Weiterführende Konzepte des Programmierens mit Java

Verschachtelte, lokale, anonyme Klassen und Aufzählungstpyen

HOCHSCHULE DER MEDIEN

Prof. Dr. Peter Thies
Prof. Dr. Christian Rathke
Hochschule der Medien (HdM)

{thies|rathke}@hdm-stuttgart.de
http://www.hdm-stuttgart.de/~rathke
http://www.prof-thies.de/

Verschachtelte Klassen

- In Java können Klassen innerhalb einer anderen Klasse definiert werden.
- Solche Klassen werden verschachtelte Klassen (*nested classes)* genannt und sehen so aus:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

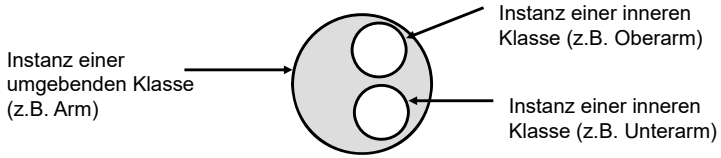
- Eine verschachtelte Klasse ist ein *Member* ihrer umgebenden Klasse und hat als solche Zugriff auf deren andere Members (Variablen und Methoden).
- Die verschachtelte Klasse kann als `private, public, protected` oder `package-private` erklärt werden.
- **Definition**: Eine verschachtelte Klasse ist eine Klasse, die Member einer anderen Klasse ist.

Warum Verschachtelte Klassen?

- Man verwendet verschachtelte (oder auch: *innere*) Klassen, um die Zusammengehörigkeit von Objekten dieser Klassen darzustellen oder zu erzwingen.
- Man sollte Klassen *innerhalb* einer anderen Klasse nur dann verwenden, wenn ein Objekt der inneren Klasse nur zusammen mit einem Objekt der äußern Klasse einen Sinn ergibt oder wenn die Existenz des inneren von der äußeren Objekts abhängt.
- Beispiel: die Klassen „Unterarm“ und „Oberarm“ als Bestandteile (Members) der Klasse „Arm“ beschreiben die Zusammengehörigkeit von den drei Objekten der jeweiligen Klasse.
- Code-Elemente der inneren Klasse(n) haben uneingeschränkten Zugriff auf die Members der äußeren Klasse, sogar wenn diese als `private` deklariert sind (dies ist aber konsistent mit der sonstigen Verwendung von `private`).

Innere Klassen

- Instanzen innerer Klassen sind wie eine Instanzen-Methode oder eine Instanzen-Variable *mit einer Instanz* ihrer umgebenden Klasse verbunden.
- Sie hat direkten Zugriff auf die Instanzen-Variablen und –Methoden dieses Objekts.
- Wegen ihrer Beziehung zu einer Instanz, kann eine innere Klasse keine statischen Members, d.h Klassen-Variable oder Klassen-Methoden, enthalten.
- Der Begriff "verschachtelte Klasse" bezieht sich dabei auf eine syntaktische Eigenschaft, d.h. der Code der einen befindet sich innerhalb des Codes der anderen.
- Der Begriff "innere Klasse" bezieht sich auf eine Beziehung zwischen den Instanzen der beteiligten Klassen:



Innere Klassen (fortg.)

- **Definition:** Eine *innere Klasse* ist eine verschachtelte Klasse, deren Instanzen innerhalb der Instanzen ihrer umgebenden Klasse existieren.
- Die Instanzen der inneren Klasse haben direkten Zugriff auf die Instanzen-Members ihrer umgebenden Instanz (z.B. wenn ihre Instanzenvariablen initialisiert werden).
- Um eine innere Klasse zu instantiieren, muss man zunächst die äußere Klasse instantiieren. Dann kann mit der folgenden Syntax ein inneres Objekt erzeugen:

```
OuterClass.InnerClass innerObject =
            outerObject.new InnerClass();
```

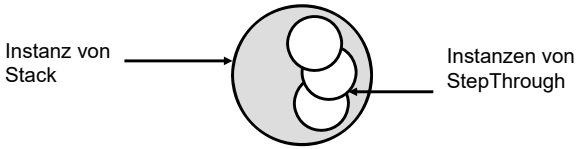

Innere Klassen: Die Definition eines Stapels (Stack) als Anwendungsbeispiel

- Ein Stack bzw. Stapel ist eine Datenstruktur, bei der man nur oben eine neues Element hinzufügen bzw. nur das oben liegende entfernen kann.
- Beispiele: Papierstapel, Tellerstapel, Kistenstapel
- Im Beispiel wird ein `StackOfInts` mit Hilfe eines Felds (Arrays) und den folgenden Methoden implementiert:
  - eine Methode namens `push` für das Hinzufügen eines Elements (einer Zahl),
  - eine Methode namens `pop` für das Entfernen eines Elements und
  - eine Methode namens `isEmpty` zum Testen, ob der Stack noch Elemente enthält.


```
public class StackOfInts {

    private int[] data;
    private int next = 0;  // index of last item in stack + 1

    public StackOfInts(int size) {
        //create an array large enough to hold the stack
        data = new int[size];
    }

    public void push(int on) {
        if (next < data.length)
           data[next++] = on;
    }
    public boolean isEmpty() {
        return (next == 0);
    }
    public int pop(){
        if (!isEmpty())
           return data[--next]; // top item on stack
        else
           return 0;
    }
    public int getStackSize() {
        return next;
    }
...
```


Anwendungsbeispiel (fortg.)

- Die Klasse StackOfInts besitzt eine innere Klasse namens *StepThrough*, mit deren Hilfe die Datenstruktur "abgewandert" werden kann.
- Dazu definiert die innere Klasse die folgenden Members:
  - einen Zähler in Form der Variable *i* zum Merken der aktuellen Position; diese wird mit 0 initialisiert;
  - die Methode *increment* zum Weiterschalten der Position,
  - die Methode *current* für den Lesezugriff auf das Element an der aktuellen Position,
  - die Methode *isLast* zum Abprüfen, ob das letzte Element erreicht wurde.



```
    …
    private class StepThrough {
          // start stepping through at i=0
          private int i = 0;

          // increment index
          public void increment() {
               if ( i < data.length) i++;
          }

          // retrieve current element
          public int current() {
               return data[i];
          }

          // last element on stack?
          public boolean isLast(){
               if (i == getStackSize() - 1)
                  return true;
               else
                  return false;
          }
    }

    …
```


Anwendungsbeispiel (fortg.)

- Die nachfolgende *main*-Methode enthält ein Anwendungsbeispiel.
- Beim "Abwandern" (Iterieren) der Datenstruktur in der while-Schleife wird
  - geprüft, ob man am letzten Element angekommen ist,
  - andernfalls wird das Element "konsumiert" und
  - das nachfolgende Element betrachtet.
- Das folgende Programm (die main-Methode)
  - instantiiert die Klasse `StackOfInts` (`stackOne`)
  - füllt die Instanz mit Integers (0, 2, 4, usw.),
  - erzeugt ein `StepThrough`-Objekt und
  - gibt mit dessen Hilfe den Inhalt von `stackOne` aus.
- Die Ausgabe des Programms ist:

  `0 2 4 6 8 10 12 14 16 18 20 22 24 26`


```
    …

    public static void main(String[] args) {

         // instantiate outer class as "stackOne"
         StackOfInts stackOne = new StackOfInts(15);

         // populate stackOne
         for (int j = 0 ; j < 15 ; j++) {
              stackOne.push(2*j);
         }

         // instantiate inner class as "iterator"
         StepThrough iterator = stackOne.new StepThrough();

         // print out stackOne[i], on one line
         while(!iterator.isLast()) {
              System.out.print(iterator.current() + " ");
              iterator.increment();
         }
         // enf of line
         System.out.println();

    }
}
```


Lokale und anonyme innere Klassen

- Es existieren zwei weitere Typen Innerer Klassen:
  1. als *lokale, innere* Klassen innerhalb von Methoden
  2. als *anonyme, innere* Klassen.

Zugriff und Sichtbarkeit

- Zugriff und Sichtbarkeit werden für innere Klasse genauso angegeben wie für die anderen Members der äußeren Klasse, also z.B. `private, public und protected.`

Lokale Klassen

- … sind Klassen, die innerhalb eines Code-Blocks definiert werden, z.B. in einem Methodenkörper
- … haben Zugriff auf die Members der „umgebenden“ Klasse
- zusätzlich können sie auf die lokalen Variablen oder die Methodenparameter des „umgebenden“ Code-Blocks zugreifen, die als „final“ deklariert sind.
- Wie innere Klassen können lokale Klassen keine statischen Members haben, da sie auf Instanzenvariablen zugreifen können.
- Interfaces können *nicht* innerhalb eines Code-Blocks definiert werden; sie sind inhärent statisch.

Beispiel für eine lokale Klasse

```
public class LocalClassExample {
    static String regularExpression = "[^0-9]";

    public static void validatePhoneNumber(
        String phoneNumber1, String phoneNumber2) {

           final int numberLength = 10;

           class PhoneNumber {
             String formattedPhoneNumber = null;
             PhoneNumber(String phoneNumber){
                String currentNumber
                   = phoneNumber.replaceAll(regularExpression, "");
                if (currentNumber.length() == numberLength)
                    formattedPhoneNumber = currentNumber;
                else
                    formattedPhoneNumber = null;
            }

            public String getNumber() {
                return formattedPhoneNumber;
            }
          }

          …
```


Beispiel für eine lokale Klasse (fortg.)

```
…
        PhoneNumber myNumber1 = new PhoneNumber(phoneNumber1);
        PhoneNumber myNumber2 = new PhoneNumber(phoneNumber2);

        if (myNumber1.getNumber() == null)
            System.out.println("First number is invalid");
        else
            System.out.println("First number is " + myNumber1.getNumber());
        if (myNumber2.getNumber() == null)
            System.out.println("Second number is invalid");
        else
            System.out.println("Second number is " + myNumber2.getNumber());

    }

    public static void main(String... args) {
        validatePhoneNumber("123-456-7890", "456-7890");
    }
}
```


Anonyme Klassen

- erlauben kompakteren Code;
- erlauben es, Klassen gleichzeitig zu definieren und zu instantiieren.
- Sie sind wie lokale Klassen ohne Namen und können nur einmalig zur Objekterzeugung verwendet werden.
- Das folgende Beispiel verwendet anonyme Klassen in den Initialisierungsanweisungen der Variablen „frenchGreeting“ und „spanishGreeting“ und eine lokale Klasse bei der Initialisierung von „englishGreeting“.

Beispiel für (lokale und) anonyme Klassen

```
public class HelloWorldAnonymousClasses {

    interface HelloWorld {
        public void greet();
        public void greetSomeone(String someone);
    }

    public void sayHello() {

        class EnglishGreeting implements HelloWorld {
            String name = "world";
            public void greet() {
                greetSomeone("world");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hello " + name);
            }
        }

        HelloWorld englishGreeting = new EnglishGreeting();

…
```


Beispiel für anonyme Klassen

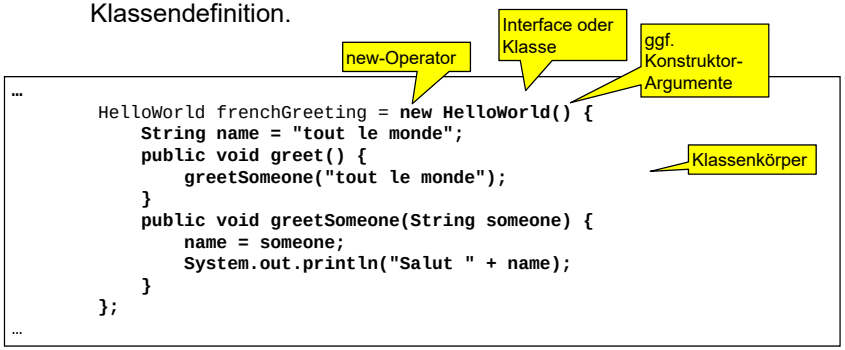
```
…
        HelloWorld frenchGreeting = new HelloWorld() {
            String name = "tout le monde";
            public void greet() {
                greetSomeone("tout le monde");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Salut " + name);
            }
        };

        HelloWorld spanishGreeting = new HelloWorld() {
            String name = "mundo";
            public void greet() {
                greetSomeone("mundo");
            }
            public void greetSomeone(String someone) {
                name = someone;
                System.out.println("Hola, " + name);
            }
        };
}
```


Beispiel für anonyme Klassen (fortg.)

```
…
        englishGreeting.greet();
        frenchGreeting.greetSomeone("Fred");
        spanishGreeting.greet();
    }

    public static void main(String... args) {
        HelloWorldAnonymousClasses myApp =
            new HelloWorldAnonymousClasses();
        myApp.sayHello();
    }
}
```


Syntax für anonyme Klassen

- Eine anonyme Klasse ist ein *Ausdruck!*, d.h. notwendigerweise Teil einer Anweisung.
- Er ist wie der Aufruf eines Konstruktors mit einer Klassendefinition.



Zugriff auf Variable umgebender Programmteile

- Eine anonyme Klasse kann of die Members der umschließende Klasse zugreifen.
- Eine anonyme Klasse kann nur auf *finale* lokale Variable zugreifen.
- Es können keine Interfaces in einer anonymen Klasse als Members definiert werden.
- Es können keine statischen Members in einer anonymen Klasse definiert werden.
- Anonyme Klassen finden oft als Implementierungen sog. Event-Handlers in graphischen Benutzungsschnittstellen Anwendung.

Aufzählungsdatentypen

- Ein *Aufzählungsdatentyp* ist ein Datentyp, dessen Werte aus einer Menge festgelegter *Konstanten* bestehen. D.h. der Datentyp ist durch die Aufzählung seiner Werte bestimmt.
- Beispiel: der Datentyp „Himmelsrichtung“ besteht aus den Werten `Norden, Sueden, Westen, Osten`)
- Die Angabe der Werte erfolgt im Körper der Datendefinition. Sie sind durch Kommas voneinander getrennt.
- Beispiel: Datentyp „Wochentag“ besteht aus den Tagen der Woche

anstelle von `class`

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

- Für solche Typendefinitionen wird das Schlüsselwort **`enum`** verwendet.

Beispiel für die Verwendung des Aufzählungstyps `Day`

```
public class EnumTest {
     Day day;

     public EnumTest(Day day) {
          this.day = day;
     }

     public void tellItLikeItIs() {
          switch (day) {
               case MONDAY: System.out.println("Mondays are bad.");
                            break;

               case FRIDAY: System.out.println("Fridays are better.");
                            break;

               case SATURDAY:
               case SUNDAY: System.out.println("Weekends are best.");
                            break;

               default:    System.out.println("Midweek days are so-so.");
                            break;
          }
     }
...
```


Beispiel für die Verwendung des Aufzählungstyps `Day` (fortg.)

```
...
     public static void main(String[] args) {
          EnumTest firstDay = new EnumTest(Day.MONDAY);
          firstDay.tellItLikeItIs();
          EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
          thirdDay.tellItLikeItIs();
          EnumTest fifthDay = new EnumTest(Day.FRIDAY);
          fifthDay.tellItLikeItIs();
          EnumTest sixthDay = new EnumTest(Day.SATURDAY);
          sixthDay.tellItLikeItIs();
          EnumTest seventhDay = new EnumTest(Day.SUNDAY);
          seventhDay.tellItLikeItIs();


     }
}
```

```
Mondays are bad.
Midweek days are so-so.
Fridays are better
Weekends are best
Weekends are best.
```

Aufzählungstypen (fortg.)

- Intern erzeugt Java für einen Aufzählungstyp eine Klasse gleichen Namens als Erweiterung von **`java.lang.Enum`** mit den Konstanten als Instanzen dieser Klasse.
- Daher kann ein solcher Typ auch keine Erweiterung eines anderen Typs (einer anderen Klasse) darstellen (Einfachvererbung!), d.h. *extends* kann in Verbindung mit Aufzählungstypen nicht (!) verwendet werden.
- Die Konstanten werden zusätzlich als "Klassenvariable" mit sich selbst als Werte eingerichtet und sind z.B. über **`Day.MONDAY`** verwendbar.
- Zusätzlich wird automatisch die Klassenmethode **`values()`** zur Verfügung gestellt, die als Ergebnis ein Feld bestehend aus den Konstanten erzeugt.

```
for (Planet p : Planet.values()) {
    System.out.printf("Your weight on %s is %f%n",
                      p, p.surfaceWeight(mass));
}
```


Aufzählungstypen (fortg.)

- Aufzählungstypen können mit einer *speziellen Form von Initialisierungen* und sowohl Konstruktoren als auch Methoden definiert werden.
- Der Konstruktor wird dann automatisch beim Erzeugen der einzelnen Konstanten mit „den Initialisierungen“ als Argumente aufgerufen.
- Außerdem können zusätzlich „normale“ Member-Variable definiert werden. Dann muss die Liste der Konstanten mit einem Strichpunkt enden.
- Im folgenden Beispiel wird der Enumerator **`Planet`** definiert.
- In seiner main-Methode berechnet er ein in der Kommandozeile angegebenes Gewicht auf der Erde für alle Planeten:

Beispiel: Enumerator Planet

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO   (1.27e+22,  1.137e6);

    private final double mass;   // in kilograms
    private final double radius; // in meters

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

 ...
```


Beispiel: Enumerator Planet (fortg.)

```
...
    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
    public static void main(String[] args) {
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
           System.out.printf("Your weight on %s is %f%n",
                             p, p.surfaceWeight(mass));
    }
}
```

Ergänzungen

Statische, verschachtelte Klassen

- Wie andere Members können auch verschachtelte Klassen als `static` deklariert werden. Dann heißen sie *statische, verschachtelte Klassen.*

```
class OuterClass {
     ...
     static class StaticNestedClass {
          ...
     }
     class InnerClass {
          ...
     }
}
```

- Ein statische, verschachtelte Klasse ist mit ihrer umgebenden Klasse verbunden und kann (wie Klassenmethoden) nicht direkt auf Instanzen-Variablen oder Instanzen-Methoden zugreifen.
- Statische, verschachtelte Klassen werden über die umschließende Klasse angesprochen: `OuterClass.StaticNestedClass`

```
OuterClass.StaticNestedClass nestedObject =
                              new OuterClass.StaticNestedClass();
```

- Nicht-statische verschachtelte Klassen heißen *innere Klassen.*

Innere Klassen: Beispiel

```
class Outer{
  String name;
  int number;

  class Inner {
    private String name;
    private String getQualifiedName() {
      return number + ":" + Outer.this.name + "." + name;
    }
  }

  public void createAndPrintInner(String iname) {
    Inner inner = new Inner();
    inner.name = iname;
    System.out.println(inner.getQualifiedName());
  }
}
```

```
public class InnerClassDemo {
  public static void main(String[] args) {
    Outer outer = new Outer();
    outer.name = "Outer";
    outer.number = 77;
    outer.createAndPrintInner("Inner");
  }
}
```


Innere Klassen: Beispiel (fortg.)

- Bei Verwendung von einfachen Variablennamen (ohne Vorsatz) wird nach lexikalischen Sichtbarkeitsregeln bestimmt, welche Variable gemeint ist.
- In der Methode `getQualifiedName` gehört
  - `name` gehört zur Instanz von `Inner`
  - `number` gehört zur Instanz von `Outer`
- Mit vorangestelltem Klassennamen und `this` kann auf die Instanz der so bezeichneten Klasse Bezug genommen werden:
  - `Outer.this.name` gehört zur Instanz von `Outer`.