PM2 Java: Innere Klassen

Innere Klassen

Was ist eine innere Klasse?

- Eine **innere** Klasse ist eine nicht statische Klassendefinition, die in eine andere Klassendefinition eingebettet ist. Die einbettende Klasse ist die **äußere** Klasse.
- **Nicht eingebettete** Klassen sind **Toplevel** Klassen.
- Innere Klassen gibt es seit Java 1.1. Sie wurden im Zuge der Umstellung des Eventmodells in der Java GUI-Bibliothek eingeführt.
- Unterschieden werden:
  1. innere Member Klassen
  2. lokale innere Klassen
  3. anonyme innere Klassen

Wozu innere Klassen?

- Bis JDK 1.0 wurden Klassen nur auf Paketebene definiert. **Unhandlich**, wenn eine Klasse nur lokale Bedeutung hatte oder wenn »auf die Schnelle eine kleine Klasse« gebraucht wurde.
- Bei den Erweiterungen für die Programmierung grafischer Oberflächen mit JDK 1.1, wurde ein flexiblerer Mechanismus benötigt. Erforderlich wurde das häufige Schreiben kleiner Programmteile (**Controller**), die nur im einem eng begrenzten Kontext (**GUI Komponente**) eine Bedeutung haben.
- Manchmal benötigt man mehrere Instanzen einer Klasse ***A***, die Zustandsinformation zur einer Klasse ***B*** verwalten und gleichzeitig Zugriff auf die Attribute von ***A*** brauchen. (Beispiel: **Iteratoren**)

Innere Member Klassen

Beispiel 1: *Haus*

- Innere Member Klassen
  1. werden innerhalb einer Klasse, der äußeren Klasse, wie normale Klassen definiert. (Beispiel 1: ***Oeffnung, Tuer, Fenster***)
  2. erlauben auch Klassenhierarchien.
  3. werden in der äußeren Klasse wie normale Klassen verwendet. (Beispiel 1: ***Tuer, Fenster*** Methode ***bauen***).

Beispiel 2: *Haus* und *HausDemo*

- Referenzen auf Objekte innerer Klassen lassen sich mit Hilfe von Objektmethoden erzeugen. (→ Beispiel 2: ***Haus.tuer()***)
- Der Typ-Name der Objekte der inneren Klassen wird gebildet, indem der inneren Klasse der Name der äußeren Klasse vorangestellt wird. Referenzen auf Objekte der inneren Klasse sind auch außerhalb der äußeren Klasse möglich, wenn die Sichtbarkeit der inneren Klasse dieses zulässt. (→ Beispiel 2: ***main*** der ***HausDemo***).

Beispiel 1: Innere Member Klasse

```
public class Haus {
    private Point2D pos;
    private Dimension2D dimension;
    private Point2D relPosTuer;
    private Point2D relPosFenster;
    private Tuer tuer;
    private Fenster fenster;

    // Sichtbarkeit von inneren Memberklassen public, protected, default, private
    public Haus(Point2D pos, Dimension2D dim, Point2D relPosTuer, Point2D relPosFenster) {

    public void bauen() {

    public String toString() {

    public abstract class Oeffnung {

    public class Tuer extends Oeffnung {

    public class Fenster extends Oeffnung {
}
```

innere Member Klassen werden wie normale Klassen definiert. Sie stehen im Body der äußeren Klasse.

Beispiel 1: Innere Member Klasse

```
public class Haus {
    private Point2D pos;
    private Dimension2D dimension;
    private Point2D relPosTuer;
    private Point2D relPosFenster;
    private Tuer tuer;
    private Fenster fenster;

    // Sichtbarkeit von inneren Memberklassen public, protected, default, private
    public Haus(Point2D pos, Dimension2D dim, Point2D relPosTuer, Point2D relPosFenster) {..

    public void bauen() {
        this.tuer = new Tuer(1, 2.2);
        this.fenster = new Fenster(1.9, 1.0);
    }

    public String toString() {..

    public abstract class Oeffnung {..

    public class Tuer extends Oeffnung {..

    public class Fenster extends Oeffnung {..

}
```

innere Member Klassen werden innerhalb der äußeren Klasse wie jede andere Klasse verwendet

Beispiel 2: Referenzen auf Objekte innerer Member Klassen

```
public class Haus {
    private Point2D pos;
    private Dimension2D dimension;
    private Point2D relPosTuer;
    private Point2D relPosFenster;
    private Tuer tuer;
    private Fenster fenster;

    // Sichtbarkeit von inneren Memberklassen public, protected, default, private
    public Haus(Point2D pos, Dimension2D dim, Point2D relPosTuer, Point2D relPosFenster) {

    public void bauen() {
        this.tuer = new Tuer(1, 2.2);
        this.fenster = new Fenster(1.9, 1.0);
    }

    public Tuer tuer(){ return tuer;}
    public Fenster fenster(){return fenster;}

    @Override
    public String toString() {
        return String.format("%s[:pos=[%.2f:%.2f],dim:=[%.2f:%.2f],%s,%s]", getClass().getSimpleName(), pos.getX(),
                pos.getY(), dimension.getWidth(), dimension.getHeight(), tuer, fenster);
    }

    public abstract class Oeffnung {

    public class Tuer extends Oeffnung {

    public class Fenster extends Oeffnung {
}
```

Methoden liefern Referenzen Auf Objekte der inneren Klassen zurück

Beispiel 2: Referenzen auf Objekte innerer Member Klassen

```
public class HausDemo {

    public static void main(String[] args) {
        Haus haus = new Haus(new Point2D(10, 0), new Dimension2D(6,9),
                                    new Point2D(1, 0), new Point2D(2,1.2));
        Haus.Fenster fenster = haus.fenster();
        Haus.Tuer tuer = haus.tuer();
        System.out.println(fenster);
        System.out.println(tuer);
    }

}
```

Referenzen auf die inneren Klassen von *Haus* außerhalb der Klasse *Haus*

Wozu innere Member Klassen?

- Innere Member Klassen haben Zugriff auf alle Attribute und Methoden der äußeren Klasse (auch die privaten Attribute und Methoden).
- Zu inneren Member Klassen lassen sich mehrere Objekte erzeugen und damit verschiedenen Zustände in unterschiedlichen Sichten auf die äußere Klasse modellieren.
- Innere Member Klassen können unabhängig von der äußeren Klasse Interfaces implementieren und Funktionalität zur äußeren Klasse hinzufügen, ohne dass die äußere Klasse das Interface implementieren muss.

Beispiel 3: Iteratoren für die Klasse *Liste<T>*

- Ein Iterator für eine Liste hat u.A. die Methoden ***hasNext()*** und ***next(). hasNext()*** und ***next()*** benötigen beide Zugriff auf private Attribute der Klasse ***Liste<T>***.
- Ein Iterator merkt sich die letzte Leseposition in einem privaten Attribut ***readCursor***.
- Wollen wir wiederholt über eine Liste iterieren, dann benötigen wir immer wieder einen „**frischen**“ Iterator, der am Anfang der Liste startet.
- Da das Iterator Muster auch für andere Klassen außer Listen verwendet wird, gibt Java für dieses Muster ein Interface vor.
- Nur die innere Klasse implementiert das ***Iterator*** Interface.

Beispiel 3: Methoden innerer Klassen sehen private Attribute der äußeren Klasse

```
public class Liste<T> implements Iterable<T> {
    private T[] elements;
    private int writeCursor=0;
    @Override
      public Iterator<T> iterator() {
        return new ListeIterator();
    }

     private class ListeIterator implements Iterator<T> {
          private int readCursor=0;
          @Override
          public boolean hasNext() {
             return readCursor<writeCursor;
          }
          @Override
          public T next() {
               if (!hasNext()) throw new NoSuchElementException();
               return elements[readCursor++];
          }
          @Override
          public void remove() {
               throw new UnsupportedOperationException();
          }
     }
}
```

Innere Klasse ListeIterator von Liste<T>. Nur diese implementiert das Iterator<T> Interface.

Zugriff auf private Attribute der äußeren Klasse Liste<T>.

Beispiel 3: Mehrfaches Iterieren über Listen mit Iterator Objekten

```
Liste<Integer> li = new Liste<Integer>();
li.add(4);li.add(7);li.add(1);li.add(13);li.add(8);li.add(5);
Iterator<Integer> iiter1 = li.iterator();
Iterator<Integer> iiter2 = li.iterator();
Integer nextI;

// gebe alle geraden Zahlen der Liste aus, bis eine Zahl >= 7 gelesen wird.
while(iiter1.hasNext()){
     if ((nextI = iiter1.next()) %2 == 0) p(nextI);
     if (nextI >= 7) break;
}
// Gebe alle geraden Zahlen der Liste aus.
// Ausgabe ist: 8
p("Mit demselben Iterator");
while(iiter1.hasNext())
    if ((nextI = iiter1.next()) %2 == 0) p(nextI);
// Gebe alle geraden Zahlen der Liste aus.
// Ausgabe ist: 4 8
p("Mit dem zweiten Iterator");
while(iiter2.hasNext())
    if ((nextI = iiter2.next()) %2 == 0) p(nextI);
```

zwei unabhängige Iteratorinstanzen für die Liste li.

```
4
Mit demselben Iterator
8
Mit dem zweiten Iterator
4
8
```

wiederholtes Iterieren mit iiter1

Iterieren mit iiter2 startet am Anfang der Liste

Innere Klassen merken sich das Objekt der äußeren Klasse, das die Instanzen der inneren Klasse erzeugt.

- Objekte der inneren Klassen haben Zugriff auf die privaten Attribute und Methoden der äußeren Klasse.
- Das Objekt der äußeren erzeugenden Klasse kann explizit angesprochen werden durch ***<NameDerÄußerenKlasse>.this***.
- Da ein Objekt einer inneren Klasse nur durch ein Objekt der äußeren Klasse erzeugt werden kann, können **innere Member Klassen nicht statisch** sein.

Beispiel 3: (cont.)

- Wir implementieren die ***remove*** Methode von ***ListeIterator***:
  - ***remove*** in ***ListeIterator*** führt das Löschen von Elementen auf die ***remove*** Methode der ***Liste*** zurück.
  - Dazu benötigen wir eine Referenz auf das äußere Objekt: ***Liste.this***

ListeIterator: Zugriff auf das Objekt der äußeren Klasse.

```
public class Liste<T> implements Iterable<T> {
      ...
      private class ListeIterator implements Iterator<T> {
            private int readCursor=0;
            private boolean nextGiven = false;

            public boolean hasNext() { …}

            public T next() {
               if (!hasNext()) throw new NoSuchElementException();
               nextGiven = true;
               return elements[readCursor++];
            }

            public void remove() {
                  if (!nextGiven) throw new IllegalStateException();
                  Liste.this.remove(--readCursor);
                  nextGiven = false;
            }
      }
}
```

merkt sich, ob mit *next* gelesen wurde.

Liste.this.remove
lösche das Element an der letzten gelesenen Position im Objekt der äußeren Klasse.
Liste wird kürzer ➔ *--readCursor*

Liste<T>.remove im Objekt der äußeren Klasse.

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

    private T[] elements;
    private int writeCursor=0;
    private int capacity;
    ...
    public T remove(int index) {
        T toRemove = elements[index];

        System.arraycopy(elements, index+1, elements, index, writeCursor-index-1);

        writeCursor--;

        elements[writeCursor]= null;
        return toRemove;
    }
    @Override
    public Iterator<T> iterator() {

        return new ListeIterator();
    }
}
```

Löschen durch Umkopieren von index +1 auf index

Liste wird kürzer ➔ writeCursor --

Liefert eine Referenz auf ein Objekt der inneren Iteratorklasse

Iterator.remove in Aktion

li : Das Objekt der äußeren Klasse

```
Liste<Integer> li = new Liste<Integer>();
li.add(4);li.add(7);li.add(1);li.add(13);li.add(8);li.add(5);
Iterator<Integer> iiter = li.iterator();
```

iiter: Das Objekt der inneren Klasse.

```
// löscht alle ungeraden Zahlen aus li mit Hilfe des Iterators
while (iiter.hasNext()){
     Integer nextI = iiter.next();
     p("next " + nextI);
     if (nextI%2 !=0){
          iiter.remove();
          p("removed " + nextI);
     }
}
```

remove auf *iiter* ruft *remove(i)* auf *li* dem Objekt der äußeren Klasse auf.

```
next 4
next 7
removed 7
next 1
removed 1
next 13
removed 13
next 8
next 5
removed 5
```

Erzeugen von Objekten innerer Member Klassen

- Sind innere Member Klassen nicht private, dann können auch außerhalb der äußeren Klasse Objekte dieser Klassen erzeugt werden.

- Dazu ist eine gesonderte Syntax für den ***new*** Operator notwendig. Das ***new*** bezieht sich auf das Objekt der äußeren Klasse.

```
public class DotNew {
    public class Inner{}
}

public class DotNewDemo {
     public static void main(String[] args) {

          DotNew dotNew = new DotNew();

          DotNew.Inner dotNewInner = dotNew.new Inner();
     }
}
```

Objekt der äußeren Klasse

Objekt der inneren Klasse.
gehört zum Objekt *dotNew*.

Schachtelung innerer Member Klassen

```
public class Outer {
      String name = "class Outer";
      public class Inner {
      String name = "class Inner";

            public class DeepInner {
                  String name = "class DeepInner";
                  public void printNames() {
                        p("\nDeepInner >> printNames ");
                        p(name); // class DeepInner
                        p(this.name); // class DeepInner
                        p(DeepInner.this.name); // class DeepInner
                        p(Inner.this.name); // class Inner
                        p(Outer.this.name); // class Outer
                        p("DeepInner >> printNames() -- DONE");
                  }
            }
            public void test3() {
                  p("\nInner >> test3()");
                  DeepInner di = new DeepInner();
                  di.printNames();
            }
      } … }
```

Innere Member Klassen können in inneren Member Klassen eingebettet sein.

Schachtelung innerer Member Klassen

```
public class Outer { ...
     public static void main(String[] args) {
          Outer outer = new Outer();
          outer.test1();
          outer.test2();
          Inner inner = outer.new Inner();
          inner.test3();
     }
     private void test1() {
          p("\nOuter >> test1()");
          Outer outer = new Outer();
          Inner inner = new Inner();
          inner.test3();
     }
     private void test2() {
          p("\nOuter >> test2()");
          Outer outer = new Outer();
          Inner inner = new Inner();
          Outer.Inner inner2 = outer.new Inner();
          Outer.Inner.DeepInner deep = inner2.new DeepInner();
          deep.printNames();
     }

}
```

```
Outer >> test1()
Inner >> test3()
DeepInner >> printNames
class DeepInner
class DeepInner
class DeepInner
class Inner
class Outer
DeepInner >> printNames() -- DONE

Outer >> test2()
DeepInner >> printNames
class DeepInner
class DeepInner
class DeepInner
class Inner
class Outer
DeepInner >> printNames() -- DONE

Inner >> test3()
DeepInner >> printNames
class DeepInner
class DeepInner
class DeepInner
class Inner
class Outer
DeepInner >> printNames() -- DONE
```

Lokale innere Klassen

- Lokale innere Klassen können in einer Methode oder in einem Block innerhalb einer Methode definiert werden.
- Sie sind dann nur in der Methode bzw. im Block erreichbar.
- Zum Selbststudium – nicht prüfungsrelevant.

Anonyme innere Klassen

- Sind eine Kombination aus
  - Definition einer lokalen Klasse
  - plus Instanziierung der lokalen Klasse
- Sind Klassen ohne Namen und daher auch nicht über den Klassennamen referenzierbar.
- Können auf Grundlage einer bestehenden Klasse oder auf Basis einer Interfaces erzeugt werden.
- Können Initialisierungen von eigenen Attributen vornehmen:
  - durch Übernahme von Werten ihres Definitions-Kontextes
  - in Instanz-Initialisierern

- **Schema**:

```
new Class / Interface() {
        Attribute
        Methoden
}
```

Anonyme innere Klassen zu einer abstrakten Klasse

```
public abstract class Anonymous {
      String name = "Anonymous";
      public Anonymous() { }
      public Anonymous(String arg) { name = arg; }
      abstract void printName(String arg);
}
public class AnonymousDemo {
      public static void main(String[] args) {
      Anonymous an1 = new Anonymous() {
            void printName(String name) {
               p(this.name + "   " + name);
            }
      };
      an1.printName("aName");
      Anonymous an2 = new Anonymous ("FirstName") {
            String innerName = "innerName";
            void printName(String arg) {
               p(name + "   "+arg+ "   " + innerName);
            }
      };
      an2.printName("SecondName");
      }
}
```

Instanzen der anonymen inneren Klassen werden
a) mit dem Default Konstruktor
b) mit einem Custom Konstruktor erzeugt.

anonyme innere Klassen zu *Anonymous*. Müssen die abstrakte Methode *printName* implementieren.

```
Anonymous   aName
FirstName   SecondName   innerName
```

Anonyme innere Klassen zu einem Interface

```
public interface AnInterface {
      void iMethod1(String arg);
      void iMethod2();
}
public class AnonymousInterfaceDemo {
      public static void main(String[] args) {

            AnInterface an3 = new AnInterface() {
                  @Override
                  public void iMethod1(String name) {
                     p("iMethod1 " + name);
                  }
                  @Override
                  public void iMethod2() {
                     p("iMethod2");
                  }
            };

            an3.iMethod1("Interface");
            an3.iMethod2();
      }
}
```

Erzeugt eine Instanz der anonymen inneren Klasse.

anonyme innere Klasse zum Interface *AnInterface*. Muss die Interface Methoden implementieren.

```
iMethod1 Interface
iMethod2
```

Initialisierung von Attributen einer anonymen inneren Klasse

```
public interface Ziel {
    String getOrt();
}

public class Paket5 {
     public Ziel ziel(final String
        wohin){
          return new Ziel(){
               private String ort=wohin;
               public String getOrt() {
               return ort;
          }
        };
    }

     public static void main(String[]
        args) {
          Paket5 p5 = new Paket5();
          Ziel ziel5 = p5.ziel("Accra");
          p(ziel5.getOrt());
     }
}
```

durch Werte des Definitionskontextes

- Anonyme innere Klassen können eigene Attribute durch Werte des Definitionskontextes initialisieren.
- Lokale Variablen des Kontextes müssen mit ***final*** gekennzeichnet werden, wenn diese in der anonymen Klasse direkt verwendet werden.

Initialisierung von Attributen einer anonymen inneren Klasse

```
public class Paket6 {
     public Ziel ziel(final String
        wohin,final float preis){
          return new Ziel(){
               private int kosten;
               private String ort =
                  wohin;
               { kosten = (int) preis;
                 if (kosten > 100) {
                 p("Nachverhandeln");
                 }
               }
          @Override
          public String getOrt() {return
             ort;}
          };
     }
     public static void main(String[]
        args) {
          Paket6 p6 = new Paket6();
          Ziel ziel6 = p6.ziel("Bremen",
             103.44f);
     }
}
```

In Instanz-Initialisieren

- Anonyme innere Klassen können eigene Attribute in Instanz-Initialisieren setzen, wenn ihnen für die Initialisierung kein Konstruktor zur Verfügung steht.
- Lokale Variablen des Kontextes müssen mit ***final*** gekennzeichnet werden, wenn diese in der anonymen Klasse direkt verwendet werden.

Zusammenfassung Innere Klassen

Top-Level Klassen		s. java-Doku
Innere Klassen	**Member Klassen**	• Klassen sind Instanzen in einer anderen Klasse. • Die Klasseninstanzen „kennen“ die äußere Klasse über eine interne Instanzvariable. • Instanz der äußeren Klasse ist sichtbar. • Keine statischen Instanzen erlaubt • Erweiterte Syntax für **new, this, super**
	Lokale Klassen	• ausgelassen
	Anonyme Klassen	• Klassen **ohne Namen, definiert innerhalb eines Ausdrucks.** • haben keine Konstruktoren. • Instanz der äußeren Klasse ist sichtbar. • Lokale Variable der äußeren Klasse sind benutzbar, sofern sie **final definiert sind** • Singletons von Klassen. • Initialisierung u.A. in Instanz Initialisierern

Identifikatoren für innere Klassen

- Der Javacompiler erzeugt für jeden Objektyp (Klasse / Interface) eine ***.class*** Datei.

- Auch für innere Klassen werden ***.class*** Dateien nach folgenden Schema erzeugt.

  <Äußere Klasse>$<Innere Klasse Level1>$ ... $<Innere Klasse Leveln>

- Für anonyme inneren Klassen und lokale Klassen generiert der Compiler aufsteigende Nummern.

Innere Member Klassen:
Liste.class
Liste$ListeIterator.class

Innere lokale Klassen:
Haus.class
Haus$Oeffnung.class
Haus$Fenster.class
Haus$Tuer.class

Innere anonyme Klassen:
AnonymousDemo.class
AnonymousDemo$1.class
AnonymousDemo$2.class

Übungen:

- Implementieren Sie den Iterator für ***Liste<T>*** als anonyme innere Klasse.
- Gegeben eine ***ArrayList<Person>***. Eine ***Person*** hat Name und Vorname und eine ***Adresse***. Die Adresse hat Strasse, Hausnummer, PLZ und Ort. Alle Attribute außer Adresse sind vom Typ ***String.***
  - Sortieren Sie die Liste in der natürlichen Ordnung für Personen.
    - Die natürliche Ordnung für Personen ergibt sich durch den Vergleich von Name, Vorname, Adresse in dieser Reihenfolge.
    - Die natürliche Ordnung von Adressen ergibt sich durch den Vergleich von Strasse, Hausnummer, PLZ und Ort in dieser Reihenfolge.
  - Sortieren Sie die Liste für Personen in der natürlichen Reihenfolge absteigend.
  - Sortieren Sie die Liste der Personen anhand der Adressen.
  - Verwenden Sie für die 2'te und 3'te Sortierung anonyme Klassen, die das Interface ***Comparator*** implementieren.