Einführung in das Programmieren (1)

1. Einleitung

Einleitende Frage:

Was haben ein Päckchen Gummibärchen und Programmieren miteinander zu tun?

Eine gute Antwort ist: Für viele Menschen wohl gar nichts. Für angehende Informatiker sehr viel, denn an Hand eines Päckchens Gummibärchen lassen sich anschaulich **Denkweisen und grundlegende Prinzipien der Informatik** erklären.

Mögliche Aufgabenstellungen zum Päckchen Gummibärchen:

1. Alle roten (gelben usw.) Gummibärchen aus dem Päckchen nehmen.
2. Alle nicht roten (nicht gelben usw.) Gummibärchen aus dem Päckchen nehmen.
3. Zählen, wie viele rote (gelbe usw) Gummibärchen in dem Päckchen sind.
4. Zählen, wie viele rote und gelbe Gummibärchen in dem Päckchen sind.
5. Zählen, wie viele Gummibärchen in dem Päckchen sind.
6. Alle Gummibärchen in dem Päckchen nach Farben sortieren.

Betrachte die Lösung der Aufgabenstellungen unter folgenden Gesichtspunkten:

- Beschreibe umgangssprachlich den genauen Ablauf, wie die Aufgabe gelöst wird.
  (vgl. Vorgangsbeschreibungen, Gebrauchsanweisungen, Kochrezepte, mathematische Methoden).
- Erkenne die Notwendigkeit, aus der umgangssprachlichen Formulierung eine „Sprache des Programmierers“ zu entwickeln.

Lösungsvorschläge zu 1.:

Kommentar zum Lösungsvorschlag:

Lösung 1:

- Schaue nach, welche Gummibärchen rot sind und nimm diese aus dem Päckchen.

Lösung 2:

1 Öffne das Päckchen.
2 Lege alle Gummibärchen auf einen Teller.
3 Suche alle roten Gummibärchen heraus und lege sie auf Seite.

Lösung 3:

1 Öffne das Päckchen
2 Entnehme ein Gummibärchen und prüfe, ob es rot ist.
3 Wenn ja, lege es auf die linke Seite; wenn nicht, lege es auf die rechte Seite.
4 Wiederhole die Schritte 2 und 3 bis das Päckchen leer ist.

2. Algorithmen und Struktogramme

2.1. Algorithmus

Beschreibungen von Abläufen enthalten **Anweisungen (=Befehle)**, die genau festlegen, was zu tun ist, in welcher Reihenfolge etwas zu tun ist, ggf. wie oft etwas zu tun ist usw.

Definition:

Eine Folge aus **elementaren, eindeutigen** und **ausführbaren Anweisungen** heißt **Algorithmus**.

Die Bausteine von Algorithmen sind **Anweisungen, Sequenzen, Wiederholungen** und **bedingte Anweisungen**.

2.2. Anweisung

Der einfachste Baustein eines Algorithmus ist eine **Anweisung.**

In jeder Programmiersprache gibt es **vordefinierte Anweisungen** wie *print, write, delete* usw. Das Entscheidende ist jedoch die Tatsache, dass man **eigene Anweisungen** schreiben kann, die so genannten **Methoden**. Jede Methode wird mit einem möglichst aussagekräftigen Namen bezeichnet.

Hinweise zu Java:
- In Java wird jede Anweisung mit einem Semikolen „;“ abgeschlossen.
- Hinter jeder Methodenbezeichnung muss „( )“ stehen. Genaueres später.
- Es gibt Konventionen bei der Namensvergabe.

Beispiele für Methodenbezeichnungen: *oeffnePackchen(); entnehmeGummibaerchen(); sortiereNachFarben();*

2.3. Sequenz

Eine Folge nacheinander auszuführender Anweisungen heißt **Sequenz.**

Beispiel: *oeffnePackchen();*
entnehmeGummibaerchen();

2.4. Wiederholung

Soll eine Anweisung oder Sequenz mehrfach durchlaufen werden, so verwendet man **Wiederholungen (= Schleifen).**

Steht von vorneherein fest, wie oft die Schleife durchlaufen wird, so spricht man von einer **Wiederholung mit fester Anzahl**. Dabei verwendet man einen Zähler für die Anzahl der Durchläufe.

Hängt die Anzahl der Schleifendurchläufe von einer Bedingung ab, so spricht man von einer **bedingten Wiederholung**. Dabei wird die Schleife so lange wiederholt, wie die Bedingung wahr ist.

Eine **Bedingung** ist ein Aufruf einer Methode, bei der eine Frage mit **wahr** oder **falsch** beantwortet wird.

Hinweise zu Java:
- Zusammenhängende Teile eines Programmcodes bezeichnet man als **Blöcke**. Blöcke sind z.B. Schleifen, Methodendefinitionen usw.
- Jeder Block beginnt mit der Blockbezeichnung und der öffnenden Klammer „{“. Das Blockende wird mit der schließenden Klammer „}“ gekennzeichnet.
- Für Wiederholungen mit fester Anzahl verwendet man meistens die **for**-Schleife.
- Struktur der **for**-Schleife:

```
for (int i = 0; i < anzahl; i++)
  {
      Anweisung1;
      Anweisung2;
      ...
  }
```

- Für bedingte Wiederholungen gibt es die **while**-Schleife oder die **do-while**-Schleife.
- Struktur der **while**-Schleifen:

```
while (Bedingung)
    {
        Anweisung1;
        Anweisung2;
        ...
    }
```

```
do
    {
        Anweisung1;
        Anweisung2;
        ...
    }
while (Bedingung);
```

Unterschied: Bei der while-Schleife wird die Bedingung am Anfang geprüft. Ist die Bedingung bereits am Anfang falsch, so wird die Schleife überhaupt nicht durchlaufen. Bei der do-While-Schleife wird die Bedingung am Ende geprüft und damit die Schleife mindestens einmal durchlaufen.

Beispiel 1: Kommentar:

```
int anzahl = 5;
oeffnePackchen();
for (int i = 0; i < anzahl; i++){
    entnehmeGummibaerchen();
    issGummibaerchen();
    }
schmeisseLeeresPaeckchenInPapierkorb();
```

Beispiel 2: Kommentar:

```
oeffnePackchen();
while (istPaeckchenNichtLeer() = = true){
    entnehmeGummibaerchen();
    issGummibaerchen();
    }
schmeisseLeeresPaeckchenInPapierkorb();
```

2.5. Bedingte Anweisung

In vielen Fällen muss in einem Programm an gewissen Stellen eine **Entscheidung** getroffen werden, wie der weitere Programmablauf sein wird.

Eine Situation, in der abhängig von einer Bedingung eine von zwei Sequenzen ausgewählt werden muss, heißt **Auswahl** oder **bedingte Anweisung**.

Hinweise zu Java:
- Struktur:

```
if (Bedingung)
    {
        Sequenz 1
    }
else
    {
        Sequenz 2
    }
```

Ist die Bedingung wahr, dann fährt das Programm mit der Sequenz 1 fort, ist die Bedingung jedoch falsch, so fährt es mit der Sequenz 2 fort.
Der else-Block kann weggelassen werden (= einseitige Bedingung).

Beispiel: Kommentar:

```
if (farbeGummibaerchen = = rot)
{
    issGummibaerchen();
}
```

2.6. Struktogramme

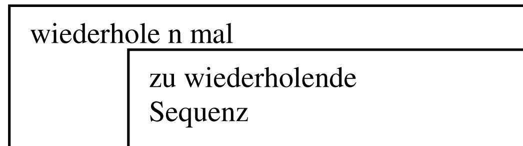
Das Ziel des Programmierens ist es, Algorithmen in die Programmiersprache zu übersetzen. Die einzelnen Programmiersprachen unterscheiden sich durch ihren Befehlssatz, ihre Formalismen, ihre Syntax usw.

Um Algorithmen zum Einen gut lesbar und zum Andern unabhängig von einer bestimmten Programmiersprache zu machen, kann man sie grafisch als so genannte **Struktogramme** darstellen.

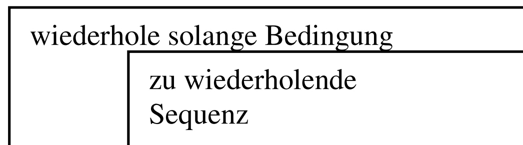
Dabei verwendet man folgende (genormte) **Symbole**:

- für eine (einzelne) Anweisung:

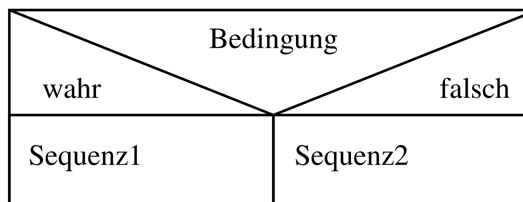
- für eine Wiederholung mit fester Anzahl:



- für eine bedingte Wiederholung:



- für eine bedingte Anweisung:

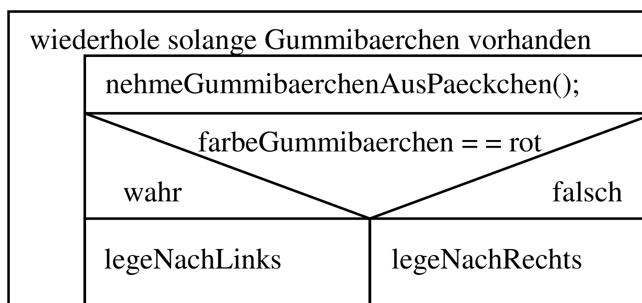


Beispiel: Alle roten Gummibärchen aus dem Päckchen nehmen.

Hauptprogramm:

Hinweis: Die Methode „*nehmeRoteGummibaerchenAusPaeckchen* " ist selbst als Struktogramm darzustellen.

Methode: *nehmeRoteGummibaerchenAusPaeckchen*



Beachte: Ein wesentlicher Gedanke beim Erstellen eines Algorithmus ist darauf zu achten, dass ein **Algorithmus überschaubar und damit möglichst kurz** bleibt. Deshalb verwendet man **eigene Anweisungen**, die **außerhalb des Hauptalgorithmus** eigene Algorithmen darstellen.
Man zerlegt somit das „große" Problem in mehrere „kleine" Probleme und fügt am Schluss alle Teile wieder zusammen. Dieses grundlegende Prinzip nennt man **„Divide and conquer"**.
Dadurch ist bei sehr großen Projekten eine Arbeit im Team überhaupt erst möglich, da sich nun jeder Mitarbeiter um die Lösung eines kleinen Problems kümmern kann.

3. Datentypen

3.1. Einführung

Was könnte das auf einem Papierfetzen bedeuten?

Eine gute Antwort ist: Nicht leicht zu sagen, ohne mehr darüber zu wissen.

Computerspeicher speichert beliebige Bitmuster. Wie bei einer Folge von Buchstaben hängt die Bedeutung einer Folge von Bits davon ab, wie sie verwendet werden.

Ein **Datentyp** ist ein Schema für die Verwendung von Bits, um Werte darzustellen. Werte sind nicht nur Zahlen, sondern jede Art von Daten, die ein Computer verarbeiten kann. Alle Werte in einem Computer werden dargestellt, indem der eine oder der andere Datentyp verwendet wird.

3.2. Primitive Datentypen

Es gibt Datentypen, die so elementar sind, dass die Art und Weise sie zu repräsentieren in der Programmiersprache eingebaut ist. Dieses sind die **primitiven Datentypen**.

Hinweise zu Java:

- Java kennt acht primitiven Datentypen. Sie heißen:

byte short int long float double char boolean

- Java ist eine "**case sensitive**" Programmiersprache, d. h. sie unterscheidet zwischen Groß- und Kleinschreibung. So ist "byte" der Name eines primitiven Datentyps, aber "BYTE" ist es nicht.

In dem Ausdruck *primitiver Datentyp* bedeutet das Wort **primitiv** "eine elementare Komponente, die verwendet wird, um andere, größere Teile" zu erstellen. Dieses Wort wird in der Informatik oft verwendet. Um eine umfangreiche Aufgabe zu lösen, sieht man sich nach primitiven Operationen um, die benötigt werden, dann verwenden man sie, um zu einer Lösung zu kommen.

3.3. Numerische primitive Datentypen

Zahlen sind von so großer Bedeutung in Java, dass 6 von 8 der primitiven Datentypen numerische Typen sind.

Als primitive Typen gibt es sowohl **Ganzzahlen** (***Integer***) als auch **Gleitpunktzahlen**. Ganzzahlen haben keinen Dezimalteil; Gleitpunktzahlen haben einen Dezimalteil. Auf dem Papier haben Ganzzahlen keinen Dezimalpunkt und Gleitpunkttypen haben einen. Aber im Hauptspeicher gibt es keine Dezimalpunkte: selbst Gleitpunktwerte werden durch Bitmuster dargestellt. Es gibt einen wesentlichen Unterschied zwischen der Methode, die verwendet wird, um ganze Zahlen darzustellen und der Methode, die verwendet wird, um Gleitpunktzahlen darzustellen.

Ganzzahlen Primitive Datentypen		
Typ	**Größe**	**Wertebereich**
byte	8 Bit	-128 bis +127
short	16 Bit	-32.768 bis +32.767
int	32 Bit	(ca.) -2 Milliarden bis +2 Milliarden
long	64 Bit	(ca.) -10E18 bis +10E18

Primitive Gleitpunkttypen		
Typ	**Größe**	**Wertebereich**
float	32 Bit	-3.4E+38 bis +3.4E+38
double	64 Bit	-1.7E+308 bis 1.7E+308

Jeder primitive Typ verwendet eine *feste* Anzahl von Bits. Das bedeutet, wenn man einen bestimmten Datentyp verwendet, dann wird immer die gleiche Anzahl von Bits verwendet, egal welcher Wert dargestellt wird.

Sehr große Werte (negative oder positive) brauchen mehr Bits zum darstellen. Das ist ähnlich wie das Schreiben von Zahlen auf Papier: große Zahlen brauchen mehr Ziffern. Wenn ein Wert mehr Bits braucht, als ein bestimmter Datentyp verwendet, dann kann er nicht durch diesen Datentyp dargestellt werden.

Hinweise zu Java:
- Wenn man mit Gleitpunktzahlen zu tun hat, sollte man fast immer Variablen vom Typ `double` verwenden. Nicht nur wegen des größeren Wertebereichs, sondern auch wegen der größeren Genauigkeit.
- **Deklaration von Variablen**: | Datentyp Variablenbezeichner = Wert; |

  Beispiele: *int z = 5;* *int zaehler = 0;* *double x = 5.3;* *double erg = -123.0;*

3.4. Der primitive Datentyp `char`

Schriftzeichen sind in der Programmierung sehr gebräuchlich. Der dafür verwendete primitive Datentyp hat den Namen `char`, um Tipparbeit zu sparen. Der `char`-Typ verwendet 16 Bit für die Darstellung der Zeichen. In vielen Programmiersprachen werden nur 8 Bit für diesen Zweck verwendet. Java verwendet 16 Bit, um auch die Schriftzeichen anderer Sprachen als der Englischen darstellen zu können. Die dafür verwendete Methode wird Unicode genannt.

Beispiel: Das 16-Bit Muster

0000000001100111 stellt als `char` das Zeichen `'g'` dar, als `short` bedeutet es die Zahl 103!

Groß- und Kleinbuchstaben werden durch unterschiedliche Muster dargestellt. Interpunktion und Sonderzeichen sind ebenfalls `char`-Daten. Es gibt auch Sonderzeichen wie das Leerzeichen, das Worte trennt.

Steuerzeichen sind Bitmuster, die das Zeilenende oder den Seitenumbruch anzeigen. Andere Steuerzeichen stellen mechanische Aktionen alter Kommunikationsgeräte (wie Fernschreiber) dar, die heutzutage selten verwendet werden, aber beibehalten werden müssen.

Der primitive Typ `char` repräsentiert ein *einzelnes* Zeichen. Er enthält keinerlei Information über den Font. Wenn man mehr als ein Zeichen gleichzeitig handhaben möchten (fast immer), muss man Objekte verwenden, die aus `char`-Daten konstruiert werden.

3.5. Der primitive Datentyp `boolean`

Ein weiterer primitiver Datentyp ist der Typ `boolean`. Er wird verwendet, um einen einzelnen true/false-Wert darzustellen. Ein boolescher Wert kann nur einen von zwei Werten haben: `true` oder `false`.

Hinweis zu Java:
- In einem Javaprogramm bedeuten die Worte `true` und `false` immer boolesche Werte. Der Datentyp `boolean` wurde nach George Boole benannt, einem Mathematiker des 19. Jahrhunderts, der entdeckte wie viele Dinge man mit true/false-Werten (auch bekannt als Bit) tun kann.

3.6. Die Klasse Strings

Unter **Strings** versteht man vereinfacht ausgedrückt einen Datentyp, der Texte darstellt. Genau genommen ist ein String eine Klasse, also ein Bauplan für Objekte mit gewissenen Attributen und Methoden (genaueres über Klassen und Objekte später).

Hinweis zu Java:
- Bei Java erzeugt man ein Stringobjekt in der Form: *String farbbezeichnung = „rot“;*

4. Methoden

4.1. Einführung

Bereits in 2.2 wurde auf die Möglichkeit, **eigene Anweisungen (= Methoden)** schreiben zu können, hingewiesen.

Warum sollte man Methoden verwenden?

1. Um den Programmcode übersichtlich zu halten.
2. Damit ein und der selbe Programmcode nicht mehrfach geschrieben werden muss.

Beispiele:
zu 1: siehe Struktogramme, Kap. 2.6.

zu 2: Betrachte die ähnlichen Aufgabensituationen:

A) Alle roten Gummibärchen aus dem Päckchen nehmen.

oeffnePaeckchen();
nehmeRoteGummibaerchenAusPaeckchen();

B) Alle gelben Gummibärchen aus dem Päckchen nehmen.

oeffnePaeckchen();
nehmeGelbeGummibaerchenAusPaeckchen()

Die beiden Methoden „nehmeRoteGummibaerchenAusPaeckchen()“ und „nehmeGelbeGummibaerchen AusPaeckchen()“ sind praktisch identisch. Der Unterschied liegt lediglich in der Farbe, die abgefragt wird.

Für solche Fälle besteht die Möglichkeit, Methoden zu verwenden, denen **beim Methodenaufruf ein oder mehrere Parameter (= Daten) übergeben** werden. Der Parameter ist hier die Farbe des Gummibärchens. Innerhalb der Methode kann dann an Hand dieses Parameters entschieden werden, was getan werden soll.

Damit genügt es, eine einzige Methode an Stelle zweier Methoden zu schreiben.

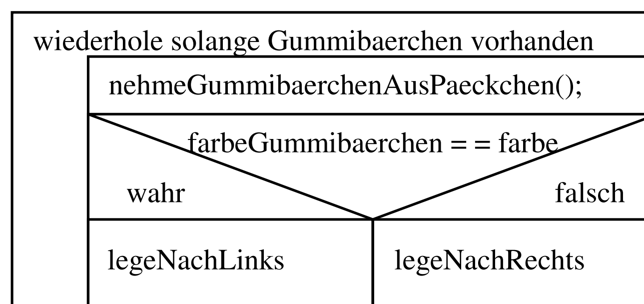
Das jeweilige *Hauptprogramm* sieht dann so aus:

oeffnePaeckchen();
nehmeGummibaerchenAusPaeckchen(rot)

bzw.

oeffnePaeckchen();
nehmeGummibaerchenAusPaeckchen(gelb)

Und die Methode *nehmeGummibaerchenAusPaeckchen* so:



4.2. Der Methodenaufruf

Beim **Methodenaufruf** stehen in den runden Klammer (...) die Parameter (= Daten), die an die aufgerufene Methode übergeben werden sollen. Wird mehr als ein Parameter übergeben, so spricht man von einer **Parameterliste**.

Beispiele: *nehmeGummibaerchenAusPaeckchen(rot);*
zeichneRechteck (10, 20, 30, 40);

Die übergebenen Daten müssen vom Datentyp sein, den die Methode erwartet. Ebenso muss die Reihenfolge der Parameter genau eingehalten werden, wiederum so, wie es die Methode erwartet.

Im zweiten Beispiel könnten die Parameter jeweils vom Datentyp `int` sein und den x-Wert, den y-Wert, die Breite und die Höhe des Rechtecks bedeuten.

4.3. Methodendefinition

Methodendefinitionen sehen wie folgt aus:

```
returntyp methodenname( parameterListe )
{
 // Java Anweisungen
 return returnwert;
}
```

Der *returntyp* ist der Typ des Werts, den die Methode an den Aufrufer der Methode zurückgibt. Methoden können Werte zurückliefern. Die **`return`**-Anweisung wird von der Methode verwendet, um an den aufrufenden Programmteil einen Wert zurückzugeben.

Der Rückgabetyp **`void`** wird verwendet, wenn eine Methode etwas tut, aber keinen Wert an den Aufrufer zurückgeben soll. Die `return`-Anweisung kann dann weggelassen werden; da die Methode automatisch zum Aufrufer zurückkehren wird, nachdem sie ausgeführt wurde.

Beispiel 1: Alle roten Gummibärchen in einem Päckchen zählen.
Hier wird das Ergebnis der Zählung an den Aufrufer zurück gegeben.

Die Methodendefinition sieht so aus:

```
int zaehleGummibaerchen(Farbe farbe)
{
    int anzahl = 0;
    //Anweisungen zum Zählen
    return anzahl;
}
```

Beispiel 2: Alle roten Gummibärchen aus dem Päckchen nehmen.
Hier wird kein Ergebnis zurück gegeben.

Die Methodendefinition sieht so aus:

```
void nehmeGummibaerchenAusPaeckchen(Farbe farbe)
{
    //Anweisungen
}
```

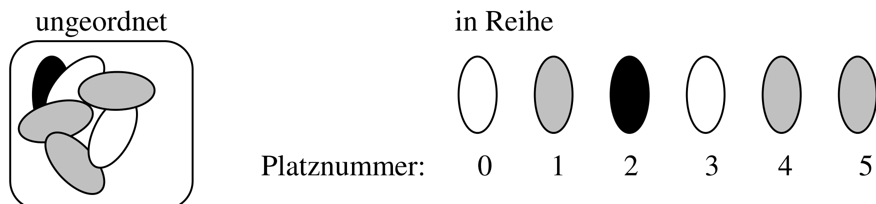
5. Arrays

5.1. Einleitung

Worin werden eigentlich die Gummibärchen aufbewahrt?

Die Gummibärchen befinden sich normalerweise in einem Päckchen, in dem sie völlig ungeordnet liegen. Zum Verwalten der Gummibärchen ist diese Unordnung ungeeignet.

Deshalb liegt es nahe, die Gummibärchen z. B. in eine Reihe zu legen. Nun kann man der Reihe nach die Plätze durchnummerieren und sich merken, welches Gummibärchen an welchem Platz liegt. Auf diese Weise können die Gummibärchen systematisch abgearbeitet und sie somit verwaltet werden.



Hinweis: Die Zählung in der Informatik beginnt i.d.R. bei 0.

5.2. Arrays

Um in Programmen große Mengen von Daten bewältigen zu können, müssen diese organisiert sein und systematisch abgearbeitet werden können. Dafür verwendet man fast immer **Arrays**.

Ein **Array** ist ein Objekt, das verwendet wird, um Listen von Werten zu speichern. Es wird aus einem zusammenhängenden Speicherblock gebildet, der in eine Anzahl von "Schlitzen" (*Slots*) unterteilt ist. Jeder Slot enthält einen Wert und alle Werte sind von dem gleichen Typ. In dem rechts stehendem Beispiel-Array enthält jeder Slot einen `int`.

	data
0	23
1	38
2	14
3	-3
4	0
5	14
6	9
7	103
8	0
9	-56

Der Name dieses Arrays ist `data`. Die Slots sind indexiert von 0 bis 9. Auf jeden Slot kann über seinen **Index** zugegriffen werden. Zum Beispiel, `data[0]` ist der Slot mit dem Index 0 (welcher den Wert 23 enthält). `data[5]` ist der Slot mit dem Index 5 (welcher den Wert 14 enthält).

Beachte:
- Die Slots sind sequentiell nummeriert und beginnen mit 0.
- Wenn es N Slots in einem Array gibt, sind die Indizes 0 bis N - 1.
- Jeder Slot eines Arrays enthält einen Wert des gleichen Typs, z. B. ein Array von `int`, ein Array von `double` usw.

5.3. Arrays verwenden

Um den Wert eines Slots zu ändern, schreibt man z. B.

data[3] = 99;

Der ursprünglich Wert vom Slot 3 ist damit überschrieben (und verloren).

Beispiel: Was bewirken diese Zeilen?
int index = 4;
data[index] = 15;

Der Wert eines Slots kann in eine Variable des gleichen Datentyps geschrieben werden.

Beispiel: *int wert = data[3];*

5.4. Arrays sind Objekte

Arrays sind Objekte und müssen deshalb deklariert werden. **Deklarationen von Arrays** sehen folgendermaßen aus:

typ[] arrayName = ***new*** *typ[laenge];*

Diese Anweisung macht zwei Dinge:
(1) Sie teilt dem Compiler mit, dass *arrayName* auf ein Array von *typ* verweisen wird.
(2) Sie *konstruiert ein Arrayobjekt*, das *laenge* Anzahl von Slots enthält.

Beispiel: *int[] data =* ***new*** *int[10];*
Diese Anweisung erzeugt ein Array `data` vom Datentyp `int` und stellt eine 0 in jeden Slot.

Ein Array kann auch in einer Anweisung deklariert, konstruiert und initialisiert werden:

int[] data = {23, 38, 14, -3, 0, 14, 9, 103, 0, -56 };

Diese Anweisung deklariert ein Array von `int`, das `data` genannt wird. Dann konstruiert sie ein `int`-Array mit 10 Slots (indexiert von 0 bis 9). Schließlich stellt sie die genannten Werte in die Slots. Der erste Wert in der **Initialisierungsliste** entspricht dem Index 0, der zweite Wert entspricht dem Index 1 und so weiter. (In diesem Beispiel bekommt `data[0]` die 23.)

Der Compiler wird die Werte in der Initialisierungsliste zählen und dementsprechend viele Slots bilden. Sobald ein Array konstruiert wurde, ändert sich die Anzahl der Slots nicht mehr, d.h. ein nachträgliches Ändern der Anzahls der Slots ist nicht möglich. Initialisierungslisten werden gewöhnlich nur für kleine Arrays verwendet.

5.5. Länge eines Array

Die Länge eines Array lässt sich mit `length` abfragen.

int laenge = arrayname.length;

Damit lässt sich nun bequem eine Schleife konstruieren, die alle Elemente des Arrays durchgeht:

```
int wert;
for ( int index= 0 ; index < array.length; index++ )
{
    wert = array[index];
    //Zum Beispiel jeden Wert ausgeben
}
```

Eine Codefragment, ähnlich des oberen, ist in Programmen *sehr* gebräuchlich.

Aufgaben: Schreibe einen Algorithmus,

1. der den größten (kleinsten) Wert des Arrays `data` ermittelt.
2. der alle Zahlen des Arrays `data` aufsummiert.
3. den Durchschnitt aller Zahlen des Arrays `data` ermittelt.

5.6. Zweidimensionale Arrays

Daten kommen häufig in zweidimensionaler Form vor. Zum Beispiel besteht ein Kurs Schülern; zu jedem Schüler gehört eine Liste von Noten. Man braucht somit einen Index für die Schüler und einen Index für die Noten.

Eine kompaktere Schreibweise verwendet den **Zeilen-** und **Spaltenindex** (*zeile, spalte)*), um eine Zelle zu bestimmen:

`notentabelle[zeile][spalte]`

Beispiele: notentabelle[0][1] hat den Wert 10;
notentabelle[2][0] hat den Wert 15;
notentabelle[2][3] existiert nicht.

	Note		
Schüler	0	1	2
0	12	10	8
1	4	6	3
2	15	11	13
3	8	8	7

Zweidimensionale Arrays sind Objekte. Eine Variable wie `notentabelle` ist eine Referenz auf ein 2D-Arrayobjekt.
Die Deklaration

int[][] notentabelle = new int[4][3] ;

erzeugt ein Arrayobjekt mit dem Arraynamen `notentabelle` mit 4 Zeilen und 3 Spalten. Alle Elemente des Arrays werden mit 0 initialisiert.

Die Deklaration

int[][]notentabelle = { {0,0,0}, {0,0,0}, {0,0,0}}, {0,0,0} };

macht genau das gleiche wie die vorherige Deklaration (und wird normalerweise nicht verwendet).

Die Deklaration

int[][]notentabelle = { {12,10,8}, {4,6,3}, {15,11,13}}, {8,8,7} };

erzeugt ein Array mit *den gleichen Dimensionen* (die gleiche Anzahl von Zeilen und Spalten) wie das vorherige Array und initialisiert die Elemente mit bestimmten Werten (vgl. Tabelle).

6. `Vector`-Klasse

6.1. Einleitung

In einem Programm werden sehr häufig Daten manipuliert, die in einer Liste aufbewahrt werden, z. B. in Arrays. Arrays sind ein wesentliches Merkmal von Java und den meisten Programmiersprachen.

Der entscheidende Nachteil von Arrays ist, dass sich ihre Länge nachträglich nicht mehr verändern lässt.

Deshalb gibt es die Klasse `Vector`, die fast wie ein Array funktioniert, aber über zusätzliche Methoden und Merkmale verfügt. Wie bei einem Array enthält ein `Vector` Elemente, auf die über einen ganzzahligen Index zugegriffen wird. Aber im Unterschied zu einem Array, kann die Größe eines `Vector`-Objekts bei Bedarf wachsen. Man kann beliebig Elemente hinzufügen, unabhängig von seiner ursprünglichen Größe. Die Größe eines `Vector`-Objekts wird automatisch wachsen, und es wird keine Information verloren gehen.

Die Elemente eines `Vector`-Objekts müssen *Objektreferenzen* sein, also z. B. `Strings`, jedoch keine primitiven Daten wie `int` oder `double`.

6.2. Konstruktoren für `Vector`-Objekte

Um eine Referenzvariable vom Typ `Vector` zu deklarieren, macht man folgendes:

```
Vector meinVector = new Vector();         //1. Fall
Vector meinVector = new Vector(15);       //2. Fall
Vector meinVector = new Vector(15,5);     //3. Fall
```

Hinweise:

- Es wird nicht festgelegt, welcher Typ von Objekt gespeichert werden soll.
- Im 1. Fall wählt das Java System die Anfangskapazität; im 2. Fall wird die Anfangskapazität auf 15 festgelegt; im 3. Fall wird die Anfangskapazität auf 15 festgelegt und die *Inkrementgröße* auf 5, d.h. bei Bedarf werden jeweils 5 neue Slots auf einmal hinzugefügt.

6.3. Methoden eines `Vector`-Objekts

Die Klasse `vector` stellt verschiedene Methoden zur Verfügung mit der die Liste manipuliert werden kann.

Methodenname	Beschreibung	Beispiel
`size()`	liefert die Größe des `Vector`-Objekts; Slots 0 bis `size()-1` enthalten Daten	*int laenge = meinVector.size();*
`addElement(Object)`	fügt ein Element am Ende der Liste an	*meinVector.addElement(„rot“);*
`setElementAt(Object, index)`	setzt ein Element an eine bestimmte Stelle der Liste	*meinVector.setElementAt(„rot“, 4);*
`insertElementAt(Object,index)`	fügt ein Element an einer bestimmten Stelle der Liste ein	*meinVector.insertElementAt(„gelb“, 1);*
`elementAt(index)`	greift auf das Element am Slot `index` zu	*meinVector.elementAt(2);*
`removeElementAt(index)`	entfernt das Element am Slot `index`	*meinVector.removeElementAt(2);*
`firstElement()`	liefert das erste Element der Liste	*meinVector.firstElement();*
`lastElement()`	liefert das letzte Elemente der Liste	*meinVector.lastElement();*
`boolean isEmpty()`	prüft, ob die Liste leer ist; ist vom Datentyp `boolean`; liefert `true`, wenn die Liste keine elemente enthält	*boolean flag = meinVector.isEmpty();*
`int indexOf(Object)`	durchsucht die Liste nach einem Element; liefert den index des ersten gefundenen Elements, sonst -1	*int i = meinVector.indexOf(„rot“);*
`clear()`	löscht alle Elemente	*meinVector.clear();*

6.4. `Enumeration`-Schnittstelle

Ein Programm muss sehr häufig auf die Elemente eines `Vector`-Objekts, eines nach dem anderen, zugreifen. Das kann mit einer Zählschleife getan werden. Aber es kann auch ein Objekt verwendet werden, das die Schnittstelle `Enumeration` implementiert. Um ein `Enumeration`-Objekt für einen `Vector` zu bekommen, verwendet man diese Methode:

elements() // gibt eine Aufzählung (Enumeration) der Komponenten des Vector-Objekts zurück.

Sobald man ein `Enumeration`-Objekt hat, können die Methoden `hasMoreElements()` und `nextElement()` verwendet werden, um sich durch die Elemente zu bewegen.

boolean hasMoreElements() // Gibt true zurück, wenn noch nicht alle Elemente besucht wurden.

Object nextElement() // Gibt das nächste Element der Aufzählung zurück.

Beispiel: Programmausschnitt, in dem alle Elemente des `Vector`-Objekts ausgegeben werden:

```
Enumeration num = meinVector.elements();
while (num.hasMoreElements())
{
   System.outprintln(num.nextElement());  //Ausgabe am Bildschirm
}
```

Aufgabe 1:

Die Farben der Gummibärchen eines Päckchens sollen in einer `vector`-Liste verwaltet werden.

Schreibe für jeden der folgenden Schritte die Programmzeilen auf und gib für jeden neuen Zustand die Belegung der Liste aus:

a) Erzeuge die `vector`-Liste mit dem Namen *farbeGummibaerchen*, der Kapazität 5 und dem Inkrement 2.
b) Füge die Gummibärchenfarben „rot“, „gelb“, „weiß“, „rot“, „rot“, „gelb“ hinzu.
c) Gib die Länge der Liste in der Variablen `len` an.
d) Füge die Farbe „rot“ an der Stelle 2 ein.
e) Setze an der Stelle 4 die Farbe „weiß“ ein.
f) Gib die Farbe am Slot 5 aus.
g) Entferne das 3. Element.
h) Entferne das erste Element mit der Farbe „gelb“.
i) Gib alle Elemente der Liste am Bildschirm aus.

Aufgabe 2:

a) Gib einen Algorithmus an, der feststellt, wie viele verschiedene Farben der Gummibärchen vorkommen und wie viele von jeder Farbe vorhanden sind.
b) Versuche soweit wie möglich, den Algorithmus im Java-Code zu schreiben.