Programmiersprache

Java

Teil 6

KW 22-23 / 2020

Detlef Wilkening
www.wilkening-online.de

Programmiersprache Java

12 Klassen-Details

12.1 Zugriffsbereiche

Java kennt 4 Zugriffs-Modifizierer. Für alle Elemente in einer Klasse haben sie folgende Bedeutung:

- **public**
  Auf diese Elemente darf jeder von überall her zugreifen.
- **protected**
  Auf diese Elemente darf von innerhalb des gleichen Packages und von den abgeleiteten Klassen her zugegriffen werden. Packages werden in Kapitel 13 eingeführt, Vererbung danach. D.h. ist dieser Modifizierer für uns zurzeit ohne Bedeutung.
- **---** (kein Zugriffs-Modifizierer entspricht einer Package Sichtbarkeit)
  Auf diese Elemente darf von innerhalb des gleichen Package her zugegriffen werden. Packages werden in Kapitel 13 eingeführt. D.h. ist dieser Modifizierer für uns zurzeit noch ohne Bedeutung.
- **private**
  Auf diese Elemente darf nur von innerhalb der Klasse und von inneren Klassen (siehe später) zugegriffen werden.

```
public class A {

  private void f();

  public void g() {
    f();                // Kein Problem, da innerhalb der Klasse
  }
```

```
}
```

```
public class B {

  public void f(A a) {
    a.f();                // Compiler-Fehler, da private
    a.g();                // Okay, da public
  }
}
```

Wichtige Bemerkung – es ist sehr sehr empfehlenswert alle Attribute private zu machen, und einen Zugriff von außen – wenn er denn überhaupt notwendig ist – über Funktionen zu regeln. Dies ist einer der wichtigsten Grundprinzipien jeglicher Software-Entwicklung und der Objektorientierung: „Information-Hiding“:

- Die interne Implementierung ist vollständig vor dem Benutzer versteckt.
- Die interne Implementierung lässt sich jederzeit ändern.
- Von außen können keine inkonsistenten Objekt-Zustände erzeugt werden.

12.2 Konstruktoren

12.2.1 Initialisierung

Wird ein Objekt erzeugt, so gibt es mehrere Möglichkeiten die Attribute zu initialisieren.

- Man macht nichts: in diesem Fall werden die entsprechenden Attribute entsprechend ihrem Typ initialisiert.
- Man kann den gewünschten Wert direkt bei der Attribut-Definition angeben.
- Eine Klasse kann Initialisierungs-Blöcke enthalten – dies wird in der Vorlesung nicht besprochen.
- Oder das Attribut wird im Konstruktor initialisiert.

```
public class A {

  private int i1;          // Default-Wert 0
  private int i2 = 42;     // Initial-Wert bei der Attribut-Definition
  private int i3;          // Sieht nach Default-Wert aus, Initialisierung dann im
Ko.

  private String s1;                              // Default-Wert null
  private String s2 = "Hallo";                    // Initial-Wert String "Hallo"
fuer s2
  private String s3 = new String("Hallo");        // Langfassung von s2
  private String s4;                              // Sieht nach null aus, aber dann
Ko.

  public A(int arg1, String arg2) {
    i3 = arg1;
    s4 = arg2;
  }

}
```

- Direkte Initialisierungen der Attribute ist vor allem dann sinnvoll, wenn eine Klasse mehrere Konstruktoren hat, und die Attribute in allen Konstruktoren die gleichen Werte bekommen.
- Das Initialisieren der Attribute in den Konstruktoren ist vor allem dann sinnvoll, wenn die

gesetzten Werte konstruktor-spezifisch sind, d.h. z.B. von den Konstruktor-Parametern abhängen.

- Das Initialisieren der Attribute in den Initialisierungs-Blöcken – was hier nicht besprochen wird – ist vor allem dann sinnvoll, wenn das Setzen in allen Konstruktoren identisch, aber zu komplex für eine direkte Initialisierung ist.

12.2.2 Konstruktor-Verkettung

Wenn eine Klasse mehrere Konstruktoren hat, so können diese oft aufeinander zurückgeführt werden, um das Programmieren zu vereinfachen.

```
public class Person {

  private String forename;
  private String surname;
  private int age;

  public Person(String fn, String sn) {
    forename = fn;
    surname = sn;
  }

  public Person(String fn, String sn, int a) {
    this(fn, sn);
    age = a;
  }

}
```

Der Aufruf eines anderen Konstruktors der gleichen Klasse geschieht über den „this“ Zeiger mit den entsprechenden Parametern und **muss** in der ersten Zeile des Konstruktors stehen.

12.3 Funktionen

12.3.1 Parameter und Rückgaben

Die integrierten Datentypen wie int, double, char,... werden per Wert übergeben („call-by-value“ – cbv) – sie stellen also eine Kopie da - Änderungen in der Funktion verändern den Wert in der aufrufenden Funktion nicht.

Bei allen anderen Parametern, die ja alle Referenz-Typen sind, werden die Argumente per Referenz übergeben („call-by-reference“ – cbr). Bei diesen referenzieren die Parameter das gleiche Objekt wie in der aufrufenden Funktion – d.h. es kann in der Funktion das Original-Objekt verändert werden.

```
public class A {

  private static void f(int i, StringBuilder s) {
    System.out.println("> f(" + i + ", " + s + ")");
    i = 12;
    s.append(" Welt");
    System.out.println("> i: " + i);
    System.out.println("> s: " + s);
  }
```

```
  public static void main(String[] args) {
    int i = 42;
    StringBuilder s = new StringBuilder("Hallo");
    f(i, s);
    System.out.println("# i: " + i);
    System.out.println("# s: " + s);
  }

}
```

```
Ausgabe
> f(42, Hallo)
> i: 12
> s: Hallo Welt
# i: 42
# s: Hallo Welt
```

Der Rückgabe-Typ void bei Funktionen gibt an, dass die Funktion nichts zurückgibt.

12.3.2 return

Ist eine Funktion nicht vom Typ void, so muss sie einen zu dem Typ passenden Wert zurückgeben. Dies geschieht mit der return Anweisung. Die Funktion wird instantan beendet.

```
public class Person
{
  private int age;

  public int getAge() {
    return alter;
  }

}
```

- In einer Funktion können beliebig viele return Anweisungen vorkommen.
- Der Rückgabewert kann beim Funktions-Aufruf ignoriert werden, d.h. er muss nicht verwendet oder ausgewertet werden.

12.3.3 this

Innerhalb jeder Element-Funktion ist automatisch das Schlüsselwort „**this**“ definiert, das eine Referenz auf das aktuelle Objekt ist, d.h. das für das die Element-Funktion aufgerufen wurde.

Eine Referenz auf das aktuelle Objekt – aus Sicht der Element-Funktion auf sich selber – wird in der Praxis häufig benötigt, z.B. um *„sich-selber“* an andere Element-Funktionen zu übergeben. Aber auch für profanere Aufgaben wie z.B. zur Aufruf-Verkettung (siehe Beispiel) wird „this“ eingesetzt.

```
public class A {

  public A myself() {
    return this;
  }

  public void f() {
  }
```

```
}
```

```
A a = new A();
a.myself().f();
```

Ein ganz typischer Anwendungsfall in Java für „this“ ist die Verwendung gleicher Namen für Attribute (d.h. Variablen im Objekt) und Parameter. In einem solchen Fall verdecken die Parameter die Attribute, da sie im Scope der Funktion liegen, während die Attribute zum Scope der Klasse gehören. Über das Schlüsselwort „this“ – d.h. über das aktuelle Objekt – können die Attribute trotz gleich-namiger lokaler Variablen angesprochen werden.

```
public class RingCounter {

  private int start;
  private int limit;
  private int delta;
  private int value;

  public RingCounter(int start, int limit, int delta) {
    this.start = start;
    this.limit = limit;
    this.delta = delta;
    this.value = start;
  }

}
```

Achtung – in Klassen-Funktionen, d.h. Funktionen mit dem Modifierer static, ist „this“ **nicht** definiert, da Klassen-Funktionen keinen Objekt-Bezug haben.

12.3.4 Klassen-Funktionen

Klassen-Funktionen – d.h. Funktionen mit dem Modifier „static“ – werden ohne Objekt-Bezug aufgerufen. Klassen-Funktionen von *fremden* Klassen müssen mit dem Klassen-Namen referenziert werden.

```
public class A {

  public static void f() {
    System.out.println("A.f");
  }

}
```

```
public class B {

  public static void f() {
    System.out.println("B.f");
  }

  public static void main(String[] args) {
    f();
    B.f();
    A.f();
  }

}
```

```
Ausgabe
B.f
B.f
A.f
```

Im Gegensatz zu einer Element-Funktion ist eine Klassen-Funktion nicht einem Objekt, sondern einer Klasse zugeordnet. Daher können Klassen-Funktionen natürlich nicht auf Attribute zugreifen, sondern nur auf Klassen-Variablen – siehe Kapitel 12.4.1.

12.4 Attribute

12.4.1 Klassen-Variablen

So wie es Element- und Klassen-Funktionen gibt, so gibt es auch Element- und Klassen-Variablen. Gegenüber Attributen haben Klassen-Variablen zusätzlich den Modifier „static“.

```
public class A {

  private static int i;
  private static String s = "Hallo";

}
```

Im Gegensatz zu einem Attribut ist eine Klassen-Variable nicht einem Objekt, sondern einer Klasse zugeordnet. Klassen-Variablen können von Element- und von Klassen-Funktionen genutzt werden.

Klassen-Variablen werden initialisiert, wenn die Klasse in die JVM geladen wird. Für die Definition der Initialisierung gibt es drei Arten:

- Man macht nichts: in diesem Fall werden die entsprechenden Klassen-Variablen entsprechend ihrem Typ initialisiert.
- Man kann den gewünschten Wert direkt bei der Klassen-Variablen -Definition angeben.
- Eine Klasse kann statische Initialisierungs-Blöcke enthalten – dies wird in der Vorlesung nicht besprochen.

12.4.2 Klassen-Konstanten

An vielen Stellen sind beim Programmieren Konstanten – gerade Integer Konstanten – sehr hilfreich. Eine Konstante ermöglicht es ihnen statt eines Literals einen sprechenden Namen zu benutzen, und den Wert der Konstanten an einer einzigen Stelle zu definieren, egal wie oft und wo sie die Konstante benutzen.

```
// Bsp ohne Konstante

double bruttoPrice = 1.19 * nettoPrice;       // (*)

...

System.out.println(value/1.19);               // (**)
```

```
// Bsp mit Konstante - Achtung, nur Pseudocode

MWST = 1.19

double bruttoPrice = MWST * nettoPrice;
```

```
...

System.out.println(value/ MWST);
```

Während in Zeile (*) noch recht offensichtlich ist, dass das Literal wohl die Mehrwert- oder Umsatz-Steuer meint, ist dies in Zeile (**) nur noch durch den Kontext erkennbar. Stattdessen ist dies im Beispiel mit Konstante in beiden Fällen sofort klar.

Ändert sich die Umsatz-Steuer, so müssen Sie im ersten Beispiel alle Stellen im Programmcode finden, die ein Literal enthalten, und sie ändern. Und hoffentlich vergessen sie keine Stelle bzw. ändern keine „19“ um, die nichts mit der Umsatz-Steuer zutun hat. Im zweiten Beispiel müssen Sie nur die eine Konstanten-Definition ändern, was viel weniger Arbeit ist und auch weniger fehlerträchtig.

Eine Konstante ist in Java eine normale Element- oder Klassen-Variable, die den Modifier „final“ bekommt. Damit lässt sich die entsprechende Variable nicht mehr ändern.

```
public class A {

  public static final int CONST_VAR = 42;

}
```

Bemerkung – Konstanten sind in Java typischerweise Klassen-Konstanten, d.h. konstante Klassen-Variablen, da die Konstante normalerweise ja für alle Objekte gleich ist. Also mit Modifier „static“.

Bemerkung – Klassen-Konstanten sind, obwohl Variablen, häufig „public“ oder „protected“, da sie meist von überall genutzt werden können sollen. Da sie konstant sind, d.h. nicht geändert werden können, ist der unbeschränkte Zugriff ja auch kein Problem.

Bemerkung – der Modifier „final“ wirkt bei Referenz-Variablen nur auf die Variable, und nicht auf das referenzierte Objekt.

Bemerkung – Klassen-Konstanten werden in Java per Konvention in Großbuchstaben geschrieben, und die einzelnen Wörter durch Underscores getrennt.

12.5 Aufzählungen

Aufzählungen werden benötigt, wenn eine Variable nur wenige feste Werte annehmen kann, die alle zur Entwicklungs-Zeit schon feststehen. Beispiele hierfür sind:

- Ausrichtung von Text in einem Absatz:
  - Linksbündig, zentriert, rechtsbündig, blocksatz
- Spielfarben in einem Kartenspiel:
  - Karo, Herz, Pik, Kreuz
- Belegung eines Feldes beim Tic-Tac-Toe:
  - Unbelegt, weiß, schwarz

- Auflistung der 16 HTML Grund-Farben:
  - Schwarz, weiß, rot, grün,…

In einem solchen Fall wünscht man sich als Programmierer einen speziellen Typ, der nur die jeweiligen festen Werte aufnehmen kann – eine Aufzählung oder Enumeration, die es seit dem JDK 1.5 in Java gibt.

- Intern werden Enums in Java mit Klassen definiert, d.h. Enums sind eigentlich nur Klassen. Daher bestimmt wie bei Klassen auch der Enum-Name den Datei-Namen, und in der Datei steht nur die Enumeration.
- Die Enumeration wird mit dem Schlüsselwort „enum“ eingeleitet, und ist meistens „public“.
- Im einfachsten Fall besteht der Enum nur aus den Aufzählungs-Konstanten, die vergleichbar zu Klassen-Konstanten sind. Als solche werden sie natürlich groß geschrieben.
- Der Enum (im Beispiel „Color“) kann wie eine Klasse genutzt werden – d.h. z.B. für Variablen-Definitionen oder in Funktions-Schnittstellen – natürlich mit Referenz-Semantik.
- Java sorgt dafür, dass nur die Aufzählungs-Konstanten in der Enum-Definition existieren. Man kann im Programm keine weiteren Aufzählungs-Konstanten anlegen. Daher funktioniert der Vergleich mit „==“ (Identität), und man muss nicht „equals“ benutzen.

```
public enum Color {
   RED,
   GREEN,
   BLUE
}
```

```
public class A {

   public static void checkColor(Color c) {
      if (c == Color.RED) {
         System.out.println("Farbe ist rot");
      } else {
         System.out.println("Farbe ist nicht rot");
      }
   }

   public static void main(String[] args) {
      Color co = Color.GREEN;
      checkColor(co);
      checkColor(Color.RED);
   }

}
```

```
Ausgabe
Farbe ist nicht rot
Farbe ist rot
```

Ein typisches Einsatzgebiet von Enums sind Switch-Anweisungen, in denen sie genutzt werden können.

```
Color c = Color.GREEN;
switch (c) {
case RED:
   System.out.println("Farbe ist rot");
   break;
case GREEN:
   System.out.println("Farbe ist gruen");
   break;
case BLUE:
   System.out.println("Farbe ist blau");
```

```
    break;
}
```

Ausgabe
```
Farbe ist gruen
```

12.5.1 Enums in Klassen

Selbst wenn wir innere Klassen noch nicht kennen (siehe späteres Kapitel) – Enums können auch innerhalb von Klassen als eine Art klassenbezogene Aufzählung definiert werden. Das Schlüsselwort „static“ ist hierbei optional.

```
public class A {
    public static enum Color1 { RED, GREEN, BLUE };
    public enum Color2 { RED, GREEN, BLUE };
}
```

12.5.2 Enums sind Klassen

Enums sind in Wirklichkeit echte Klassen, und können als solche auch definiert und genutzt werden. Daher man kann die Enum-Konstanten mit weiteren Informationen (Attributen) und Element-Funktionen anreichern:

```
public enum Color {

    RED(255,0,0), GREEN(0,255,0), BLUE(0,0,255);

    private int r,g,b;

    private Color(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }

    public int getRedValue() {
        return r;
    }

}
```

Achtung – Konstruktoren von Enumerationen müssen „private“ sein, da nur die Klasse die Enumerations-Konstanten erzeugen darf.

12.5.3 Weiteres

- Im JDK wurden außerdem noch statische Imports eingeführt, die man u.a. mit Enums nutzen kann. Aus Zeitmangel werden wir keine statischen Imports besprechen.
- Enums bringen von sich aus einige Funktionen mit sich, wie z.B. die Klassen-Funktionen:
  - „values“, die ein Array aller Enum-Konstanten zurückgibt.
  - „valueOf“, die die Enum Konstante zum übergebenen String zurückgibt.

```
for (Color c : Color.values()) {
    System.out.println(c);
}

Color c = Color.valueOf("RED");
```

```
System.out.println("-> " + c);
```

```
Ausgabe
RED
GREEN
BLUE
-> RED
```

12.5.4 Historie

Erst mit Java 5.0 (JDK 1.5) sind Aufzählungs-Typen in Java eingeführt worden. Bis dahin wurden meist Integer-Konstanten für Aufzählungen verwandt, typischerweise so:

```
public class Text {

  public static final int LEFT = 1;
  public static final int CENTER = 2;
  public static final int RIGHT = 3;
  public static final int BLOCK = 4;

  public void setAlignment(int arg) {
    ...
  }

}
```

Obwohl diese Lösung mit Integer-Konstanten alles andere als gut ist (fehlende Typsicherheit), sollten Sie sie kennen, denn sie findet sich in vielen altem Code und vor allem auch in den Schnittstellen der Java Bibliothek, denn diese wurde zum größten Teil schon in den JDKs 1.0 bis 1.4 definiert.

13 Packages

Packages sind neben Klassen das zweite Modul-Konzept von Java. Während Klassen wiederverwendbare Komponenten darstellen und einen Aspekt der realen Welt abbilden, sind Packages ein reines Strukturierungsmittel, das konzeptionell zusammengehörige Klassen zu einer Einheit verbindet.

13.1 Package-Anweisung

In Java ist jede Klasse automatisch Bestandteil eines Packages - indem sie eine .java Datei mit einer Package-Anweisung wie z.B. „*package packagename;*“ beginnen, geben sie automatisch das Package an, zu dem die Klasse gehört.

```
package mypackage;        // Package Anweisung

public class A {          // Die Klasse A liegt jetzt im Package „mypackage“
}
```

Achtung – eine Package-Anweisung **muss** immer die erste Anweisung in einem Quelltext sein. Daher vorher darf es nur Kommentar- und Leerzeilen geben.

Hinweis – Klassen in Quelltexten ohne Package-Anweisungen liegen im sogenannten Default-Package – siehe Kapitel 13.5.

Hinweis – denken sie daran, dass jedes Package auf der Datei-Systeme-Ebene einem Verzeichnis entsprechen muss.

Hinweis – per Konvention sollten Package Namen immer klein beginnen und klein geschrieben werden, z. B.: „nameeinespackage“.

13.2 Klassen in Packages

Der vollständige Name einer Klasse ist nicht nur der Klassen-Name, sondern beinhaltet auch den bzw. die Package-Namen, durch den Punkt-Operator getrennt – Beispiel „java.util.Date“. Dies wird auch der „vollständig-referenzierte“ oder auch der „vollständig-qualifizierte“ Name genannt.

Um eine Klasse zu benutzen gibt es drei Möglichkeiten:
- Benutzung des vollständig qualifizierten Namens.
- Benutzung einer Import-Anweisung für die Klasse.
- Benutzung einer Import-Anweisung für das gesamte Package der Klasse.

13.2.1 Benutzung des vollständig qualifizierten Namens

Sie können immer jede Klasse über ihren voll referenzierten Namen ansprechen.

```
java.util.Date d = new java.util.Date();
```

13.2.2 Benutzung einer Import-Anweisung

Sie können eine Klasse in den Namensraum ihrer Datei importieren. Dafür können sie mit dem Schlüsselwort *„import“* Import-Anweisungen an den Anfang ihrer Datei schreiben. Import-Anweisungen müssen nach der Package-Anweisung stehen, wenn eine solche vorhanden ist. Aber sie müssen vor jeder Klassen- oder Interface-Definition erfolgen.

Mit *import* und exakt referenzierter Klasse importieren Sie eine Klasse:

```
import java.util.Date;

...

Date d = new Date();
```

Alternativ können sie mit einer Import-Anweisung auch alle Symbole eines Package in den Namensraum ihrer Datei importieren. Hierfür muss statt des Klassen-Namens in der Import-Anweisung ein „*“ angegeben werden.

```
import java.util.*;
```

```
...

Date d = new Date();
Vector v = new Vector();
```

Hinweis – bevorzugen Sie einzelne Imports. Denn wenn Sie alle Symbole eines Packages mit "*" importieren, ist natürlich die Gefahr von Namenskonflikten viel größer.

13.2.3 Klassen im gleichen Package

Klassen im gleichen Package brauchen weder importiert noch vollständig qualifiziert werden – sie sind immer automatisch bekannt und können einfach durch Benutzung des Klassen-Namens angesprochen werden.

13.3 Language-Package

Z. B. die Klassen String und Object sind Teil des Packages **java.lang**, trotzdem konnten wir sie benutzen ohne dieses Package importiert zu haben. Das Package java.lang wird immer automatisch importiert, ohne dass Sie sich darum kümmern müssen.

13.4 Verschachtelung

Packages können natürlich wieder in einander verschachtelt sein. Wollen Sie z. B. eine Klasse Model dem Package generator im Package report zuordnen, so müssen sie nur folgende package-Anweisung am Anfang Ihrer Datei Model.java unterbringen:

```
package report.generator;
```

Wollen sie diese Klasse in einem anderen Package nutzen, so haben sie natürlich folgende drei Möglichkeiten:

```
report.generator.Model m = new report.generator.Model();
```

```
import report.generator.Model;
...
Model m = new Model();
```

```
import report.generator.*;
...
Model m = new Model();
```

13.5 Default-Package

Und was passiert, wenn sie keine Package-Anweisung benutzen? Nichts - auch das ist korrekter Code.

Für kleinere Projekte bzw. schnelle Tests wurde zur Arbeitserleichterung in Java das Feature eingeführt, dass in diesem Fall alle diese Klasse in einem Default-Package der Entwicklungsumgebung landen. Sie brauchen auch keine Import-Anweisung anzugeben – das Default-Package wird immer automatisch importiert.

Dem Java-Compiler ist die physikalische Realisation des Default-Packages übrigens freigestellt - er braucht nur eins, kann aber auch mehrere Packages anlegen und diese dann über mehrere Unterverzeichnisse zu verteilen.

13.6 Verzeichnisse

Packages müssen physikalisch auf der Platte durch Unterverzeichnisse abgebildet werden. Eine Klasse „first.second.third.ClassName“ muss also in einer Datei „ClassName.java“ in der Verzeichnis-Struktur „first/second/third“ liegen.

Moderne Entwicklungsumgebungen wie z.B. Eclipse können die notwendige Verzeichnis-Struktur automatisch im Hintergrund erzeugen, und legen z.B. neue Dateien automatisch richtig ab.

