Abschnitt 6: Klassen, Objekte und Methoden in Java

Überblick

Klassendefinition

- Eine Klassendefinition in Java besteht aus dem Klassennamen und dem Klassenrumpf, der wiederum aus zwei Teilen besteht:
  - Die Attributdeklaration enthält die Liste der Attribute der Klasse. Jedes Attribut hat einen Namen und einen Typ. Dafür sind entweder primitive Typen, Reihungen oder andere Klassen (Objekttypen) erlaubt.
  - Die Methodendeklaration enthält die Liste der Methoden der Klasse. Als Typen für die Methodenparameter sind wiederum primitive Typen, Reihungen oder Objekttypen erlaubt.
- Bei allen Attributen und Methoden muss zusätzlich die *Sichtbarkeit* spezifiziert werden, d.h. ob andere Klassen das entsprechende Attribut oder die entsprechende Methode sehen und benutzen können. Dies ist für die Kapselung wichtig.
  - Das Schlüsselwort **`private`** bedeutet, dass die entsprechenden Attribute/Methoden von außerhalb nicht sichtbar sind.
  - Das Schlüsselwort **`public`** bedeutet, dass die entsprechenden Attribute/Methoden von außerhalb sichtbar sind.

Klassendefinition

- Beispiel: Eine Definition der Klasse “Auto”

```
public class Auto
{
  // Attribute

  /**
   * Der Modellname des Autos.
   */
  private String name;

  /**
   * Das Jahr der Erstzulassung des Autos.
   */
  private int erstzulassung;

  /**
   * Die Leistung des Autos in PS.
   */
  private int ps;

  // Methoden

...

}
```

Klassendefinition

- Konventionen:
  - Klassennamen beginnen mit Großbuchstaben.
  - Attributsnamen und Methodennamen beginnen mit kleinen Buchstaben.
  - Konstantennamen bestehen komplett aus Großbuchstaben.

Überblick

Objekterzeugung

- Um ein Objekt der Klasse `Auto` zu erzeugen, muss man an der ensprechenden Stelle im Programm eine Variable vom Typ der Klasse deklarieren und ihr mit Hilfe des **`new`**-Operators ein neu erzeugtes Objekt zuweisen:

```
Auto meinKombi;
meinKombi = new Auto();
```

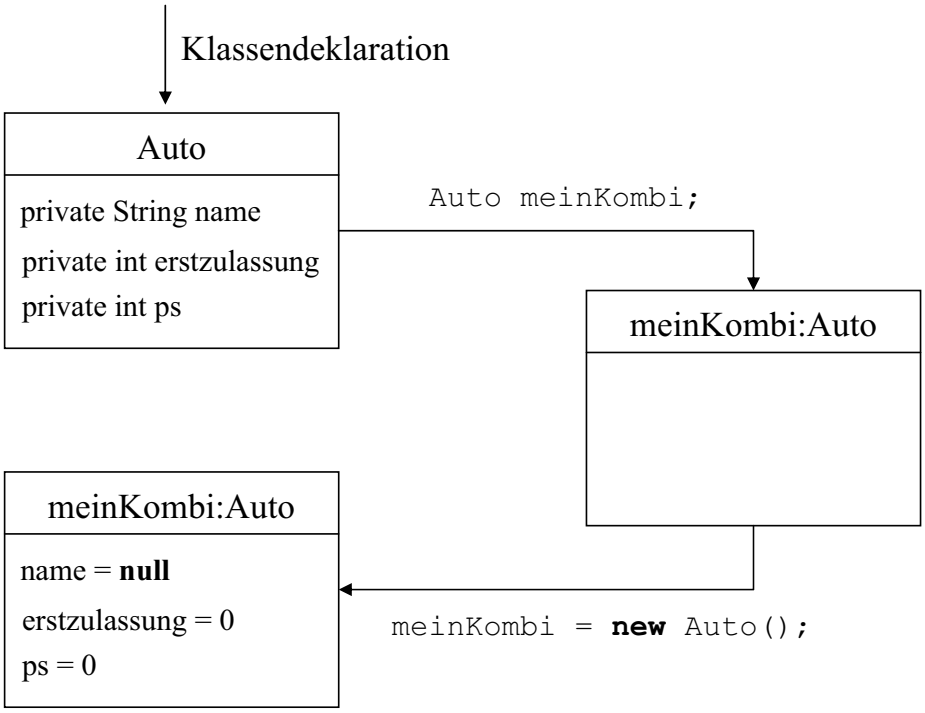
- Die erste Anweisung ist eine klassische Variablendeklaration. Anstelle eines primitiven Datentyps wird hier der Name einer zuvor definierten Klasse verwendet (Objekttyp).
- Die Variable `meinKombi` wird auf den Speicher gelegt mit einer Referenz auf die Halde. Dort wird später das konkrete Objekt gespeichert sein, das noch nicht generiert ist.
- Die zweite Anweisung generiert das Objekt mittels des **`new`**-Operators.
- Da Arrays auch Objekte sind, wissen Sie nun auch, warum man bei der Erzeugung eines Arrays den **`new`**-Operator benötigt (wenn man kein Literal benutzt).

Objekterzeugung

- Nach der Generierung eines Objekts haben alle Attribute mit primitiven Datentypen des Objekts zunächst ihre entsprechenden Standardwerte (siehe Arrays).
- Attribute mit Objekttypen haben den Standardwert **`null`**, die “leere Referenz”.
- Auf den Zustand und die Methoden eines Objekts kann nun wie folgt zugegriffen werden:
  - Auf alle mit **`private`** versehenen Attribute und Methoden kann außerhalb der Klasse *nicht* zugegriffen werden.
  - Auf alle mit **`public`** versehenen Attribute und Methoden kann außerhalb der Klasse über die Punktnotation zugegriffen werden: gäbe es z.B. ein Attribut **`public int`** `kaufpreis` in der Deklaration der Klasse `Auto`, so könnte man mit `myKombi.kaufpreis` darauf lesend/schreibend zugreifen.
- Offensichtlich widerspricht der direkte Zugriff auf den Zustand (Attribute) eines Objekts dem Prinzip der Kapselung. Daher ist es guter oo Programmierstil, alle Attribute als **`private`** zu deklarieren.

Objekterzeugung

- Beispiel:



Objekterzeugung

- Nochmals zu Arrays:
  - Eigentlich (im Sinne der Kapselung) sollten Attribute als **`private`** vereinbart werden.
  - Was verbirgt sich hinter `beispielArray.length`?
  - Eigentlich ist das ein Attribut, auf das man von außerhalb der Klasse zugreifen kann, also **`public`** ist.
  - Allerdings kann man (aus naheliegenden Gründen) hier nicht schreibend sondern nur lesend zugreifen.
- Tatsächlich sind Arrays besondere Objekte, zu denen es keine “echte” Klassendefinition gibt.
- Daher unterscheidet sich die “Schnittstelle” (die **`public`** Methoden/Attribute) von Arrays zu den Schnittstellen klassischer Objekte.
- Arrays werden ansonsten genauso behandelt wie “normale” Objekte.

Gleichheit von Objekten

- Was bedeutet, dass zwei Objekte gleich sind?
- Intuitiv: dass ihre Attribute dieselben Werte haben, d.h. dass sie denselben Zustand haben.

```
Auto meinKombi;
meinKombi = new Auto();
Auto meinSportwagen;
meinSportwagen = new Auto();

boolean vergleich = meinKombi == meinSportwagen;  // (*)

meinKombi = meinSportwagen;

vergleich = meinKombi == meinSportwagen;          // (**)
```

- Was ist der Wert der Variablen `vergleich` an der Stelle `(*)`?

Gleichheit von Objekten

- Der Wert der Variablen `vergleich` an der Stelle `(*)` ist **`false`**!
- Warum? Sowohl das Objekt `meinKombi` als auch das Objekt `meinSportwagen` haben doch als Attributwerte die Standardwerte, also haben sie insgesamt denselben Zustand.
- Richtig! Aber wir sind einem call-by-reference-Effekt bei Objekten auf den Leim gegangen.
- `meinKombi` ist eine Variable, die die Referenz zu einem speziellen Objekt speichert. Ebenso `meinSportwagen`. Beide speichern natürlich unterschiedliche Referenzen, denn die Variablen repräsentieren ja unterschiedliche Objekte.
- Der Wert der Variablen `vergleich` an der Stelle `(**)` ist dagegen **`true`**.

Gleichheit von Objekten

- **Achtung:** Es gibt zwei “Arten” von Gleichheit von Objekten bzw. Objektvariablen (Variablen mit Objekttyp):
- Gleichheit: der Zustand der entsprechenden Objekte beider Objektvariablen ist gleich.
- Identität: beide Objektvariablen verweisen auf die gleiche Speicheradresse.
- Der Operator == prüft den zweiten Fall (Identität). Er kann also nicht dazu benutzt werden, abzufragen, ob die Objekte zweier Objektvariablen gleich bzgl. ihres Zustands sind.
- Dies wird oft übersehen und ist daher eine häufige Fehlerquelle!
- Um Gleicheit von Objekten zu prüfen, benötigt man andere Konstrukte, die wir später kennenlernen werden.

Überblick

Methoden

- Frage: Wenn der Zustand (alle Attribute) des Objekts gekapselt ist, wie kann ich dann auf den Zustand zugreifen bzw. wie kann ich ihn verändern?
- Antwort: Mit Hilfe der Methoden.
- Die Methoden einer Klasse definieren das Verhalten der Objekte der Klasse.
- Die Methoden einer Klasse haben Zugriff auf alle Attribute und Methoden der Klasse (auch auf den **`private`**-Teil).
- Methoden können wiederum von außen sichtbar (**`public`**) oder nicht sichtbar (**`private`**) sein. Die Methoden, die das Verhalten der Objekte spezifizieren, sollten alle sichtbar sein. Darüberhinaus kann es natürlich auch noch nicht sichtbare Hilfsmethoden geben.
- Beispiel: Klasse `Auto`

Methoden

```
public class Auto
{
  ... // siehe oben

  /**
   * Gibt das Jahr der Erstzulassung des Auto-Objekts zur&uuml;ck.
   * @return  das Jahr der Erstzulassung des Autos.
   */
  public int getErstzulassung()
  {
    return erstzulassung;
  }

  /**
   * Ver&auml;ndert das Jahr der Erstzulassung des Auto-Objekts.
   * @param jahr  das Jahr der Erstzulassung des Autos.
   */
  public void setErstzulassung(int jahr)
  {
    erstzulassung = jahr;
  }

  /**
   * Berechnet das Alter (in Jahren) des Auto-Objekts.
   * @param aktuellesJahr  das aktuelle Jahr, z.B. 2007.
   * @return  das Alter des Autos.
   */
  public int alter(int aktuellesJahr)
  {
    return aktuellesJahr - erstzulassung;
  }

}
```

Methoden

- Alle Methoden sind sichtbar und haben ihrerseits Zugriff auf alle nicht sichtbaren Attribute der Klasse Auto.
- Kapselung: Das Jahr der Erstzulassung des Autos ist in diesem Beispiel nur über die Methoden `getErstzulassung` bzw. `setErstzulassung` von außerhalb lesend/schreibend zugreifbar. Dieser Zugriff ist durch die beiden Methoden wohldefiniert.

Methodenaufruf

- Die **`public`** Methoden können wie bereits erwähnt mit der Punktnotation aufgerufen werden.

```
Auto golf = new Auto();
golf.setErstzulassung(1998);
int e = golf.getErstzulassung(); // Wert von e ist 1998
int a = golf.alter(2007); // Wert von a ist 9
```

- Wie man auf der Folie davor sieht, darf eine Methode auf die Attribute (und auch auf die anderen Methoden) der Klasse ohne Punktnotation zugreifen.
- Tatsächlich bezieht der Compiler alle Variablen `x`, die nicht in Punktnotation verwendet werden, auf das Objekt **`this`**, d.h. `x` wird eigentlich als **`this`**`.x` interpretiert. Ausnahme: es gibt eine lokale Variable (z.B. in einem Methodenrumpf), die genauso wie ein Klassenattribut heißt.

Methodenaufruf

- Bei **`this`** handelt es sich um eine Art Objektvariable (also eine Referenz auf ein Objekt), die beim Anlegen eines Objekts automatisch generiert wird.
- **`this`** zeigt auf das aktuelle Objekt und wird dazu verwendet, die eigenen Methoden und Attribute anzusprechen.
- **`this`** kann auch explizit verwendet werden:

```
public int alter(int aktuellesJahr)
{
  return aktuellesJahr - this.erstzulassung;
}
```

- **`this`** muss explizit verwendet werden, wenn gleichnamige lokale Variablen (z.B. Parameter) verwendet werden.

Methodensignatur

- Die *Signatur* einer Methode setzt sich zusammen aus
  - dem Namen der Methode,
  - der Reihenfolge und Typen der Eingabeparameter,
  - dem Ausgabetyp.
- Methoden können grundsätzlich den selben Namen habe, solange sie sich in ihrer Parameterliste unterscheiden. Man spricht in diesem Fall von *Überladen*.
- Die beiden Methoden

```
public int alter(int aktuellesJahr)
{
  return aktuellesJahr - this.erstzulassung;
}
public int alter(double aktuellesJahr)
{
  return (int) aktuellesJahr - this.erstzulassung;
}
```

sind verschieden.

Methodensignatur

- Dagegen sind die beiden Methoden

```
public int alter(int aktuellesJahr)
{
  return aktuellesJahr - this.erstzulassung;
}
public double alter(int aktuellesJahr)
{
  return (double) (aktuellesJahr - this.erstzulassung);
}
```

nicht verschieden!

Konstruktoren

- Bei unserer Beispielklasse `Auto` ist das Jahr der Erstzulassung von außen über die Methoden `setErstzulassung` veränderbar.
- Dies ist vermutlich nicht besonders glücklich: Der Wert des Attributs `erstzulassung` sollte einmal initialisiert werden können, dann aber nicht mehr verändert werden können.
- D.h., die Methode `setErstzulassung` sollte nicht zur Verfügung stehen.
- Dann kann das Attribut `erstzulassung` aber auch nicht verändert werden, schließlich ist es von außerhalb nicht sichtbar.

Konstruktoren

- Dieses Dilemma wird durch die *Konstruktoren* gelöst (auch wenn es nicht der Grund ist, warum es Konstruktoren überhaupt gibt).
- Konstruktoren sind spezielle Methoden, die zur Erzeugung und Initialisierung von Objekten aufgerufen werden können.
- Konstruktoren sind Methoden *ohne* Rückgabwert (nicht einmal **`void`**), die den Namen der Klasse erhalten.
- Konstruktoren können eine beliebige Anzahl von Eingabe-Parametern besitzen und überladen werden.
- Ein Konstruktor, der z.B. das Jahr der Erstzulassung als Parameter übergeben bekommt und das Attribut `erstzulassung` entsprechend initialisiert, löst also unser oben angesprochenes Dilemma.

Konstruktoren

```
public class Auto
{
  /** Der Modell-Name des Autos */
  private String name;
  /** Das Jahr der Erstzulassung des Autos */
  private int erstzulassung;
  /** Die Leistung des Autos in PS */
  private int ps;

  /**
   * Konstruktor. Generiert ein neues Objekt Auto.
   * @param name           der Name des Modells.
   * @param erstzulassung  das Jahr der Erstzulassung.
   * @param ps             die Leistung in PS.
   */
  public Auto(String name, int erstzulassung, int ps)
  {
    this.name = name;
    this.erstzulassung = erstzulassung;
    this.ps = ps;
  }

  // Methoden (jetzt ohne "setErstzulassung")
  ...
}
```

Konstruktoren

- Aufruf:

```
Auto golf;
golf = new Auto("Golf",1998,102);
int e = golf.getErstzulassung(); // Wert von e ist 1998
int a = golf.alter(2007); // Wert von a ist 9
```

- Wird kein expliziter Konstruktor deklariert, gibt es einen Default-Konstruktor (den wir bisher benutzt haben und der keine Eingabeparameter hat – `Auto()` in unserem Beispiel).
- Wird mindestens ein expliziter Konstruktor vereinbart, steht kein Default-Konstruktor zur Verfügung!
- **Bemerkung:** Neben den Konstruktoren gibt es noch die Möglichkeit, initiale Attributwerte in der Klassendefinition direkt zu vereinbaren (Die Attributsdefinitionen sehen dann so aus wie Variablenvereinbarungen mit Initialisierung). Offensichtlich löst dies aber nicht das vorher angesprochene Dilemma.

Überblick

Zusammenspiel von imperativen und objektorientierten Aspekten in Java

- Wie passen unsere imperativen Konstrukte aus Teil I hier dazu?
- Es gibt in Java *statische* und *nicht-statische* Elemente einer Klasse.
- Die statischen Elemente (Methoden oder Attribute) sind mit dem Schlüsselwort **`static`** gekennzeichnet.
- Fehlt **`static`**, ist das entsprechende Element (Methode/Attribut) nicht-statisch.
- Statische Elemente existieren unabhängig von Objekten, wogegen nicht-statische Elemente an die Existenz von Objekten gebunden sind.
- Werden statische Variablen von einem Objekt verändert, ist diese Veränderung auch in allen anderen Objekten sichtbar.

Zusammenspiel von imperativen und objektorientierten Aspekten in Java

Beispiel:

- Ein imperatives Programm `Programm` in Java besteht aus einer Klassendefinition für die Klasse `Programm` mit einer statischen `main`-Methode.
- Diese Methode existiert unabhängig von Objekten der Klasse `Programm`.
- Die Klasse `Auto` in unserem vorherigen Beispiel spezifiziert ein nicht-statisches Attribut `name`.
- Dieses Attribut existiert nur dann, wenn es mindestens ein Objekt der Klasse `Auto` gibt. Für jedes existierende Objekt der Klasse `Auto` existiert ein Attribut `name`.
- Statische und nicht-statische Elemente können “nebeneinander” verwendet werden.
- Für statische Elemente stehen natürlich auch **`private`** und **`public`** zur Spezifikation der Sichtbarkeit zur Verfügung.

Zusammenspiel von imperativen und objektorientierten Aspekten in Java

Beispiel: Kontoverwaltung

- Klasse `Konto` spezifiziert die Konten einer Bank.
- Für jedes neueröffnete Konto wird eine neue, fortlaufende Kontonummer vergeben.
- Wie kann man dies realisieren?
- Z.B. mit einer statischen Variablen `aktuelleKNR`, die bei jeder neuen Kontoeröffnung gelesen wird (um die neue Kontonummer zu bestimmen) und anschließend inkrementiert wird. Dieses Attribut sollte **`private`** sein, damit nur die Objekte der Klasse darauf zugreifen können.
- Das heißt die Klasse `Konto` wird neben den nicht-statischen Attributen, die für jedes neue Objekt neu angelegt werden (z.B. für den Namen des Kontoinhabers) auch das statische Attribut `aktuelleKNR` haben, das unabhängig von den existierenden Objekten der Klasse `Konto` existiert und verwendet werden kann.

Zusammenspiel von imperativen und objektorientierten Aspekten in Java

```
public class Konto
{
  /* ** Statische (objekt-unabhaengige) Attribute ** */
  private static int aktuelleKNR = 1;

  /* ** Nicht-statische (objekt-abhaengige) Attribute ** */
  private String kundenName;
  private double kontoStand;
  private int kontoNR;
  ... // weitere Attribute

  /* ** Konstruktor ** */
  public Konto(String kundenName)
  {
    this.kundenName = kundenName;
    kontoStand = 0.0;
    kontoNR = aktuelleKNR;
    aktuelleKNR++;
  }

  /* ** Methoden ** */
  ...
}
```

Beispiel: Wrapper-Klassen für primitive Typen

- Zu jedem primitiven Datentyp in Java gibt es eine korrespondierende (sog. *Wrapper*-) Klasse, die den primitiven Typ in einer oo Hülle kapselt.
- Es gibt Situationen, bei denen man diese Wrapper-Klassen anstelle der primitiven Typen benötigt. Z.B. werden in Java einige Klassen zur Verfügung gestellt, die eine (dynamische) Menge von beliebigen Objekttypen speichern können. Um darin auch primitive Typen ablegen zu können, benötigt man die Wrapper-Klassen.
- Zu allen numerischen Typen und zu den Typen **`char`** und **`boolean`** existieren Wrapper-Klassen.

Beispiel: Wrapper-Klassen für primitive Typen

Wrapper-Klasse	Primitiver Typ
`Byte`	**`byte`**
`Short`	**`short`**
`Integer`	**`int`**
`Long`	**`long`**
`Double`	**`double`**
`Float`	**`float`**
`Boolean`	**`boolean`**
`Character`	**`char`**
`Void`	**`void`**

Beispiel: Wrapper-Klassen für primitive Typen

- Zur Objekterzeugung stellen die Wrapper-Klassen haupsächlich zwei Konstruktoren zur Verfügung:
  - Für jeden primitiven Typ `type` stellt die Wrapperklasse `Type` einen Konstruktor zur Verfügung, der einen primitiven Wert des Typs `type` als Argument fordert:
    z.B. `public Integer(int i)`
  - Zusätzlich gibt es bei den meisten Wrapper-Klassen die Möglichkeit, einen String zu übergeben:
    z.B. `public Integer(String s)` wandelt die Zeichenkette `s` in einen Integer um, z.B. `"123"` in den **`int`**-Wert `123`.
- Kapselung:
  - Der Zugriff auf den Wert des Objekts erfolgt ausschließlich lesend über entsprechende Methoden:
    z.B. `public int intValue()`
  - Die interne Realisierung (wie wird der Wert gespeichert) ist dem Benutzer verborgen.
  - Insbesondere kann der Wert des Objekts nicht verändert werden.

Beispiel: Wrapper-Klassen für primitive Typen

- Statische Elemente (u.a.):
  - Wichtige Literale aus dem enstrechenden Wertebereich, z.B. Konstanten
    `public static int MAX_VALUE` bzw.
    `public static int MIN_VALUE`
    für den maximal/minimal darstellbaren **`int`**-Wert

    oder z.B. Konstanten
    `public static double NEGATIVE_INFINITY` bzw.
    `public static double POSITIVE_INFINITY`
    für $-\infty$ und $+\infty$
  - Hilfsmethoden wie z.B.
    `static double parseDouble(String s)` der Klasse `Double` wandelt die Zeichenkette `s` in ein primitiven **`double`**-Wert um und gibt den **`double`**-Wert aus.

Zusammenfassung

Sie kennen jetzt:

- Die Grundidee der oo Modellierung mit Objekten, die, bzgl. ähnlichem Verhalten und ähnlichem Zustand, zu Klassen zusammengefaßt werden.
- Die Umsetzung der oo Modellierung mit Objekten und Klassen in Java.
- Die Beschreibung des Verhaltens von Objekten mit Methoden in Java.
- Das Zusammenspiel von statischen (imperativen) und objektorientierten Konzepten in Java.