutzen.

Lokale Klassen

```
public class A {
    int x = 42;

    public void doSomething() {
        final int i = 4711; // in B sichtbar
        int k = 2; // in B nicht sichtbar!
        class B {
            int j = i * x;
        } // end of class B
    } // end of method doSomething()
}
```

Lokale Klassen

```
public class H {
    String t = "text";
    public void m() {
        final String mt = "in m";
        class C {
            void h() {
                System.out.println( t );//Instanzvariable
                System.out.println( mt );//mt ist final
            }
        }// end of class C
        C in_m = new C();
        in_m.h();
    }// end of method m

    public static void main( String[] args ) {
        H h = new H();
        h.m();
    }
}
```

Ausgabe:

```
text
in m
```

Anonyme Klassen

"Einweg-Klassen"

Anonyme Klasse werden wie lokale Klassen innerhalb von Anweisungsblöcken definiert

haben keinen Namen. Sie entstehen immer zusammen mit einem Objekt

haben keinen Konstruktor

haben die gleichen Beschränkungen wie Lokale Klassen.

Anonyme Klassen

Die Syntax für Anonyme Klassen ist:

***new**-expression **class**-body*

Für die Anonyme Klasse kann man keine extends- oder implements-Klauseln angeben.

Anonyme Klassen

Beispiel:

```
public void doSomething() {
      Runnable r = new Runnable() {
          public void run() {...}
      };
      r.run();
}
```