Typhierarchien, Exceptions und mehr zu Klassen

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

In diesem Kapitel befassen wir uns mit der Frage, wie Typhierarchien modelliert und umgesetzt werden können. Wir werden anhand der Beispiele sehen, dass es diese Frage nicht immer einfach zu beantworten ist.
Die Modellierung und Implementierung von Exceptions in Java ist ein Beispiel für eine komplexe Typhierarchie. Wir besprechen die einzelnen Arten von Exceptions und Fehlerbehandlung und auch die Möglichkeit eigene Exceptions in dieser Typhierarchie einzubringen und in Programmen zu verwenden.
Außerdem stellen wir weitere Aspekte von Klassen vor: (statische) Klassenattribute - und methoden und geschachtelte Klassendeklarationen.

1 Umsetzung von Klassifikationen

Die Klassen bzw. Begriffe in einer Klassifikation können im Java-Programm durch Interface- oder Klassentypen realisiert werden. Ob Interface- oder Klassentypen gewählt werden, hängt vom konkreten Anwendungsfall ab.
Verwendung von Interfaces in Java:

- Man will sich nicht auf bestimmte Implementierungsteile festlegen.
- Der neue Typ dient als Supertyp von Klassen mit mehreren Supertypen.

Verwendung von Klassen in Java:

- Objekte sollen von dem Typ erzeugt werden; dies kann jedoch durch die Verwendung von abstrakten Klassen verhindert werden, falls es unerwünscht ist.
- Vererbung an Subtypen soll ermöglicht werden; es erlaubt die Verwendung von Code für mehrere verwandte Klassen, die Ähnlichkeiten aufweisen.

1.1 Beispiel

Wir betrachten als Beispiel noch einmal die Klassifikation von Personen an der Universität aus Kapitel 12 bestehend aus:

- Person, Printable, Student, Angestellte
- WissAngestellte und VerwAngestellte
- RegulaererStudent und AustauschStudent

Frage 1:

- Visualisieren Sie die entsprechende Typhierarchie, wobei Studenten und Angestellte ein Subtyp von Person und Printable sein sollen; Person soll jedoch kein Subtyp von Printable sein.
- Wie kann man diese Typhierarchie implementieren? Welche Typen sollen als Interfaces, welche als (abstrakte) Klassen implementiert werden?

1. Variante Nur die unterste Ebene ("Blätter") der Klassifikation (`RegulaererStudent`, `AustauschStudent`, `WissAngestellte`,`VerwAngestellte`) werden durch Klassen realisert, alle anderen durch Interfaces.

```
interface Printable {
  void print();
}

interface Person {
  String getName();
  String getBirthdate();
}

interface Angestellte extends Person, Printable { ... }
interface Student     extends Person, Printable { ... }

class WissAngestellte   implements Angestellte { ... }
class VerwAngestellte   implements Angestellte { ... }
class RegulaererStudent implements Student   { ... }
class AustauschStudent  implements Student   { ... }
```

2. Variante Bis auf den Typ `Printable` realisieren wir alle Typen durch Klassen:

```
interface Printable { ... }

class Person        { ... }

class Student     extends Person implements Printable { ... }
class Angestellte extends Person implements Printable { ... }

class WissAngestellte   extends Angestellte { ... }
class VerwAngestellte   extends Angestellte { ... }

class RegulaererStudent extends Student   { ... }
class AustauschStudent  extends Student   { ... }
```

Variante 1 legt lediglich fest, welche Methoden Objekte vom Typ `Person`, `Printable`, `Student` etc. bereitstellen müssen. Diese Methoden sind außerdem alle `public`. Die jeweiligen Implementierungen müssen von den Klassen, die in der Typhierarchie ganz unten erscheinen, definiert werden.
In der 2. Variante können die Klassen `Student` und `Angestellte` sowie deren Subklassen Attribute und Methoden von `Person` erben. Es ist außerdem möglich, `Person`-Objekte zu erzeugen, die weder `Student` noch `Angestellte` sind. Dies kann unerwünscht sein und durch ein Umwandeln von `Person` in eine abstrakte Klasse verhindert werden. In dieser Variante kann der Typ `Printable` nicht einfach zusätzlich in eine Klasse umgewandelt werden, da jede Klasse immer nur von maximal **einer** Superklasse erben kann. Es ist in diesem Beispiel sinnvoll, `Printable` durch ein Interface zu realisieren und nicht `Person`, da es im Vergleich zu `Person` schwieriger ist für die `print()`-Methode eine Implementierung zu finden, die von allen `Printable`-Objekten genutzt werden kann. Variante 1 erlaubt es flexibel, dass weitere Klassen die `Student`- bzw. `Angestellte`-Interface implementieren, auch wenn sie bereits Subklassen einer anderen Klasse sind. Es ist beispielsweise schwierig in der 2. Variante Tutoren zu implementieren, da diese sowohl Studierende als auch Angestellte sind, aber nur eine dieser Klassen als Superklasse haben können.
Die beiden Varianten zeigen zwei Extreme: Variante 1 verwendet möglichst nur Interfaces, Variante 2 verwendet möglichst nur Klassen. Es sind natürlich auch weitere Abstufungen möglich (z.B. Umsetzung von `Person` als Interface und `Student` und `Angestellte` als Klasse).

1.2 Das Quadrat-Rechteck-Problem

In Kapitel 15 haben wir eine Klassifizierung von geometrischen Figuren vorgestellt. Diese umfasste bisher nur Kreise und Quadrate. Wir wollen dieser Modellierung nun beliebige Rechtecke hinzufügen.

```
class Rectangle extends AFigure {
  private double length;
  private double width;
  ...
  double area() {
    return length * width;
  }
}

class Square extends AFigure {
  private double length;
  ...
  double area() {
    return length * length;
  }
}
```

Die Klassen `Square` und `Rectangle` weisen viele Gemeinsamkeiten auf, allerdings können sich bei Rechtecken Höhe und Breite voneinander unterscheiden.

Wie sollte die Vererbungs-/Typrelation zwischen den Klassen `Square` und `Rectangle` aussehen?

Variante 1 `Square` als Superklasse von `Rectangle`

```
class Square extends AFigure {
  private double length;
  ...
  double area() {
    return length * length;
  }
}
class Rectangle extends Square {
  private double width;
  ...
  double area() {
    return length * width;
  }
}
```

- Die Variante erlaubt die Vererbung des `length`-Attributs.
- Methoden wie `area()`, `draw()`, etc. müssen überschrieben werden.
- Semantik ist problematisch: Ein Rechteck ist im Allgemeinen kein Quadrat!

Variante 2 `Rectangle` als Superklasse von `Square`

```
class Rectangle extends AFigure {
  private double length;
  private double width;
  Rectangle (CartPt loc, double length, double width) {
    super(loc);
    this.length = length;
    this.width = width;
  }
  double area() {
    return length * width;
  }
}
class Square extends Rectangle {
  Square (CartPt loc, double length) {
    super(loc, length, length);
  }
}
```

- Methoden müssen nicht überschrieben werden, aber spezialisierte Methoden könnten effizienter sein.
- `Square`-Objekte haben nicht verwendbare, unnötige Attribute.
- Die Invariante `length == width` muss für `Square`-Objekte immer sichergestellt sein!

- Das Substitutionsprinzip erfordert, dass man in dieser Variante ein Quadrat überall dort verwenden kann, wo ein Rechteck erwartet wird. Wenn es nur Methoden gibt, die auf die Attribute lesend zugreifen, dann folgt diese Variante dem Substitutionsprinzip. Die Semantik von Operationen wie `setWidth()` vs. `setLength()` ist aber unklar: Soll bei Verändern der Breite bzw. Höhe aus dem Quadrat ein Rechteck werden?!

Außer den hier vorgestellten Varianten sind noch weitere Alternativen denkbar:

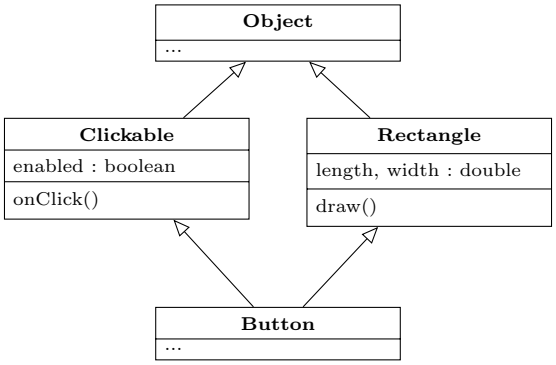
- Verzicht auf Vererbungsbeziehung
  - Führt zu Code-Duplikation
  - Keine Subtyp-Polymorphie anwendbar
- Einführung einer zusätzlichen gemeinsamen Superklasse, die die Gemeinsamkeiten abstrahiert
  - Unnötiges Aufblähen der Klassenhierarchie
  - Keine Subtyp-Polymorphie anwendbar zwischen Quadrat und Rechteck

Ohne weiteren Kontext lässt sich nicht einfach bestimmen, welche Variante am sinnvollsten gewählt werden sollte.

1.3 Mehrfachvererbung

Übernimmt eine Klasse Programmteile von mehreren anderen Klassen spricht man von ***Mehrfachvererbung*** (engl. *multiple inheritance*).

Beispiel Schaltflächen in Benutzeroberflächen sind zum einen Rechtecke, aber auch anklickbare Objekte, denen ein Verhalten beim Anklicken zugeordnet werden kann.



Mehrfacherbung ist problematisch, da es zu Namenskonflikten bei Attributen und Methoden kommen kann, Konstruktoren Attribute verschieden initialisieren können etc.

Da in Java (bis Version 1.7) Mehrfachvererbung nicht möglich ist, muss man die Modellierung durch Einfachvererbung und mehrfache Subtypbildung über Interfaces umsetzen.
Seit Java 8 können Interfaces default-Implementierungen (Standardimplementierung) von Methoden enthalten. Dadurch kann eine Art von Mehrfachvererbung umgesetzt werden. Wenn zwei Interfaces default-Methoden mit gleichen Typsignaturen haben, muss in der abgeleiteten Klasse eine eigene Implementierung angegeben werden. Diese kann auf eine Implementierung im Interface zurückgreifen.

```
interface A {
  default void m() {
     System.out.println("Hier in A.m()");
  }
}

interface B {
  default void m(){
     System.out.println("Hier in B.m()");
  }
}

class C implements A, B {
   public void m(){
      A.super.m();
   }
}
```

2 Ausnahmebehandlung mit Exceptions

Wie im Abschnitt zur "Terminierung" bereits angesprochen, kann die Auswertung eines Ausdrucks bzw. die Ausführung einer Anweisung:

- normal terminieren,
- in eine Ausnahmesituation kommen und abrupt terminieren, oder
- nicht terminieren.

Es gibt drei Arten von Ausnahmesituationen:

- Vom Programmierer schwer zu kontrollierende und zu beseitigende Situationen (z.B. Speichermangel)
- Programmierfehler (z.B. Null-Dereferenzierung, Verletzung von Indexgrenzen)
- Zeitweise nicht verfügbare Ressourcen, anwendungsspezifische Ausnahmen, die behebbar sind (z.B. Netzwerkunterbrechungen)

Programmiersprachen gehen mit Ausnahmesituation ganz unterschiedlich um. Sie können entweder behandelt werden oder zu einem Programmabbruch führen (engl. *abort*).
Java bietet Sprachmittel für die ***Ausnahmebehandlung*** (engl. ***exception handling***). Dabei spielen drei Aspekte eine Rolle:

1. Wann/wie werden Ausnahmen ausgelöst?
2. Wie kann man sie abfangen?
3. Welcher Ausnahmetyp ist passend bzw. wie kann man neue Ausnahmetypen deklarieren?

Auslösen von Ausnahmen Das Auslösen einer Ausnahme kann sprachdefiniert (z.B. NullPointer, IndexOutOfBounds) oder durch eine Anweisung spezifiziert sein. In Java gibt es zum Auslösen von Ausnahmen die `throw`-Anweisung.

Syntax:

```
Anweisung → throw  Ausdruck ;
```

Dabei muss der Ausdruck ein Ausnahmeobjekt als Ergebnis liefern.

Semantik:

- Werte zunächst den Ausdruck aus.
- Löst die Auswertung eine Ausnahme aus, ist dies die Ausnahme, die von der Anweisung ausgelöst wird.
- Andernfalls löse die Ausnahme aus, die das Ergebnis des Ausdrucks ist.

Abfangen von Ausnahmen Die try-catch-Anweisung dient dem Abfangen und Behandeln von Ausnahmen.

Syntax:

```
Anweisung →
    try {
      DeklAnweisListe
    } catchKlauselListe

CatchKlauselListe →
    CatchKlausel CatchKlauselListe
  | ϵ

CatchKlausel →
    catch ( ≪ Typ ≫ ≪ Bezeichner ≫ ) {
        DeklAnweisListe
    }
```

Semantik:

- Führe die Anweisung im `try`-Block aus.
- Löst dies eine Exception aus, wird ein entsprechendes `Exception`-Objekt erzeugt.

- Ist der Typ dieses Objekt in der Liste der `catch`-Klauseln aufgeführt, wird das Objekt an den Bezeichner der entsprechenden `catch`-Klausel gebunden und diese `catch`-Klausel ausgeführt. Andernfalls wird die Exception an den umfassenden Ausführungskontext weitergereicht.

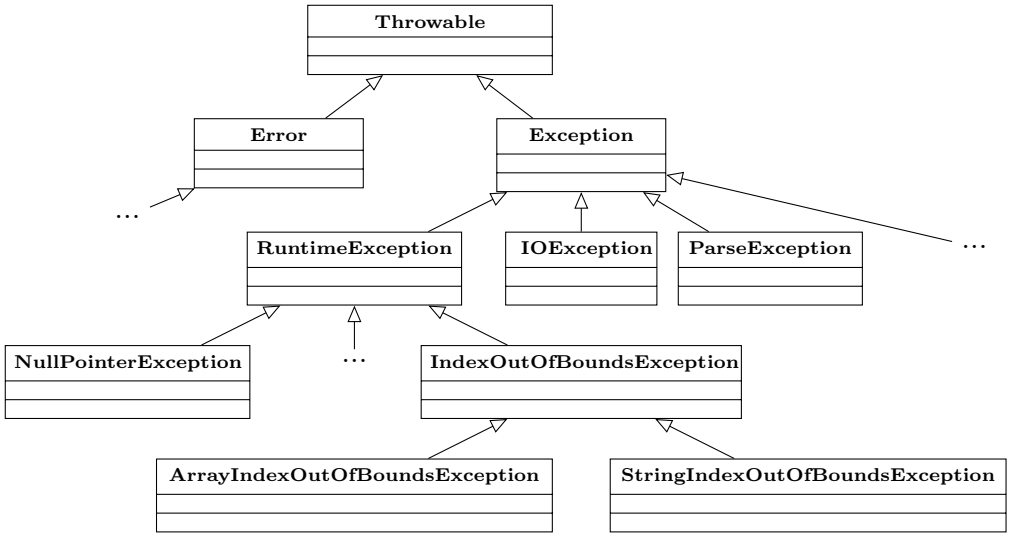
Hier sehen wir ein Beispiel dazu:

```
3  void myMethod (String[] s) {
4     try {
5       StdOut.println(s[0]);
6       StdOut.println(s[1]);
7    } catch (NullPointerException e) {
8       StdOut.println("s is null");
9    } catch (IndexOutOfBoundsException e){
10      StdOut.println("s too small");
11   }
12 }
```

Tritt eine Ausnahme vom Typ `NullPointerException` oder `IndexOutOfBoundsException` im `try`-Block (Zeile 5-6) auf, wird ein entsprechendes Objekt `x` erzeugt. In diesem Fall wird die Ausnahme *gefangen*. Das Objekt wird an den Bezeichner `e` der entsprechenden `catch`-Klausel gebunden und diese `catch`-Klausel dann ausgeführt (für weitere Details siehe Abschnitt 2.2).

2.1 Benutzerdefinierte Exceptions

Ausnahmen werden in Java durch Objekte repräsentiert; d.h. wenn eine Ausnahme auftritt, wird ein entsprechendes Ausnahmeobjekt erzeugt. Die Verwendung von benutzerdefinierten Exceptions unterscheidet sich nicht von denen, die Java bereitstellt. Benutzerdefinierte Ausnahmen müssen in die Typhierarchie der Java-Exceptions eingegliedert werden.
Ausschnitt aus der Typhierarchie der Exceptions:



Klassifizierung von Ausnahmen und Fehlern

- Die Klasse `Throwable` aus dem Packet `java.lang` ist die Superklasse aller Ausnahmen- und Fehlerklassen in Java.
- `Error`s sind schwerwiegende Fehler, die in der Regel nicht vom Programm abgefangen und behandelt werden sollten.
- `Exception`s sollten vom Programm abgefangen und geeignet behandelt werden.

Beispiel Ausnahmebehandlung bei Integer-Overflow

```
1  public class UeberlaufException extends Exception { }
2
3  public class ExceptionTest {
4   public static void main(String[] args) {
5    try {
6     int ergebnis = 0;
7     int m = Integer.parseInt(args[0]);
8     int n = Integer.parseInt(args[1]);
9     long aux = (long) m + (long) n;
10    if (aux > Integer.MAX_VALUE) {
11       throw new UeberlaufException(); // selbstdefiniert
12    }
13    ergebnis = (int) aux;
14   } catch (IndexOutOfBoundsException e) {
15      System.out.println("Zuwenig Argumente");
16   } catch (NumberFormatException e) {
17      System.out.println("Parameter ist keine int-Konstante");
18   } catch (UeberlaufException e) {
19      System.out.println("Ueberlauf bei Addition");
20   }
21  }
22 }
```

2.2 Reihenfolge der Exception-Handler

Ein Handler für Exceptions einer Klasse `X` behandelt auch Exceptions aller von `X` abgeleiteten Klassen (Stichwort: Subtyp-Polymorphie). Die Reihenfolge der `catch`-Blöcke ist daher relevant bei der Ausnahmebehandlung. Zunächst müssen die Handler für die am meisten spezialisierten Klassen aufgelistet werden und dann mit zunehmender Generalisierung die Handler für die entsprechenden Superklassen. Dies wird vom Compiler überprüft, um Fehler zu vermeiden.
Beispiel Hier noch einmal eine Variante des Overflow-Beispiels, bei dem zunächst die `IndexOutOfBoundsException` abgfangen wird, alle anderen Exceptions werden durch die zweite `catch`-Klausel abgefangen.

```
3  public class UeberlaufException extends Exception {
4    UeberlaufException() {
5      super("Ueberlauf bei der Integer-Addition");
6    }
7  }
8  public class ExceptionTest {
9   public static void main(String[] args) {
10   try {
11    int ergebnis = 0;
12    int m = Integer.parseInt(args[0]);
13    int n = Integer.parseInt(args[1]);
14    long aux = (long) m + (long) n;
15    if (aux > Integer.MAX_VALUE) {
16       throw new UeberlaufException(); // selbstdefiniert
17    }
18    ergebnis = (int) aux;
19   } catch (ArrayIndexOutOfBoundsException e) {
20      System.out.println("Zuwenig Argumente");
21   } catch (Exception e) {
22      System.out.println("Ein Fehler ist aufgetreten");
23      e.getMessage();
24   }
25  }
```

- Der `Exception`-Konstruktor kann einen String-Parameter nehmen, der eine Beschreibung des Fehlers enthält.
- Die Fehlerbeschreibung kann durch die Methode `getMessage()` in der Fehlerbehandlung ausgelesen werden.

2.3 Checked und Unchecked Exceptions

In Java zeigt die `throws`-Klausel in einer Methodensignatur an, dass eine Methode Exceptions auslöst bzw. weitergibt, ohne sie selbst abzufangen.
Beispiel Folgendes Programm verwendet Exceptions um die Gültigkeit von Parametern zu überprüfen. Falls die Methode `checkPreconditions()` eine Exception auslöst, wird diese an die aufrufende Methode `average()` weitergeleitet. Diese wiederum leitet sie an die `main()`-Methode weiter.

```
3  public static void main(String[] args) throws Exception {
4      double[] a = {};
5      System.out.println("avg = " + average(a));
6  }
7  static double average(double[] a) throws Exception {
8      checkPreconditions(a);
9      int sum = 0;
10     for (int i = 0; i < a.length; i++) {
11         sum += a[i];
12     }
13     return sum / a.length;
14 }
15 static void checkPreconditions(double[] a) throws Exception {
16     if (a == null) {
17         throw new Exception("Array darf nicht null sein");
18     }
19     if (a.length == 0) {
20         throw new Exception("Array darf nicht leer sein");
21     }
22 }
```

In Java unterscheidet man zwei Arten von Exceptions:

- ***Checked Exceptions*** müssen, wenn sie in einer Methode geworfen werden, entweder dort in einem Exception Handler behandelt werden, oder aber in einer `throws`-Klausel in der Methodensignatur angegeben werden. Methodensignaturen können auch mehrere Exceptions durch Komma getrennt enthalten. Dies wird vom Compiler überprüft.
- ***Unchecked Exceptions*** müssen nicht entsprechend deklariert werden. Nur Exceptions vom Typ `RuntimeException` und `Error` sind unchecked.

2.4 Stacktraces

Im obigen Programm führt die Ausführung zu einem Programmabbruch, weil die Ausnahme nirgendwo in einem `try`-Block behandelt wird. Als Fehlermeldung wird ein sogenannter *Stacktrace* angezeigt:

```
Exception in thread "main" java.lang.Exception: Array darf nicht leer sein
    at ExceptionTest.checkPreconditions(ExceptionTest.java:20)
    at ExceptionTest.average(ExceptionTest.java:8)
    at ExceptionTest.main(ExceptionTest.java:5)
```

Stacktraces sind folgendermaßen strukturiert:

- Die erste Zeile enthält die Fehlermeldung mit Klassenname der Ausnahme (hier: `java.lang.Exception`)
- Die folgende Zeilen bestehen aus einer Liste der Methoden-Aufrufe, die beim Werfen der Ausnahme aktiv waren. Jede dieser Zeilen hat die Form:

```
atKlassenname.Methodenname(Dateiname:Zeilennummer)
```

Die Zeilennummern geben jeweils die aktuelle Position im Programm zum Zeitpunkt der Ausnahme an.

2.5 Vorteile von Exceptions

Die Verwendung von Exceptions in Java hat einige Vorteile:

- Der Standardfall in der Ausführung einer Methode wird von der Fehlerbehandlung syntaktisch getrennt. Damit wird der Code besser verständlich und übersichtlicher.
- Falls eine Methode mit der Fehlersituation nicht umgehen kann, wird der Fehler an den Aufrufer weitergegeben in der Hoffnung, dass dieser den Fehler behandeln kann. Findet sich keine passende `catch`-Klausel, führt der Fehler zum Programmabbruch.
- Programmierer können auf verschiedene Fehlertypen unterschiedlich reagieren. Diese Fehlertypen können auch zu Gruppen zusammengefasst werden und so eine gleiche Behandlung erfahren. Dies vereinfacht wiederum die Fehlerbehandlung.

Exceptions sollten nicht eingesetzt werden, um den Kontrollfluß im regulären Ausführungsfall zu kontrollieren. Schreiben Sie bitte **niemals** folgende Art von Code:

```
...
try {
  if (i == 0) {
    throw new Exception("i ist 0");
  } else {
    throw new Exception("i ist ungleich 0");
  }
} catch (Exception e) {
  StdOut.println(e);
}
```

Der Exception-Mechanismus ist führt zu vielen zusätzlichen Instruktionen bei der Ausführung und ist in vielen Programmiersprachen daher vergleichsweise ineffizient. Außerdem führt es zu unübersichtlichem Programmstrukturen. Exceptions sollten daher nur verwendet werden, wenn es sich tatsächlich um eine Fehlersituation handelt.

3 Klassenattribute und -methoden

Wir haben bereits zu Beginn der Vorlesung gesehen, dass Methoden mit dem Schlüsselwort `static` gekennzeichnet werden können. Dies gilt auch für Attribute. Diese statischen Methoden bzw. Attribute nennt man auch ***Klassenattribute/variablen bzw. -methode***. In diesem Abschnitt diskutieren wir den Unterschied zwischen Instanzattributen bzw. -methoden von Objekten und Klassenattributen bzw. -methoden.

3.1 Klassenattribute

Jedes Objekt, das von derselben Klasse instantiiert wird, hat für jedes seiner Attribute eine eigene Kopie. Im folgenden Beispiel hat jede Übungsgruppe einen eigenen Namen und eine eigene Anzahl von Teilnehmern. Bisweilen ist es aber auch wünschenswert Attribute zu haben, die für alle Objekte einer Klasse gleich sind. Bei der Übungsgruppen ist die maximale Anzahl an Teilnehmern ein solches Attribut. Dieses Attribut ist mit der *Klasse* assoziiert, nicht mit den einzelnen Objekten. Es gibt nur eine Instanz/Kopie dieses Attributs. Dieses kann von allen Methoden der Klasse verändert werden, d.h. sowohl in Instanzmethoden als auch in Klassenmethoden (bei `public` Attributen auch von außerhalb der Klasse). Klassenattribute können verwendet werden, auch wenn keine Objekte dieser Klasse erzeugt wurden.

Beispiel: Übungsgruppen

```
class Uebungsgruppe {
  static int maxTeilnehmer = 30;
  private String gruppenname;
  private Set<String> teilnehmer;

  Uebungsgruppe(String gruppenname){
    this.gruppenname = gruppenname;
    this.teilnehmer = new HashSet<String>();
  }  ...

  int freiePlaetze() {
    return maxTeilnehmer - teilnehmer.size();
  }
}

...
public static void main (String[] args) {
    // Verwendung ohne Erzeugung von Objekten
    int i = Uebungsgruppe.maxTeilnehmer;

    Uebungsgruppe u = new Uebungsgruppe("Montagsgruppe");
    Uebungsgruppe u2 = new Uebungsgruppe("Dienstagsgruppe");
    System.out.println("Freie Plaetze montags: " + u.freiePlaetze());
    System.out.println("Freie Plaetze dienstags: " + u2.freiePlaetze());

    // Veraendern des Klassenattributs
    Uebungsgruppe.maxTeilnehmer = 32;
    System.out.println("Freie Plaetze montags: " + u.freiePlaetze());
    System.out.println("Freie Plaetze dienstags: " + u2.freiePlaetze());
}
```

Klassenattribute werden folgendermaßen deklariert:

```
static Typausdruck Attributname;
```

Die Variable kann innerhalb der Klasse mit dem Attributnamen, außerhalb mittels

```
Klassenname . Attributname
```

angesprochen werden.

Die Lebensdauer der Klassenattribute entspricht der Lebensdauer der Klasse (d.h. vom Laden der Klasse bis zum Programmende). Sie werden beim Laden der Klasse initialisiert, sobald die Klasse das erste Mal verwendet wird.

Beispiel: Matrikelnummer `Student`-Objekte sollen mit einer fortlaufenden Matrikelnummer versehen werden. Der Zähler für die Matrikelnummer wird als Klassenattribut realisiert.

```
class Student {
    // Instanzattribut fuer die Matrikelnummer
    private int matrikel;
    private String name;

    // Klassenattribut fuer den Zaehler;
    // initialisiert fuer 8stellige Matrikelnummern
    private static int matrikelCount = 10000000;

    Student(String name) {
      this.name = name;
      this.matrikel = matrikelCount++;
    } ...
}
```

Klassenattribute haben Ähnlichkeit mit globalen Variablen in der prozeduralen Programmierung. Sie erhöhen die Abhängigkeiten zwischen verschiedenen Code-Teilen, sind schwieriger zu testen, und weniger flexibel.

> **Frage 2:** Zu welchen Problemen führt die Verwendung des Klassenattributs `matrikelCount` im obigen Beispiel, wenn Studiernde verschiedener Universitäten verwaltet werden sollen?

Klassenattribute werden sinnvoll eingesetzt, um mittels Modifikator `final` globale Konstanten zu definieren (siehe z.B. `Math.PI`, `Integer.MAX_VALUE`).

3.2 Klassenmethoden

Klassenmethoden (statische Methoden) haben wir bereits in Kapitel 2 als Prozeduren eingeführt. Um Klassenmethoden zu verwenden muss keine Objektinstanz erzeugt werden. Sie besitzen keinen impliziten Parameter (`this`), da auch sie mit der Klasse und nicht mit den von der Klasse instantiierten Objekten assoziiert sind.
Klassenmethoden können nur auf Klassenattribute, Parameter und lokale Variable zugreifen. Wie Klassenattribute werden sie durch das Schlüsselwort `static` gekennzeichnet und über den Klassennamen adressiert. Innerhalb der Klasse kann der Klassenname entfallen.

Beispiel: Klassenmethoden Die `String`-Klasse besitzt Klassenmethoden / Prozeduren, die eine (explizite) Konvertierung von Werten elementarer Datentypen in Strings erlauben:

```
class String {
    ...
    static String valueOf(long l)  { ... }
    static String valueOf(float f) { ... }
    ...
}
```

Die Anwendung bzw. der Aufruf `String.valueOf( (float)(7./9.) )` liefert den String `"0.7777778"`.

Beispiele: Klassenattribute, -methoden Charakteristische Beispiele für Klassenattribute und -methoden liefert die Klasse `System`, die eine Schnittstelle von Programmen zur Umgebung bereitstellt:

```
class System {
  final static InputStream in  = ...;
  final static PrintStream out = ...;
  static void exit(int status)    { ... }
  static long currentTimeMillis() { ... }
}
```

Die Klasse `PrintStream` besitzt Methoden `print` und `println`:

```
System.out.print("Das erklaert die Syntax");
System.out.println(" von Ausgaben!");
```

Zusammenfassender Überblick

- Instanzmethoden können Instanzvariablen (Attribute von Objekten) und Instanzmethoden direkt verwenden.
- Instanzmethoden können Klassenattribute und -methoden direkt verwenden.
- Klassenmethoden können Klassenattribute und -methoden direkt verwenden.
- Klassenmethoden können Instanzvariablen oder -methoden **nicht** direkt vewerden; sie benötigen dazu eine Objektreferenz; ebenso kann `this` innerhalb von Klassenmethoden nicht verwendet werden, da es keine entsprechende Objektinstanz gibt.

4 Geschachtelte Klassen

Klassen- (und auch Interface-Definitionen) können zur besseren Strukturierung von Code geschachtelt werden, so dass "Hilfsklassen" bei ihrer Verwendungsstelle deklariert werden und bei Bedarf auch als Implementierungsdetails verborgen werden können (Stichwort: Information Hiding).
Eine Klasse heißt in Java ***geschachtelt*** (engl. *nested*), wenn sie innerhalb der Deklaration einer anderen Klasse deklariert ist:

- als Komponente (ähnlich einem Attribut) (sogenannte statische und innere Klassen),
- als ***lokale*** Klassen (ähnlich einer lokalen Variablen),
- als ***anonyme*** Klassen bei der Objekterzeugung.

Ist eine Klasse nicht geschachtelt, nennen wir sie ***global*** (engl. *top-level*).
Wir betrachten hier als Beispiel die statischen geschachtelten Klassen, anhand derer wir die grundlegenden Konzepte erläutern.

Statische Klassen Geschachtelte Klassen heißen ***statisch***, wenn sie mit dem Modifikator `static` deklariert sind. Statische Klasse haben die gleiche Bedeutung und Verwendung wie globale Klassen. Sie können erben und Interfaces implementieren und generisch sein.
Achtung: Eine geschachtelte Klasse übernimmt **nicht** die Superklassen und Interfaces der umschließenden Klasse!
Allerdings ergeben sich andere Sichtbarkeits- und Zugriffsbereiche. Insbesondere gilt:

- Geschachtelte Klassen können als privat deklariert werden, so dass sie außerhalb der umfassenden Klasse nicht verwendet werden können.
- Zugreifbare statische geschachtelte Klassen können über zusammengesetzte Namen außerhalb der umfassenden Klasse angesprochen werden (z.B. `Map.Entry` für die Einträge der Maps aus den Java Collections).
- Geschachtelte Klassen können auf private Attribute und Methoden der umfassenden Klasse zugreifen. Wie bei statischen Methoden kann eine statische Klasse nicht direkt auf Instanzvariablen oder - methoden zugreifen, sondern benötigt dazu eine Objektrefernz.

Beispiel: Statische Klassen Ein typischer Anwendungsfall für geschachtelte statische Klassen sind die Knotenklassen für verlinkte Listen und Bäume.

```
public class LinkedList {
   private Node entries = null;
   private int size = 0;

   private static class Node {
      int elem;
      Node next;
   }
   int getFirst() { ... }
   ...
}
```

Bei dieser Implementierung ist der Typ `Node` nur innerhalb von `LinkedList` sichtbar und zugreifbar. Statische geschachtelte Klassen können sich **nicht** auf die Typparameter der äußeren Klasse beziehen. Im folgenden Beispiel kann in der Klasse `Node` der Typparameter `T` nicht verwendet werden; stattdessen wird `Node` über einen Typen `S` parametrisiert, der in der Klasse `LinkedList` mit `T` instantiiert wird.

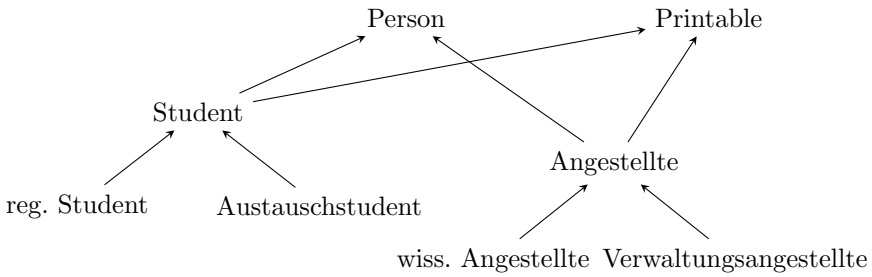
```
public class LinkedList<T> {
   private Node<T> entries = null;
   private int size = 0;

   private static class Node<S> {
      S elem;
      Node<S> next;
   }

   T getFirst() { ... }
   ...
}
```

Hinweise zu den Fragen

Hinweise zu Frage 1:



Hinweise zu Frage 2: Das Klassenattribut `matrikelCount` ist allen Student-Objekten gemein. Es kann nicht zwischen den Studierenden verschiedener Universitäten unterscheiden. Bei einer Erweiterung des Systems muss der Matrikelzähler daher anders realisiert werden (z.B. als Attribut der Universitätsklasse).