

Inner Class

```
1 public class OuterClass {
2     private int var;
3     public class InnerClass {
4         void methodA() {};
5     }
6     public void methodB() {};
7 }
```

- ▶ Instanz von **InnerClass** kann auf alle Member von **OuterClass** zugreifen.
- ▶ Wenn **InnerClass** **static** deklariert wird, kann man nur auf statische Member zugreifen.
- ▶ Statische innere Klassen sind im Prinzip normale Klassen mit zusätzlichen Zugriffsrechten.
- ▶ Nichtstatische innere Klassen sind immer an eine konkrete Instanz der äußeren Klasse gebunden.

Beispiel – Zugriff von Außen

```
1 class OuterClass {
2     private int x = 1;
3     public class InnerClass {
4         void show() {
5             System.out.println("x = " + x);
6         } }
7     public void showMeth() {
8         InnerClass b = new InnerClass();
9         b.show();
10 } }
11 public class TestInner {
12     public static void main(String args[]) {
13         OuterClass a = new OuterClass();
14         OuterClass.InnerClass x = a.new InnerClass();
15         x.show();
16         a.showMeth();
17 } }
```

"TestInner.java"

Um von außen ein Objekt der inneren Klasse zu erzeugen, muss man erst ein Objekt der äußeren Klasse generieren.
Dann erzeugt man ein Objekt der Klasse z.B. durch `objOuter.new InnerClass()`, wobei wir hier annehmen, dass **InnerClass** einen Defaultkonstruktor hat.

Beispiel – Zugriff von Außen

```
1 class OuterClass {
2     private static int x = 1;
3     public static class InnerClass {
4         void show() {
5             System.out.println("x = " + x);
6         } }
7     public void showMeth() {
8         InnerClass b = new InnerClass();
9         b.show();
10 } }
11 public class TestInnerStatic {
12     public static void main(String args[]) {
13         OuterClass a = new OuterClass();
14         OuterClass.InnerClass x =
15             new OuterClass.InnerClass();
16         x.show(); a.showMeth();
17 } }
```

"TestInnerStatic.java"

Eine Anwendung von öffentlichen, inneren Klassen sind z.B. **Adapterklassen** (vgl. **Iterator**).

Normalerweise erzeugt man keine Objekte einer inneren Klasse von außen. Stattdessen bietet häufig die äußere Klasse eine Funktion, die ein Objekt der inneren Klasse zur Verfügung stellt.

Häufig (vgl. verkettete Liste mit innerer Klasse **ListElem**) werden innere Klassen aber auch nur zur Datenkapselung eingesetzt und sind dann privat.

Local Inner Class

Eine **lokale, innere Klasse** wird innerhalb einer Methode deklariert:

```
1 public class OuterClass {
2     private int var;
3     public void methodA() {
4         class InnerClass {
5             void methodB() {};
6         }
7     }
8 }
```

- ▶ Kann zusätzlich auf die **finalen** Parameter und Variablen der Methode zugreifen.

Man kann nicht von außen auf die Klasse zugreifen. Deshalb machen modifier wie **private**, **public** keinen Sinn und sind nicht erlaubt.

Beispiel - Iterator

```
1 interface Iterator<T> {
2     boolean hasNext();
3     T next();
4     void remove(); // optional
5 }
```

- ▶ Ein Iterator erlaubt es über die Elemente einer Kollektion zu iterieren.
- ▶ Abstrahiert von der Implementierung der Kollektion.
- ▶ `hasNext()` testet, ob noch ein Element verfügbar ist.
- ▶ `next()` liefert das nächste Element (falls keins verfügbar ist wird eine `NoSuchElementException` geworfen).
- ▶ `remove()` entfernt das zuletzt über `next()` zugegriffene Element aus der Kollektion.

Falls die Kollektion das Entfernen von Elementen nicht erlaubt, bleibt `remove()` unimplementiert und liefert bei Aufruf eine Exception.

Beispiel - Iterator

`curr` zeigt auf das Element, das beim letzten Aufruf von `next()` zurückgegeben wurde.

```
1 public class TestIterator implements Iterable<Integer> {
2     Integer[] arr;
3     TestIterator(int n) {
4         arr = new Integer[n];
5     }
6     public Iterator<Integer> iterator() {
7         class MyIterator implements Iterator<Integer> {
8             int curr = arr.length;
9             public boolean hasNext() { return curr>0;}
10            public Integer next() {
11                if (curr == 0)
12                    throw new NoSuchElementException();
13                return arr[--curr];
14            }
15        }
16        return new MyIterator();
17    }
```

Das Interface `Iterable<Integer>` „verspricht“ die Implementierung der Funktion `Iterator<Integer> iterator()`.

"TestIterator.java"

Beispiel - Iterator

Anwendung des Iterators:

```
18 public static void main(String args[]) {
19     TestIterator t = new TestIterator(10);
20     for (Iterator<Integer> iter = t.iterator();
21         iter.hasNext();) {
22         Integer i = iter.next();
23         System.out.println(i);
24     }
25     for (Integer j : t) {
26         System.out.println(j);
27     }
28 }
29 }
```

Erweiterte Syntax der For-Schleife. Da `t` das Interface `Iterable<Integer>` implementiert, kann man mit dieser Syntax über die Elemente der Kollektion iterieren.

"TestIterator.java"

In diesem Fall wird nur 20 mal null ausgegeben...