Programmiersprache

Java

2022 / Teil 5

Detlef Wilkening
www.wilkening-online.de

Programmiersprache Java

10 Arrays

Arrays sind schon kurz am Anfang eingeführt worden. Hier wollen wir sie jetzt detaillierter besprechen.

- Arrays sind in Java Objekte, d.h. sie stellen keinen „elementaren“ Daten-Typ dar.
- Array-Variablen sind Referenz-Variablen.
- Auf die Array-Elemente kann sowohl lesend als auch schreibend mit dem Operator [int] zugegriffen werden.
- Array-Zugriffe mit dem Operator [int] sind 0-basiert.
- Zugriffe auf nicht existente Elemente lösen die Exception „IndexOutOfBoundsException“ aus – siehe Kapitel 10.6.
- Arrays haben ein Attribut „length”, dass die Anzahl an Elementen im Array enthält.
- Arrays sind von der Klasse „Object“ abgeleitet – siehe späteres Kapitel.

10.1 Deklaration

Arrays können in Java auf zwei Arten deklariert werden: Die eckigen Klammern, die ein Array als solches identifizieren, können sowohl am Typ als als auch am Bezeichner stehen. Die typische Syntax in Java ist die erste Variante.

```
int[] a1;      // Typische Java Syntax
int a2[];      // Auch moeglich, sieht man aber selten
```

Initialisierung

Es gibt zwei Arten, Arrays zu initialisieren:

1. Literale Initailisierung
Bei der literalen Initialisierung wird hinter der Deklaration in geschweiften Klammern eine Liste der Initialwerte aufgeführt. Der Compiler erzeugt implizit ein Array entsprechender Größe und weist die Werte dem Array zu.

Es darf dabei keine Größe in den eckigen Klammern der Variablen-Deklaration angegeben werden, auch nicht die Richtige.

```
int[] a = { 1, 2, 3, 4 };
boolean[] b = { true, false, true };
double[2] d = { 1.4, 3.1 };          // Compiler-Error - Groessenangabe nicht
erlaubt
```

Das Array a bekommt hier automatisch die Größe 4, das Array b die Größe 3 zugewiesen. Ausserdem werden die Arrays mit den aufgeführten Werten initialisiert.

Die literale Initialisierung darf nur bei der Deklaration einer Array-Variablen benutzt werden. Für spätere Neubesetzung muss die dynamische Initialisierung benutzt werden.

```
int[] a = { 1, 2, 3, 4 };
...
a = { 1, 2, 6, 8 };           // Compiler-Error
                              // Hier ist nur die dynamische Initialisierung
erlaubt
```

2. Dynamische Initialisierung
Bei der dynamischen Initialisierung legt man selber das Array mit new, Typname und gewünschter Größe in eckigen Klammern an – das Array wird dabei mit den Defaultwerten des Typ‘s initialisiert.

```
int[] a = new int[6];
boolean[] b = new boolean[2];
```

Hierbei wird mit new ein entsprechender Speicherplatz reserviert (a - Größe 6 / b - Größe 2) und die Array-Elemente werden mit den Default-Werten der jeweiligen Datentypen gefüllt.

Auch bei der dynamischen Initialisierung können die Initialisierungswerte in Form einer Liste angegeben werden. Dabei darf keine Größenangabe in den eckigen Klammern stehen und die Liste muss direkt hinter den eckigen Klammern folgen. Die Größe des Arrays bestimmt der Compiler implizit aus der Anzahl an Werten in der Liste.

```
int[] a = new int[] { 2, 5, 9 };
boolean[] b = new boolean[2] { true, false }; // Compiler-Error - wegen
Groessenangabe
```

10.2 Arrays haben ein Attribut: length

Array-Objekte haben keine Funktion und genau ein Attribut, das nur lesenden Zugriff erlaubt: ‚length' liefert die Anzahl an Elementen im Array zurück.

```
int[] a = { 1, 2, 3, 4 };
System.out.println(a.length);
```

Ausgabe
```
4
```

10.3 Arrays durchlaufen

Seit Java 5 (JDK 1.5) gibt es einen neuen Schleifen-Typ für Container und Arrays, der schon kurz vorgestellt wurde. Wollen Sie einfach nur über das gesamte Array laufen, dann nutzen Sie diesen Typ. Alternativ können Sie natürlich eine normale For-Schleife mit Schleifen-Zähler nutzen.

```
int[] a = { 1, 2, 3, 4 };

System.out.println("Neue Schleife: ");
for (int n : a) {
   System.out.print(n + " ");
}
System.out.println();

System.out.println("Alte Schleife: ");
for (int i=0; i<a.length; i++) {
   System.out.print(a[i] + " ");
}
System.out.println();
```

Ausgabe
```
Neue Schleife: 1 2 3 4
Alte Schleife: 1 2 3 4
```

10.4 Arrays sind Referenzen

Auch Arrays sind keine elementaren Datentypen, d.h. unterliegen auch Array-Variablen der Referenz-Semantik.

```
int[] a = { 1, 2, 3, 4 };
int[] b = a;
a[2] = 42;

for (int i : a) {
   System.out.print(i + "  ");
}
System.out.println();

for (int i : b) {
   System.out.print(i + "  ");
}
System.out.println();
```

Ausgabe
```
1  2  42  4
1  2  42  4
```

Im Beispiel zeigen beide Array-Variablen auf das gleiche Array. Der schreibende Zugriff auf das dritte Element von a (a[2] = 42;) ändert damit auch das Array b.

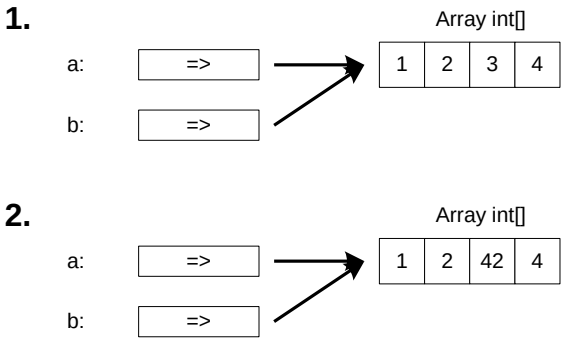

Abb. 10-1 : Referenz-Semantik bei Arrays

10.5 Arrays können mehrdimensional sein

Natürlich können Arrays auch mehrdimensional sein. Sie sind kein spezieller Datentyp, sondern werden als Array-von-Arrays implementiert. Die Deklaration der Variablen enthält dann für jede Dimension ein Klammen-Paar, die Initialisierungsliste besteht dann aus einer Liste von Listen. Der Zugriff erfolgt über eine entsprechende Anzahl von Index-Operatoren [].

```
int[][] a = { {1,2}, {3, 4}, {5,6} };
for (int[] aa : a) {
   for (int i : aa) {
      System.out.print(i + " ");
   }
   System.out.println();
}
```

```
Ausgabe
1 2
3 4
5 6
```

Achtung – seien Sie sich darüber im klaren, dass Java die Elemente eines mehrdimensionales Arrays nicht bündig in einem Block im Speicher ablegt, sondern es ein echtes Array-aus-Arrays ist.

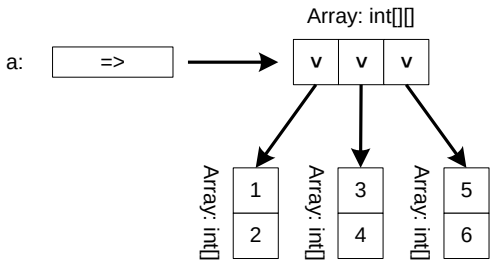

Abb. 10-2 : Speicher-Layout eines mehrdimensionalen Arrays

Arrays müssen nicht unbedingt rechteckig sein

Aus dem oben gesagten folgt natürlich auch, dass Arrays nicht unbedingt rechteckig sein müssen.

```
int[][] a = { {1}, {2, 3}, {4, 5, 6, 7} };
for (int[] aa : a) {
   for (int i : aa) {
      System.out.print(i + " ");
   }
   System.out.println();
}
```

```
Ausgabe
1
2 3
4 5 6 7
```

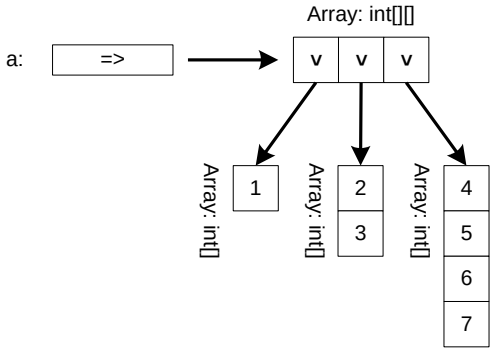

Abb. 10-3 : Speicher-Layout eines nicht-rechteckigen Arrays

10.6 Fehlerhafter Index

Der Zugriff auf ein nicht existentes Objekt führt java-typisch zu einer Exception („java.lang.IndexOutOfBoundsException“). Zugriffe mit fehlerhaftem Index führen also nicht zu potentiellen Problemen wie in manchen anderen Programmier-Sprachen, sondern werden sauber gemeldet.

```
int[] array = { 1, 2 };
array[0] = 0;            // Okay
array[1] = 0;            // Okay
array[2] = 0;            // Laufzeit-Fehler => wirft Exception
array[-1] = 0;           // Laufzeit-Fehler => wirft Exception
```

10.7 Vergleiche

Werden zwei Arrays mit dem Operator „==“ verglichen, so wird nur die Identität verglichen, da es sich um Referenz-Objekte handelt – siehe früheres Kapitel. Auch die Benutzung der von „Object“ geerbten Element-Funktion „equals“ ändert daran nichts, da das Default-Verhalten der Vergleich auf Identität ist – siehe späteres Kapitel – und die Elementfunktion für Arrays nicht überschrieben worden ist.

Für Werte-Vergleiche auf Arrays muss die Funktion „java.util.Arrays.equals“ benutzt werden.

11 Klassen

11.1 Motivation

Um nicht-triviale Probleme in den Griff zu bekommen, muss man die Probleme in kleine überschaubare Einheiten zerlegen (Stichwort „teile und herrsche“), die möglichst unabhängig von einander sein sollten (Stichwort „Entkopplung“). Das heißt, man muss in seinem Programm Abstraktionen einziehen, die Komplexität verbergen (am besten so kapseln, dass gar keine Zugriff auf die Interna möglich ist – Stichwort „Information-Hiding“) und auf eine einfache Art Funktionalität zur Verfügung stellen. Ein Beispiel dafür sind z.B. die Container-Klassen von Java, die die gesamten für die Speicherung und Verwaltung von Objekten notwendigen Daten, Techniken und Mechanismen verbergen.

In unserem normalen Leben arbeiten wir doch auch so. Wenn sie z.B. ein Haus bauen, dann werden weder sie, noch der Architekt, und wohl auch nicht der Maurer hierbei die Eigenschaften von Elementarteilchen wie Quarks oder Bosonen berücksichtigen. Jeder arbeitet mit den Abstraktions-Ebenen, die für sein Problem angepasst sind. Der eine denkt halt in Elementarteilchen, dem nächsten passen Atom-Kerne und Elekronenhüllen. Wieder andere denken in Molekülen, noch andere z.B. in Werkstoffeigenschaften. Ein Ingenieur wechselt dann sich in die Sichtweise von – tief unten – Schrauben und Zahnrädern. Weiter oben wird dann z.B. in Getriebe oder Motoren gedacht. Usw, usw. . Jeder arbeitet halt mit den Abstraktions-Ebenen, die für sein Problem angepasst sind. Und alles darunter interessiert ihn nicht, bzw. wird aktiv versucht auszublenden, damit das Problem überschaubar bleibt.

Einer der Hauptgedanken der Objekt-Orientierung ist daher die Idee, abgeschlossene gekapselte Einheiten programmieren und anbieten zu können, bei denen der Benutzer sich keine Gedanken mehr über das Innenleben machen muss, sondern sie einfach über ihre Schnittstelle benutzt.

Damit ist das Ziel klar – es muss möglich sein:
- abgeschlossene gekapselte Einheiten zu definieren,
- diesen Einheiten eine Schnittstelle zu geben,
- sie sollte einfach zu implementieren sein, und
- einfach zu benutzen sein.

Dies wird in Java – wie in allen OO Sprachen – über Klassen und Objekte erreicht. Hierbei sind
- **Klassen** eine allgemeine Beschreibung einer bestimmten Art Objekte. Eine Klasse Auto beschreibt ganz allgemein alle Autos, sowohl meine alte Karre, einen neues Familienauto, als auch das Cabrio meines Kollegen. Allen diesen Autos ist gemein, dass sie Autos sind, und sich durch gemeinsame Fähigkeiten und Merkmale wie Größe, Leistung, Verbrauch,

Farbe, usw. auszeichnen.
- **Objekte** dagegen sind ein konkretes Exemplar (Instanz) einer Klasse. Daher meine alte Karre ist ein konkretes Exemplar der Klasse Auto.

11.2 Klassen-Entwurf

Bevor sie eine eigene Klasse entwerfen, sollten sie sich immer einige Gedanken über den Sinn, die Repräsentation und die Implementation der Klasse machen.

1. Überlegung - Art der Objekte:
- Was für Objekte soll die Klasse repräsentieren?

2. Überlegung - Schnittstelle:
- Was soll mit den Objekten der Klasse machbar sein?
- Wie sollen sich die Objekte verhalten?
- Welche Schnittstelle benötigt die Klasse dafür?

3. Überlegung - Vererbung:
- Gibt es Klassen, die eine vergleichbare Schnittstelle haben?
- Gibt es Klassen, die ähnliche Funktionalität aufweist?
- Gibt es Klassen, die diese Objekte in allgemeinerer Form repräsentieren?
- Kann eine abstrakte Schnittstelle konkrete Klassen verbergen?

Diese Frage können wir uns erst stellen, wenn wir Vererbung kennen – siehe späteres Kapitel. Ich habe sie hier aber aufgeführt, damit sie hier alle relevanten Fragen auf einen Schlag sehen.

4. Überlegung - Implementierung:
- Was soll mit den Objekten der Klasse machbar sein?
- Wie sollen sich die Objekte verhalten?
- Wie implementiere ich diese Funktionalitäten?
- Welchen inneren Aufbau wähle ich dafür?

Bemerkung – die zweite und vierte Überlegung beruhen auf den gleichen Fragen, aber einmal aus der Sicht eines Benutzers der Klassen, und das andere Mal aus der Sicht des Implementierers gestellt.

Bemerkung – in einem größeren Kontext betrachtet würde man die Überlegungen 1 bis 3 als Analyse und die Überlegungen 2 bis 4 als Design betrachten.

11.3 Beispiel

11.3.1 Wurm-Programm

Vielleicht kennen sie kleine Gimmick-Programme, die z.B. gerne als Bildschirmschoner eingesetzt werden. Hier konkret geht es um ein Programm, das einen Wurm über den Bildschirm krabbeln lässt. Dabei soll der Wurm, der den Bildschirm rechts verlässt links wieder

hinein krabbeln, und umgekehrt. Verlässt der Wurm den Bildschirm oben, so soll er unten wieder hinein krabbeln, und umgekehrt. Für den Wurm bildet der Bildschirm also eine unendliche Fläche, bei der die jeweils gegenüberliegenden Ränder aufeinander abgebildet werden.

Da wir zurzeit noch kein GUI programmieren können, und auch sonst noch ziemlich viel Java-Werkzeug fehlt, beschränken wir uns hier mit einem ganz kleinen und sehr stark abgespeckten *Wurm* Programm:

- Statt mit einem GUI Fenster arbeiten wir nur mit der Konsole.
- Statt eines Wurms wird hier nur ein Zeichen gezeichnet – der Stern ‚*'.
- Statt der Bildschirmbreite setzen wir eine feste Breite von 40 Zeichen an.
- Statt mit einer x/y-Koordinate wird hier nur x variiert.
- Statt den alten Wurm zu löschen, geben wir eine neue Zeile aus.

Übrig bleibt also nur ein Programm mit ungefähr folgender Ausgabe:

```
*
 *
  *
   *
    ...
                                    *
                                     *
                                      *
*
 *
  ...
```

Alles in allem ist das neue Programm nicht mehr sehr eindrucksvoll, aber es hat noch genügend Probleme für den Anfang:

- Um die Ausgabe einzurücken, definieren wir eine Hilfs-Funktion „spaces“, die einen StringBuffer der entsprechenden Größe zurückgibt. Die Funktion ist mit ihrer Schleife nicht besonders elegant, aber manchmal muss man so programmieren.
- Damit die Ausgabe nicht nur so dahin-fliegt, müssen wir das Programm verlangsamen. Nach jeder Ausgabe legen wir es daher für 30 ms schlafen. Auch hierfür gibt es eine extra Funktion „sleep“, die für uns den „Thread.sleep“ Aufruf kapselt, und den notwendigen try/catch-Block übernimmt.
- Statt einem festen Programm-Ende oder z.B. einer Tastatur-Abfrage aufs Programm-Ende arbeiten wir mit einer Endlos-Schleife. Das Programm muss daher auf der Konsole mit „Strg+C“ oder in der IDE abgebrochen werden.

```
public class Appl {

  public static StringBuffer spaces(int len) {
    StringBuffer sb = new StringBuffer(len);
    for (int i=0; i<len; i++) {
      sb.append(' ');
    }
    return sb;
  }

  public static void sleep() {
    try {
      Thread.sleep(30);
    } catch (Exception x) {
```

```
    }
  }

  public static void main(String[] args) {
    System.out.println("Ein Stern laeuft ueber den Bildschirm");

    int pos = 0;
    while (true) {
      System.out.println(spaces(pos) + "*");
      sleep();
      ++pos;
      if (pos>40) {
        pos=0;
      }
    }
  }

}
```

Das eigentliche *Programm* reduziert sich so auf eine Initalisierung und eine Endlos-Schleife mit 6 Zeilen:

```
int pos = 0;                                  // (*)
while (true) {
  System.out.println(spaces(pos)+"*");
  sleep();
  ++pos;
  if (pos>40) {                               // (**)
    pos=0;                                    // (***)
  }
}
```

An welcher Stelle könnte dieses Programm durch eine weitere Abstraktion vereinfacht werden?

Immerhin die Hälfte des Programms beschäftigt sich mit dem Positions-Zähler „pos“. Statt eines „int“ Elements wird hier ja ein sogenannter Ring-Zähler benötigt, und der größte Teil der Programmlogik beschäftigt sich jetzt mit dem Problem einen Ring-Zähler zu simulieren. Ausserdem ist nicht schön, dass sich die zwei wichtigsten Parameter des Ring-Zählers an mehreren Stellen wiederfinden: Start- und End-Wert (*), (**) und (***).

11.3.2 Überlegung 1

Damit ist Überlegung 1 „Art der Objekte“ beantwortet: wir wollen einen Ring-Zähler implementieren, d.h. eine Art Integer-Variable die bei Erreichen eines bestimmten Wertes wieder auf einen Startwert springt.

Bemerkung – mancher wird nun denken: „Was soll das?“ Die Schleife ist doch übersichtlich, das Problem klar, die Lösung einfach – wozu hier Aufwand investieren? Aber diese Einstellung greift zu kurz, denn:

- Nicht immer sind alle Code-Vorkommen eines Ring-Zählers so klar und nah beieinander lokalisiert. Nicht selten finden sich diese Stellen in realen Programmen verteilt in mehreren Stellen wieder. Und in solchen Fällen ist dem Leser nicht immer klar, warum hier solcher Code steht. Wir haben es hier halt mit einem sehr einfachen Beispiel-Programm zu tun.
- Und selbst wenn der Code und die Lösung einfach sind: man kann auch einfache Sachen vergessen oder falsch machen.

- Ein fertiger Ring-Zähler – wir reden hier jetzt von der zweiten Verwendung der Klasse, nicht von ihrer ersten – ist ein fertiger Ring-Zähler. Er ist getestet und im Einsatz und läuft – es ist ein fertiger Ring-Zähler. Man kann ihn nehmen und benutzen. Das nennt man Wiederverwendung und sollte ein Ziel jeder Software-Entwicklung sein.
- Und warum müssen Abstraktions-Ebenen einen großen Abstand haben, und die Einheiten dazwischen kompliziert damit sie sich lohnen? Sind sie groß und kompliziert, so sind sie wahrscheinlich auch fehlerhaft. Sind sie groß und kompliziert, so passen sie sicher nicht beim nächsten Problem.
- Und bedenken sie – es ist unser erstes Beispiel. Das ändert nichts daran, dass ein Ring-Zähler als wiederverwendbare Einheit (Klasse) Sinn macht. Aber es gibt sicher auch noch komplexere Klassen.

11.3.3 Überlegung 2

Die Überlegung 2 zielt dann auf die Schnittstelle der Klasse:

- Was soll mit den Objekten der Klasse machbar sein?
- Wie sollen sich die Objekte verhalten?
- Welche Schnittstelle benötigt die Klasse dafür?

Hier ist die Lösung einfach:

- Bei der Erzeugung des Ring-Zählers müssen Start- und Grenzwert definiert werden können.
- Der aktuelle Wert muss abfragbar sein.
- Der Wert muss incrementiert werden können, wobei eine Überschreitung des Grenzwerts implizit wieder zum Startwert führen muss.

Programm-technisch könnte dies aus der Sicht des Benutzers so aussehen:

```
RingCounter rc = new RingCounter(0, 40);
while (true) {
  System.out.println(spaces(rc.get()) + "*");
  sleep();
  rc.add();
}
```

6 Zeilen statt 9, und außerdem noch der Vorteil eine if-Anweisung los geworden zu sein. Und jede Kontrollstruktur bringt halt Komplexität und potentielle Fehlerquellen mit sich.

11.3.4 Überlegung 3

Da wir noch keine Vererbung kennen, ist diese Überlegung zurzeit noch nicht relevant.

11.3.5 Überlegung 4

So bleibt noch die Frage nach der Implementierung, d.h. wie implementiere ich das?

11.4 Klassen-Implementierung

11.4.1 Definition

Zuerst muss eine Klasse definiert werden.

Syntax Kassen-Definition

```
[Modifizierer] class klassenname {
  ...
}
```

Achtung – eine Klassen-Definition hat immer ihre eigene Datei, die genauso wie die Klasse mit der Extension ‘java’ heißen muss. Dies ist ein gern gemachter Anfänger-Fehler. Wenn sich ihre Klasse also nicht compilieren lässt - überprüfen sie erstmal die Namen.

Die für Klassen möglichen Modifizierer werden später aufgeführt und erklärt. Im Augenblick sollten sie hier immer nur ‘public’ verwenden.

Beispiele:

```
public class RingCounter {
  ...
}
```

```
public class Person {
  ...
}
```

Innerhalb einer Klasse können Konstruktoren, Element-Funktionen, Klassen-Funktionen, Attribute, innere Klassen, und einige weitere Elemente vorhanden sein. Eine Klasse kann aber auch einfach leer sein.

11.4.2 Benutzung

Ist eine Klasse definiert, so kann sie im gleichen Package einfach unter ihrem Typ-Namen benutzt werden. Achtung – Variablen der Klasse sind immer Referenz-Variablen. Und erzeugt werden Objekte der Klasse mit dem „new“ Operator.

```
public class A {
}
```

```
public class Appl {

  public static void main(String[] args) {
    A a = new A();
    f(a);
    a = g();
  }

  public static void f(A a) {
  }

  public static A g() {
    return new A();
  }
```

```
}
```

11.4.3 Garbage Collector

Erzeugte Objekte müssen nicht explizit gelöscht werden. Dies macht die JVM automatisch. Dafür gibt es im System den sogenannten Garbage-Collector, der automatisch im Hintergrund alle nicht mehr referenzierten Objekte löscht. Den Vorgang des *Einsammelns* nicht mehr *benötigter* Objekte nennt man Garbage-Collection. Je nach Implementierung in der JVM wird diese von Zeit zu Zeit aufgerufen, oder läuft kontinuierlich im Hintergrund.

11.4.4 Element-Funktionen

Leere Klassen sind meist ziemlich sinnlos. Um mit den Objekten der Klasse agieren zu können, sollten diese Element-Funktionen haben.

Syntax Funktions-Definition

```
[Modifizierer] rueckgabetyp funktionsname(parameterliste) {
  // Funktions-Rumpf
}
```

Gegenüber den bisher verwendeten Klassen-Funktionen unterscheiden sich Element-Funktionen durch den fehlenden Modifier „static“, und dass sie nur mit Objekt-Bezug aufgerufen werden können.

```
public class A {

  public static void f() {
  }

  public void g() {
  }

  public static void main(String[] args) {

    f();              // Okay, da Klassen-Funktion
    g();              // Compiler-Fehler, da Element-Funktion -> Objekt-Bezug
noetig

    A a = new A();
    a.g();            // Okay, g() wird fuer das Objekt a aufgerufen
  }

}
```

Die für Funktionen möglichen Modifizierer werden später aufgeführt und erklärt. Im Augenblick sollten sie bei Element-Funktionen immer nur ‘public’ verwenden, bei Klassen-Funktionen „public“ und „static“.

Und wie sieht unser Ring-Zähler jetzt aus?

```
public class RingCounter {

  public void add() {
    ...                          // Die Implementierung ist noch offen
```

```
  }

  public int get() {
    ...                          // Die Implementierung ist noch offen
  }

}
```

11.4.5 Warum Element-Funktionen?

Weshalb gibt es überhaupt Element-Funktionen? Warum nimmt man nicht einfach Klassen-Funktionen, und spart sich den Aufwand mit den Objekten?

Man kann beliebig viele Objekte einer Klasse erzeugen kann, die alle voneinander abhängig sind. Element-Funktionen beziehen sich jeweils nur auf das Objekt, für das sie aufgerufen werden, d.h. jedes Objekt kann (und hat) seinen eigenen von allen anderen unabhängigen Status. Klassen-Funktionen beziehen sich auf die Klasse, für die es nur einen Status gibt. Ohne Objekte und Element-Funktionen wäre sowas nicht möglich.

```
StringBuilder s1 = new StringBuilder("Hallo Welt");
StringBuilder s2 = new StringBuilder("Java");
System.out.println("Die Laenge von \"" + s1 + "\" ist " + s1.length());
System.out.println("Die Laenge von \"" + s2 + "\" ist " + s2.length());
```

„s1“ und „s2“ sind zwei unabhängige Objekte, die unterschiedliche Strings repräsentieren. Nur Funktionen mit Objekt-Bezug können wissen, welches Objekt sie denn nun *bearbeiten* sollen.

11.4.6 Attribute

Natürlich hält ein Objekt Daten. Der Ring-Zähler z.B. muss sowohl seinen Start- und Endwert kennen, als auch seinen aktuellen Wert. Objekt-spezifische Daten – d.h. Variablen in Objekten – werden Attribute genannt. Element-Funktionen können einfach unter Verwendung des Attribut-Namens auf die Attribute zugreifen und sie benutzen.

Achtung – in Java wird auch gerne der historisch begründetet Name „Feld“ statt „Attribute“ verwandt.

Syntax Attribut-Definitionen

```
[Modifizierer] typ name;
```

Beispiel

```
public class A {

  private int i;
  private String s;

  public void set(int arg1, String arg2) {
    i = arg1;
    s = arg2;
  }

  public void print() {
    System.out.println(i + " => " + s);
```

```
  }

}
```

```
public class Appl {

  public static void main(String[] args) {

    A a1 = new A();
    A a2 = new A();

    a1.set(1, "Hallo");
    a2.set(2, "Java");

    a1.print();
    a2.print();
  }

}
```

```
Ausgabe
1 => Hallo
2 => Java
```

Die für Attribute möglichen Modifizierer werden später aufgeführt und erklärt. Im Augenblick sollten sie bei Attributen immer nur ‘private’ verwenden.

Und wie sieht unser Ring-Zähler jetzt aus?

```
public class RingCounter {

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

  public void add() {
    if (++value>limit) {
      value=start;
    }
  }

  public int get() {
    return value;
  }

}
```

Jetzt bleibt nur noch die Frage offen, wie denn die Attribute sinnvoll initalisiert werden, d.h. mit den gewünschten Start- und End-Werten? Zurzeit werden sie als „int“s alle mit „0“ initialisiert.

Hinweis – Attribute werden auch oft Member, Attributes, Element-Variablen, Objekt-Variablen oder (speziell in Java) auch Felder genannt.

11.4.7 Konstruktoren

Um ein Objekt bei der Erzeugung sauber zu initialisieren, gibt es spezielle Methoden - die Konstruktoren. Sie haben keinen Rückgabetyp und heissen immer wie die Klasse. Sie werden automatisch **immer** beim Erzeugen eines neuen Objektes aufgerufen.

```
public class Person {
```

```
  private String forename;
  private String surename;
  private int age;

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

  public void print() {
    System.out.println("Name: " + forename + " " + surename);
    System.out.println("Alter: " + age);
  }

}
```

```
Person p = new Person("Detlef", "Wilkening", 16);   // Das Alter stimmt nicht
p.print();
```

Ausgabe

```
Name: Detlef Wilkening
Alter: 16
```

Nur wenn sie **keinen** Konstruktor definieren, erzeugt der Compiler automatisch einen sogenannten Standard-Konstruktor, der ohne Argumente aufgerufen werden kann. Der Konstruktor wird entsprechend denen bei **new** angegebenen Argumenten ausgewählt.

```
public class A {
}
```

```
Person p = new Person();  // Compiler-Fehler, dieser Konstruktor existiert hier
nicht

A a = new A();       // Klasse A hat einen automatisch erzeugten Standard-
Konstruktor
```

Und wie sieht unser Ring-Zähler jetzt aus?

```
public class RingCounter {

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

  public RingCounter(int s, int l) {
    start = s;
    limit = l;
    value = s;
  }

  public void add() {
    if (++value>limit) {
      value=start;
    }
  }

  public int get() {
    return value;
  }

}
```

Nun ist die Klasse vollständig und unser Beispiel funktioniert.

Bemerkung – die Ring-Zähler Klasse ist in diesem Zustand natürlich nicht wirklich in einem produktiven Zustand. Z.B. kann man sie mit einem End-Wert, der kleiner als der Start-Wert ist, problemlos aushebeln.

