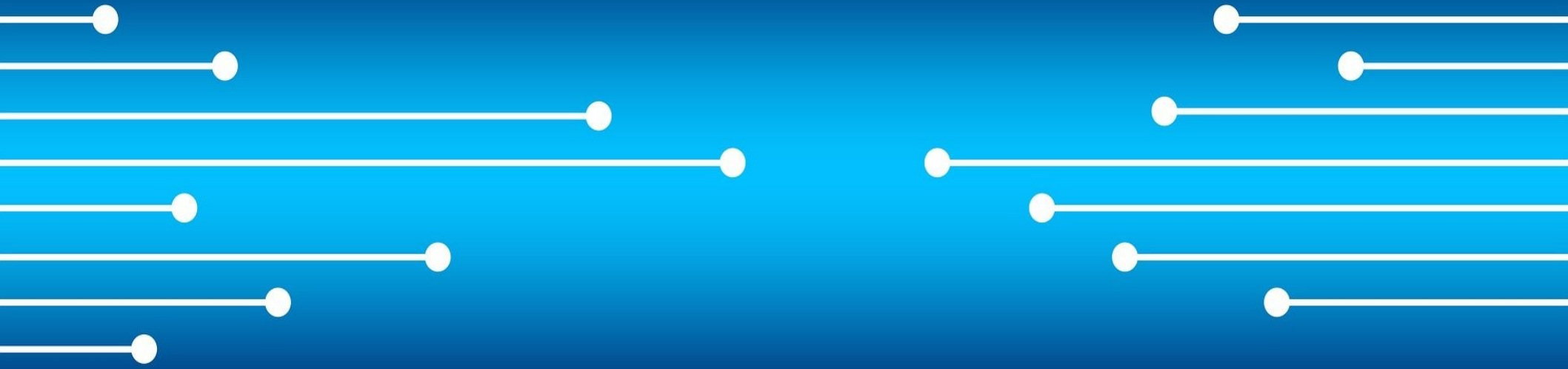
Datenverarbeitung

Teil des Moduls 5CS-DPDL-20



Prof. Dr. Deweß

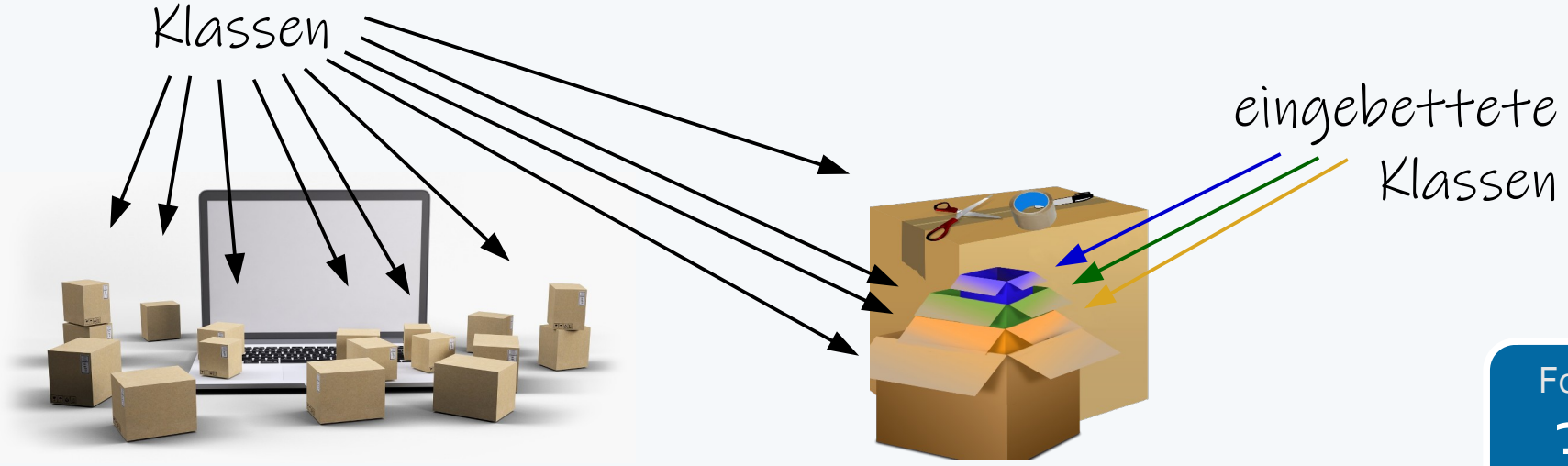
Thema 2

Klassen, Schnittstellen und Objekte für Java-Profis

- Klassen und Schnittstellen in Klassen und Schnittstellen
- Konstruktion von Objekten mittels „Java Builder Pattern“

Klassen in Klassen

- in Java kann man eine Klasse X in einer Klasse Y definieren
- diese Klasse X heißt dann „eingebettet“ bzw. „geschachtelt“ (engl. „nested“)
- Einbettung kann über mehrere Ebenen gehen
- **Anmerkung:** Wir haben uns bisher nur „Top-Level-Klassen“, also objekt- und methodenunabängige Klassen, angeschaut. „Top-Level-Klasse“ ist aber nicht das Gegenstück zu einer eingebetteten Klasse, da manche eingebetteten Klassen auch wie „Top-Level-Klassen“ sind.

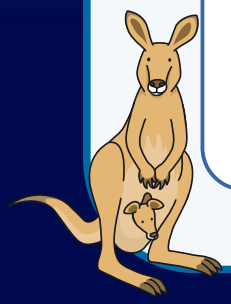


Eingebettete Klassen – Zweck?

- Logik: wenn eine Klasse X nur zusammen mit einer anderen Klasse Y benötigt wird
  (Hauptanwendung: Ereignissteuerung bei graphischen Benutzerschnittstellen)

```
class Kangaroo {

  ...

  class BabyKangaroo {
    ...
  }

}
```

```
class MyFrame extends JFrame {

  ...

  private class ClickListener
    implements ActionListener {
    ...
  }

}
```



Eingebettete Klassen – Zweck?

- Logik: wenn eine Klasse X nur zusammen mit einer anderen Klasse Y benötigt wird (Hauptanwendung: Ereignissteuerung bei graphischen Benutzerschnittstellen)
- Sichtbarkeit/Kapselung:
  - wenn speziell eine Klasse X auf private Variablen oder Methoden einer anderen Klasse Y zugreifen muss, wohingegen sonst kein Zugriff darauf erfolgen soll, dann kann man Klasse X in Y einbetten und so diesen speziellen Zugriff ermöglichen
  - wenn verborgen werden soll, dass eine Klasse X innerhalb einer Klasse Y benötigt wird, dann kann man X in Y einbetten und so vor äußeren Zugriffen schützen

```
class EnclosingClass {
  private int enclosingValue = 1;
  ...
  class InnerClass {
    private int innerClassValue;
    ...
  }
}
```

Zugriff auf „enclosingValue" durch „InnerClass" möglich („InnerClass" ist bzgl. „EnclosingClass" klassenintern)

[aber „EnclosingClass" kann nicht direkt auf „innerClassValue" zugreifen, da „EnclosingClass" bzgl. „InnerClass" nicht klassenintern ist; aber über ein Objekt geht das trotzdem]

Eingebettete Klassen – Zweck?

- Logik: wenn eine Klasse X nur zusammen mit einer anderen Klasse Y benötigt wird (Hauptanwendung: Ereignissteuerung bei graphischen Benutzerschnittstellen)
- Sichtbarkeit/Kapselung:
  - wenn speziell eine Klasse X auf private Variablen oder Methoden einer anderen Klasse Y zugreifen muss, wohingegen sonst kein Zugriff darauf erfolgen soll, dann kann man Klasse X in Y einbetten und so diesen speziellen Zugriff ermöglichen
  - wenn verborgen werden soll, dass eine Klasse X innerhalb einer Klasse Y benötigt wird, dann kann man X in Y einbetten und so vor äußeren Zugriffen schützen
- Wartbarkeit: gemeinsam benutzter Quellcode wird näher zusammen gebracht und erleichtert dabei das Nachvollziehen und die Wartung von Quellcode (oder es passiert genau das Gegenteil ;-))
- Faulheit: es muss ggf. weniger Quellcode geschrieben werden (wobei eigentlich anderweitig nutzbare Klassen dann aber nicht anderweitig nutzbar sind und aufwendig neu implementiert werden müssen :-( )

Arten eingebetteter Klassen

```
package de.baleipzig.nested;

public class EnclosingClass {
  static class StaticNestedClass {
    ...
  }
  class InnerClass {
    ...
  }
}
```

Hauptunterscheidung:

- statisch eingebettete Klassen („static nested classes“)
- nicht-statisch eingebettete Klassen, die traditionell als „innere Klassen“ („inner classes“) bezeichnet werden

Arten eingebetteter Klassen

Wissen Sie noch, was **„static“** in Java allgemein bedeutet?

Hauptunterscheidung:

- statisch eingebettete Klassen („static nested classes“)
- nicht-statisch eingebettete Klassen, die traditionell als „innere Klassen“ („inner classes“) bezeichnet werden

Statisch eingebettete Klasse

- eingebettete Klasse, die (analog zu anderen statischen Klassenmitgliedern) durch das Schlüsselwort „static" gekennzeichnet wird

  Anmerkung: eine Klasse kann nur „static" sein, wenn sie eine eingebettete Klasse ist, da „static" außerhalb einer Klasse keinen Sinn ergibt
- wie bei „static" üblich wird prinzipiell keine Instanz der umschließenden Klasse benötigt
- daher können auch nur statische Mitglieder der umschließenden Klasse direkt angesprochen werden und nicht-statische nur über ein Objekt (daher werden diese Klassen relativ selten benutzt)
- Zugriff (wie bei anderen Klassenmitgliedern üblich) über Name der umschließenden Klasse mit einem Punkt dahinter
- entsprechend Instanzerzeugung durch:

```
NameUmschliessendeKlasse.NameEingebetteteKlasse objektName =
  new NameUmschliessendeKlasse.NameEingebetteteKlasse();
```

- verhält sich wie eine „Top-Level-Klasse", die aus was für Gründen auch immer in eine andere Klasse eingebettet wurde

Statisch eingebettete Klasse - Beispiel

```
package de.baleipzig.nested;
public class DemoStaticNestedClass {
  private int enclosingValue = 1;
  private static int enclosingStaticValue = 2;
  public static class StaticNestedClass {
    private int staticNestedValue = 3;
    private static int staticNestedStaticValue = 4;
    public void staticNestedClassMessage() {
      System.out.println("Hello from a method in StaticNestedClass.");
      System.out.println("Only general access to static values of enclosing class: "
        + enclosingStaticValue);
      DemoStaticNestedClass testEnclosingObjInSNClass = new DemoStaticNestedClass();
      System.out.println("Access to non-static values through object: "
        + testEnclosingObjInSNClass.enclosingValue);
    }
  }
  public static void main(String[] args) {
    DemoStaticNestedClass.StaticNestedClass testNestedObject =
      new DemoStaticNestedClass.StaticNestedClass();
    testNestedObject.staticNestedClassMessage();
  }
}
```

könnte auch anders sein, z.B. **private**

Aufgabe 1: Versuchen Sie, innerhalb von „staticNestedClassMessage“ auch die Variablen von „DemoStaticNestedClass“ auszugeben.

Zusatz: Was müssten Sie tun, wenn die statischen Variablen beider Klassen namensgleich wären?

Statisch eingebettete Klasse - Übung

praxisnahe Aufgaben, alles ist etwas schwammig und unvollständig gegeben; wenn Sie ohne „Herztabletten" bis zur Rente kommen wollen, dann regen Sie sich bitte nicht darüber auf, sondern krempeln die Ärme hoch und „ran ans Werk"

Aufgabe 2

Schreiben Sie eine Klasse „Student" mit:

- Instanzvariablen „name" und „javaExperience"
- Methode „getExaminationScore", die einen double-Wert „extraExperience" als Parameter übernimmt und basierend darauf und auf „javaExperience" einen double-Wert zurückliefert
- statisch eingebetter Klasse „ExtraExperience" mit:
  - Instanzvariable „extraHoursLearned"
  - Methode „getExtraExperience", die als Parameter einen „Student" bekommt und basierend auf den Instanzvariablen beider Klassen einen Bonus für die Methode „getExaminationScore" ermittelt und als double-Wert zurückgibt.

Zusatz: Methode „addExtraHoursLearned" noch implementieren und im Test nutzen

Implementieren Sie nötige Konstruktoren, Getter und Setter und überlegen Sie sich passende Methodeninhalte.

Aufgabe 3 - Testen Sie Ihre Klasse (z.B. über „`public static void main(String[] args)`")

Erzeugen Sie zwei normale (ein erfahrener und ein unerfahrener) und einen faulen Studenten und passend dazu zwei Instanzen der Klasse „ExtraExperience" (eine für die normalen und eine für den faulen Studenten). Nutzen Sie für alle Studenten die Methode „getExaminationScore", wobei für den Parameter jeweils auf „getExtraExperience" der passenden Instanz zurückgegriffen werden soll, und geben Sie jeweils den erhaltenen Wert auf der Konsole aus.

Statisch eingebettete Klasse – Lösung (1-3)

aus Platzgründen etwas unübersichtlich mit abgekürzten Variablennamen – Sie machen das bitte „schöner"

```
package de.baleipzig.nested;
public class Student {
  //Instanzvariablen, Konstruktoren, Getter und Setter …
  public double getExaminationScore(double extraExperience) {
    return javaExperience + extraExperience;
  }

  public static class ExtraExperience {
    private double extraHoursLearned;
    //Konstruktoren ...
    public double getExtraExperience(Student ourStudent) {
      return extraHoursLearned * ourStudent.getJavaExperience();
    }
  }

  public static void main(String[] args) {
    Student normalStudA = new Student("Der schlaue Fleißmeister", 10.0);
    Student normalStudB = new Student("Der unerfahrene Fleißmeister", 1.0);
    Student.ExtraExperience normalStudExp = new Student.ExtraExperience(20.0);
    System.out.println("Werte unseres normalen, schlauen Studies: " +
      normalStudA.getExaminationScore(normalStudExp.getExtraExperience(normalStudA)));
    System.out.println("Werte unseres normalen, unerfahrenen Studies: " +
      normalStudB.getExaminationScore(normalStudExp.getExtraExperience(normalStudB)));
    Student lazyStudi = new Student("Herr Faultier", 10.0);
    Student.ExtraExperience lazyStudExp = new Student.ExtraExperience(1.0);
    System.out.println("Werte unseres faulen Studies: " +
      lazyStudi.getExaminationScore(lazyStudExp.getExtraExperience(lazyStudi)));
  }
}
```

nicht-statisch eingebettete Klasse = Innere Klasse

- eingebettete Klasse, die weder explizit noch implizit „static" ist
- **da eine innere Klasse nicht „static" ist, wird prinzipiell zur Erzeugung einer Instanz davon eine Instanz der umschließenden Klasse benötigt**
- Instanzerzeugung erfolgt z.B. so:

```
NameUmschliessendeKlasse objektNameUmschliessendeKlasse =
  new NameUmschliessendeKlasse();

NameUmschliessendeKlasse.NameInnereKlasse objektNameInnereKlasse =
  objektNameUmschliessendeKlasse.new NameInnereKlasse();
```

- **aber: wenn keine Instanz der inneren Klasse benötigt wird, muss auch keine Instanz der umschließenden Klasse erzeugt werden**
  - seit Java SE 16 (JEP 395: Records – siehe https://openjdk.org/jeps/395) können innere Klassen auch statische Mitglieder haben, z.B.: `public static int numberOfInnerObjects = 0;`
  - Zugriff darauf erfolgt z.B. mit: `NameUmschliessendeKlasse.NameInnereKlasse.numberOfInnerObjects;`
- alle Mitglieder der umschließenden Klasse können direkt angesprochen werden (daher sind die meisten eingebetteten Klassen sogenannte „Innere Klassen")
- Zugriff auf Variablen und Methoden der inneren Klasse über Name der (inneren) Instanz bzw. vollen Name der Klasse mit einem Punkt dahinter gefolgt vom Variablen- bzw. Methodennamen

Innere Klassen bzgl. Klassen - Arten

Innere Member-Klasse:

Innere Klasse, die parallel zu den anderen Klassenmitgliedern innerhalb einer umschließenenden Klasse, aber nicht innerhalb einer Methode einer umschließenden Klasse steht

Innere Lokale Klasse:

Innere Klasse, die innerhalb einer Methode einer umschließenden Klasse steht

Innere Anonyme Klasse:

Innere Klasse, die namenslos innerhalb einer umschließenden Klasse steht

Member-Klasse (eine Innere Klasse)

```
package de.baleipzig.nested;
public class DemoMemberInnerClass {
  private int enclosingValue = 1;
  private static int enclosingStaticValue = 2;
  public class MemberClass {
    private int innerClassValue;
    private static int innerClassStaticValue;
    public void memberClassMessage() {
      System.out.println("Hello from a method in MemberClass.");
      System.out.println("General access to values of enclosing class: ");
      System.out.println("enclosingValue: " + enclosingValue);
      System.out.println("enclosingStaticValue: " + enclosingStaticValue);
    }
  }
  public static void main(String[] args) {
    DemoMemberInnerClass testEnclosingObject = new DemoMemberInnerClass();
    DemoMemberInnerClass.MemberClass testMemberObject =
      testEnclosingObject.new MemberClass();
    testMemberObject.memberClassMessage();
  }
}
```

könnte auch anders sein, z.B. **private**

Aufgabe 4: Versuchen Sie, innerhalb von „memberClassMessage" auch die Variablen von „MemberClass" auszugeben.

Zusatz: Was müssten Sie tun, wenn die statischen Variablen beider Klassen namensgleich wären?

Member-Klasse - Übung

Aufgabe 5

Schreiben Sie in Ihrer eben erstellten „DemoMemberInnerClass" eine weitere Member-Klasse „Safe" mit:

- einer statischen Variablen „numberOfSafes", die mit `0` initialisiert wird
- einer statischen Methode, die eine Nachricht auf der Konsole ausgibt und keinen Rückgabewert liefert

Versuchen Sie, innerhalb der main-Methode der umschließenden „DemoMemberInnerClass" sowohl den Wert der statischen Variablen „numberOfSafes" anzuzeigen als auch die statische Methode Ihrer Member-Klasse aufzurufen, ohne explizit ein Objekt der umschließenden Klasse zu erstellen.

Aufgabe 6

Schreiben Sie im gleichen Paket eine „TestClass" mit main-Methode und versuchen Sie von dort aus, sowohl den Wert der statischen Variablen „numberOfSafes" anzuzeigen als auch die statische Methode Ihrer Member-Klasse aufzurufen, ohne explizit ein Objekt der umschließenden Klasse zu erstellen.

Zusatzaufgabe

Lagern Sie auch die bisher in „DemoStaticNestedClass" in der main-Methode durchgeführten Tests ordentlich in Ihre „TestClass" aus.

Member-Klasse – Lösung (4-6)

```
package de.baleipzig.nested;
public class DemoMemberInnerClass {
  ...
  public class Safe {
    public static int numberOfSafes = 0;

    public static void staticShow() {
      System.out.println("Du kommst auch ohne Objekt rein ;-)");
    }

  }
  public static void main(String[] args) {
    ...
    System.out.println("Anzahl Safes: " + Safe.numberOfSafes);
    Safe.staticShow();
  }
}

package de.baleipzig.nested;
public class TestClass {
  public static void main(String[] args) {
    System.out.println("Anzahl Safes: " + DemoMemberInnerClass.Safe.numberOfSafes);
    DemoMemberInnerClass.Safe.staticShow();
  }
}
```

Lokale Klasse (eine Innere Klasse)

```
package de.baleipzig.nested;

public class DemoLocalInnerClass {
  public void enclosingMethod () {
    System.out.println ("Inside outer method");

    class LocalClass {
      void localClassDemoMethod () {
        System.out.println ("Inside local inner method");
      }
    }

    LocalClass myLocalObject = new LocalClass ();
    myLocalObject.localClassDemoMethod ();
  }

  public static void main (String[]args) {
    DemoLocalInnerClass testEnclosingObject = new DemoLocalInnerClass ();
    testEnclosingObject.enclosingMethod ();
  }

}
```

kein Zugriffsmodifizierer

Aufgabe 7: Versuchen Sie, eine lokale Klasse mit Konstruktor zu schreiben und zu nutzen.

Aufgabe 8: Versuchen Sie, in der lokalen Klasse eine statische Variable mit Initialwert `0` hinzuzufügen und diese bei jedem Konstruktoraufruf um 1 zu erhöhen.

Zusatz: Kann eine lokale Klasse „abstract" sein? Erstellen Sie ggf. ein Demo-Beispiel.

Lokale Klasse (eine Innere Klasse) – Lösung (7-8)

```
...
class LocalClass {
  void localClassDemoMethod () {
    System.out.println ("Inside local inner method");
  }
  private int testNumber = 0;
  public static int staticTest =  0;

  public LocalClass() {
    testNumber = 7;
    staticTest = staticTest + 1;
  }

  void localClassDemoMethod () {
    System.out.println ("Inside local inner method with testNumber " +
     testNumber + " and staticTest " + staticTest);
  }
}
...
```

Lokale Klasse können Konstruktoren und statische Mitglieder haben.

Lokale Klassen können auch „abstract“ sein, was allerdings ungewöhnlich ist.

Anonyme Klasse (eine Innere Klasse)

- innere Klasse, die namenslos innerhalb einer umschließenden Klasse steht
- **da eine anonyme innere Klasse namenslos ist, muss sie gleichzeitig deklariert und initialisiert werden und kann auch nur einmal benutzt werden**
- ist syntaktisch Teil eines Ausdruck, der mit einem Semikolon abgeschlossen werden muss
- Ausdruck „anonyme Klasse" besteht aus:
  - new-Operator
  - Angabe einer zu erweiternden Klasse bzw. einer zu implementierenden Schnittstelle
  - runden Klammern mit ggf. Parametern
  - geschweiften Klammern mit der Klassendeklaration

Anonyme Klassen werden meist zur Implementierung von Schnittstellen mit wenigen Methoden eingesetzt, vor allem bei der Programmierung graphischer Benutzeroberflächen (z.B. zur Anpassung des Verhaltens von Eingabefeldern); da graphische Benutzeroberflächen erst in einer späteren Vorlesung ausführlich behandelt werden, schauen wir uns anonyme Klassen nur oberflächlich an

Anonyme Klasse (eine Innere Klasse)

```
package de.baleipzig.nested;

public abstract class Ship {
  public abstract void giveAHint();
}
```

```
package de.baleipzig.nested;

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

    Ship rowboat = new Ship () {
      public void giveAHint () {
        System.out.println ("Ich bin ein Ruderboot.");
      }
    };

    rowboat.giveAHint();
  }
}
```

kein Name und kein Zugriffsmodifizierer; nur Angabe entweder

- genau EINER erweiterten Klasse bzw.
- genau EINER implementierten Schnittstelle

Aufgabe 9: Fügen Sie der Klasse „Ship" zwei Instanzvariablen „name" und „type" vom Typ „String" und entsprechende Getter, Setter und Konstruktoren zu.

Aufgabe 10: Geben Sie Ihrem „rowboat" einen Namen, z.B. „noname", überschreiben Sie in Ihrer anonymen Klasse „getName()", so dass für unser „rowboat" dann beispielsweise Ruderboot „noname" statt noname zurückgegeben wird, und geben Sie dies aus.

Anonyme Klasse (eine Innere Klasse) – Lsg. 9

```
package de.baleipzig.nested;
public abstract class Ship {
  private String name;
  private String type;
  public abstract void giveAHint();
  public Ship() {
    this("unknown", "unknown");
  }
  public Ship(String aName, String aType) {
    name = aName;
    type = aType;
  }
  public String getName() {
    return name;
  }
  public void setName(String aName) {
    name = aName;
  }
  public String getType() {
    return type;
  }
  public void setType(String aType) {
    type = aType;
  }
}
```

Anonyme Klasse (eine Innere Klasse) – Lsg. 10

```
package de.baleipzig.nested;

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

    Ship rowboat = new Ship () {

      public void giveAHint () {
        System.out.println ("Ich bin ein Ruderboot.");
      }

      public String getName () {
        return "Ruderboot \"" + super.getName() + "\"";
      }
    };

    rowboat.setName("noname");
    rowboat.giveAHint();
    System.out.println(rowboat.getName());

  }
}
```

nicht klausurrelevant:

Kann „rowboat" in der anonymen Klasse problemlos eigene Methoden haben?

Kann eine anonyme Klasse eigene Konstruktoren haben?

(Und ist dies überhaupt nötig?)

Anonyme Klasse (eine Innere Klasse) – Lsg. 10

```
package de.baleipzig.nested;

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

    Ship rowboat = new Ship () {

      public void giveAHint () {
        System.out.println ("Ich bin ein Ruderboot.");
      }

      public String getName () {
        return "Ruderboot \"" + super.getName() + "\"";
      }
    };

    rowboat.setName("noname");
    rowboat.giveAHint();
    System.out.println(rowboat.getName());

  }
}
```

nicht klausurrelevant: da „rowboat" ein „Ship" ist, können auf normalem Weg nur Methoden von „Ship" und keine eigenen benutzt werden; um eigene Methoden aufzurufen, müsste man diese direkt bei der Initialisierung aufrufen, Reflections benutzen oder den neuen „var"-Identifizierer nutzen, also **„var rowboat" satt „Ship rowboat"** schreiben; auch Konstruktoren sind aufgrund der Namenslosigkeit nicht möglich, dafür gibt es ersatzweise aber Exemplarinitialisierungsblöcke

Eingebettete Klassen – Übung „Klasse in Klasse“

Aufgabe 11

Schreiben Sie in Ihrem Paket eine Klasse „Port“ mit:

- Instanzvariablen:
  - „name“ passenden Typs
  - „registeredShips“ als „List<ShipAtPort>“, die mit einer neuen „ArrayList<>()“ initialisiert wird
- Gettern, einem Setter für „name“ und einem Konstruktor, der „name“ als Parameter hat
- einer Methode „addRegisteredShip“, die als Parameter ein „ShipAtPort“ hat, nichts zurückliefert und das übergebene Ship der „registeredShips“-Liste hinzufügt
- einer inneren Klasse „ShipAtPort“, die vom eben erstellten „Ship“ erbt, mit:
  - eigener statischer Variable „numberOfRegisteredShips“, die mit `0` initialisiert wird
  - eigener Instanzvariable „numberInRegister“
  - Konstruktor, der Name und Bootstyp entgegennimmt und mit Hilfe des Konstruktors aus „Ship“ setzt und außerdem bei jedem Aufruf „numberOfRegisteredShips“ um eins erhöht und zusätzlich „numberInRegister“ sinnvoll setzt
  - sinnvollem Standardkonstruktor
  - überschriebener toString-Methode, die einen sinnvollen String mit dem Hafenname und allen Schiffsinformationen zurückliefert

Aufgabe 12

Testen Sie Ihre Klassen, indem Sie einen Hafen mit einem Namen anlegen, mittels „addRegisteredShip“ ihrem Hafen ein „Ruderboot“ mit einem Namen hinzufügen und Sich dann die Schiffe Ihres Hafens anzeigen lassen.

Eingebettete Klassen – Lösung 11/12 (1)

```
package de.baleipzig.nested;

import java.util.ArrayList;
import java.util.List;

public class Port {

  private String name;
  private List<ShipAtPort> registeredShips = new ArrayList<>();

  public List<ShipAtPort> getRegisteredShips() {
    return registeredShips;
  }

  public String getName() {
    return name;
  }

  public void setName(String aName) {
    name = aName;
  }

  public Port(String aName) {
    name = aName;
  }

  public void addRegisteredShip(ShipAtPort newShip) {
    registeredShips.add(newShip);
  }

  public class ShipAtPort extends Ship { … }

  public static void main(String[] args) { … }

}
```

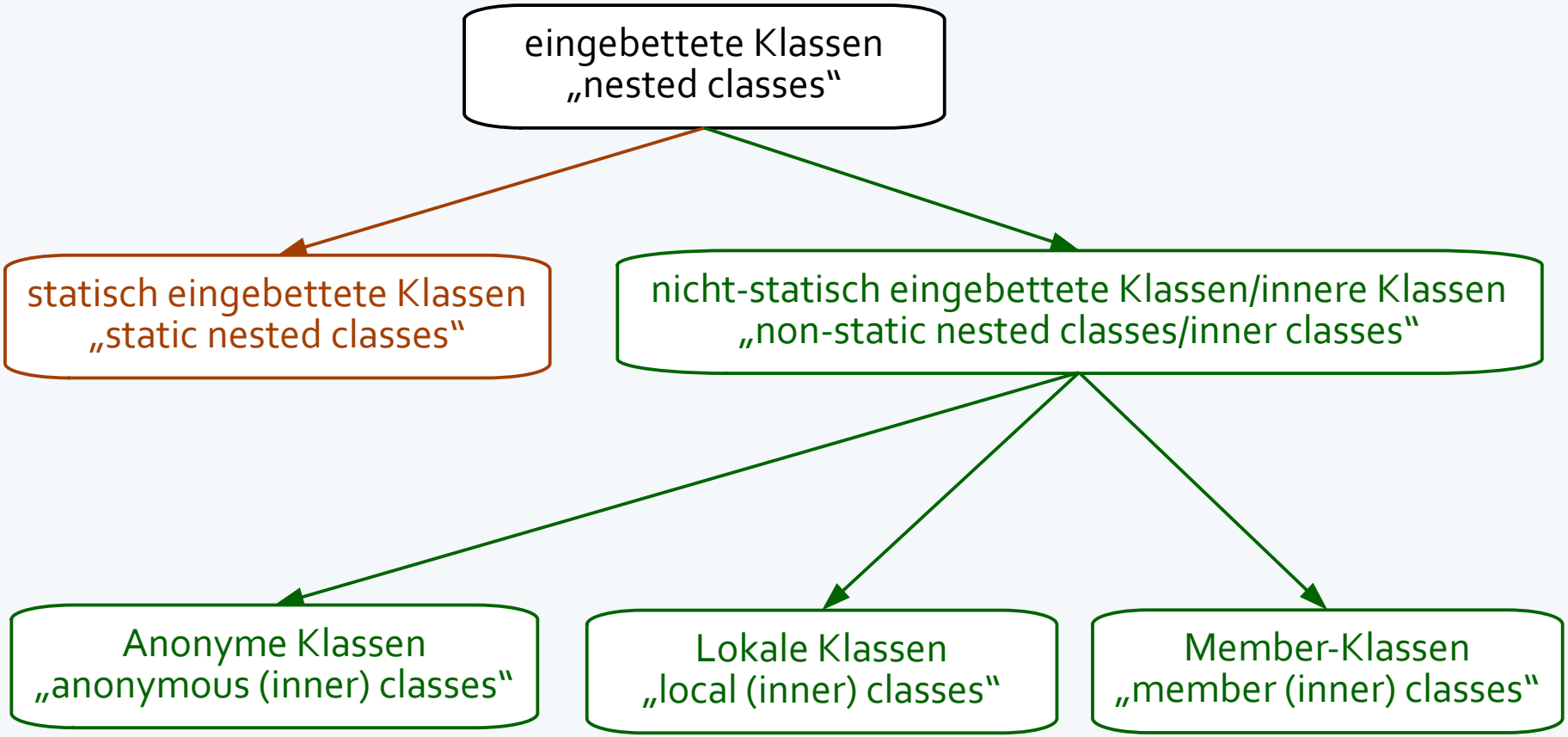
Eingebettete Klassen – Lösung 11/12 (2)

```
...
public class Port {
  public class ShipAtPort extends Ship {
    public static int numberOfRegisteredShips = 0;
    private int numberInRegister;

    public ShipAtPort() {
      this("unknown","unknown");
    }

    public ShipAtPort(String aName, String aType) {
      super(aName, aType);
      numberOfRegisteredShips = numberOfRegisteredShips + 1;
      numberInRegister = numberOfRegisteredShips;
    }

    public void giveAHint() {
      System.out.println("Hier liege ich ...");
    }

    @Override
    public String toString() {
      return "ShipAtPort [register number = " + numberInRegister +
        " at port " + name + ", name = " + getName() +
        ", type = " + getType() + "]";
    }
  }
  ...
}
```

Eingebettete Klassen – Lösung 11/12 (3)

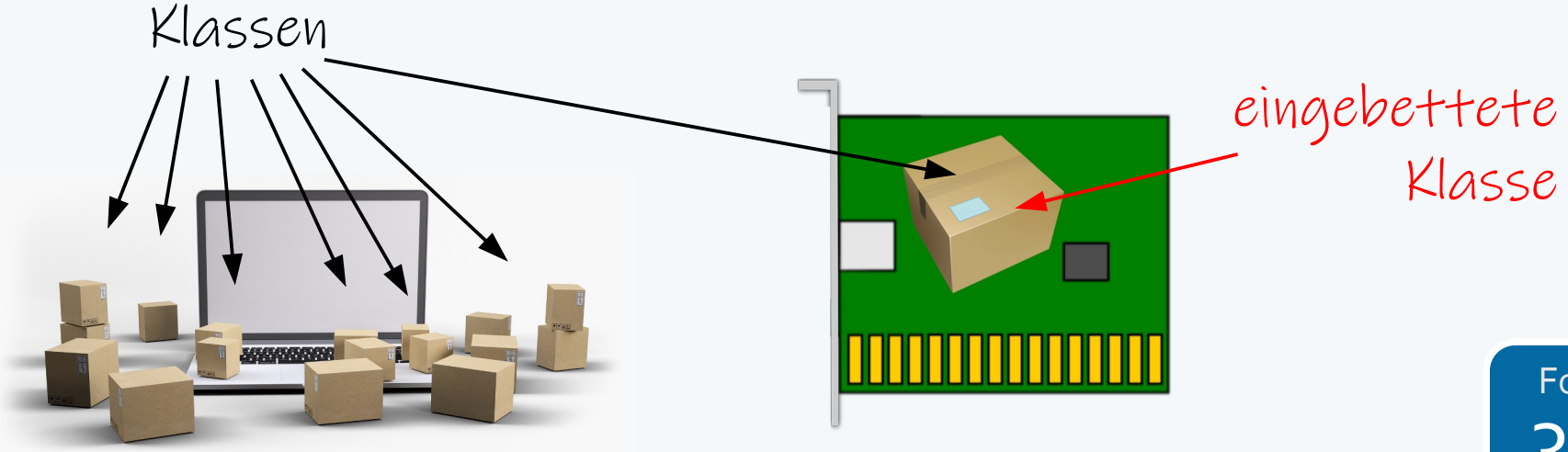
```
...

public class Port {

  ...

  public static void main(String[] args) {
    Port leipzigPort = new Port("Cityhafen");
    leipzigPort.addRegisteredShip(leipzigPort.new ShipAtPort("Mein Kahn",
      "Ruderboot"));
    System.out.println(leipzigPort.getRegisteredShips());
  }

}
```

Eingebettete Klassen - Überblick



Klassen in Schnittstellen

- in Java kann man eine Klasse X auch in einer Schnittstelle Y definieren
- diese Klasse X heißt dann „eingebettet" bzw. „geschachtelt" (engl. „nested")
- Einbettung kann über mehrere Ebenen gehen
- **Anmerkung:** Zweck davon kann z.B. sein, dass Klassen als Hilfsklassen für Schnittstellen zur Verfügung gestellt werden oder dass die Benutzung bestimmter Klassen an die Implementierung der umschließenden Schnittstelle gebunden sind.



Klassen in Schnittstellen

```
package de.baleipzig.nested;
public interface Loadable {
  public abstract void load();

  public class CargoHold {
    boolean cargoHoldOpen = false;
    ...
  }

  static public class Helper {
    static public void startMessage() {
      System.out.println("Ich werde beladen.");
    }
    static public void finishMessage() {
      System.out.println("Ich bin fertig beladen.");
    }
  }

}
```

kann/sollte weggelassen werden, da sowieso nur „public“ möglich ist

Aufgabe 13:

Implementieren Sie in „CargoHold“ die Methoden „statusCargoHold()“, „openCargoHold()“ und „closeCargoHold()“.

Klassen in Schnittstellen – Lösung 13

```
...
public class CargoHold {

  boolean cargoHoldOpen = false;

  public boolean statusCargoHold() {
    return cargoHoldOpen;
  }

  public void openCargoHold() {
    cargoHoldOpen = true;
  }

  public void closeCargoHold() {
    cargoHoldOpen = false;
  }

}

...
```

Aufgabe 14:

Erstellen Sie eine Klasse „CargoShip", die von „Ship" erbt und „Loadable" implementiert.

Ein „CargoShip" soll dabei ein „cargoHold" passenden Typs haben, der beim „load" zunächst geöffnet, dann „per Systemausgabe als Ersatzhandlung" beladen wird und anschließend wieder geschlossen wird.

Ergänzen Sie weitere fehlende Quellcodeteile sinnvoll.

Testen Sie Ihre neue Klasse zunächst direkt innerhalb der main-Methode der Klasse, indem Sie dort ein „CargoShip" erstellen und beladen.

Zusatz:

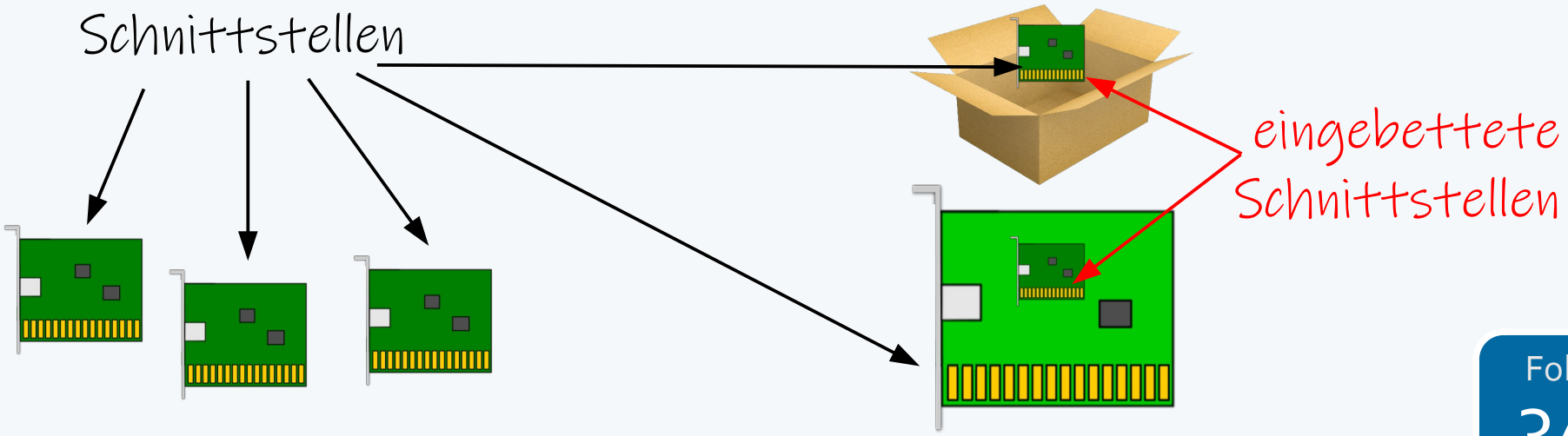
Nutzen Sie beim „load" auch Ihre Methoden der „Helper"-Klasse von „Loadable".

Klassen in Schnittstellen – Lösung 14

```
package de.baleipzig.nested;

public class CargoShip extends Ship implements Loadable {

  Loadable.CargoHold cargoHold = new Loadable.CargoHold();

  @Override
  public void load() {
    cargoHold.openCargoHold();
    Loadable.Helper.startMessage();
    System.out.println("Achtung, ich werde beladen.");
    Loadable.Helper.finishMessage();
    cargoHold.closeCargoHold();
  }

  @Override
  public void giveAHint() {
    System.out.println("Ich bin ein Frachter ... ");
  }

  public static void main(String[] args) {
    CargoShip meinFrachter = new CargoShip();
    meinFrachter.load();
  }

}
```

Schnittstellen in Klassen oder in Schnittstellen

- man kann (dann z.B. auch „private“) Schnittstellen in Klassen definieren; Zweck ist oft, eine Schnittstelle derart an eine Klasse zu binden, dass man Programmierern, die die Schnittstelle nutzen, ermöglicht, auch die Funktionalitäten der umschließenden Klasse zu nutzen bzw. allgemein Klassenfunktionalitäten mit einer Schnittstelle zu verbinden
- man kann (dann nur öffentliche [„public“]) Schnittstellen in Schnittstellen definieren; Zweck ist, dass man verwandte Schnittstellen mit gemeinsamer Funktionalität/gemeinsamem Zweck zusammen gruppiert
- **Anmerkung:** Schnittstellen in Klassen oder in Schnittstellen sind relativ selten und die Umsetzung ist zwar intuitiv, aber trotzdem nicht klausurrelevant



Klassen, Schnittstellen und Objekte für Java-Profis

- Klassen und Schnittstellen in Klassen und Schnittstellen
- **Konstruktion von Objekten mittels „Java Builder Pattern“**

Objekte mittels "Java Builder Pattern"

- reale Objekte sind vielfältig und Merkmale können optional oder unbekannt sein
- Objekte mit vielen optionale Eigenschaften gleichen Datentyps lassen sich mit Konstruktoren und überladenen Konstruktoren schlecht erzeugen

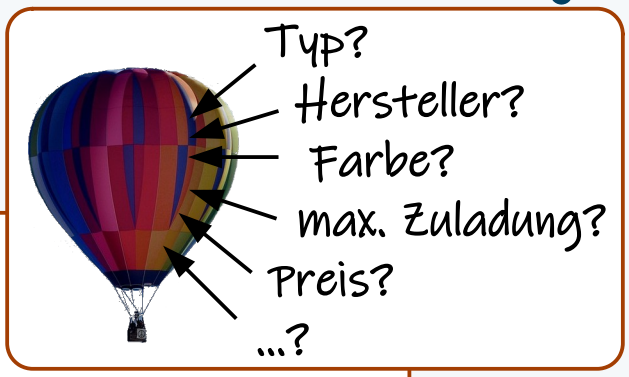
Objekte mittels "Java Builder Pattern"

- reale Objekte sind vielfältig und Merkmale können optional oder unbekannt sein
- Objekte mit vielen optionale Eigenschaften gleichen Datentyps lassen sich mit Konstruktoren und überladenen Konstruktoren schlecht erzeugen

zur übersichtlichen und wenig fehleranfälligen Objekterzeugung nutzt man **eingebettete Klassen im Rahmen des „Java Builder Pattern"**

Objekte mittels "Java Builder Pattern"

- **Idee:** Klasse für eigentliches Objekt
- statisch eingebettete Hilfsklasse zur Objekterzeugung




```java
package de.baleipzig.nested;

public class Balloon {

  private String type; //benoetigt
  private String manufacturer; //optional
  private double payload; //optional

  private Balloon(BalloonBuilder builder) {
    type = builder.type;
    manufacturer = builder.manufacturer;
    payload = builder.payload;
  }

  public static class BalloonBuilder{
    //... benoetigte Parameter setzen, gegebene optionale Parameter
    //... verarbeiten und über Methode „Balloon“ erzeugen
  }

  //Getter und Setter ...

}
```

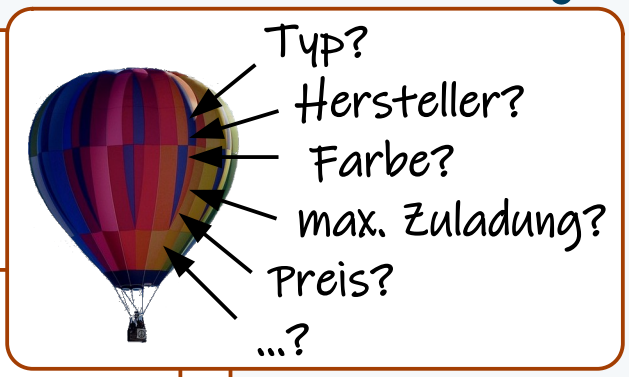
nur über Builder nutzbar

Objekte mittels "Java Builder Pattern"

```
private Balloon(BalloonBuilder builder) {
  type = builder.type;
  manufacturer = builder.manufacturer;
  payload = builder.payload;
}
```

Zugriff klassenintern über Objekt mgl.

```
public static class BalloonBuilder{
  private String type; //benoetigt
  private String manufacturer;  // optional (null)
  private double payload;// optional (0)

  public BalloonBuilder(String aType) {
    type = aType; //benoetigte Werte sind Pflicht
  }

  public BalloonBuilder manufacturer(String aManufacturer) {
    manufacturer = aManufacturer; //optionalen Wert über Methode
    return this;
  }

  public BalloonBuilder payload(double aPayload) {
    payload = aPayload; //optionalen Wert über Methode
    return this;
  }

  public Balloon build() {
    return new Balloon(this); //letztendlich „Balloon“ erzeugen
  }
}
```



hier brauchen Sie „this“, um auf die Instanz vom „BalloonBuilder“ zuzugreifen

Objekte mittels "Java Builder Pattern"

Wie erzeuge ich nun ein Objekt?

```
package de.baleipzig.nested;

public class TestClass {

  public static void main(String[] args) {
    //langer Weg zum Ballon
    Balloon.BalloonBuilder longBalloonBuilder =
      new Balloon.BalloonBuilder("Heißluftballon");
    longBalloonBuilder = longBalloonBuilder.payload(700);
    longBalloonBuilder = longBalloonBuilder.manufacturer("BA BallonProfi");
    Balloon longTestBalloon = longBalloonBuilder.build();

    //kurzer Weg zum Ballon
    Balloon shortTestBalloon =
      new Balloon.BalloonBuilder("Heißluftballon").
        payload(700).manufacturer("BA BallonProfi").build();
  }
}
```

für den kurzen Weg müssen Sie „BallonBuilder" jeweils als Rückgabewert der Methoden haben

Aufgabe 15: Versuchen Sie, „Balloon" und „BalloonBuilder" derart zu erweitern, dass optional für einen Ballon noch eine Farbe und ein Preis angegeben werden können.

Zusatz: Implementieren Sie alle Getter und Setter für „Balloon".

Objekte mittels "Java Builder Pattern"

```
...
private String color;
private double price;

private Balloon(BalloonBuilder builder) {
  ...
  color = builder.color;
  price = builder.price;
}
```

```
public static class BalloonBuilder{
  ...
  private String color;
  private double price;

  public BalloonBuilder color(String aColor) {
    color = aColor;
    return this;
  }

  public BalloonBuilder price(double aPrice) {
    price = aPrice;
    return this;
  }
  ...
}
```

```
...
```

Objekte mittels "Java Builder Pattern"

Aufgabe 16:

Erstellen Sie für eigene Objekte mit mind. 4 Merkmalen eine Klasse, bei der zur Objekterzeugung das „Java Builder Pattern" benutzt wird. Erstellen Sie testweise ein Objekt.

Zusatz:

Sorgen sie dafür, dass für ein Objekt der Klasse mit `System.out.println(Objekt)` die Objekteigenschaften des betreffenden Objektes angezeigt werden.

Vielen Dank für Ihre Aufmerksamkeit.