Abschnitt 7: Arrays und Strings

Motivation

- Bisher haben wir einzelne Objekte der Grunddatentypen (Menge der bereitgestellten Sorten) verarbeitet.
- Oft werden aber auch Mengen bzw. Multimengen von Objekten verarbeitet (z.B. bei unseren Wechselgeldalgorithmen, die die Ausgabe als Folge von EUR-Münzen/-Scheinen modelliert hatten).
- Unsere bisherigen Konzepte geben die Verarbeitung von (Multi-)Mengen von Objekten allerdings nicht her.
- In einer Programmiersprache ist üblicherweise eine einfache *Datenstruktur* eingebaut, die es ermöglicht, eine Reihe von Werten (gleichen Typs) zu modellieren.

Motivation

- Datenstrukturen unterscheiden sich typischerweise anhand ihres inneren Aufbaus (Struktur) und den (Grund-)Operationen, die für diese Datenmengen zur Verfügung stehen.
- Typische Datenstrukturen sind:
  - Felder oder Reihungen (Array)
  - Listen (List)
  - Schlangen (Queue)
  - Keller (Stack)
- Die Art der inneren Darstellung (Strukturierung) hat meist erheblichen Einfluss auf die Effizienz der Verarbeitungsalgorithmen.
- Wir werden die Unterschiede später nochmal anreißen, ansonsten sollten diese Gegenstand der Vorlesung „Datenstrukturen und Algorithmen“ sein.

Motivation

- Im imperativen Paradigma wird normalerweise das *Array* (*Reihung*, *Feld*) zur Verfügung gestellt.
- Wir schauen uns Arrays zunächst ganz allgemein (als Konzept) an, um uns dann Arrays in Java zu widmen.
- Eine spezielle Form von Arrays sind *Zeichenketten* (in Java *Strings*), bei denen die Objekte der Menge vom Typ $CHAR$ (bzw. in Java `char`) sind.

Überblick

Reihungen

- Ein Array (Reihung, Feld) ist ein Tupel von Komponentengliedern gleichen Typs, auf die über einen Index direkt zugegriffen werden kann.
- Formal handelt es sich um eine Folge, d.h. ein Array von Objekten einer bestimmten Sorte S ist ein Element aus S^*.
- D.h. eine Reihung mit n Komponenten über einer Sorte S kann als Abbildung von der Indexmenge I_n auf die Menge S interpretiert werden.

Reihungen

- Um einen Datentyp (Sorte) „Array über S“ einzuführen, sodass wir mit ihm „vernünftig“ arbeiten (programmieren) können, sollten wir uns überlegen, welche Grundoperationen (analog zu unseren bisherigen Grunddatentypen) für diese Arrays allgemein gelten sollten.
- Diese Operationen sollten die wesentliche Eigenschaften von Arrays, die sie von anderen Datentypen unterscheiden, widerspiegeln, insbesondere:
  - Die Länge eines Arrays ist fix, d.h. sie können nicht dynamisch wachsen oder schrumpfen (d.h. die Menge I_n verändert sich nicht).
  - Arrays erlauben direkten Zugriff (lesend/schreibend) auf ihr i-tes Element ($i \in I_n$).
- Diese könnten wir dann z.B. in ein entsprechendes Modul schreiben und allgemein zur Verfügung stellen.

Datenstrukturen

- Die typischen Grundoperationen von Arrays sind:
  - Der Zugriff auf die obere Grenze des Indexbereichs (Länge).
  - Der Zugriff auf die i-te Komponente (Projektion).
  - Ein Konstruktions-Operator, der ein Array einer gewissen Länge (d.h. die Indexmenge ist damit festgelegt) erzeugt. Wie besprochen ist die Länge des Arrays dann nicht mehr veränderbar (statisch).
  - Das Verändern des i-ten Elements.
- Das Verhalten dieser Operationen ließe z.B. durch axiomatische Spezifikation definieren.
- Wir geben zumindest die Signaturen an.
- Dazu bezeichnen wir die Menge aller Arrays über der Sorte S mit

  array S

  und nehmen an, dass $I_n \subset \mathbb{N}$[16]

[16] Dazu später mehr!!!

Datenstrukturen

- Dann wären:
  - Der Zugriff auf die obere Grenze des Indexbereichs (Länge):
    UP : **array** $S \to \mathbb{N}$
  - Der Zugriff auf die i-te Komponente (Projektion):
    $PROJ$: **array** $S \times \mathbb{N} \to S$
  - Ein Konstruktions-Operator, der ein Array (mit einem speziellen Objekt aus S) einer gewissen Länge (d.h. die Indexmenge ist damit festgelegt) erzeugt:
    $INIT : \mathbb{N} \times S \to$ **array** S
    mit $INIT(n, a) = (a, a, \ldots, a) \in S^n$
  - Das Verändern des i-ten Elements
    ALT : **array** $S \times \mathbb{N} \times S \to$ **array** S
    mit $ALT(x, i, a) = (x_1, \ldots, x_{n-1}, a, x_{n+1}, \ldots, x_n)$
    für $x = (x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n)$.

Datenstrukturen

- Mit diesen Grundoperationen könnten wir jetzt weitere Algorithmen entwickeln, wie z.B. die Suche eines Objekts $a \in S$ in einem Array $x \in$ **array** S:

procedure $enthalten(x : \textbf{array}\ S, a : S) \to \mathbb{B}$
  output gibt $TRUE$ falls a in x enthalten, sonst $FALSE$
  body
    var $gefunden : \mathbb{B}, i : \mathbb{N};$
    $gefunden \leftarrow FALSE;$
    $i \leftarrow 1;$
    while $\neg gefunden \wedge i \leq UP(x)$ **do**
      if $PROJ(x, i) = a$ **then** $gefunden \leftarrow TRUE;$ **endif**
      $i \leftarrow i + 1;$
    enddo
    return $gefunden;$

Überblick

Achtung

- Die Umsetzung in Java erfolgt 1-zu-1, allerdings ist zu beachten:
- Wie bei den meisten Programmiersprachen beginnt die Indexmenge in Java bei 0, d.h. für eine n-elementige Reihung gilt die Indexmenge $I_n = \{0, \ldots, n-1\}$.
  (Die Operation UP würde für ein Array der Länge n also $n-1$ zurückgeben!)
- In Java sind Arrays *semidynamisch*, d.h. ihre Größe kann zur Laufzeit (=dynamisch) festgesetzt werden, danach aber nicht mehr geändert werden (=statisch), d.h. dynamisches Wachstum von Arrays ist auch in Java nicht möglich!!!

Reihungen

- Beispiel: Eine **char**-Reihung `gruss` der Länge 13:

`gruss`:	'H'	'e'	'l'	'l'	'o'	','	' '	'W'	'o'	'r'	'l'	'd'	'!'
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12

ist formal eine Abbildung

$$\texttt{gruss} : \{0, 1, \ldots, 12\} \to \textbf{char}$$

$$i \mapsto \begin{cases} \text{'H'} & \text{falls} \quad i = 0 \\ \text{'e'} & \text{falls} \quad i = 1 \\ \vdots & \\ \text{'!'} & \text{falls} \quad i = 12 \end{cases}$$

Reihungen

- Der Typ eines Arrays, das den Typ `<type>` enthält, wird in Java als `<type>[]` notiert (statt **array**`<type>`).
- Beispiel: ein Array mit Objekten vom Typ **`int`** ist vom Typ **`int`**`[]`.
- Variablen und Konstanten vom Typ `<type>[]` können wie gewohnt vereinbart werden:

```
<type>[] variablenName;
```

(Konstanten wie immer mit dem Zusatz **`final`**)

Reihungen

- Vorsicht: Arrays sind sog. *Referenztypen*, die wir schon mal angesprochen haben.
- Auf dem Zettel einer Variable steht nicht der Wert (also das Array direkt) sondern eine Referenz (Arrays werden nämlich, da sie i.d.R. unterschiedlich groß sind, auf einem besonderen Bereich im Speicher, der Halde (Heap), verwaltet – dazu aber später noch mehr).
- Dies führt zu einem call-by-reference-Effekt (auch dazu später mehr)!!!

Reihungen

Erzeugung (Konstruktions-Operator):

- Es gibt nicht *den* Konstruktions-Operator *INIT*.
- Wie alle Referenztypen werden Arrays bei ihrer Initialisierung *erzeugt*.
- Die Initialisierung (Erzeugung) eines Arrays kann dabei auf verschiedene Arten erfolgen.
- Die einfachste ist, alle Elemente der Reihe nach in geschweiften Klammern `{}` aufzuzählen:

```
<type>[] variablenName = {element1, element2, ...}
```

wobei die einzelnen `element1`, `element2`, etc. Literale (Werte) oder Variablen vom Typ `<type>` sind.

Reihungen

Zugriff auf das i-te Element (Projektion):

- Die Operation *PROJ* für den Zugriff auf das `i`-te Element eines Arrays `a` notiert man in Java durch den Ausdruck `a[i]`.
- Dabei ist `i` vom Typ **`int`** (bzw. vom Compiler implizit in **`int`** cast-bar)
- Der Wert des Ausdrucks `variablenName[i]` ist der Wert des Arrays `variablenName` an der Stelle `i`.
- Der Typ des Ausdrucks `variablenName[i]` ist der Typ, über dem das Array `variablenName` gebildet wurde.
- Beispiel:

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

  Der Ausdruck `a[1]` hat den Wert `2` und den Typ **`int`**.

Reihungen

Verändern des i-ten Elements:

- Die Operation ALT gibt es nicht explizit.
- Vielmehr ist die Projektion auf das `i`-te Element eines Arrays `a` (`a[i]`) auch für den schreibenden Zugriff gedacht.
- Man kann `a[i]` also nicht nur als Ausdruck verstehen, sondern diesem selbst auch einen Wert zuweisen (es handelt sich nämlich letztlich um eine Variable).
- Beispiel:

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

`a[1] = 6;` verändert das Array `a` zu `{ 1 , 6 , 3 , 4 }`.

Reihungen

Zugriff auf die Länge:

- Die Operation UP ist in Java etwas anders umgesetzt: es gibt eine Konstante mit Namen `length`, die anzeigt, wie viele Elemente im Array enthalten sind (Vorsicht: das sind nicht $n-1$ sondern n!!!), d.h. es wird nicht die obere Grenze der Indexmenge sondern die Länge des Arrays gespeichert.
- Der Typ dieser Konstante ist **`int`**.
- Der Name der Konstanten ist zusammengesetzt aus dem Namen des Arrays und dem Namen `length`, d.h. die Länge eines Arrays `a` ist der Wert des Ausdrucks `a.length`.
- Beispiel:

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

Der Ausdruck `a.length` hat den Wert `4` und den Typ **`int`**.

Reihungen

- Beispiel

```
char a = 'a';
char b = 'b';
char c = 'c';
char[] abc = {a, b, c};
System.out.print(abc[0]); // gibt den Character 'a' aus,
                          // den Wert des Array-Feldes
                          // mit Index 0. Allgemein: array[i]
                          // ist Zugriff auf das i-te Element
System.out.print(abc.length);      // gibt 3 aus
int[] zahlen = {1, 3, 5, 7, 9};
System.out.print(zahlen[3]);       // gibt die Zahl 7 aus
System.out.print(zahlen.length);   // gibt 5 aus
```

Reihungen

- Oft legt man ein Array an, bevor man die einzelnen Elemente kennt.
- Die Länge muss man dabei angeben: `char[] abc = new char[3];`
- Das Schlüsselwort **`new`** ist hier verlangt (es bedeutet in diesem Fall, dass eine neue (leere) Referenz angelegt wird — es ist also so etwas wie ein Konstruktions-Operator).
- Dann kann man das Array im weiteren Programmverlauf (durch Verändern der einzelnen Elemente) füllen:

```
abc[0] = 'a';
abc[1] = 'b';
abc[2] = 'c';
```

Reihungen

- Dass Arrays in Java *semidynamisch* sind, bedeutet: Es ist möglich, die Länge erst zur Laufzeit festzulegen.
- Beispiel

```
// x ist eine Variable vom Typ int
// deren Wert bei der Ausfuehrung
// feststeht, aber ggfs. noch nicht beim
// Uebersetzen des Programmcodes (Kompilieren)
// (z.B. weil x ein Eingabeparameter ist)
char[] abc = new char[x];
```

Reihungen

- Was passiert, wenn man ein Array anlegt

```
int[] zahlen = new int[10];
```

  aber nicht füllt? Ist das Array dann leer?
- Nein: es gibt in Java keine leeren Arrays.
- Ein Array wird immer mit den Standardwerten des jeweiligen Typs initialisiert.
- Das spätere Belegen einzelner Array-Zellen ist also immer eine Änderung eines Wertes (durch eine Wertzuweisung):

```
int[] zahlen = new int[10];
System.out.print(zahlen[3]);    // gibt 0 aus
zahlen[3] = 4;
System.out.print(zahlen[3]);    // gibt 4 aus
```

Reihungen

- Das legt den Schluss nahe, dass die einzelnen Elemente des Arrays wiederum Variablen (Zettel) sind, die ich beschreiben und ablesen kann.
- So in etwa kann man sich das auch tatsächlich vorstellen.
- In

```
int[] zahlen = { 1 , 2 , 3 }
```

ist die Variable `zahlen` ein (radierbarer) Zettel, auf dem ein Haufen weitere Zettel, nämlich die der einzelnen Elemente `zahlen[i]`, liegen.
- Genauer gesagt enthält `zahlen` die Adresse der Zettel, der einzelnen Elemente `zahlen[i]` und einen zusätzlichen (nicht-radierbaren) Zettel mit der Länge des Arrays (`zahlen.length`).
- Die einzelnen (radierbaren) Zettel `zahlen[1]`, enthalten die konkreten Werte.

Reihungen

- Das ganze Ausmaß dieser Zettelwirtschaft nochmal bildlich an der Tafel (Oh Gott!!!)

Reihungen

- Es gelten die üblichen Eigenschaften für Zettel (Variablen/Konstanten), also z.B. ist auch so etwas erlaubt:

```
char[] abc = { 'a',  'b',  'c'};
char[] de = { 'e',  'e'};
abc = de; // d.h. de wird der Wert von abc zugewiesen
```

- Und jetzt eben Achtung: statt einem konkreten Wert wird hier eine Referenz zugewiesen!

Reihungen

- Auch Array-Variablen kann man als Konstanten deklarieren.
- Dann kann man der Variablen keinen neuen Wert zuweisen:

```
final char[] ABC = { 'a',  'b',  'c'};
final char[] DE = { 'd',  'e'};
ABC = DE; // ungueltige Anweisung: Compilerfehler
```

- Aber Achtung: einzelne Array-Komponenten sind normale Variablen (Zettel), man kann ihnen also einen neuen Wert zuweisen:

```
ABC[0] = 'd';                          // erlaubt
ABC[1] = 'e';                          // erlaubt
System.out.print(ABC.length);   // gibt 3 aus
System.out.print(ABC[0]);       // gibt 'd' aus
System.out.print(ABC[1]);       // gibt 'e' aus
System.out.print(ABC[2]);       // gibt 'c' aus
```

Reihungen

- Hä? Wie passt denn das mit unserer Intuition der Zettel(-wirtschaft) zusammen?
- Sehr gut sogar:
- Wie gesagt, eine Variable vom Typ `<type>[]` ist ein Zettel, auf dem die Referenz zu weiteren Zetteln (die Elemente) steht.
- Konstanten sind nicht radierbare Zettel, d.h. der Zettel `<type>[]` auf dem die Referenz zu den anderen Zettel steht, ist dann nicht radierbar; die Zettel, die referenziert werden, aber natürlich schon.
- Natürlich . . .
- Auch hier lohnt sich nochmal die Visualisierung der Zettel.

Reihungen

- Mit Arrays kann man also wunderbar programmieren:
- Beispiel: der Algorithmus *enthalten*

```
public static boolean seqSearch(int[] x, int a) {
      boolean gefunden = false;
      int i = 0;
      while(!gefunden && i < a.length) {
            if(x[i] == a) {
                  gefunden = true;
            }
            i++;
      }
      return gefunden;
}
```

Versuchen Sie es selbst mal mit einer **`for`**-Schleife!

Reihungen

- Beispiel: Summe der Elemente in einem **`int`**-Array

```
public static int summeElemente(int[] x) {
      int erg = 0;
      for(int i = 0; i < x.length; i++) {
          erg = erg + x[i];
      }
      return erg;
}
```

Reihungen

- Beispiel: Wechselgeldalgorithmus

  Zur Erinnerung:

  - Eingabe eines Rechnungsbetrags $1 \leq r \leq 100$.
  - Gesucht ist das Wechselgeld zu einer Bezahlung von r mit einem 100-EUR-Schein als Menge an 1 EUR, 2 EUR Münzen sowie 5 EUR Scheinen (mit dem Ziel möglichst wenige Münzen/Scheine auszugeben).
  - Als Ergbnis wollten wir eine Folge an 1er, 2er und 5er ausgeben. Das könnten wir jetzt mit einem Array implementieren.

  Dadurch handeln wir uns allerdings ein Problem ein:

  - Arrays sind ja semi-dynamisch, d.h. wir müssen in Abhängigkeit von r zunächst bestimmen, wieviel Scheine/Münzen auszugeben sind (d.h. wie lang das Ergebnis-Array wird).

Reihungen

- Beispiel: Wechselgeldalgorithmus (cont.)

  Aber die Lösung hatten wir wenigstens schon:

  - Der ganzzahlige Quotient $q_1 = DIV(100 - r, 5)$ ist die Anzahl der 5-EUR-Scheine im Wechselgeld.
  - Der Rest $r_1 = MOD(100 - r, 5)$ ist der noch zu verarbeitende Wechselbetrag. Offensichtlich gilt $r_1 < 5$.
  - r_1 muss nun auf 1 und 2 aufgeteilt werden, d.h. analog bilden wir $q_2 = DIV(r_1, 2)$ und $r_2 = MOD(r_1, 2)$.
  - q_2 bestimmt die Anzahl der 2-EUR-Münzen und r_2 die Anzahl der 1-EUR-Münzen.

  D.h. die Länge des Ergebnis-Arrays ist $q_1 + q_2 + r_2$.

Reihungen

- Beispiel: Wechselgeldalgorithmus (cont.)

  Fragt sich nur noch, wie wir das Ergebnis-Array zu befüllen haben:

  - q_1 ist die Anzahl der 5-EUR-Scheine, d.h. die Stellen $0, \ldots, (q_1 - 1)$ sind mit der Zahl 5 zu belegen.
  - q_2 ist die Anzahl der 2-EUR-Münzen, d.h. die Stellen $q_1, \ldots, (q_1 + q_2) - 1$ sind mit der Zahl 2 zu belegen.
  - r_2 ist die Anzahl der 1-EUR-Münzen, d.h. die Stellen $(q_1 + q_2), \ldots, (q_1 + q_2 + r_2) - 1$ sind mit der Zahl 1 zu belegen.

Reihungen

- Beispiel: Wechselgeldalgorithmus (cont.)

```
public static int[] wechselGeld(int r) {
    int q1 = (100 - r) / 5;
    int q2 = ((100 - r) % 5) / 2;
    int r2 = ((100 - r) % 5) % 2;
    int[] erg = new int[q1 + q2 + r2];

    for(int i=0; i<q1; i++) {
        erg[i] = 5;
    }
    for(int i=q1; i<q1+q2; i++) {
        erg[i] = 2;
    }
    for(int i=q1+q2; i<erg.length; i++) {
        erg[i] = 1;
    }
    return erg;
}
```

Reihungen

- Eine Variante: nehmen wir an, wir schreiben diesen Algorithmus für einen Wechselgeldautomaten.
- Der Automat will nur wissen, wie viele 1er, 2er und/oder 5er er ausgeben soll (das tatsächliche Ausgeben erfolgt auf Basis dieser Informationen).
- Dazu verändern wir die Darstellung des Ergebnisses: wir geben ein Array der Länge 3 aus; an die Stelle 0 schreiben wir die Anzahl der 5er, and die Stelle 1 die Anzahl der 2er und an die Stelle 2 die Anzahl der 1er:

```
public static int[] wechselGeldVar(int r) {
      int q1 = (100 - r) / 5;
      int q2 = ((100 - r) % 5) / 2;
      int r2 = ((100 - r) % 5) % 2;
      int[] erg = new int[3];
      int[0] = q1;
      int[1] = q2;
      int[2] = r2;
      return erg;
}
```

Reihungen

- Da auch Arrays einen bestimmten Typ haben z.B. `gruss : char[]` kann man auch Reihungen von Reihungen bilden.
- Diese Gebilde heißen auch *mehrdimensionale* Arrays.
- Mit einem Array von Arrays lassen sich z.B. Matrizen modellieren.

```
int[] m0 = {1, 2, 3};
int[] m1 = {4, 5, 6};
int[][] m = {m0, m1};
```

- man ist dabei nicht auf „rechteckige“ Arrays beschränkt:

```
int[] m0 = {0};
int[] m1 = {1, 2};
int[] m3 = {3, 4, 5};
int[][] m = {m0, m1, m2};
```

Überblick

Zeichenketten

- Wir hatten bereits diskutiert, dass Zeichenketten nicht nur zur Darstellung von Daten benutzt werden können; sie können selbst Gegenstand der Datenverarbeitung sein.
- *Zeichenketten* (*Strings*) sind letztlich Arrays über dem Typ **`char`** (Folgen über der Menge der druckbaren Zeichen).
- Java stellt einen eigenen Typ `String` für Zeichenketten zur Verfügung, d.h. es gibt eine eigene Sorte (mit Operationen) für Zeichenketten in Java, wir können mit diesem Typ ganz normal „arbeiten“.
- Der Typ `String` ist kein primitiver Typ, sondern wieder ein Referenztyp, genauer eine Klasse von Objekten, ein sog. *Objekttyp*.

Zeichenketten

- Betrachten wir folgendes Beispiel:

```
public class HelloWorld {
     public static final String GRUSS = "Hello World";
     public static void main(String[] args) {
          System.out.println(GRUSS);
     }
}
```

- In der Deklaration und Initialisierung

  `public static final String GRUSS = "Hello, World!";`

  entspricht der Ausdruck `"Hello, World!"` einer speziellen Schreibweise für ein konstantes Array `char[13]`, das in einen Typ `String` *gekapselt* ist.
- Achtung: Die Komponenten dieses Arrays können nicht mehr (durch Neuzuweisung) geändert werden.

Zeichenketten

- Obwohl `String` (wie Arrays) kein primitiver Typ ist, wird dieser Typ in Java sehr ähnlich wie ein primitiver Typ behandelt:
- Z.B. können Werte dieses Typs (wie bei primitiven Typen) durch Literale gebildet werden (in `" "` eingeschlossen).
- Beispiele für Literale der Sorte `String` in Java:
  - `"Hello World!"`
  - `"Kroeger"`
  - `"Guten Morgen"`
  - `"42"`
- Literale und komplexere Ausdrücke vom Typ `String` können durch den (abermals überladenen!) Operator `+` konkateniert werden:
  - `"Guten Morgen, "+"Kroeger"` ergibt die Zeichenkette `"Guten Morgen, Kroeger"`

Zeichenketten

- `String` ist in der Tat ein klassisches Modul, dass verschiedene Operationen (statische Methoden) über dieser (und weiterer) Sorte(n) bereitstellt.
- Ein paar davon schauen wir uns im Folgenden an (einige später), ansonsten sei wieder auf die Dokumentation der API verwiesen:
- Typcast, um aus primitiven Typen Strings zu erzeugen

  ```
  static String valueOf(<type> input)
  ```

  wobei u.a.

  `<type>` $\in \{$ **`boolean`**, **`char`**, **`char[]`**, **`int`**, **`long`**, **`float`**, **`double`** $\}$.

  Bei der Konkatenation eines Strings mit einem Literal eines primitiven Typs (z.B. `"Note: "+1.0`) werden diese Methoden (hier: **`static`** `String valueOf(`**`double`**` d)`) implizit verwendet.

Zeichenketten

- Länge der Zeichenkette durch die Methode `int length()` (Achtung, anders als bei Arrays ist das tatsächlich eine Methode!!!))
- Die Methode `char charAt(int index)` liefert das Zeichen an der gegebenen Stelle des Strings
  Dabei hat das erste Element den Index 0 und das letzte Element den Index `length() - 1` (wie beim unterliegenden Array).
- Beispiele:
  - der Ausdruck `"Hello, World!".length()` hat den Wert: `13`
  - der Ausdruck `"Hello, World!".charAt(10)` hat den Wert: `'l'`

Zeichenketten

- Nun verstehen Sie auch endlich offiziell den Parameter der `main`-Methode eines Java Programms.
- In Programmbeispielen haben wir bereits die `main`-Methode gesehen, die das selbständige Ausführen eines Programmes ermöglicht.
- Der Aufruf `java KlassenName` führt die `main`-Methode der Klasse `KlassenName` aus (bzw. gibt eine Fehlermeldung falls, diese Methode dort nicht existiert).
- Die `main`-Methode hat immer einen Parameter, ein `String`-Array, meist als Eingabe-Variablen `args`.
- Dies ermöglicht das Verarbeiten von Argumenten, die über die Kommandozeile übergeben werden.

Zeichenketten

- Der Aufruf

  `java KlassenName <Eingabe1> <Eingabe2> ... <Eingabe_n>`

  füllt das `String`-Array (Annahme, der Eingabeparameter heißt `args`) automatisch mit den Eingaben

```
args[0] = <Eingabe1>
args[1] = <Eingabe2>
...
args[n-1] = <Eingabe_n>
```

Zeichenketten

- Beispiel für einen Zugriff der main-Methode auf das Parameterarray

```
public class Gruss {

      public static void gruessen(String gruss) {
            System.out.println("Der Gruss ist: "+gruss);
      }

      public static void main(String[] args) {
           gruessen(args[0]);
      }
}
```

Dadurch ist eine vielfältigere Verwendung möglich:

- `java Gruss "Hello, World!"`
- `java Gruss "Hallo, Welt!"`
- `java Gruss "Servus!"`