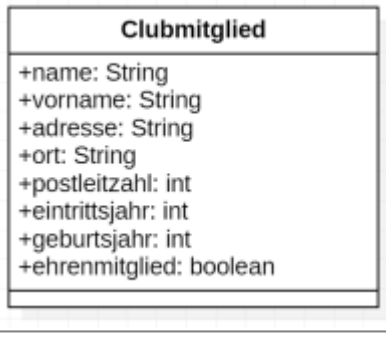
LU07b - Vom eigenen Datentyp zur Klasse

Repetition eigene Datentypen

Wir kennen bereits die eigenen Datentypen. Unsere eigenen Datentypen haben einen Namen und haben eigene Attribute/Eigenschaften.

Der eigene Datentyp **Clubmitglied** weist z.B. die folgenden Attribute auf

Attribute	**UML**
Name, Vorname, Adresse, Ort: `String`	
Postleitzahl, Eintrittsjahr, Geburtsjahr: `int`	
Ehrenmitglied: `boolean`	

In Java sieht dieser Datentyp wie folgt aus:

```
public class Clubmitglied {
  String name;
  String vorname;
  String strasse;
  int hausnummer;
  String ort;
  int postleitzahl;
  int eintrittsjahr;
  int geburtsjahr;
  boolean ehrenmitglied;
}
```

Eigene Datentypen zu Klassen umbauen

Defaultkonstruktor

In einer Klasse ist der Java Konstruktor eine Methode, die den gleichen Namen wie die Klasse hat. Sie wird dazu verwendet, ein neues Objekt dieser einen Klasse zu erstellen. Das ist der Grund, weshalb jede Klasseninstanz einen Konstruktor haben muss.

Programmierst du den Defaultkonstruktor nicht selbst, so
erstellt der Compiler des Programms Java automatisch einen

sogenannten Defaultkonstruktor. Dabei ist die Methode ohne Parameter und hat keinen Inhalt. Ebenfalls kannst du den speziellen Konstruktor nicht sehen, da er nur im Hintergrund abläuft.

Der Defaultkonstruktor für die Klasse Clubmitglied würde folgendermassen aussehen:

```
public class Clubmitglied {
  String name;
  String vorname;
  String strasse;
  int hausnummer;
  String ort;
  int postleitzahl;
  int eintrittsjahr;
  int geburtsjahr;
  boolean ehrenmitglied;

  public Clubmitglied(){}
}
```

Vielleicht ist Ihnen aufgefallen, dass Sie diesen Defaultkonstruktor bereits kennen und auch schon aufgerufen haben.

```
public void run(){
  Clubmitglied mitglied1;
  mitglied1 = new Clubmitglied ();
}
```

Jedes mal, wenn Sie einen eigenen Datentyp (hier Clubmitglied) erstellt haben, haben Sie bereits den Defaultkonstruktor aufgerufen.

Konstruktoren

Nehmen wir an, Sie möchten ein Clubmitglied erstellen. So wäre es doch praktisch, wenn sie bereits dem Konstruktor den Wert für `Vorname` und `Nachname` übergeben könnten. Dazu müssen Sie einen Konstruktor ergänzen, sodass er Parameter entgegen nehmen kann.

Der Aufruf zur Instanzierung und zur Speicherung des neuen Objekts in der Variablen clubmitglied1 wäre dabei folgender:

```
public void run(){
  Clubmitglied mitglied1;
  mitglied1 = new Clubmitglied ("Peter", "Meier");
```

```
}
```

Damit werden die String „Peter“ und „Meier“ mitgegeben. Nun sucht das Programm nach dem passenden Konstruktor in der Klasse Clubmitglied. Hier wird nach einer Methode gesucht, die Clubmitglied heißt und zwei Parameter des Types String verarbeiten kann.

Der Name und die Parameter, welche die Methode bekommen kann, werden auch Signatur einer Methode genannt. Die Signatur muss immer eindeutig sein, das bedeutet, sie darf es nur einmal in einer Klasse geben.

In unserem Beispiel würde der Konstruktor sich wie folgt verändern:

```
public class Clubmitglied {
  String name;
  String vorname;
  String strasse;
  int hausnummer;
  String ort;
  int postleitzahl;
  int eintrittsjahr;
  int geburtsjahr;
  boolean ehrenmitglied;

  public Clubmitglied(){
  }

  public Clubmitglied(String vorname, String nachname){
    this.vorname = vorname;
    this.nachname = nachname;
  }
}
```

Das ist der zur Instanziierung passende Java Konstruktor. Damit kannst du zwei String Werte bei der Instanziierung übergeben.

Mit dem Befehl:

```
this.vorname = vorname;
```

sorgst du lediglich dafür, dass in das Attribut `vorname` der Klasse der Wert aus dem Parameter `vorname` gespeichert wird.

Überladene Konstruktoren

Wie bei normalen Methoden , kann auch die Konstruktormethode überladen werden. Schauen wir uns wieder ein Beispiel an. Angenommen Sie möchten ein Objekt `clubmitglied1` und ein Objekt `clubmitglied2` erstellen. Weiterhin gehen wir davon aus, dass Sie den Namen und Vornamen, des `clubmitglied1` schon kennen. Vom `clubmitglied2` hingegen wissen Sie keinen Nachnamen,

weshalb das entsprechende Attribut auf „unbekannt“ gesetzt werden soll.

In Java kannst du das folgendermaßen instanzieren:

```
public void run(){
  Clubmitglied mitglied1, mitglied2;
  mitglied1 = new Clubmitglied ("Peter", "Meier");
  mitglied2 = new Clubmitglied ("Manuel");
}
```

Der Code für den Java Konstruktor könnten Sie so umsetzen:

```
public class Clubmitglied {
  String name;
  String vorname;
  String strasse;
  int hausnummer;
  String ort;
  int postleitzahl;
  int eintrittsjahr;
  int geburtsjahr;
  boolean ehrenmitglied;

  public Clubmitglied(){
  }

  public Clubmitglied(String vorname, String nachname){
    this.vorname = vorname;
    this.nachname = nachname;
  }

  public Clubmitglied(String vorname){
    this.vorname = vorname;
    this.nachname = "unbekannt";
  }
}
```

Damit hätten Sie Ihren Konstruktor überladen. Sie benutzen zweimal den gleichen Methodenaufruf, in dem Fall `Clubmitglied()`, jedoch durch die unterschiedlichen Parameter zwei unterschiedliche Signaturen. Deshalb wird für das `clubmitglied1` der erste und für das `clubmitglied2` der zweite Konstruktor angesprochen.

Mit der Referenz this. zeigen Sie dem Programm an, dass sie sich auf ein Attribut beziehen. Also das Attribut, das Sie am Anfang der Klasse im Bereich der Eigenschaften deklariert haben.

Sichtbarkeit der Attribute

Klassen, Methoden sowie Attribute können mit einem Sichtbarkeits-Modifier ergänzt werden. Im Modul 319 interessieren uns nur die Sichtbarkeiten `private` und `public`

- `private` Attribute oder Methoden sind nur von innerhalb der Klasse zugänglich.
- `public` Attribute oder Methoden sind auch von ausserhalb zugänglich.

Wir gewöhnen uns an

- Attribute immer mit `private` zu deklarieren.
- Methoden die nur innerhalb der Klasse verwendet werden auch als `private` zu deklarieren.
- Methoden die Funktionen für die Klasse zur Verfügung stellen als `public` zu deklarieren.

Für unsere Klasse `Clubmitglied` würde die Umsetzung so aussehen:

```
public class Clubmitglied {
  private String name;
  private String vorname;
  private String strasse;
  private int hausnummer;
  private String ort;
  private int postleitzahl;
  private int eintrittsjahr;
  private int geburtsjahr;
  private boolean ehrenmitglied;

  public Clubmitglied(){
  }

  public Clubmitglied(String vorname, String nachname){
    this.vorname = vorname;
    this.nachname = nachname;
  }

  public Clubmitglied(String vorname){
    this.vorname = vorname;
    this.nachname = "unbekannt";
  }
}
```

Da wir die Attribute jetzt auf `private` gesetzt haben, können wir nicht mehr direkt auf die Attribute zugreifen. Dieses Problem kann durch `Getter` und `Setter` gelöst werden.

Getter und Setter

In Klassen werden die Attribute mit dem Sichtbarkeits-Modifier `private` versehen, da hier nichts von außerhalb der Klasse geändert werden soll. Die Attribute sollen nicht so einfach zugänglich sein. Die Kontrolle ob und wie die Werte der Attribute geändert werden können soll in der Hand der Klasse bleiben. Dafür gibt es die sogenannten `getter` und `setter` Java Methoden.

Um dir die Funktionsweisen dieser speziellen Methoden zu zeigen, bauen wir uns die Klasse `Mensch`. Sie soll als Attribut einen `Namen` haben. Außerdem soll sie einen Konstruktor, der den Namen der Klassenvariablen zuweist, besitzen. Danach kommt eine Setter und eine Getter Methode.

```
public class Mensch{

    //Attribut
    private String name;
    private boolean isMann;

    //Konstruktor
    public Mensch(String name){
        this.name = name;
    }

    /**
     * Setter: Setzt den Namen
     * @param name neuer Name
     */
    public void setName(String name){
        this.name = name;
    }

    /**
     * Getter: Gibt den Namen zurück
     * @return name Name des Menschen
     */
    public String getName(){
        return this.name;
    }

    /**
     * Setter für das Geschlecht
     * @param isMann für das Geschlecht
     */
    public void setIsMann(boolean isMann) {
        this.isMann = isMann;
    }

    /**
     * Getter für das Geschlecht
```

```
     * @return das Geschlecht
     */
    public boolean isMann() {
        return isMann;
    }
}
```

Getter und Setter Methoden sind beide `public`, weil du ja von einer Klasse außerhalb auf diese zugreifen möchtest. Es ist dabei Konvention ein `get` oder `set` vor den Namen der Methode zu schreiben. Der Name der Methode ist dabei gleich dem Variablennamen der Variable, die in der Methode behandelt werden soll. In unserem Fall entsteht so der Name `setName()` und `getName()`. Bei Boolean werden die Vorsilben `set` und `is` verwendet, in unserem Fall `setIsMann()` und `isMann()`.

Die Set-Methode gibt an den keinen Wert zurück, sie ist also `void`. Sie weißt lediglich dem Attribut `name` einen neuen Wert zu. Daher ist es wichtig, dass der Methode einen Parameter des Types `String` mit dem neuen Namen übergeben wird.

Die Get-Methode hingegen gibt den aktuellen Wert des Attributes zurück. Das ist eine Möglichkeit die Belegung der Variablen zu erfahren. Der return-Wert entspricht hier also dem Datentyp der Variable.

Gültigkeitsbereich (Scope) von Variablen

Wir unterscheiden zwischen drei verschiedenen Variablen-Typen, die sich danach unterscheiden, wo sie im Code der jeweiligen Klasse deklariert werden. Der Ort, wo wir eine Variable erzeugen, hat nämlich entscheidenden Einfluss auf ihren Gültigkeitsbereich.

- **Attribute:** Werden auch Instanzvariablen genannt. Diese werden direkt zu Beginn einer Klasse deklariert. Wir können diese Variablen in der gesamten Klasse verwenden.
- **Parameter-Variablen:** Diese Variablen werden als Parameter von Methoden innerhalb der runden Klammern im Methoden-Kopf deklariert. Sie können nur in der entsprechenden Methode verwendet werden.
- **Lokale Variablen:** Werden innerhalb eines Methoden-Körpers deklariert. Sie können von der Zeile ihrer Deklaration an bis zum Ende der Methode verwendet werden.

Es ist am besten, wenn wir uns das Prinzip einmal konkret anhand einer kleinen Java-Klasse anschauen:

```
public class Fahrzeug {

    // Instanzvariablen:
    private String bezeichnung;
    private double geschwindigkeit;

    // Konstruktor mit 2 Parameter-Variablen:
    public Fahrzeug(String bezeichnung, double
geschwindigkeit){
        this.bezeichnung = bezeichnung;
        this.geschwindigkeit = geschwindigkeit;
```

```
    }

    // Methode mit einer Parameter-Variablen und
    // einer lokalen Variablen:
    public double getFahrtdauer(int fahrtstrecke){
        double zeit = (fahrtstrecke / geschwindigkeit) * 60;
        return zeit;
    }
}
```

Folgende Variablen kommen in dieser Beispielklasse vor:

- **Attribute:** Die Klasse Fahrzeug hat insgesamt zwei Instanzvariablen, nämlich `bezeichnung` und `geschwindigkeit`. Wir können auf diese drei Variablen von überall aus in unserer Klasse zugreifen und das tun wir auch in unserem Code. Im Konstruktor nämlich initialisieren wir die beiden Instanzvariablen. Auch in der Methode `getFahrtdauer()` greifen wir auf die Instanzvariable `geschwindigkeit` zu. Wir können es also nochmal sagen: Instanzvariablen gehen immer!
- **Parametervariablen:** In unserer Beispielklasse gibt es drei Parametervariablen. Zwei werden im Kopf des Konstruktors (`bezeichnungP`, `geschwindigkeitP`) und die dritte (fahrtstrecke) im Kopf der Methode `getFahrtDauer()` deklariert. Jede Parameter-Variable ist ausschließlich in der Methode verfügbar, zu der sie gehört.
- **Lokale Variablen:** Kommen wir nun zum dritten Variablen-Typ, den lokalen Variablen. Von diesen haben wir genau eine in unserem Code - erkennen Sie, wo sie sich befindet? Richtig: `zeit` heißt sie und ist innerhalb der Methode `getFahrtdauer()` deklariert und mit dem Ergebniswert der Rechnung initialisiert. Man könnte Parametervariablen und lokale Variablen auch „Methoden-Variablen“ nennen, da sie ja fest zu den Methoden gehören, in denen sie erzeugt wurde und auch nur dort benutzbar bzw. sichtbar sind

Es gibt in der Verwendung von Parametervariablen und lokalen Variablen aber einen feinen Unterschied: Auf Parametervariablen können in der gesamten Methode zugegriffen können. Lokale Variablen dagegen innerhalb „ihrer“ Methode erst in den Code-Zeilen nach der Initialisierung (also nachdem sie erstellt wurde).

Was macht man, wenn **Attribut und Parameter oder lokale Variable denselben Namen** haben? Sie sehen die Lösung im Konstruktor des Fahrzeug. Wir verwenden das Schlüsselwort `this` um auf das Attribut zu zeigen.

Merken Sie sich:** Wir verwenden das Schlüsselwort `this` um innerhalb von Mehtoden oder Konstruktoren auf ein Attribut zu zeigen.**

© Kevin Maurizi

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/modul/archiv/m319/learningunits/lu07/lu07b-eigenerdatentypzuklasse

Last update: **2025/06/23 07:45**