LU07a - Methoden

Sinn von Methoden

Wieso soll man überhaupt ein Programm in Methoden aufteilen? Zunächst bedeutet das schreiben der Methoden scheinbar einen Mehraufwand. Der Grund wird offensichtlich, wenn man ein wenig weiter schaut.

Wiederhole Dich nicht (DRY)

Das DRY-Prinzip lautet: Don't repeat yourself! Jedesmal wenn Sie den gleichen Code mehrfach schreiben, erhöht sich das Risiko von Fehlern. Angenommen Sie haben in Ihrem Programm den gleichen Codeteil 5x geschrieben. Nun stellen Sie fest, dass ein Fehler in diesem Codeteil ist oder Sie wollen etwas ergänzen. So ist die Gefahr gross, dass Sie die Korrektur 4x vornehmen und die letzte Stelle vergessen.

Daher wird alles, was Sie mehrfach benötigen, in eine Methode gepackt.

Teilaufgaben erhöhen die Übersicht

Es ist einfacher ein Programm zu verstehen, wenn es in viele, klar definierte Teilaufgaben unterteilt ist. Ein Sourcecode mit mehreren hundert Zeilen an einem Stück, ist extrem schwer zu lesen.

Als Faustregel gilt: **Eine Methode sollte höchstens 12-15 Anweisungen enthalten**. Hinzu kommen der Methodenkopf, Deklarationen und allfällige Kommentarzeilen. Damit bleibt die Methode als Ganzes kompakt und übersichtlich.

Wartung

Nachdem ein Programm geschrieben wurde, ist es oftmals jahrelang im Einsatz. Während dieser Zeit müssen Fehler korrigiert und weitere Funktionen eingebaut werden. Durch die Abgrenzung von Teilaufgaben wird das Programm übersichtlicher. Es ist auch Jahre später viel einfacher, Änderungen vorzunehmen.

Recycling

Methoden können sehr einfach wiederverwendet werden. Sie müssen also nicht jedes mal das ganze Programm neu erfinden. Vielmehr können sie Teile aus bestehenden Programmen übernehmen.

Beispielsweise stehen in Java Dutzende von Bibliotheken mit vorgefertigten Klassen zur Verfügung. Diese Klassen hat ein anderer Softwareentwickler erstellt und wir können Sie nun nutzen. Damit ersparen Sie sich viele Stunden Arbeit.

Vorgehen

Gedankenspiel: Auftrag einem Mitarbeiter erteilen

Sie wollen einem Kollegen den Auftrag erteilen, ihnen einen Kaffee aus dem Automaten zu holen. Dieser Kollege weiss, wo der Automat steht und wie man den Kaffeeautomaten bedient.

- Was müssen sie dem Kollegen alles mitteilen bzw. mitgeben?
- Was wollen sie erhalten, wenn der Auftrag erledigt ist?

Notieren sie sich ihre Überlegungen zu diesen Punkten.

Lösungsidee

Was muss ich dem Kollegen mitteilen bzw. mitgeben?

- Ob ich einen Kaffee oder einen Espresso möchte.
- Ob ich Rahm und/oder Zucker möchte.
- Das Geld für den Automaten.
- Meine Kaffeetasse.

Was will ich erhalten?

- Meine Kaffeetasse mit dem gewünschten Kaffee drin.

Fassen wir das in Pseudocode zusammen könnte der Auftrag in etwa so aussehen:

```
double geld = 4.50
String art = "Espresso"
bool rahm = false
bool zucker = true
Tasse tasse = new Tasse()

kaffee =
arbeitskollege.bringKaffee(geld,
art, rahm, zucker, tasse);
```

Aufgabe der Methode definieren

Als ersten Schritt überlegen Sie sich, welche Teilaufgabe diese Methode erledigen soll. Diese Teilaufgabe muss sich in einem kurzen Satz prägnant umschreiben lassen. Unklare oder schwammig definierte Teilaufgaben führen in der Regel zu schlechten Methoden.

- Vermeiden Sie Füllwörter wie Ding, etwas, machen, ...
- Eine Methode hat genau eine Aufgabe ⇒ Beschreibungen mit **und** bzw. **oder** vermeiden.

Aus der Definition der Aufgabe lässt sich auch ein sinnvoller Name für die Methode ableiten.

Beispiele

- Beschreibung: Die Methode gibt einen String an der aktuellen Cursorposition aus.
    - Name: putString
- Beschreibung: Die Methode prüft, ob die Eingabe des Benutzers eine Zahl ist.
    - Name: isNumber

Schnittstelle definieren

Eine Methode ist ein eigenständiger Programmblock mit eigenem Speicherbereich und eigenen Variablen. Sämtliche Daten, welche die Methode zur Verarbeitung benötigt, werden als Argumente übergeben. Auch die Resultate der Verarbeitung muss die Methode über ihre Schnittstelle an den Aufrufer zurück geben.

Zur Definition der Schnittstelle der Methode müssen Sie folgende Aufgaben erledigen :

Sichtbarkeit der Methode festlegen

Sichtbarkeit: Häufig wird einer der Modifier `private`, `public`, `default` oder `protected` benutzt. Um programme im Modul 319 umzusetzen werden wir lediglich die Modifier `private` und `public` verwenden. Zusammengefasst bestimmt `private`, dass du nur von innerhalb der Klasse auf die entsprechende Methode oder Variable zugreifen kannst. `public` ist das Gegenteil dazu, hier kannst du auch von außerhalb der Klasse auf sie zugreifen.

Static: Du hast das `static` in der Main Methode bereits kennengelernt. Wir können statische Main Methoden oder Variablen als unabhängig von der jeweiligen Klasse, in der sie geschrieben sind, bezeichnen. Das bedeutet wiederum, dass nicht ein extra Objekt der Klasse erzeugt werden muss, bevor diese Variable oder Methode benutzt werden kann. Lässt du das static weg, so muss zuerst eine Instanz der Klasse für die weitere Verarbeitung der jeweiligen Java Methoden oder Variablen erzeugt werden.

Bezeichner der Methode definieren

Frei wählbarer Name der Methode. Sollte sich aus der Funktion der Methode herleiten lassen

(Sprechend sein). Methodennamen sind klein geschrieben. Handelt es sich um mehr als ein Wort, so ist die CamelCaseSchreibweise angebracht.

Returnwert der Methode festlegen

Eine Methode kann **genau einen** Returnwert liefern. Der Returnwert kann das Resultat einer Verarbeitung sein.

Hat die Methode keinen Return-Wert so muss die Methode mit `void` gekennzeichnet werden. Gibt die Methode jedoch einen Wert zurück, ist an dieser Stelle der Datentyp des Return-Wertes zu deklarieren.

Beispiel:

Die Methode `liesEingabe` liest alle Tasten, die der Benutzer drückt. Sobald der Benutzer die `Return`- oder `Enter`-Taste drückt, gibt die Methode eine Zeichenkette mit allen eingegebenen Zeichen zurück. Als Returnwert wählen wir hier ein `String`-Objekt.

```
String liesEingabe(......) {
    String eingabe;
    eingabe = new String("");

    ....

    return eingabe;
}
```

Eine andere Möglichkeit ist, dass der Returnwert Auskunft darüber gibt, ob die Verarbeitung erfolgreich war. Diese Art des Returnwerts macht vor allem Sinn, wenn eine Methode etwas prüft.

Beispiel: istZahl

Die Methode `istZahl` prüft, ob die Eingabe des Benutzers eine Zahl ist oder nicht. Dabei gibt es nur zwei mögliche Resultate: Ja oder Nein. Somit macht es Sinn, dass die Methode einen bool'schen Wert als Returnwert liefert.

```
bool istZahl(.......) {
    if (.....) {   // Es ist eine Zahl
        return true;
    } else {
        return false;
    }
}
```

Parameter festlegen

Zur Schnittstelle gehören auch die Parameter, welche der Aufrufer an die Methode weiter gibt. Die Parameter werden zusammen mit dem Bezeichner(Name) der Methode angegeben. Jedes Parameter benötigt:

- Einen Datentyp, z.B. int, String, ...
- Einen Bezeichner

```
boolean checkLogin(String benutzername, String passwort) {
    ...
}
```

Zu jedem Argument in der Schnittstelle wird innerhalb der Methode automatisch eine lokale Variable deklariert. Im Beispiel `checkLogin` stehen also innerhalb der Methode die Referenzvariablen `benutzername` und `passwort` zur Verfügung.

Beim Aufruf der Methode gibt der Aufrufer die entsprechenden Werte für diese Variablen mit. Diese Werte werden automatisch in die entsprechenden lokalen Variablen gespeichert.

Methode realisieren

Nun gehen Sie daran, die Methode zu realisieren. Dabei empfiehlt es sich, gezielt vorzugehen:

1. Vereinbarung Schnittstelle

Schreiben Sie zuerst die Vereinbarung der Schnittstelle. Das ist die Codezeile mit Sichtbarkeit, Returnwert, Bezeichner und Parameter:

```
private bool istGross(int zahl) {
  ...
}
```

2. Methodenkopf schreiben

Nun kommt der Kommentarblock mit den Angaben zur Methode. In vielen Entwicklungswerkzeugen wird ein Teil dieses Kommentarblocks automatisch erzeugt.

Schreiben Sie

```
 /**
```

in die Zeile oberhalb der Schnittstellen-Vereinbarung und drücken Sie `Enter`

```
/**
  *
```

```
  * @param  int zahl
  * @return
  */
private bool istGross(int zahl) {
  ...
}
```

Die Schlüssel für die Parameter und den Returnwert werden automatisch eingefügt. Nun müssen Sie nur noch die Beschreibungen ergänzen.

```
/**
  * Prüft ob die Zahl eine grosse Zahl ist
  * @param  int zahl   Zu prüfende Zahl
  * @return            true=Es ist eine grosse Zahl
  */
private bool istGross(int zahl) {
  ...
}
```

3. Logik programmieren

Zuletzt programmieren Sie die Logik der Methode. Betrachten Sie jede Methode wie ein eigenständiges kleines Programm, mit:

- Variablen
- Anweisungen
- Returnwert

Eine Methode muss innerhalb einer Klasse definiert werden. Die Methode gehört dann zu dieser Klasse.

```
public class Auto(){
  public static void main(String[] args){}
  private void run(){}
  private void print(){}
}
```

Methoden aufrufen

Rufen wir Methoden auf, so müssen wir unterscheiden ob die Methode in der eigenen oder in einer fremden Klasse definiert ist. In der eignen Klasse muss die Methode lediglich mit dem Namen angesprochen werden.

Beispiel:

```
public class Methode(){

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

  /**
   * Hauptprogramm das ausgeführt wird.
   */
  private void run(){
    addieren(5,7);
  }

  /**
    * Addiert zwei Werte
    * @param zahl1 Summand 1
    * @param zahl2 Summand 2
    */
  private void addieren(int zahl1, int zahl2){
    System.out.println((zahl1+zahl2));
  }
}
```

Gehört die Methode zu einer anderen Klasse, so muss die Methode über das Objekt angesprochen werden.

Beispiel:

```
public class Methode(){

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

  /**
   * Hauptprogramm das ausgeführt wird.
   */
  private void run(){
    Auto auto = new Auto();
    auto.fahren();
  }
}


public class Auto(){
  int leistung;
  String marke;
  String modell;

  /**
    * Lässt das Auto Fahrgeräusche machen.
    */
  public void fahren(){
```

```
    System.out.println("BrummBrumm");
  }

}
```

Methoden überladen

In Java können in einer Klasse mehrere Methoden mit dem gleichen Namen aber unterschiedlichen Parametern definiert werden. Das heißt, trotz gleichem Methodennamen ist die Signatur unterschiedlich. Dieses Vorgehen bezeichnet man als Überladen einer Methode.

Beispiel So könnte man sich beispielsweise eine Methode „summe“ vorstellen, welche die Summe von übergebenen Werten bildet und auf der Konsole ausgibt. Es könnten zwei Methoden „summe“ definiert werden, eine nimmt zwei und die andere drei Parameter zum Summieren entgegen.

```
public class Methode(){

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

  /**
   * Hauptprogramm das ausgeführt wird.
   */
  private void run(){
    addieren(5,7);
    addieren(5,7,4);

  }

  /**
    * Addiert zwei Werte
    * @param zahl1 Summand 1
    * @param zahl2 Summand 2
    */
  private void addieren(int zahl1, int zahl2){
    System.out.println((zahl1+zahl2));
  }

    /**
    * Addiert drei Werte
    * @param zahl1 Summand 1
    * @param zahl2 Summand 2
    * @param zahl3 Summand 3

    */
  private void addieren(int zahl1, int zahl2, int zahl3){
    System.out.println((zahl1+zahl2+zahl3));
  }
```

```
}
```

Java entscheidet also anhand der übergebenen Parameter (genauer anhand des Datentyps und der Anzahl der Parameter), welche der vorhandenen Methoden zur Laufzeit ausgeführt werden soll.

Der Rückgabetyp wird zur Unterscheidung nicht verwendet, es kommt lediglich auf die Anzahl und Art der Parameter an.
