4. Objektorientierte Programmierung

- In Abschnitt 3 ging es um fundamentale Basiskonzepte von Java, wie es sie in jeder anderen gängigen Programmiersprache so oder so ähnlich auch gibt.

- In Abschnitt 4 nun geht es um ”fortgeschrittene“ Konzepte, die es nur in sogenannten *objektorientierten* Programmiersprachen gibt.

  → Die ”wahre“ Welt der Java–Programmierung.

4.1. Mehr zu Methoden

- Aus anderen Programmiersprachen sind *Unterprogramme* (engl. *subroutines*), *Funktionen* und *Prozeduren* bekannt.
- Sie bezeichnen alle im Grunde dasselbe abstrakte Konzept:
  - ◊ Einzelne Teile des gesamten Quelltextes werden zu einer Einheit zusammengefasst.
  - ◊ Eine solche Einheit kann von anderen Stellen des Quelltextes aus *aufgerufen* werden.
  - ◊ Die ganze Kommunikation zwischen aufrufender Stelle und dieser Einheit läuft über eine kleine, fest umrissene Schnittstelle: Parameter und (ggf.) Rückgabewert.
- Das Äquivalent in Java sind die *Methoden*.

Methoden und Klassen

- Im Gegensatz zu anderen Programmiersprachen gibt es in Java keine isolierten Unterprogramme.
- Statt dessen gehört jede Methode zu einer festen Klasse.
- Der vollständige Name einer Methode ergibt sich durch
  - Voranstellen des Pfades der zugehörigen Klasse
  - separiert wie üblich durch einen Punkt.
- ***Beispiele***:
  - `java.applet.Applet.paint`
  - `java.lang.StringBuffer.append`
  - `java.lang.Thread.sleep`

Klassenpfade

Die Menge aller Klassen ist grundsätzlich hierarchisch organisiert.

Beispiel:

- Es gibt eigentlich überhaupt keine Klasse `Window`,
- sondern eine Klasse `java.awt.Window`.
- Namensbestandteil
  - `java`: Die Klasse gehört zum Standardumfang von Java dazu.
  - `awt`: ”Abstract Windowing Toolkit“, d.h. Die Klasse gehört (keine Überraschung) zum Werkzeugkasten für Fensterbasteleien.

Weiteres Beispiel: `java.lang.StringBuffer`

Namensbestandteil `lang` (”language“): Die Klasse gehört zu den Kernbausteinen der Programmiersprache Java.

Importieren von Klassendefinitionen

- In Java Source Files sieht man häufig Klassennamen ohne Klassenpfad.
- Wie lässt sich das mit der Aussage der letzten Folie vereinbaren?

Antwort:

- Nach einer `import`–Zeile am Anfang des Java Source Files braucht man den dort angegebenen Klassenpfad nicht mehr mit anzugeben.
- *Beispiel*:

```
import java.awt.*
Window win;  ← gleich "java.awt.Window win;"
```

- Speziell für `java.*` und `java.lang.*` betrachtet der Java–Compiler die `import`–Zeilen automatisch als gegeben.

4.1.1 Signatur und Überladung

Die *Signatur* einer Methode setzt sich zusammen aus

- dem vollständigen Namen mit Klassenpfad gemäß vorheriger Folie,
- der Anzahl der Parameter (potentiell auch gar keine Parameter),
- den Typen der Parameter in ihrer Reihenfolge in der Parameterliste,
- dem Rückgabetyp der Methode (bzw. `void`),
- der (potentiell leeren) Liste der in der `throws`–Klausel angegebenen Exception–Typen
- sowie den *Modifiern* der Methode wie `public` und `static`

Anmerkung:

`throws`–Klauseln sind bis jetzt noch gar nicht eingeführt und Modifier bisher noch nicht ernsthaft betrachtet worden.

→ Wird später nachgeholt.

Überladung

Zwei (oder mehr) Methoden dürfen in Java denselben vollständigen Namen haben, das heißt, sie dürfen

- zur selben Klasse gehören und zugleich
- mit demselben Identifier als Namen der Methode bezeichnet sein, wenn
  - ◊ sie sich entweder in der Anzahl der Parameter unterscheiden
  - ◊ oder (falls die Anzahl gleich ist) wenigstens die Liste der Typen der Parameter sich unterscheidet.

→ Eine derart duplizierte Methode heißt *überladen*.

Beispiel

```
public class MeineKlasse
 {
   public void f ()                    { ... }
   public void f ( int x )             { ... }
   public void f ( double x )          { ... }
   public void f ( int x, double y )   { ... }
   public void f ( double x, int y )   { ... }
 }
```

→ Alle diese Methoden dürfen in derselben Klasse `MeineKlasse` mit demselben Identifier `f` bezeichnet werden.

Nicht hinzugefügt werden zu obiger Klasse `MeineKlasse` dürfen zum Beispiel folgende Methoden:

```
public          int f   ( double x ) { ... }
private         void f  ( double x ) { ... }
public static   void f  ( double x ) { ... }
public          void f  ( double x ) { ... }
```

Beispiel 2

- Es gibt insgesamt zehn Methoden mit dem vollständigen Namen `java.lang.StringBuffer.append`.
- *Konkrete Beispiele*:
  - **Mit einem einzelnen `String`–Parameter**: Fügt die Zeichenkette im Parameter an die momentan in `StringBuffer` gehaltene Zeichenkette an.

    → Die auf den bisherigen Folien verwendete Variante von `java.lang.StringBuffer.append`.
  - **Mit einem einzelnen `char`–Parameter**:
    Fügt dieses Zeichen hinten an.
  - **Mit einem einzelnen `double`–Parameter**:
    Fügt eine Zeichenkette hinten an, die den numerischen Wert dieses `double`–Parameters als Zeichenkette darstellt.
- Alle diese Beispiele haben die gleiche Anzahl Parameter: `1`.
  - Aber die Typen unterscheiden sich.

Warum nur in den Parametern?

Frage:

- Warum müssen sich zwei Methoden mit dem gleichen vollständigen Namen unbedingt in der Parameterliste unterscheiden?
- Warum reicht es nicht, wenn sie sich im Rückgabetyp, der `throws`–Liste oder den Modifiern unterscheiden?

Antwort:

- Dann kann der Compiler nicht mehr für jeden Aufruf zweifelsfrei entscheiden, welche Methode nun eigentlich gemeint ist.
- Einen Unterschied in der `throws`–Klausel oder der Modifier–Liste allein könnte man einem Aufruf einer Methode überhaupt nicht ansehen.
- Speziell den Rückgabetyp kann der Compiler nicht erkennen, wenn der Rückgabewert einer Methode beim Aufruf unter den Tisch fällt.

Beispiel

```
public class MeineKlasse {
   public int f  ( int n ) { ... }
   public char f ( int n ) { ... }
      // Verboten!
}
...

MeineKlasse meinObjekt = new MeineKlasse();
meinObjekt.f(1);
```

Erläuterung:

- *Problem*:
  Welche der beiden Varianten der Methode `MeineKlasse.f` ist denn nun mit `meinObjekt.f(1)` oben gemeint?
- *Lösung in Java*:
  Durch das Verbot, Methoden allein durch Variation des Rückgabetyps zu überladen, ergibt die Deklaration der zweiten Methode `f` eine Fehlermeldung vom Compiler.

Anmerkungen zur Klarstellung

- Selbstverständlich
  - ◊ dürfen sich überladene Methoden in Rückgabetyp, Modifiern und `throws`–Liste unterscheiden,
  - ◊ nur eben nicht darin *allein*,
  - ◊ sondern auf jeden Fall müssen sich auch die Parameterlisten voneinander unterscheiden.
- Selbstverständlich
  - ◊ dürfen Methoden aus *verschiedenen* Klassen identischen Namen und zugleich identische Parameterliste haben,
  - ◊ und in diesem Fall dürfen sie sich dann (müssen aber nicht) auch in Rückgabetyp, Modifiern und `throws`–Liste beliebig unterscheiden,

Signatur und Interpreter

- Ein Java–Interpreter wie `java` oder `appletviewer`
  - bekommt den Namen einer Java–Klasse als Argument beim Aufruf mit und
  - erwartet als Einstiegspunkt immer eine Methode dieser Klasse mit ganz bestimmter Signatur.
- *Konkret*:
  - `java` erwartet eine Methode namens `main` mit Rückgabetyp `void`, mit einem (einzigen) Parameter vom Typ `String`–Array und mit den Modifiern `public` und `static`.
  - `appletviewer` erwartet eine Methode namens `paint` mit Rückgabetyp `void`, einem Parameter vom Typ `Graphics` und dem Modifier `public`.
  - Beide erwarten dabei eine leere `throws`–Liste.

Signatur und Interpreter (2)

- Diese erwartete Methode wird vom jeweiligen Interpreter als Programmstart aufgerufen.
- Wenn ihre Abarbeitung beendet ist, ist das Java–Programm zu Ende.
- Wenn die jeweils erwartete Methode nicht *mit genau der erwarteten Signatur* in der Klasse vorhanden ist,
  - ◊ bricht der Interpreter sofort ab und
  - ◊ gibt eine Fehlermeldung aus, dass ”die–und–die Methode“ nicht gefunden wurde.
- Da man sich erfahrungsgemäß oft bei den Details der Signatur irrt, schauen viele Interpreter genauer hin und geben ggf. eine Fehlermeldung, die
  - ◊ nicht einfach besagt, dass die ”die–und–die Methode“ nicht gefunden wurde,
  - ◊ sondern die besagt, dass die fragliche Methode ”die–und–die Signatur“ hat.

Wiederholung: Applets

- Wir haben jetzt gesagt, dass ein Interpreter immer eine Einstiegsmethode hat.
- Genauer muss es heißen: *mindestens* eine.
- *Konkretes Beispiel*:
  WWW–Browser und das Programm `appletviewer` erwarten noch weitere Methoden neben "`paint`" (ebenfalls mit exakt vorgegebener Signatur) und rufen diese Methoden in bestimmten Situationen auf.
- Zum Beispiel:
  - `void`–Methode `init` mit leerer Parameterliste.
    → Wird beim Start des Applets aufgerufen (noch vor `paint`).
  - `void`–Methode `repaint`:
    → Wann immer der Inhalt des Fensters neu zu zeichnen ist.
- Diese Methoden müssen aber nicht implementiert werden
- sondern werden in der Klasse Applet implementiert und durch die Angabe von `extends Applet` ererbt.

4.1.2 Klassen- vs. Objektmethoden

Zunächst zur Syntax:

- Eine *Klassenmethode* erkennt man daran, dass der Modifier ”`static`“ vor dem Rückgabetyp steht.

- Die Methoden, die wir bisher in den Übungen selbst gebastelt haben, hatten meistens noch kein `static`

  - außer es war in der Angabe gefordert (z.B. für `main`)

  → Waren also alles Objektmethoden, keine Klassenmethoden.

- ”`static`“ ist nicht gerade ein sehr intuitives Schlüsselwort für Klassenmethoden.

  → Tatsächlich wieder eine Altlast aus C/C++.

Beispiele für Klassenmethoden

- `java.lang.Character.isLowerCase` und `java.lang.Character.toUpperCase`
- `java.lang.Thread.sleep`,
- `java.awt.Color.getHSBColor`: Bekommt Werte für Farbton, Sättigung und Helligkeit als drei Parameter und liefert ein Objekt vom Typ `Color`, also in RGB–Kodierung:

```
Color c = Color.getHSBColor (1,1,1);
```

→ Violett mit voller Helligkeit und Sättigung.

Klassenmethoden vs. Methoden einer Klasse

- „Klassenmethode“ und „Methode einer Klasse“ bedeuten nicht genau dasselbe.
- Wie gesagt, gehört ja **jede** Methode zu einer Klasse.
  - In diesem Sinne ist also nicht nur jede Klassenmethode eine „Methode einer Klasse“,
  - sondern auch jede Objektmethode ist genauso eine „Methode einer Klasse“.
- Wann immer von „Methoden einer Klasse“ die Rede ist, sind daher Objekt- wie Klassenmethoden gleichermaßen gemeint.

Beispiel zur Syntax

```
public class MeineKlasse
{
   public void objektMethode ()
   {
      System.out.println ( "Hello 1" );
   }

   public static void klassenMethode ()
   {
      System.out.println ( "Hello 2" );
   }
}
```

Erläuterung:

- `objektMethode` ist eine Methode wie bisher bekannt.
- Wie gesagt, ist der syntaktische Unterschied bei Klassenmethoden zunächst einmal nur das `static`.

Was sind Klassenmethoden?

- Klassenmethoden sind im Grunde nichts anderes als Unterprogramme (Funktionen, Prozeduren), wie es sie in anderen Programmiersprachen auch gibt.
- Der wesentliche Unterschied ist, dass in Java eben Unterprogramme nur in Form von Methoden von Klassen möglich sind.

Beispiel: Quadratwurzelberechnung (`sqrt` = square root)

- `y := sqrt(x);` in Pascal,
- `y = sqrt(x);` in C,
- `y = java.lang.Math.sqrt(x);` in Java.

Beispiel: Klasse `java.lang.Math`

http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Math.html

- Diese Klasse dient im wesentlichen als Zusammenfassung für diverse grundlegende mathematische Funktionen (alle realisiert als Klassenmethoden).
- *Beispiele*:
  - Sinus:

```
y = java.lang.Math.sin(x);
```

  - Kosinus:

```
y = java.lang.Math.cos(x);
```

  - Potenzbildung x^y:

```
z = java.lang.Math.pow(x,y);
```

  - Absolutbetrag $|x|$:

```
y = java.lang.Math.abs(x);
```

  - Maximum aus zwei Zahlenwerten:

```
z = java.lang.Math.max(x,y);
```

Anm: Es reicht auch `Math.sin` (ohne `java.lang`)

Klassenmethode mit Variablen

```
char   c1  = 'a';
Character  c = new Character('b');

char c2 = java.lang.Character.toUpperCase (c1); // (1)
char c3 = Character.toUpperCase (c1);           // (2)
char c4 = c.toUpperCase(c1);                    // (3)
```

- **Achtung:** Fehler im Skriptum (Folie 402)
  - `java.lang.Character.toUpperCase(char c)` ist eine Klassenmethode der Klasse `Character`
    - Aufruf: → oben
  - `java.lang.String.toUpperCase()` ist eine Objektmethode der Klasse `String`
    - Aufruf:

```
String s  = new String("mach mich gross");
String s2 = s.toUpperCase();
```

Klassenmethode mit Variablen

```
char   c1  = 'a';
Character  c = new Character('b');

char c2 = java.lang.Character.toUpperCase (c1); // (1)
char c3 = Character.toUpperCase (c1);           // (2)
char c4 = c.toUpperCase(c1);                    // (3)
```

→ In den Zeilen (1)–(3) passiert immer dasselbe

(1) Der Aufruf von `toUpperCase` in der Initialisierung von `c2` ist wie gehabt.

(2) Der Aufruf bei der Initialisierung von `c3` ist analog, `java.lang` muß man nicht angeben

(3) Die Variante in der Initialisierung von `c4` ist neu:

- Man kann eine Klassenmethode auch mit dem Namen einer Variablen anstelle des Klassennamens aufrufen.
- Aber egal ob Klassenname oder Variablenname: Es macht in der Auswirkung absolut keinen wie auch immer gearteten Unterschied. Die Semantik ist in beiden Fällen dieselbe.

Aufruf von Klassenmethoden mit Objekten

- Wozu ist dann ein Aufruf mit einer Variablen möglich:
  - ◊ Wenn man eine Methode mit einem Variablennamen aufruft, braucht man sich keine Gedanken darum zu machen, ob dies nun eine Klassen- oder Objektmethode ist.
  - ◊ Wenn eine Objektmethode (einfach durch Einfügen von `static`) nachträglich zu einer Klassenmethode gemacht wird, braucht kein Stück Java–Quelltext bei der Verwendung der Methode deswegen geändert zu werden.

- **Beachte jedoch:**
  Klassenmethoden dürfen nicht auf Datenkomponenten des Objekts hinter der Variablen, mit der sie aufgerufen wurden, zugreifen. Sie müssen daher auch nicht mit einem Objekt aufgerufen werden.

  Nur Objekte haben konkrete Werte für die Datenkomponenten!

Beispiel

```
public class MeineKlasse
  {
    public static void f ()
      {
        System.out.println ( "Hello" );
      }
  }

 ...

MeineKlasse.f ();   // Wie bisher

MeineKlasse meinObjekt = new MeineKlasse ();
meinObjekt.f ();    // Auch ok!
```

Objektmethoden

- Können im Gegensatz zu Klassenmethoden nur mit dem Namen einer Variablen der Klasse, nicht mit dem Namen der Klasse selbst aufgerufen werden:

```
StringBuffer str = new StringBuffer ( "Hello" );

// Ok:
str.append ( ", World" );

// Verboten:
StringBuffer.append ( ", World" );

// Auch verboten:
java.lang.StringBuffer.append ( ", World" );
```

- *Terminologie*: Wir sagen, die Methode `append` ist auf `str` *angewandt* worden.

  → Genauer gesagt, auf das Objekt, auf das `str` verweist

- Oft wird in solchen Fällen schlampig verkürzt von dem ”Objekt `str`“ gesprochen.

Wieso nicht mit Klassennamen?

```
StringBuffer str = new StringBuffer ( "Hello" );

str.append(", World");                  // Erlaubt
StringBuffer.append(", World");         // Verboten!
```

Erläuterung: to append = hinten anhängen

- In der Zeile mit Kommentar ”Erlaubt“ ist eine der `append`–Methoden von `StringBuffer` auf das Objekt `str` angewandt worden.
- Die Semantik der (insgesamt zehn) Methoden mit dem vollständigen Namen `java.lang.StringBuffer.append` besagt ja gerade, dass an ein konkretes Objekt etwas angehängt werden soll.
- Es macht daher überhaupt keinen Sinn, ”`append`“ wie in der Zeile mit Kommentar ”Verboten!“ ohne ein Objekt aufzurufen, an das der Parameter angehängt werden soll.

Objektmethoden

- Eine Objektmethode darf auf das Objekt, auf das es angewandt wurde, sowie auf dessen Komponenten lesend und verändernd zugreifen.
- Natürlich muss eine Objektmethode nicht auf das Objekt (und dessen Komponenten) zugreifen, mit dem es aufgerufen wurde.

  → In diesem Fall gäbe es kein Problem mit einem Aufruf einer Objektmethode ohne Objekt.
- *Entscheidung* beim Design von Java:

  Eine Objektmethode darf dennoch generell nicht ohne Objekt aufgerufen werden.
- Eine Methode, die nicht auf das Objekt und seine Komponenten zugreift, kann man ja einfach zu einer Klassenmethode machen.

  → Dazu reicht ja aus, ein `static` einzufügen.

Objekte als Methoden-Parameter

- Im Grunde ist `str` nichts anderes als ein zweiter Parameter der Methode `append`, der
  - ◊ nicht in der Parameterliste auftaucht,
  - ◊ sondern vor den Methodennamen geschrieben wird,
  - ◊ mit einem Punkt davon getrennt.

- Die syntaktische Konvention

```
str.append(", World");
```

  ist reiner ”syntaktischer Zucker“ zur Unterstreichung, dass
  - ◊ `append` eine Methode der Klasse von `str` ist und
  - ◊ der Parameter `str` herausragende Bedeutung für die Logik des Aufrufs von `append` hat.

Objekte als Methoden-Parameter

- Man hätte es zum Beispiel durchaus stattdessen so festlegen können, dass
  - der Aufruf

    ```
    append ( str, ", World" );
    ```

    lautet, wie man es aus anderen Programmiersprachen gewohnt ist,
  - mit der Regel, dass der erste Parameter derjenige ist, auf den die Methode angewandt wird.
- Man hat sich aber in Java (und in den anderen gängigen objektorientierten Programmiersprachen) für diese eher ”krasse“ syntaktische Hervorhebung von `str` als wichtigstem Parameter von `append` entschieden.

Gegenseitiger Aufruf von Methoden

```
public class MeineKlasse {
   int n;

   public static void meineKlassenMethode1 () {
      System.out.println ( "Hallo" );
   }
   public static void meineKlassenMethode2 () {
      meineKlassenMethode1();
   }
   public void meineObjektMethode1 () {
      n = 1; // Darf nur Objektmethode!
      meineKlassenMethode2();
   }
   public void meineObjektMethode2 () {
      meineObjektMethode1();
   }
}

...
MeineKlasse meinObjekt = new MeineKlasse();
meinObjekt.meineObjektMethode2();
```

Erläuterungen

- Durch den Aufruf `meinObjekt.meineObjektMethode2()` auf der vorherigen Folie werden implizit auch die anderen Methoden von `MeineKlasse` auf `meinObjekt` angewandt.

- Insbesondere wird `meinObjekt.n` durch den Aufruf von

  ```
  MeineKlasse.meineObjektMethode1
  ```

  innerhalb von

  ```
  MeineKlasse.meineObjektMethode2
  ```

  auf `1` gesetzt.

Erläuterungen

- Wenn eine Objektmethode in einer anderen Objektmethode ohne den vorangestellten (mit ”`.`“ abgetrennten) Namen einer Variablen wie `meinObjekt` aufgerufen wird, dann wird der Aufruf auf dasselbe Objekt wie der letztere Aufruf angewandt.

- Das ginge auch gar nicht anders:

```
MeineKlasse meinObjekt1 = new MeineKlasse();
MeineKlasse meinObjekt2 = meinObjekt1;
MeineKlasse meinObjekt3 = new MeineKlasse();
```

→ Zum Beispiel die Situation in `MeineKlasse.meineObjektMethode2`:

Soll es darin nun `meinObjekt1.meineObjektMethode1` heißen oder was sonst?

Aufruf von Objektmethoden durch Klassenmethoden

- Eine Objektmethode kann jederzeit eine Klassenmethode auf diese Art und Weise aufrufen.

Umgekehrt:

- Der Aufruf einer Objektmethode durch eine Klassenmethode ist strikt verboten.
  - ◊ Das wäre auch semantischer Unsinn:
  - ◊ Eine Klassenmethode kann ja auch ohne Anwendung auf ein Objekt aufgerufen werden.
  - ◊ Eine Objektmethode hingegen darf grundsätzlich nur durch Anwendung auf ein Objekt aufgerufen werden.
- *Mit anderen Worten*:
  - ◊ Wenn eine Klassenmethode ohne Objekt aufgerufen wird
  - ◊ und in dieser Klassenmethode eine Objektmethode aufgerufen werden dürfte,
  - ◊ dann würde dieser Objektmethode das notwendige Objekt fehlen.