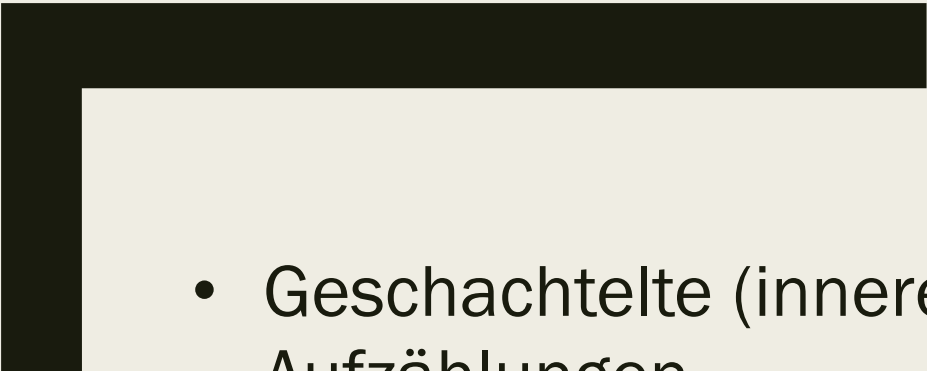





INNERE KLASSEN

- 
- Geschachtelte (innere) Klassen, Schnittstellen, Aufzählungen
 - Mitglieds- oder Elementklasse
 - Lokale Klassen
 - Anonyme innere Klassen
- 

Klassen in Java



INNERE KLASSEN

Innere Klasse

- Klassen, Aufzählungen(Enums), Schnittstellen(Interfaces) können als Typen eingebettet werden. (Innere Klassen)
- Die Motivation dazu:
 - *Details sollen versteckt sein*
 - *Typdeklaration der keine Sichtbarkeit braucht*
- Klasse In, die in eine Klasse Out gesetzt wird:

```
class Out{  
|   class In{  
    }  
}
```

4 Typen von Inneren Klassen

Statische innere Klasse	<pre>class Out{ static class In{ } }</pre>
Mitgliedsklasse	<pre>class Out{ class In{ } }</pre>
Lokale Klasse	<pre>class Out{ void out() { class In{ } } }</pre>
Anonyme innere Klasse	<pre>class Out{ void out() { new Runnable() { public void run() {} }; } }</pre>

Innere Klassen

- Das Gegenteil von geschachtelten Klassen heißt Top-Level-Klasse
 - *Top-Level and Nested Classes.*
- Die Laufzeitumgebung kennt nur Top-Level-Klassen
- Geschachtelte innere Klassen werden zu ganz „normalen“ Klassen kompiliert

STATISCHE INNERE KLASSEN UND SCHNITTSTELLEN

Statische innere Klassen und Schnittstellen

- Statische innere Klassen – nested top-level class
- Top-level: Klassen können das Gleiche wie „normale“ klassen oder Schnittstellen (Unterpaket mit Namensraum)
- Zur Erzeugung von statischen inneren Klassen sind keine Objekte der äußeren Klasse nötig.
- Die inneren (nicht statischen) Typen benötigen einen Verweis auf das äußere Objekt.
- Oracle´s Spezifikation nach, sind die statischen inneren Klassen keine „echten“ inneren Klassen.

Statische innere Klassen und Schnittstellen

```
public class Lampe {  
  
    static String s ="Licht AN";  
    int i = 1;  
  
    static class Gluehbirne{  
        void ausgabe() {  
            System.out.println(s);  
            //Cannot make a static reference to the non-static field i  
            //System.out.println(i);  
        }  
    }  
  
    public static void main(String[] args) {  
        Lampe.Gluehbirne    birne1 = new Lampe.Gluehbirne();  
        Gluehbirne          birne2 = new Lampe.Gluehbirne();  
        Lampe.Gluehbirne    birne3 = new Gluehbirne();  
        Gluehbirne          birne4 = new Gluehbirne();  
    }  
}
```

Statische innere Klassen und Schnittstellen

- Die statische innere Klassen Gluehbirne hat Zugriff auf alle anderen statischen Eigenschaften der äußeren Klasse Lampe, hier als auf das statische Attribut s.
- Zugriff auf Instanz-Attribute ist aus der statischen inneren Klasse nicht möglich(Klasse, im gleichen Paket)
- Zugriff von außen auf innere Klassen:

ÄußereKlasse.InnereKlasse

- Die Äußere und Innere Klassen dürfen nicht den selben Namen verwenden.

Statische innere Klassen und Schnittstellen

- Zulässige Modifier:
 - *Abstract, final und die Sichtbarkeitsmodifizierer*
- Innere Klassen dürfen public, paketsichtbar, aber auch private oder protected sein.
- Eine private statische innere Klasse ist wie ein privates statisches Attribut. Sie kann nur von der umschließenden äußeren Klasse gesehen werden, aber nicht von anderen Top-Level-Klassen
- Protected an statischen inneren Typen ermöglicht für den Compiler einen etwas effizienteren Bytecode.

MITGLIEDS- ODER ELEMENTKLASSEN

Mitglieds- oder Elementklassen

- Mitgliedsklassen (engl. Member class) ist vergleichbar mit einem Attribut.

```
public class Haus {  
  
    private String eigentuemer = "Ich";  
  
    class Raum {  
        void ok() {  
            System.out.println(eigentuemer);  
        }  
  
        // The method error cannot be declared static; static methods can only be  
        // declared in a static or top level type  
        // static void error() { }  
    }  
}
```

Mitglieds- oder Elementklassen

- Java schreibt eine spezielle Form für die Erzeugung mit new vor, die folgendes allgemeine Format besitzt:

```
refHaus.new Raum();
```

- Die Referenz vom Typ der äußeren Klasse. Um in der statischen main()-Methode vom Haus eine Raum Instanz zu erzeugen, schreiben wir:

```
Haus refHaus = new Haus();  
Raum refRaum = refHaus.new Raum();
```

- Oder kurz:

```
Raum refRaum = new Haus().new Raum();
```

Mitglieds- oder Elementklassen

- Der Compiler generiert aus inneren Typen normale Klassendateien, mit synthetischen Methoden
- Der Compiler generiert neue Namen nach dem Muster:
ÄußererTyp\$InnererTyp, das heißt, ein Dollar-Zeichen trennt die Namen vom äußeren und inneren Typ. Genauso heißt die entsprechende *.class-Datei* auf der Festplatte nach dem Kompilieren.

Mitglieds- oder Elementklassen

- Die Klasse Raum hat Zugriff auf alle Eigenschaften von Haus, auch auf die privaten.
- Innere Mitgliedsklassen dürfen keine statischen Eigenschaften deklarieren. Der Versuch führt zu einem Compilererror:

```
12 // The method error cannot be declared static; static methods can only be
13 // declared in a static or top level type
14 static void error() { }
```


Mitglieds- oder Elementklassen

- Um ein Exemplar von Raum zu erzeugen, muss ein Exemplar der äußeren Klasse existieren – (darf auch eine Anonyme Instanz sein)
- Im Konstruktor oder in einer Instanzmethode der äußeren Klassen kann mit dem new-Operator ein Exemplar der inneren Klasse erzeugt werden.
- Von außerhalb – oder von einem Statischen Block der äußeren Klasse – muss bei Elementklassen sichergestellt sein, dass es ein Exemplar der äußeren Klasse gibt.

Mitglieds- oder Elementklassen (this)

- Zugriff einer inneren Klasse **In** auf die this-Referenz der sie umgebenden Klasse **Out**
 - *Out.this*
- Überdecken die Variablen der inneren Klasse die Variablen der äußeren Klasse
 - *Können wir mit Out.this.Eigenschaft an die Eigenschaften der äußeren Klasse Out gelangen.*

Mitglieds- oder Elementklassen (this)

```
public class MoebliertesHaus {  
    String s = "Haus";  
    class Raum {  
        String s = "Raum";  
        class Stuhl {  
            String s = "Stuhl";  
            void output() {  
                System.out.println(s);  
                System.out.println( this.s);  
                System.out.println(Stuhl.this.s);  
                System.out.println(Raum.this.s);  
                System.out.println(MoebliertesHaus.this.s);  
            }  
        }  
    }  
}  
  
public static void main(String[] args) {  
    new MoebliertesHaus().new Raum().new Stuhl().output();  
}
```

Stuhl
Stuhl
Stuhl
Raum
Haus

Mitglieds- oder Elementklassen (this)

- Instanzen der inneren Klassen Raum und Stuhl lassen sich wie folgt erstellen:

```
public static void main(String[] args) {  
    MoebliertesHaus h = new MoebliertesHaus();  
    MoebliertesHaus.Raum r = h.new Raum();  
    MoebliertesHaus.Raum.Stuhl s = r.new Stuhl();  
  
    // Hier rufen wir die output Methode von Stuhl nun auf  
    s.output();  
    // oder in einen Schritt mit einer anonymen Instanz  
    new MoebliertesHaus().new Raum().new Stuhl().output();  
}
```

Mitglieds- oder Elementklassen (this)

- MoebliertesHaus.Raum.Stuhl bedeutet nicht automatisch, dass MoebliertesHaus ein Paket mit dem Unterpaket Raum ist, in dem die Klasse Stuhl existiert.
- Die Doppelbelegung des Punkts verbessert die Lesbarkeit nicht, es droht Verwechslungsgefahr zwischen inneren Klassen und Paketen.
 - *Deshalb sollten die Namenskonventionen beachtet werden:*
 - Klassennamen beginnen mit Großbuchstaben
 - Paketnamen mit Kleinbuchstaben
- Für das Beispiel MoebliertesHaus und Raum erzeugt der Compiler die Dateien MoebliertesHaus.class und MoebliertesHaus\$Raum.class.
- Damit innere Klassen an Attribute der äußeren gelangen, generiert der Compiler automatisch in jedem Exemplar der inneren Klasse eine Referenz auf die zugehörige Instanz der äußeren Klasse
- Mit der Referenz kann die innere Klasse auch auf nicht-statische Attribute der äußeren Klasse zugreifen.

Mitglieds- oder Elementklassen (this)

- Für die innere Klasse ergibt sich folgendes Bild in MoebliertesHaus\$Raum.class
 - *Mit dem Java Decompiler in die innere Klasse geschaut.*

```
class Haus$Raum
{
    Haus$Raum(Haus paramHaus) {}

    void ok()
    {
        System.out.println(Haus.access$0(this.this$0));
    }
}
```

Mitglieds- oder Elementklassen (this)

- Ist in einer Datei nur eine Klasse deklariert, kann diese nicht private sein. Private innere Klassen sind aber legal.
- Statische Hauptklassen gibt es auch nicht, aber innere statische Klassen sind legitim.
- Die folgende Tabelle fasst die erlaubten Modifizierer noch einmal kompakt zusammen:

Modifizierer erlaubt auf	Äußere Klassen	Inneren Klasse
Public	Ja	Ja
Protected	Nein	Ja
Package / Default	Ja	Ja
Private	Nein	Ja
Static	Nein	Ja
final	Ja	Ja
abstract	Ja	Ja

Modifizierer erlaubt auf	Äußere Schnittstellen	Inneren Schnittstellen
Public	Ja	Ja
Protected	Nein	Ja
Package / Default	Ja	Ja
Private	Nein	Ja
Static	Nein	Ja
final	Nein	Nein
abstract	Ja	Ja

Mitglieds- oder Elementklassen (this)

- Äußere Klassen können auf private Eigenschaften der inneren Klasse zugreifen

```
public class NotSoPrivate
{
    private static class Family { private String dad, mom; }

    public static void main( String[] args )
    {
        class Node { private Node next; }

        Node n = new Node();
        n.next = new Node();

        Family ullenboom = new Family();
        ullenboom.dad = "Heinz";
        ullenboom.mom = "Eva";
    }
}
```

Mitglieds- oder Elementklassen (this)

- Innere Klassen können auf alle Attribute der äußeren Klasse zugreifen. Eine innere Klasse wird als ganz normale Klasse übersetzt.
- Eine Inner Klasse kann auch auf die privaten Eigenschaften der äußeren Klasse zurückgreifen.
 - *Diese Designentscheidung wurde lange kontrovers Diskutiert und gilt immer noch als umstritten.*
- Wie geht der zugriff auf private eigenschaften der äußeren Klasse?
 - *Der Trick ist, dass der Compiler eine synthetische statische Methode in der äußeren Klasse einführt.*

```
{  
    System.out.println(Haus.access$0(this.this$0));  
}  
}
```

Mitglieds- oder Elementklassen (this)

- Die statische Methode `access$0()` ist der Helfershelfer, der für ein gegebenes Haus das private Attribut nach außen gibt. Da die innere Klasse einen Verweis auf die äußere Klasse pflegt, gibt sie diesen beim gewünschten Zugriff mit, und die `access$0()` – Methode erledigt den Rest.
- Für weitere private Attribute würde der Compiler weitere nummerierte `access$n` – Methoden generieren.

LOKALE KLASSEN

Lokale Klassen

- Lokale Klassen sind innere Klassen
- Sie werden direkt in Anweisungsbköcken von Methoden, Konstruktoren und Initialisierungsblöcken gesetzt.
- Lokale Schnittstellen gibt es nicht.

Lokale Klassen

```
public class SpassInnerhalb {  
  
    public static void main(String[] args) {  
  
        int i = 2;  
        final int j = 3;  
        //i = 4; // <--- dies zerstört das effectively final  
        class Innerhalb{  
            Innerhalb(){  
                System.out.println( j );//<-- muss final  
                System.out.println( i );//<-- oder effectively final sein  
            }  
        }  
        new Innerhalb();  
    }  
}
```

Lokale Klassen

- Deklaration der inneren Klasse **Innerhalb** wie eine Anweisung.
- Sichtbarkeitsmodifizierer sind ungültig.
- Keine Klassenmethoden und allgemeine statische Attribute, bis auf finale Konstanten.
- Kann auf Methoden der äußeren Klasse und auf finale bzw. effectively final Variablen sowie finale Parameter zugreifen.
- Die Lokale Klasse muss implementiert werden, bevor sie genutzt werden kann.

ANONYME INNERE KLASSEN

Anonyme innere Klassen

- Haben keinen Namen
- Erzeugen immer automatisch eine Instanz
- Klassendeklaration und Instanzerzeugung sind zu einem Sprachkonstrukt verbunden.
- Die allgemeine Notation ist wie folgt:

```
new KlassenOderSchnittstelle() { /* Implementierung */ };
```

Anonyme innere Klassen

- In den geschweiften Klammern werden:
 - *Methoden und Attribute deklariert*
 - *Methoden überschrieben*
- new wird gefolgt vom Namen der Klasse oder Schnittstelle
- Keine Extends- oder Implements-Angaben und eigene Konstruktoren möglich.
- Instanzmethoden und finale statische Variablen sind erlaubt.

Anonyme innere Klassen

1. `new Klassenname(Optional Argumente) {...}`
 - *Anonyme Klasse ist eine Subklasse von Klassenname*
 - *Argumente für den Konstruktor der Basisklasse notwendig, wenn z.B. die Oberklasse keinen Standardkonstruktor besitzt.*
2. `new Schnittstellename() {.....}`
 - *Anonyme Klasse erbt von Object*
 - *Ohne Implementierung der Schnittstellenoperationen (Fehler) läge eine abstrakte innere anonyme Klasse vor.*

Anonyme innere Klassen

BSP: Unterklasse von java.awt.Point, sie überschreibt die toString()-Methode

```
Point p = new Point(10, 20){  
  
    @Override  
    public String toString() {  
        return "x: " + x + " y: " + y;  
    }  
  
    public double quote(){  
        return (double)x/y;  
    }  
};
```

Unterklasse von Point wird aufgebaut – Name der inneren Klasse fehlt.

Einziges Exemplar der anonymen Klasse lässt sich über die Variable p weiterverwenden.

Anonyme innere Klassen

- Der neue Anonyme Typ hat eine Methode `quote()`
 - *Diese kann direkt aufgerufen werden*
 - *Die `quote()`-Methode ist sonst unsichtbar, da der Typ anonym ist*
- Nur Methoden der Oberklasse(hier Objekt) beziehungsweise der Schnittstelle sind bekannt.
 - *Eine Anwendung kann mit Reflection auf die Methoden zugreifen.*

Anonyme innere Klassen

- Für innere Anonyme Klassen erzeugt der Compiler eine normale Klassendatei.
- Notationen für Klassennamen:
 - *InnerToStringDate\$1.*
 - *Falls es mehr als eine innere Klasse gibt, folgen \$2, \$3 und so weiter.*
 - ÄußereKlasse\$InnereKlasse geht wegen dem anonym nicht
- Bsp. Für nebenläufige Programme, gibt es die Klasse Thread und die Schnittstelle Runnable:
 - *Die Schnittstelle Runnable hat die Methode run(), diese wird in den parallel abzuarbeiteten Programmcode gesetzt. (mit einer inneren anonymen Klasse, die Runnable implementiert)*

```
new Runnable() { // Anonyme Klasse extends Object implements Runnable
    public void run() {
    }
};
```

Anonyme innere Klassen

Bsp. Exemplar kommt in den Konstruktor der Klasse Thread. Thread wird mit start() gestartet.

```
new Thread( new Runnable() {  
    @Override public void run() {  
        for ( int i = 0; i < 10; i++ )  
            System.out.printf( "%d ", i );  
        }  
    } ).start();  
  
for ( int i = 0; i < 10; i++ )  
    System.out.printf( "%d ", i );
```

Ausgabe: 0 0 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9

Anonyme innere Klassen

- Jede anonyme Klasse kann einen eigenen Konstruktor deklarieren (keine direkten) mit Hilfe von Exemplarinitialisierungsblöcken.
- Anonyme Klassen können keinen direkten Konstruktor haben.
- Bp.: Die anonyme Klasse ist eine Unterklasse von Point und initialisiert im Konstruktor einen Punkt mit den Koordinaten -1, -1. Aus diesem speziellen Punkt-Objekt lesen wir dann die Koordinaten wieder aus:

```
java.awt.Point p = new java.awt.Point() { { x = -1; y = -1; } };

System.out.println( p.getLocation() ); // java.awt.Point[x=-1,y=-1]

System.out.println( new java.awt.Point( -1, 0 )
{
    {
        y = -1;
    }
}.getLocation() ); // java.awt.Point[x=-1,y=-1]
```


Anonyme innere Klassen

- Im „anonymen Konstruktor“ ist kein `super()`
- `Super` wird automatisch in den Initialisierungsblock eingesetzt
- Parameter für die gewünschte Variante des (überladenen) Oberklasse-Konstruktors werden am Anfang der Deklaration der anonymen Klasse angegeben.

Bsp.

```
System.out.println( new Point(-1, 0) { { y = -1; } }.getLocation() );
```

Anonyme innere Klassen

- Objekt BigDecimal wird initialisiert
- Im Initblock der anonymen Unterklasse geben wir anschließend den Wert mit der geerbten toString()-Methode aus:

```
new java.math.BigDecimal( "12345678901234567890" ) {  
    { System.out.println( toString() ); }  
};
```

Anonyme innere Klassen

- Lokale und innere Klassen können auf lokale Variablen (finale Parameter der umschließenden Methode lesend zugreifen)
- Veränderung mit Trick möglich. Zwei Lösungen:
 - *Nutzung eines finalen Feldes der Länge 1, für das Ergebnis.*
 - *Nutzung von AtomicXXX-Klassen oder dem `java.util.concurrent.atomic`-Paket, die ein primitives Element oder eine Referenz aufnehmen.*

Lambda & Anonyme innere Klassen

- Mit Lambda Ausdrücken können wir ebenfalls „innere Anonyme Klassen“ erzeugen.
- Allerdings unterliegt diese vorgehensweise Einschränkungen.
 - *Das abzuleitende muss eine Interface sein.*
 - *Das Interface muss von der Struktur her ein Funktional Interface sein.*
 - *Wir können keine weiteren Methoden einbringen.*
- Formal Juristisch gilt diese variante nicht als „Innere Anonyme Klasse“

INNERE KLASSEN

Innere Klassen (this)

- this in Unterklassen

```
public class Shoe
{
    void out()
    {
        System.out.println( "Ich bin der Schuh des Manitu." );
    }

    class LeatherBoot extends Shoe
    {
        void what()
        {
            Shoe.this.out();
        }
    }
}
```

Innere Klassen (this)

- this in Unterklassen

```
@Override
void out()
{
    System.out.println( "Ich bin ein Shoe.LeatherBoot." );
}

public static void main( String[] args )
{
    new Shoe().new LeatherBoot().what();
}
}
```