universität innsbruck

Innere Klassen

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

Innere Klassen

- Bisher: Klassen und Schnittstellen jeweils in einer Datei enthalten (Top-Level Klassen)
- Alternative: innere Klassen (auch geschachtelte Klassen genannt)

```
public class Outer {
    public class Inner {
        …
    }
}
```

Motivation

- Ziel und Verwendung innerer Klassen
  - Definition von Hilfsklassen möglichst nahe an der Stelle, wo sie gebraucht werden.
  - Logisches Gruppieren von Klassen, die nur an einem Ort verwendet werden
  - Zur besseren Kapselung
  - Zur Erhöhung der Lesbarkeit und Wartbarkeit des Codes
- Nachteile
  - Klassen werden aufgeblasen
  - Verringert wiederum die Wartbarkeit

Allgemeines (1)

- Eine innere Klasse wird innerhalb einer umschließenden Klasse definiert.
- Innere Klassen existieren nur um die umschließende Klasse zu unterstützen.
- Falls eine innere Klasse auch außerhalb des Kontexts der umschließenden Klasse Verwendung finden könnte, sollte sie stattdessen als eigene Top-Level Klasse Implementiert werden.

Allgemeines (2)

- Arten
  - Statische innere Klassen
  - Elementklassen
  - Lokale Klassen
    - Mit Klassennamen
    - Anonym
- Neben Klassen können auch Enums, Records und Interfaces geschachtelt werden.
- Der Name einer inneren Klasse bzw. eines Interfaces darf nicht identisch mit dem einer äußeren Klasse sein.
- Klassen können beliebig tief ineinander verschachtelt werden.
  - Nicht alle Kombinationen sind erlaubt!
  - Verschachtelte innere Klassen werden nur sehr Selten benutzt!
- Innere Klassen können je nach Art auf bestimmte Komponenten der umfassenden Klasse zugreifen.

Statische Innere Klassen

- Sind wie statische Member der Klasse zu handhaben.
- Sind unabhängig von Exemplaren der umgebenden Klasse.
- Erlauben keinen Zugriff auf nicht-statische Elemente der äußeren Klasse.
- Werden auch als nested Top-Level Klassen bezeichnet.

```
class Outer
{
    . . . . .
    static class Inner
    {
        . . . . .
    }
}
```

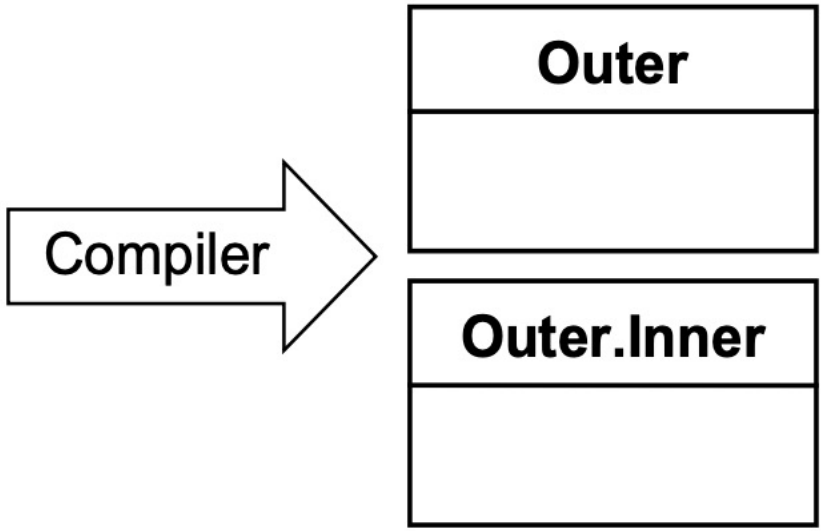


Abbildung übernommen aus „Java als erste Programmiersprache“, Goll & Heinisch

Statische Innere Klassen Beispiel (1)

```
public class OuterClass {
    private static int outerCount = 1;

    public static class StaticInnerClass {
        private static int innerCount = 3;

        public static void print() {
            System.out.println("sum: " + (innerCount + outerCount));
            ++outerCount;
            ++innerCount;
        }
    }
}
```

```
OuterClass.StaticInnerClass.print();
StaticInnerClass.print(); // Compile-Error!
```

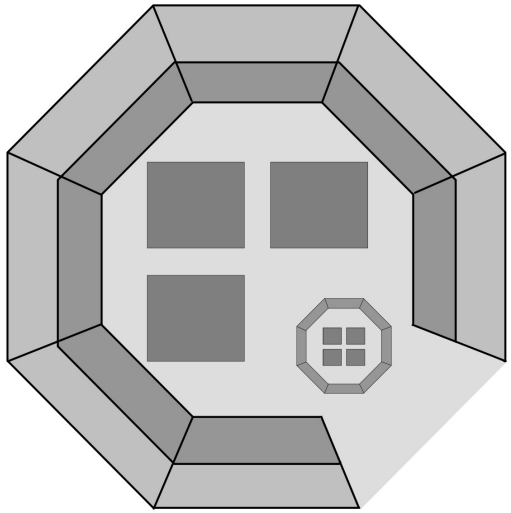
Statische Innere Klassen Beispiel (2)

```
public class LinkedList<T> implements Iterable<T> {

    private final Node<T> root = new Node<>(null, null);

    …

    private static class Node<T> {
        private final T element;
        private Node<T> next;

        private Node(T element, Node<T> next) {
            this.element = element;
            this.next = next;
        }
    }
}
```

src/at/ac/uibk/pm/generics/linkedlist/LinkedList.java

Elementklassen

- Werden auch Member-Klassen oder Mitgliedsklassen genannt.
- Sind Elemente der äußeren Klasse.
- Sind an ein Exemplar der äußeren Klasse gebunden.



Objekt einer Elementklasse in
einem umschließenden Objekt.

Abbildung übernommen aus „Java als erste Programmiersprache“, Goll & Heinisch

Elementklassen – Regeln (1)

- Elementklassen haben Zugriff auf alle Objektvariablen und Methoden der äußeren Klasse, inklusive der als `private` deklarierten (und umgekehrt).

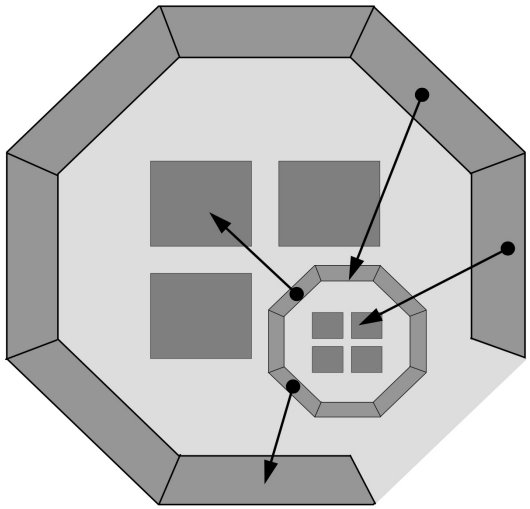


Abbildung übernommen aus „Java als erste Programmiersprache“, Goll & Heinisch

Elementklassen – Regeln (2)

- Elementklassen können ineinander verschachtelt werden.
- Elementklassen können Subklassen von äußeren Klassen sein.
- Ein Exemplar einer Elementklasse ist mit genau einem vorher erschaffenen Exemplar der äußeren Klasse verbunden.
- In einem Exemplar der äußeren Klasse können beliebig viele Exemplare einer Elementklasse „enthalten“ sein.

Elementklassen Beispiel (1)

```java
public class Outer {
    private final String name = "Outer";

    private class Inner {
        private String name;

        private String getQualifiedName() {
            return Outer.this.name + "." + this.name;
        }
    }
    public void createAndPrintInner(String innerName) {
        Inner inner = new Inner();
        inner.name = innerName;
        System.out.println(inner.getQualifiedName());
    }
}
```

```java
public final class InnerClassApplication {
    public static void main(final String[] args) {
        Outer outer = new Outer();
        outer.createAndPrintInner("Inner");
    }
}
```

Ausgabe:

```
Outer.Inner
```

src/at/ac/uibk/pm/innerclasses/memberclass/Outer.java

Elementklassen Beispiel (2)

```java
public class OuterClass {
    private String context = "OuterClass";

    public class InnerClass {
        private String context = "InnerClass";

        public class NestedInnerClass {
            private String context = "NestedInnerClass";

            public void print() {
                System.out.println(context);
                System.out.println(this.context);
                System.out.println(NestedInnerClass.this.context);
                System.out.println(InnerClass.this.context);
                System.out.println(OuterClass.this.context);
            }
        }
    }
}
```

```java
public class NestedInnerClassesApplication {
    public static void main(String[] args) {
        new OuterClass().new InnerClass().new NestedInnerClass().print();
    }
}
```

Ausgabe:

```
NestedInnerClass
NestedInnerClass
NestedInnerClass
InnerClass
OuterClass
```

src/at/ac/uibk/pm/innerclasses/memberclass/OuterClass.java

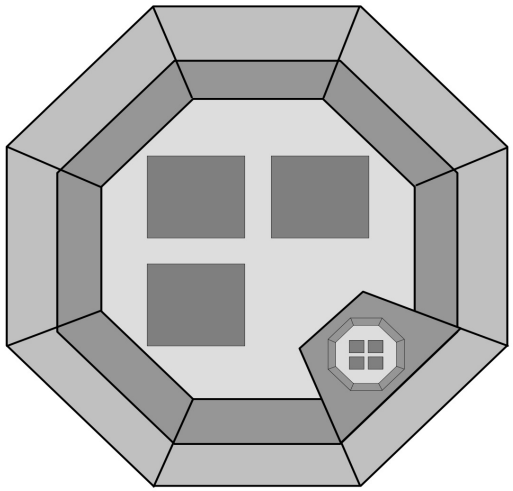
Elementklassen Beispiel (3)

```
public class LinkedList<T> implements Iterable<T> {
    private final Node<T> root = new Node<>(null, null);
    private long modificationCounter = 0;

    private static class Node<T> {
        private final T element;
        private Node<T> next;
        …
    }
    private class LinkedListIterator implements Iterator<T> {
        private Node<T> iteratorNode = root.next;
        private final long iteratorModificationCounter = modificationCounter;

        @Override
        public boolean hasNext() {
            if (modificationCounter != iteratorModificationCounter) {
                throw new ConcurrentModificationException();
            }
            return iteratorNode != null;
        }
        @Override
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            T element = iteratorNode.element;
            iteratorNode = iteratorNode.next;
            return element;
        }
    }
}
```

Elementklasse

Lokale Klassen

- Sind nur im Block gültig, in dem sie deklariert wurden und können daher weder `private`, `protected`, `public` noch `static` sein.
- Sind entweder mit Namen versehen oder anonym.



Objekt einer lokalen Klasse in einer Instanzmethode des umschließenden Objektes.

Outer

```
private static int i;

static void methode1()
{
    
}
static void methode2()
{
   . . . . .
}
```

Objekt einer lokalen Klasse in einer Klassenmethode der umschließenden Klasse.

Abbildung übernommen aus „Java als erste Programmiersprache“, Goll & Heinisch

Lokale Klassen – Regeln

- Lokale Klassen haben Zugriff auf:
  - Alle statischen Variablen und Methoden der äußeren Klasse.
  - Konstruktoren der äußeren Klasse.
  - Mit `final` deklarierte bzw. effektiv finale lokale Variablen oder Parameter in ihrem Gültigkeitsbereich.
- Lokale Klassen in nicht statischen Methoden haben zusätzlich Zugriff auf:
  - Alle Objektvariablen und Methoden der äußeren Klasse.
- Lokale Klassen können bis auf Konstanten (`static final`) keine statischen Variablen oder Methoden besitzen.

Lokale Klasse Beispiel

```
public class OuterClass {
    public static void main(String[] args) {
        final int value1 = 1;
        int value2 = 2;
        int value3 = 3;
        int value4 = 4;
        ++value3;

        class LocalInnerClass {
            int innerValue1 = 10;
            LocalInnerClass() {
                System.out.println(value1);
                System.out.println(value2);
                System.out.println(value3); // Compile-Error!
                value4 += 4; // Compile-Error!
                System.out.println(innerValue1);
            }
        }

        new LocalInnerClass();
        System.out.println(value1 + value2 + value3 + value4);
        System.out.println(innerValue1); // Compile-Error!
    }
}
```

src/at/ac/uibk/pm/innerclasses/localclass/OuterClass.java

Anonyme Klassen

- Sind lokale Klassen ohne Namen.
- Können für die lokale Implementierung von Interfaces und Klassen verwendet werden.
  - Für die Wahrung der Lesbarkeit sollten diese Klassen im Allgemeinen nur wenige Zeilen lang sein.
  - Beispiel: GUI-Programmierung, Comparator
  - In der Regel sollten Lambdas bei Interfaces mit nur einer abstrakten Methode den anonymen Klassen vorgezogen werden.
- Sind im Vergleich zu anderen lokalen Klassen aufgrund ihrer Namenlosigkeit weiter eingeschränkt.

Anonyme Klassen – Regeln (1)

- Deklaration und Anlegen des einzigen zugehörigen Exemplars sind untrennbar miteinander verbunden.
- Die Deklaration erlaubt kein `extends` oder `implements` und erfolgt auf eine der folgenden Arten:
  - Durch

```
… new SuperClassName(…) { … } …
```

    wobei die Superklasse einen Konstruktor haben muss, der eine zu den Argumenten passende Signatur hat.
  - Durch

```
… new InterfaceName() { … } …
```

    wobei die anonyme Klasse das Interface vollständig implementieren muss und direkte Subklasse von Object sein wird.

Anonyme Klassen – Regeln (2)

- Eine anonyme Klasse hat keine explizit deklarierten Konstruktoren.
- Es sind keine `instanceof` und andere Tests, die den Klassennamen benötigen, möglich.
- Zusätzliche Methoden (nicht in der Superklasse bzw. im Interface deklariert) können nicht von außen aufgerufen werden.

Anonyme Klassen Beispiel

```
Collections.sort(shoppingList, new Comparator<Product>() {
    @Override
    public int compare(Product p1, Product p2) {
        return p1.getName().compareTo(p2.getName());
    }
});
```

Quellen

- Joachim Goll, Cornelia Heinisch: **Java als erste Programmiersprache**, Springer Vieweg, 8. Auflage, 2016
- Joshua Bloch: **Effective Java**, Addison-Wesley Professional, 3. Auflage, 2018
- Christian Ullenboom: **Java ist auch eine Insel: Einführung, Ausbildung, Praxis**, Rheinwerk Verlag, 16. Auflage, 2022 (Java 17)