

# Programmierzettel

## *Geschachtelte Klassen*

Dominikus Herzberg

Version 1.1, 2021-07-05

# Lerninhalte

Geschachtelte Klassen .....	1
Statisch geschachtelte Klassen ( <i>nested static classes</i> ) .....	2
Innere Klassen ( <i>inner member classes</i> ) .....	3
Lokale Klassen ( <i>inner local classes</i> ) .....	4
Anonyme Klassen ( <i>inner anonymous classes</i> ) .....	5

# Geschachtelte Klassen

Neben den sogenannten Top-Level-Klassen erlaubt es Java zudem, Klassen innerhalb der folgenden Sprachkonstrukte einzuschachteln (*nested classes*), was u.a. die Möglichkeiten bietet, Klassen im Code zu kapseln und logisch zu gruppieren.

¥ Im Rumpf einer Klasse kann man als Member weitere Klassen deklarieren

! die Klasse ist entweder als **static** deklariert (*nested static class*)

! oder nicht, dann nennt man sie innere Member-Klasse (*inner member class*)



Merke, eine Top-Level-Klasse kann nicht **static** sein.

*Syntaktische Grundlage: Klasse als Member einer Klasse*

ClassMemberDeclaration:

Ê | FieldDeclaration

Ê | MethodDeclaration

Ê | ClassDeclaration

Ê | InterfaceDeclaration

Ê | `;`

¥ Innerhalb eines Codeblocks **{ }** wie z.B. im Rumpf einer Methode kann man eine sogenannte innere lokale Klasse anlegen (*inner local class*)

*Syntaktische Grundlage: Klasse als gültige Block-Anweisung*

BlockStatement:

Ê LocalVariableDeclarationStatement

Ê | ClassDeclaration

Ê | Statement

¥ Zur unmittelbaren Erzeugung einer Instanz kann man an geeigneten Stellen im Code eine anonyme Klasse deklarieren (*inner anonymous class*)

*Syntaktische Grundlage: Der optionale ClassBody*

UnqualifiedClassInstanceCreationExpression:

Ê `new` [TypeArguments]

Ê ClassOrInterfaceTypeToInstantiate `(` [ArgumentList] `)` [ClassBody]

# Statisch geschachtelte Klassen (*nested static classes*)

Beispiel für statisch geschachtelte Klasse

```
class Top {  
    static class Nested {  
        int i = 1;  
        void inc() { ++this.i; }  
    }  
    int k = 3;  
    Nested n = new Nested();  
}
```

Das Wichtigste in Kürze

- ⌘ Die beiden Klassen `Top` und `Nested` stehen in keinerlei Zusammenhang, sprich `Nested` kann nicht auf Variablen von `Top` zugreifen. Es ist so als sei `Nested` eine Top-Level-Klasse
- ⌘ `Nested` kann lediglich auf statische Members von `Top` zugreifen
- ⌘ Instanzen werden ganz "normal" erzeugt
  - ! von "außen" mit `new Top.Nested()`
  - ! von "innen" mit `new Nested()` (siehe vorletzte Codezeile)

JShell

```
jshell> Top.Nested nested = new Top.Nested()  
nested ==> Top$Nested@3c09711b  
  
jshell> Top top = new Top()  
top ==> Top@b97c004  
  
jshell> top.n.inc()  
  
jshell> top.n.i  
$5 ==> 2
```

# Innere Klassen (*inner member classes*)

Beispiel für innere Klasse

```
class Top {  
    class Inner {  
        int i = k; !  
    }  
    int k = 3;  
    Inner n = new Inner();  
}
```

! Beachte: Die innere Klasse kann auf das Feld der äußeren Klasse zugreifen!

Das Wichtigste in Kürze

¥ Die Klasse `Inner` ist von außen nicht adressierbar und damit nicht instanzierbar

! `new Top.Inner()` geht nicht

¥ Eine Instanz der lokalen Klasse kann nur aus dem Kontext der umgebenden Klasse erzeugt werden

! Beispiel: `new Top().n`

! Beispiel: `new Top().new Inner()`

¥ Eine Instanz der lokalen Klasse hat Zugriff auf die Members der sie erzeugenden Top-Level-Instanz.

! `new Top().n.i` liefert den Wert von `k`

JShell

```
jshell> Top top = new Top()  
top ==> Top@3c09711b  
  
jshell> top.n  
$3 ==> Top$Inner@b97c004  
  
jshell> top.n.i  
$4 ==> 3
```

## Was macht der Compiler mit einer inneren Klasse?

Eine innere Klasse wird vom Compiler aus ihrem Kontext gelöst und auf Top-Level-Ebene angelegt. Im Fall der inneren Klasse wird dem Konstruktor ein weiterer Parameter vom Typ der äußeren Klasse hinzugefügt. Beim Konstruktoraufruf wird die Instanz der äußeren Klasse übergeben. Über diesen Weg hat die innere Klasse Zugriff auf Variablen und Methoden der äußeren Klasse.

# Lokale Klassen (*inner local classes*)

In einem Codeblock `{ }` kann eine sogenannte lokale Klasse deklariert werden!

*Beispiel f r lokale Klasse*

```
void method(int arg) {  
    int k = 3;  
    class Local {  
        void show() { System.out.printf("arg = %d, k = %d\n", arg, k); }  
    }  
    new Local ().show();  
}
```

*Das Wichtigste in K rze*

-   Die Klasse ist nur im Rumpf des Blocks adressierbar und instanziiierbar, siehe Codezeile mit `new Local ()`.
-   Die Instanz der lokalen Klasse kann auf Variablen des sie umgebenden Kontextes zugreifen, sofern sie `final` oder *effectively final* sind.
-   Es gibt kein `static` f r lokale Klassen.

Eine Variable ist *effectively final*, wenn man sie auch mit dem Schl sselwort `final` versehen k nnte, ohne dass sich der Java-Compiler beschwerte; sprich, der Wert der Variablen wird *effektiv* nur einmal gesetzt.

 

Aufgepasst, die Anweisungen in einem Block werden von "oben nach unten" ausgef hrt. So muss `int k = 3;` vor `Local` deklariert und initialisiert sein, nicht danach.

*JShell*

```
jshell> method(42)  
arg = 42, k = 3
```

## Warum die Zugriffsbeschr nkung auf (*effectively*) `final`?

Eine lokale Klasse ist eine innere Klasse. In Fall der lokalen Klasse kann (und darf) der Compiler keine Zugriffe zu Variablen z.B. aus dem Methodenkontext einer ganz anderen Klasse haben, aus der der Compiler sie genommen hat. Da sich (effektiv) finale Variablen nicht  ndern, sind Wertkopien zur  bergabe an die lokale Klasse jedoch unproblematisch.

# Anonyme Klassen (*inner anonymous classes*)

Anonyme Klassen können an all den Stellen im Code zum Einsatz kommen, wo eine Klasse mit `new` instanziiert werden darf. Der Code der Klasse wird zusammen mit dem `new`-Operator angegeben und an Ort und Stelle sowohl deklariert als auch instanziiert. Da es keine mit `class` erzeugte und damit mit einem Bezeichner angelegte Klassendeklaration gibt, spricht man von einer *anonymen* Klasse (im Sinne von "namenlose" Klasse).

Die anonyme Klasse nimmt in ihrer Deklaration entweder Bezug auf ein Interface, das sie implementiert, oder auf eine Klasse, die sie als Unterklasse erweitert.

*Beispiel für anonyme Klasse, die ein Interface implementiert*

```
interface Hi able {  
    String sayHi ();  
}  
  
Hi able hi = new Hi able() {  
    public String sayHi () { return "Hi"; } !  
};  
// j shell > hi . sayHi () ==> "Hi "
```

! Da ein Interface implementiert wird, muss die Methode `public` sein.

*Beispiel für anonyme Klasse, die eine Oberklasse erweitert*

```
class Hi Sayer { }  
  
var hi = new Hi Sayer() { !  
    String sayHi () { return "Hi"; }  
};  
// j shell > hi . sayHi () ==> "Hi "
```

! Hier ist ein `var` nötig: Es gibt keine andere Möglichkeit, Variable `hi` vom Typ der anonymen Unterklasse zu deklarieren; sonst lässt sich `sayHi` nicht aufrufen. Wenn die Oberklasse einen entsprechenden Konstruktor hat, könnten in den runden Klammern Argumente übergeben werden

*Das Wichtigste in Kürze*

- ¥ Folgen einem `new` nach der Argumentliste ein Paar geschweifte Klammern, so handelt es sich um eine anonyme Klasse; in den geschweiften Klammern befindet sich die Deklaration der Klasse entweder im Sinne einer Implementierung oder einer Erweiterung.
- ¥ Das `new` erzeugt direkt die Instanz der anonymen Klasse.
- ¥ Die Instanz der anonymen Klasse kann auf Variablen des umgebenden Kontextes zugreifen, sofern sie `final` bzw. *effectively final* sind. Anonyme Klassen sind ebenfalls innere Klassen, für die dieselbe Argumentation gilt wie für lokale Klassen.