Programmierung in Java (Grundwissen)

Website:	Ludwig-Georgs-Gymnasium	Gedruckt von:	Sarah Geppert
Kurs:	Informatik Q 25/27	Datum:	Samstag, 23. August 2025, 14:24
Buch:	Programmierung in Java (Grundwissen)		

Inhaltsverzeichnis

1. Vorwort

In der Informatik geht es nicht primär um das Erlernen einer konkreten Programmiersprache, sondern um die Systematik ein Problem lösen zu können.

Die verschiedenen Programmiersprachen dienen nur als Hilfsmittel, um die Probleme konkret lösen zu können. Je nach Problemstellung können verschiedene Programmiersprachen in Frage kommen. Jede hat ihre eigenen Vorzüge.

In diesem Buch geht es um verschiedene grundlegende Konzepte, die in den meisten Programmiersprachen umgesetzt werden können. Hat man das Konzept verstanden, ist eine Übertragung in die unterschiedlichen Sprachen relativ leicht möglich.

Bei konkreten Beispielen wird auf die Programmiersprachen Python und Java eingegangen, wobei grundlegende Kenntnisse in Python vorausgesetzt werden und deshalb meist nicht ausführlich erläutert werden.

Und eine kleine "Warnung" vorweg: In der Informatik gibt es im Gegensatz zur Mathematik für gleiche Dinge unterschiedliche Begriffe, welche synonym verwendet werden können. Im nachfolgenden wird versucht, die gängisten Begriffe aufzuführen.

2. Algorithmen

Ein Algorithmus ist eine endliche, eindeutige Folge von Anweisungen, die zur Lösung eines Problems oder zur Durchführung einer Aufgabe dient. Algorithmen sind die Basis jedes Computerprogramms. Sie können in verschiedenen Formen dargestellt werden, darunter Pseudocode oder in einer spezifischen Programmiersprache. Die wichtigsten Eigenschaften eines Algorithmus sind:

- **Terminiertheit** (Endlichkeit): Ein Algorithmus muss nach einer endlichen Anzahl von Schritten enden.
- **Finitheit** (Ausführbarkeit): Ein Algorithmus wird durch eine endliche Anzahl an Schritten beschrieben.

- **Determinismus** (Eindeutigkeit): Jeder Schritt des Algorithmus muss klar und eindeutig definiert sein.
- **Determiniertheit** (Vorhersehbarkeit): Ein Algorithmus ist determiniert, wenn er bei jeweils gleichen Voraussetzunge stehts das gleiche Ergebnis liefert.

3. Programmierparadigmen (fakultativ)

Zuerst einmal muss man zwischen verschiedenen grundlegenden Programmierparadigmen unterscheiden. Sie Unterscheiden sich in ihren Vorgehensweisen und Konzepten.

Definition: Programmiersprachen Paradigmen sind ein Satz von Konzepten oder Denkmustern, die den Stil und die Methodik der Programmierung bestimmen. Sie definieren, wie Aufgaben strukturiert und Probleme gelöst werden. Jedes Paradigma betont bestimmte Aspekte des Programmierens und definiert Regeln und Prinzipien, die bei der Entwicklung von Software eingehalten werden sollten.

Die Wahl des Paradigmas ist stark abhängig von dem Problem und den Anforderungen an den Lösungsweg.

I. Imperatives Paradigma

Das imperative Paradigma ist eines der ältesten und am weitesten verbreiteten Paradigmen in der Programmierung. Es basiert auf dem Konzept der Zustandsveränderung durch Anweisungen. Programme werden als eine Abfolge von Befehlen geschrieben, die den Zustand des Systems Schritt für Schritt verändern.

- **Beispiele:** C, Pascal, Fortran, COBOL
- **Kernprinzip:** Sequentielle Ausführung von Anweisungen zur Manipulation des Programmzustands.
- **Grenzfall: Prozedurale Programmierung** ist eine Unterkategorie des imperativen Paradigmas, bei der Programme in Prozeduren oder Funktionen modularisiert werden. Diese Prozeduren sind wiederverwendbare Codeblöcke, die bestimmte Aufgaben ausführen.

II. Deklaratives Paradigma

Das deklarative Paradigma konzentriert sich darauf, **was** ein Programm tun soll, statt **wie** es dies tun soll. Es beschreibt das Ziel oder das gewünschte Ergebnis, ohne die Schritte zu spezifizieren, die benötigt werden, um dieses Ziel zu erreichen.

a. Funktionales Paradigma

Das funktionale Paradigma ist eine Form des deklarativen Paradigmas, das den Fokus auf mathematische Funktionen legt. Es vermeidet Zustandsänderungen und mutable Daten, was zu einer sauberen, vorhersehbaren Programmierung führt.

- **Beispiele:** Haskell, Lisp, Erlang
- **Kernprinzip:** Funktionen sind erstklassige Objekte und können als Argumente übergeben oder zurückgegeben werden. Programme bestehen aus der Anwendung und Komposition von Funktionen.
- **Grenzfall: Referentielle Transparenz** ist ein zentraler Begriff, bei dem der Funktionsaufruf durch seinen Rückgabewert ersetzt werden kann, ohne das Verhalten des Programms zu verändern.

b. Logisches Paradigma

Das logische Paradigma basiert auf formaler Logik. Programme bestehen aus einer Menge von Aussagen, die beschreiben, was wahr ist. Der Programmierer definiert Regeln und Relationen, und die Ausführung des Programms besteht darin, logische Schlüsse zu ziehen.

- **Beispiele:** Prolog
- **Kernprinzip:** Programme bestehen aus Fakten und Regeln. Die Ausführung ist eine Suche nach Beweisen, die die Anfragen des Benutzers erfüllen.
- **Grenzfall: Constraint-Programmierung** kann als eine Mischung aus logischer und deklarativer Programmierung betrachtet werden, bei der das System unter Berücksichtigung von Einschränkungen eine Lösung findet.

III. Objektorientiertes Paradigma

Das objektorientierte Paradigma modelliert Programme als eine Sammlung von Objekten, die miteinander interagieren. Jedes Objekt ist eine Instanz einer Klasse und enthält sowohl Daten (Attribute) als auch Verhaltensweisen (Methoden).

- **Beispiele:** Java, C++, Python
- **Kernprinzip:** Kapselung und Vererbung. Objekte kommunizieren durch das Senden von Nachrichten (Methodenaufrufen).
- **Grenzfall: Multi-Paradigma-Sprachen** wie Python und Scala unterstützen sowohl objektorientierte als auch funktionale Programmierung,

was es ermöglicht, beide Ansätze innerhalb eines Programms zu kombinieren.

IV. Nebenläufiges und paralleles Paradigma

Nebenläufigkeit und Parallelität beziehen sich auf Programme, die mehrere Berechnungen gleichzeitig durchführen. Dieses Paradigma ist besonders wichtig in modernen Anwendungen, die auf Mehrkernprozessoren laufen und hohe Leistung benötigen.

- **Beispiele:** Erlang (für Nebenläufigkeit), OpenMP (für Parallelität)
- **Kernprinzip:** Unabhängige Prozesse oder Threads, die gleichzeitig ausgeführt werden. Dies kann die Effizienz und Reaktionsfähigkeit eines Programms verbessern.
- **Grenzfall: Asynchrone Programmierung** ist eine spezielle Form der Nebenläufigkeit, bei der Operationen nicht blockierend sind und der Programmfluss nicht auf die Fertigstellung einer Operation warten muss.

V. Event-basiertes Paradigma

Das event-basierte Paradigma wird häufig in der Entwicklung von Benutzerschnittstellen und ereignisgesteuerten Systemen verwendet. Programme reagieren auf Ereignisse, wie Benutzereingaben oder Sensorwerte.

- **Beispiele:** JavaScript (für Web-Entwicklung), Node.js
- **Kernprinzip:** Programme bestehen aus einer Schleife, die auf Ereignisse wartet und entsprechende Reaktionen auslöst.
- **Grenzfall: Reaktive Programmierung** kombiniert Event-basierte und funktionale Programmierung, um komplexe asynchrone Datenflüsse zu handhaben.

VI. Grenzfälle und Hybride

Es gibt viele Sprachen und Ansätze, die mehrere Paradigmen kombinieren. Diese Multi-Paradigma-Sprachen ermöglichen es Programmierern, je nach Problemstellung das geeignetste Paradigma zu wählen oder sogar verschiedene Paradigmen innerhalb eines Projekts zu mischen.

- **Beispiel:** Python unterstützt imperatives, objektorientiertes und funktionales Programmieren. Scala kombiniert objektorientierte und funktionale Konzepte.
- **Kernprinzip:** Die Flexibilität, unterschiedliche Ansätze zu integrieren, kann zu leistungsfähigeren und anpassungsfähigeren Programmen führen.

Jedes Programmierparadigma bietet einzigartige Vorteile und hat spezifische Anwendungsgebiete. Während einige Probleme mit einem bestimmten Paradigma effizienter gelöst werden können, bieten Grenzfälle und hybride Ansätze die Möglichkeit, die Stärken verschiedener Paradigmen zu kombinieren, um flexible und leistungsstarke Programme zu entwickeln. Das Verständnis dieser Paradigmen ist entscheidend, um die geeigneten Werkzeuge und Methoden für die jeweilige Programmieraufgabe auszuwählen.

3.1. Python nur objektorientiert?

Ja, Python ist eine objektorientierte Programmiersprache, allerdings ist sie nicht ausschließlich objektorientiert, sondern unterstützt mehrere Paradigmen, darunter auch funktionale und imperative Programmierung.

Objektorientierte Merkmale von Python

Python ermöglicht die Definition von Klassen, die als Bauplan für Objekte dienen. Diese Klassen können Attribute (Daten) und Methoden (Funktionen) enthalten. Objekte in Python sind Instanzen dieser Klassen und können auf die in den Klassen definierten Methoden und Attribute zugreifen.

Einige zentrale Merkmale der objektorientierten Programmierung (OOP), die Python unterstützt:

1. Daten und Methoden sind in Klassen gebündelt. Attribute und Methoden einer Klasse können öffentlich oder privat sein, was den Zugriff und die Manipulation von Daten kontrolliert.
2. Vererbung: Klassen können von anderen Klassen erben und deren Eigenschaften und Methoden übernehmen. Dies fördert die Wiederverwendbarkeit und Erweiterbarkeit von Code.
3. Polymorphie: Python unterstützt Polymorphie, was bedeutet, dass unterschiedliche Klassen Methoden mit demselben Namen haben können, die jedoch unterschiedliche Implementierungen aufweisen.

4. Dynamische Typisierung: Python ist dynamisch typisiert, was bedeutet, dass der Typ eines Objekts zur Laufzeit bestimmt wird. Dies ist nicht direkt ein Merkmal von OOP, aber es beeinflusst, wie Klassen und Objekte in Python verwendet werden.

Flexibilität von Python

Während Python alle wichtigen Konzepte der objektorientierten Programmierung unterstützt, ist es nicht darauf beschränkt. Python ermöglicht auch:

- Imperative Programmierung: Man kann einfache Sequenzen von Anweisungen schreiben, die direkt den Zustand des Programms ändern, ohne dass Klassen oder Objekte benötigt werden.
- Funktionale Programmierung: Python unterstützt auch funktionale Programmierkonzepte wie First-Class-Funktionen, Funktionen höherer Ordnung, und Features wie `map()`, `filter()`, und `reduce()`.

3.2. Quellcode, Übersetzer, Maschinencode

Wenn Algorithmen in einer (höheren) Programmiersprache formuliert werden, wird dieser Text Quellcode (auch Quelltext) genannt. Damit ein Computer ein solches Programm versteht, muss es zunächst in Maschinensprache übersetzt werden. Ein solcher Übersetzer ist selbst ein (von Menschen geschriebenes) Programm, welches je nach Art der Übersetzung Compiler oder Interpreter genannt wird. Compiler übersetzen Quellcode zunächst vollständig und führen ihn danach auf einmal aus. Interpreter übersetzen Quellcode zeilenweise in Maschinensprache und führen diesen dann schrittweise aus. Python ist eine Sprache, die einen Interpreter besitzt.

3.3. Python vs Java

Wie bereits erwähnt hat jede Programmiersprache ihre Dasseinberechtigung, da sie je nach Kontext mehr oder weniger geeignet sein kann. In der Schule beschäftigen wir uns mit zwei wichtigen Programmiersprachen: Java und Python. Beide finden eine große Anwendung. Im folgenden wollen wir die beiden Sprachen gegenüberstellen.

Python:
- wurde 1991 entwickelt mit dem Fokus auf Einfachheit und Lesbarkeit des Codes
- Anwendungsgebiete: Software-Entwicklung, KI und Machine Learning
- Vorteile: Lesbarkeit (übersichtlicher Syntax), Open Source, Flexibilitäte
- Nachteile: Fehleranfälligkeit

Java:
- wurde 1995 entwickelt
- Anwendungsgebiete: mobile Anwendungen, Webserver, eingebettete Systeme
- Vorteile: Portabilität, Sicherheit
- Nachteile: Wortanzahl, Leistung

Jetzt werden wir auf ein paar konkretere Aspekte eingehen:
- *Benutzerfreundlichkeit*: Python liegt durch simpleren Ansatz vorne
- *Syntax*: Beispiel Code für die Aufgabe "Guten Tag! Das ist Java/Python."

```java
class Hallo {
    public static void main(String[] args) {
        System.out.println("Guten Tag! Dies ist Java.");
    }
}
```

```
>>> print("Guten Tag! Dies ist Python.")
```

- *Performance*: Java ist schneller, da Java den Code im Vorfeld bearbeitet und liefert Bytecode aus (statischer Ansatz); Python erstellt den Bytecode während der Ausführung (dynamischer Ansatz)
- *Umsetzung*: Java ist eine kompilierte Sprache (übersetzt Code zunächst komplett in Maschinensprache und führt ihn dann aus). Bei Python handelt es sich um eine interpretierte Sprache (Anweisungen werden übersetzt und direkt ausgeführt). Das macht die Sprache zwar übersichtlicher, aber eben auch etwas langsamer.
- *Stabilität*: beide Sprachen sind stabil. Projekte in Java sind besonders gut geschützt und stabil durch Sicherheitsfeatures, Analysetools und Kompatibilität
  mit älteren Versionen.

4. Grundlegende Sprachelemente

In diesem Kapitel geht um grundlegende Elemente, die bei der Programmierung immer wieder eine wichtige Rolle spielen: Variablen, Datentypen und Operatoren.

4.1. Variablen und Zählschleifen

Variablen müssen vor der ersten Verwendung **deklariert** werden. Bei der Deklaration einer Variablen wird ihr Bezeichner und ihr Datentyp folgendermaßen festgelegt: *Datentyp Bezeichner*.
Beispiele: String alter; oder int x;

Hinweise für sinnvolle Bezeichner:

- bestehen aus UNICODE-Zeichen
- dürfen nicht mit Ziffern beginnen
- dürfen viele Sonderzeichen nicht enthalten
- sollten aussagekräftig sein
- CamelCase verwenden
- gültig: x, a5n, anzahl_noten, nameDerNachbarin, ...
- ungültig: 5stunden, Name+Vorname, Mein Name, ...

Variablen sollten, wenn möglich bzw. sinnvoll, bei ihrer Deklaration **initialisiert** werden, d.h. mit einem Wert belegt. Beispiele:

- String name = "Ludwig Georg";
- int alter = 24;
- int hoehe = 5, breite = 6, laenge = 10;

4.2. Datentypen

In den Programmiersprachen wird meist zwischen verschiedenen Datentypen unterschieden.

Datentypen in Java

Datentyp	Wertebereich
int	ganze Zahlen, -2^{31} bis $2^{31} - 1$
float	Dezimalzahlen, $-3,4 * 10^{38}$ bis $3,4 * 10^{38}$, Größe 32 Bit
double	Kommazahlen, $-1,8 \cdot 10^{308}$ bis $1,8 \cdot 10^{308}$, Größe 64 Bit
char	Unicode-Zeichen (z. B. 'a', 'z', '+', ...)
boolean	Wahrheitswert (true oder false)
String	Zeichenketten (z. B. "Hallo", "Ludwig Georg", " ", ...)

Dass hier alle Datentypen bis auf String klein geschrieben werden, ist kein Versehen. Bei den kleingeschriebenen Datentypen handelt es sich um sogenante **primitive Datentypen**. Sie sind in Java grundlegend und werden direkt im Speicher als einfache Werte gespeichert. Sie haben keine Methoden und sind nicht Teil des Objektmodells von Java.

Bei dem Datentyp "String" handelt es sich um eine Klasse und somit um einen **Referenzdatentyp**. Das bedeutet, dass String-Objekte auf der Grundlage von Referenzen auf den Speicherplatz verwiesen werden, in dem die tatsächlichen Zeichen gespeichert sind. In Java werden Klassen nach Konvention immer groß geschrieben. Genaueres folgt dazu beim Thema "Objektorientierte Prgrammierung".

Datentypen in Python: Unterschiede zu Java

- **Python** hat eine einheitliche Objektstruktur, bei der selbst primitive Datentypen wie `int` und `float` als Objekte behandelt werden. Dies bietet Flexibilität, kann aber auch zu unterschiedlichen Speicher- und Leistungseigenschaften führen.
- **Wrapper-Klassen** in Java (z.B. `Integer`, `Double`) sind in Python nicht erforderlich, da primitive Typen bereits als Objekte behandelt werden.

4.3. Typumwandlung In Java

Implizite Typumwandlung zwischen Zahlentypen

```
1 double pi = 2.14;
2 int n = 1;
3 pi = pi + n; // pi = 3.14
```

(Die Integer-Variable n wurde automatisch bei der Berechnung in den ”gr¨oßeren” Datentyp double umgewandelt.)

explizite Typumwandlung zwischen Zahlentypen

```
1 double e = 2.7;
2 int n = (int) e; // n = 2
```

Umwandlung zwischen Zeichenketten und Zahlen

```
1 int i = 42;
2 String s = Integer.toString(i); // s = ”42”
3 String z = ”23”;
4 int j = Integer.parseInt(z); // j = 23
```

4.4. Operatoren

Bei der Programmierung kommen in unterschiedlichen Zusammenhänge verschiedene Operatoren zum Einsatz. Im Folgenden gibt es eine kleine Übersicht über die wichtigsten Operatoren.

1. Arithmetische Operatoren

Diese Operatoren werden für grundlegende mathematische Operationen verwendet.

Java	Beschreibung	Beispiel
+	Addition	a + b
-	Subtraktion	a - b
*	Multiplikation	a * b
/	Division	a / b
%	Modulo (Rest einer Division)	a % b
++	Inkrement (Erhöhung um 1)	a++ oder ++a
--	Dekrement (Verringerung um 1)	a-- oder --a

Die Operatoren unterscheiden sich somit nicht zu denen in Python. Hier sei aber auf einen Unterschied noch hingewiesen: In Python wird `**` für Potenzen verwendet; Java verwendet stattdessen `Math.pow(x, y)`. Es wird hier also eine Methode der Klasse Math aufgerufen.

2. Zuweisungsoperatoren

Diese Operatoren werden verwendet, um Werte Variablen zuzuweisen.

Operator	Beschreibung	Beispiel
=	Zuweisung	a = 10
+=	Addition und Zuweisung	a += 5
-=	Subtraktion und Zuweisung	a -= 3

3. Vergleichsoperatoren

Vergleichsoperatoren werden verwendet, um zwei Werte miteinander zu vergleichen und geben ein booleanes Ergebnis (`true` oder `false`) zurück.

Operator	Beschreibung	Beispiel
==	Gleichheit	a == b
!=	Ungleichheit	a != b
>	Größer als	a > b
<	Kleiner als	a < b
>=	Größer oder gleich	a >= b
<=	Kleiner oder gleich	a <= b

4. Logische Operatoren

Logische Operatoren werden verwendet, um boolesche Ausdrücke zu kombinieren. Hier gibt es deutliche Unterschiede zwischen den beiden Programmiersprachen.

Java	Python	Beschreibung
&&	and	Logisches UND
||	or	Logisches ODER
!	not	Logisches NICHT

Zur Erinnerung, was sich genau hinter diesen Operatoren verbirgt:

und:

&&	false	true
false	false	false
true	false	true

oder:

||	false	true
false	false	true
true	true	true

Negation:

a	!a
false	true
true	false

Beispiele:

- 0 ≤ note ≤ 15 : (0 <= note) && (note <= 15)
- Gegenteil: !((0 <= note) && (note <= 15))
- boolean entschuldigt = krank || beurlaubt;
- boolean unentschuldigt = !krank && !beurlaubt;

5. Konkatenation (Aneinanderreihung)

Unter Konkatenation versteht man die Aneinanderreihung von Zeichenketten.

```
1 String nachricht = "Hallo" + name + ", ";
2 nachricht = nachricht + "Alter: " + alter;
```

5. Flussdiagramm vs Struktogramm

Eine gute Möglichkeit, komplexe Probleme darzustellen, ist die Benutzung grafischer Unterstützung. Hierfür werden zwei Möglichkeiten in der Informatik gerne verwendet: Programmablaufpläne (Flussdiagramme) und Struktogramme (Nassi-Shneiderman-Diagramm).

Programmablaufplan (Flussdiagramm)

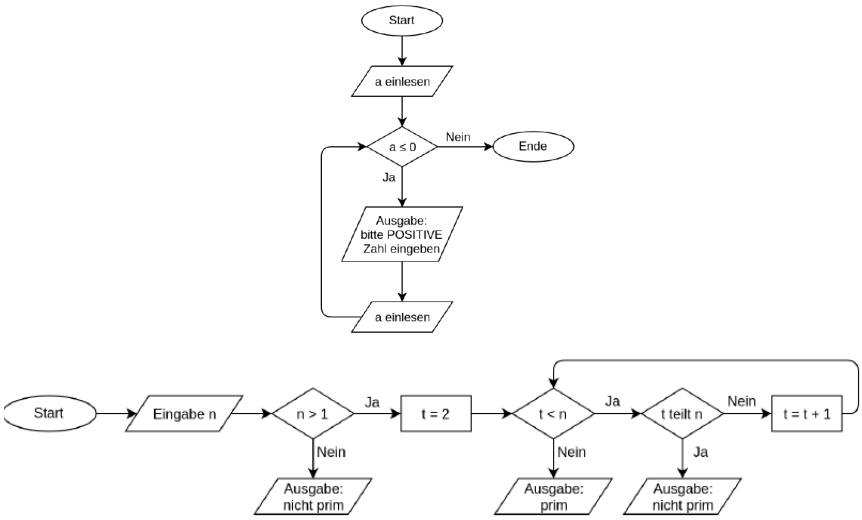
Ein Programmablaufplan wird aus sechs grundlegenden Bestandteilen zusammengebaut, welche immer durch das gleiche Symbol dargestellt werden. Je nach Quelle, können noch weitere Symbole hinzukommen, wir beschränken uns jedoch auf die hier angegebenen.

Für die Notation von größeren Algorithmen sind Programmablaufpläne ungeeignet, da ihnen unter anderem die expliziten Darstellungsmittel für Schleifen fehlen.

Symbole:

- Oval: Start/Ende eines Prozesses
- Rechteck: Anweisung/Funktion/Aktion
- Raute: Verzweigung/Abfrage/Entscheidung
- Pfeil: Verbindungselement
- Parallelogramm: Daten/Eingabe/Ausgabe

Beispiele:

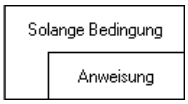


Struktogramm (Nassi-Shneiderman-Diagramm)

Jede Aktion (jedes Strukturelement) eines Algorithmus wird durch einen Block dargestellt. Die Blöcke werden aneinandergereiht oder können ineinander geschachtelt werden. Die ersten Basisdarstellungen werden im folgenden aufgezeigt.

Struktogramme sind ineinander verschachtelte Rechtecke. Willst du ein Struktogramm verfeinern, so zeichnest du einfach in das Struktogramm ein weiteres hinein, das dann angibt, was dein Programm-tun soll. Links siehst du den einfachsten Typ: erst wird Anweisung 1 ausgeführt, dann Anweisung 2, u.s.w.

Das Struktogramm zur (einseitigen) **Verzweigung**, hier wird der if-Zweig ausgeführt, falls die Bedingung wahr ist. Sonst -siehe das %-Zeichen- passiert nichts!



Und dann war da noch die **Schleife** . Solange die Bedingung wahr ist, wird die Anweisung im Schleifenrumpf ausgeführt, wobei du für die Anweisung wieder ein beliebiges Struktogramm einsetzen darfst.

Ausführliches Material zu den Struktogrammen inklusive Übungsaufgaben: Skript von Pellatz

Online Struktogramem erstellen: https://dditools.inf.tu-dresden.de/ovk/Informatik/Programmierung/Grundlagen/Struktogramme.html

6. "Hallo Welt" in Java

Das erste Programm was man klassisch beim Erlernen einer Programmiersprache schreibt, ist die Ausgabe von "Hallo Welt!".

Eine Anleitung für die Arbeit mit Eclipse ist im moodle zu finden. Der ausführbare Code in Java sieht lautet:

```
[1] public class HalloWelt {
[2]     public static void main(String[] args) {
[3]         System.out.println("Hallo, Welt!");
[4]     }
[5] }
```

Wenn man nun den Code mit Python vergleicht, fällt direkt der "komplexere" Aufbau auf. Damit man hier nicht mit völligem Blackbox-Wissen arbeitet, kommt hier schon eine erste Erklärung der einzelnen Bestandteile. Einige Aspekte werden dann zu einem späteren Zeitpunkt im Rahmen der objektorientierten Programierung erläutert. Den Aufbau des Codes kann man mit den verschiedemem Schichten einer Zwiebel vergleichen: project > package > class > main > Quelltext.

In unserem Beispiel ist die Zeile 3 unser eigentlicher Quelltext, in welchem festgelegt ist, was das Programm machen soll. Möchte man eine Textausgabe haben, wird dies durch die Funktion "System.out.println()" erreicht. (Hierbei wird von der Klasse System die Funktion aufgerufen, näheres später).

In Zeile 2 wird die main-class definiert. In einem Programm können mehrere Klassen implementiert werden, aber nur die main-class ist für die Ausführung des Codes zuständig. Diese ist immer gleich aufgebaut, weshalb man sie zum Beispiel bei Eclipse automatisch generieren lassen kann.

In Zeile 1 wird die Klasse definiert. Hierbei sollte man beachten, dass der Klassenname immer groß geschrieben wird. "Public" beschreibt die Zugriffsrechte.

7. Kontrollstrukturen allgemein

Kontrollstrukturen sind Konstrukte in einer Programmiersprache, die den Fluss der Ausführung eines Programms steuern. Die drei grundlegenden Arten von Kontrollstrukturen sind:

- **Sequenz:** Die Ausführung von Anweisungen in der Reihenfolge, in der sie im Code erscheinen.
- **Iteration**: Schleifen, die es ermöglichen, eine oder mehrere Anweisungen wiederholt auszuführen (for- und while-Schleifen).
- **Selektion**: Bedingte Anweisungen, die den Programmfluss basierend auf bestimmten Bedingungen verzweigen (if-Anweisung).

8. (Bedingte) Anweisungen

Die *if-Anweisung* in Java ist eine grundlegende Kontrollstruktur, mit der man Entscheidungen in einem Programm treffen kann. Sie ermöglicht es, bestimmte Codeabschnitte nur dann auszuführen, wenn eine bestimmte Bedingung erfüllt ist.

Die Syntax der if-Anweisung sieht folgendermaßen aus:
if (Bedingung) {

```
    // Code, der ausgeführt wird, wenn die Bedingung wahr ist
}
```

Hier prüft das Programm die Bedingung innerhalb der Klammern `(Bedingung)`. Wenn diese Bedingung `true` (wahr) ist, wird der Codeblock innerhalb der geschweiften Klammern `{}` ausgeführt. Wenn die Bedingung `false` (falsch) ist, wird dieser Codeblock bersprungen, und das Programm macht mit dem nächsten Befehl nach der `if`-Anweisung weiter.

Einseitig bedingte Anweisung

Eine einseitige bedingte Anweisung prüft eine Bedingung und führt einen bestimmten Codeblock nur dann aus, wenn diese Bedingung wahr (`true`) ist. Wenn die Bedingung falsch (`false`) ist, passiert nichts, und das Programm fährt mit dem nächsten Befehl nach der `if`-Anweisung fort. Beispiel: Man möchte in einem Programm überprüfen, ob eine Person alt genug ist, um zu wählen:

```
int alter = 18;

if (alter >= 18){
      System.out.println("Du bist alt genug, um zu wählen!");
}
```

In diesem Beispiel wird die Bedingung `alter >= 18` geprüft. Da `alter` den Wert 18 hat, ist die Bedingung wahr, und die Nachricht "Du bist alt genug, um zu wählen!" wird auf dem Bildschirm ausgegeben.

Zweiseitig bedingte Anweisung:

Eine **zweiseitige bedingte Anweisung** bietet zwei alternative Pfade: einen für den Fall, dass die Bedingung wahr ist, und einen anderen für den Fall, dass die Bedingung falsch ist. Diese Struktur wird mit `if` und `else` umgesetzt.

Manchmal möchte man, dass dein Programm auch dann etwas tut, wenn die Bedingung nicht wahr ist. Dafür kann man eine *else-Anweisung* verwenden:

```
if (alter >= 18) {
    System.out.println("Du bist alt genug, um zu wählen!");
} else {
    System.out.println("Du bist noch nicht alt genug, um zu wählen.");
}
```

Hier gibt das Programm "Du bist noch nicht alt genug, um zu wählen." aus, wenn die Bedingung `alter >= 18` falsch ist.

Geschachtelte Anweisung

Bei einer **geschachtelte** (oder **verschachtelte**) `if`-Anweisung liegt eine `if`-Anweisung innerhalb einer anderen `if`-Anweisung. Dies ist nützlich, um komplexere Bedingungen abzubilden, bei denen mehrere Prüfungen nacheinander durchgeführt werden müssen.

Beispiel: Man möchte nicht nur überprüfen, ob eine Person alt genug ist, um zu wählen, sondern auch, ob sie in einem bestimmten Alter ist, um z.B. ein bestimmtes Wahlrecht zu haben, wie z.B. für besondere Wahlen, die nur ab einem höheren Alter zugänglich sind.

```
int alter = 22;

if (alter >= 18) {
    System.out.println("Du bist alt genug, um zu wählen.");

    if (alter >= 21) {
        System.out.println("Du darfst auch an den besonderen Wahlen teilnehmen.");
    } else {
        System.out.println("Du darfst jedoch noch nicht an den besonderen Wahlen teilnehmen.");
    }

} else {
    System.out.println("Du bist noch nicht alt genug, um zu wählen.");
}
```

Vorteile der geschachtelten Anweisung:

- Flexibilität: Du kannst komplexere logische Entscheidungen treffen, indem du mehrere Ebenen von Bedingungen prüfst.
- Strukturierte Entscheidungen: Jede geschachtelte `if`-Anweisung wird nur ausgeführt, wenn die äußere Bedingung erfüllt ist, was zu einem klaren und strukturierten Entscheidungsprozess führt.

9. Schleifen

Schleifen sind essenzielle Kontrollstrukturen in Java, die es ermöglichen, einen Codeblock wiederholt auszuführen, solange eine bestimmte Bedingung erfüllt ist. Sie sind besonders nützlich, wenn du wiederholende Aufgaben automatisieren oder über Datenstrukturen wie Arrays oder Listen iterieren möchtest. In Java gibt es verschiedene Schleifenarten, aber zwei der am häufigsten verwendeten sind die **while-Schleife** und die **for-Schleife**.

while-Schleife

Die *while-Schleife* führt einen Codeblock so lange aus, wie eine angegebene Bedingung wahr (`true`) ist. Sie eignet sich besonders, wenn die Anzahl der Wiederholungen nicht von vornherein feststeht und von einer Laufzeitbedingung abhängt. Man bezeichnet sie daher auch als Bedingungsschleife.

Syntax der while-Schleife:

```
while (Bedingung) {
    // Code, der wiederholt ausgeführt wird
}
```

Beispiel:

```
int zahl = 0;

while (zahl < 5) {
    System.out.println("Zahl ist: " + zahl);
    zahl++; // Erhöht den Wert von zahl um 1
}
```

In diesem Beispiel wird die Schleife so lange ausgeführt, wie `zahl` kleiner als 5 ist. Bei jedem Durchlauf wird der aktuelle Wert von `zahl` ausgegeben und anschließend um 1 erhöht. Die Schleife endet, sobald die Bedingung `zahl < 5` nicht mehr erfüllt ist.

for-Schleife

Die *for-Schleife* ist eine kompaktere Schleifenform, die häufig verwendet wird, wenn die Anzahl der Iterationen bekannt ist oder wenn du über eine Datenstruktur wie ein Array iterieren möchtest. Sie kombiniert die Initialisierung einer Schleifenvariable, die Bedingungsprüfung und die Aktualisierung der Schleifenvariable in einer Zeile. Man bezeichnet sie deshalb auch als Zählschleifen.

Syntax der for-Schleife:

```
for (Initialisierung; Bedingung; Aktualisierung) {
    // Code, der wiederholt ausgeführt wird
}
```

Beispiel:

```
for (int i = 0; i < 5; i++) {
    System.out.println("i ist: " + i);
}
```

In diesem Beispiel wird die Schleife genau fünf Mal ausgeführt. Sie startet bei `i = 0`, überprüft bei jedem Durchlauf, ob `i < 5` wahr ist, und erhöht dann `i` um 1. Die Schleife endet, sobald die Bedingung `i < 5` nicht mehr erfüllt ist.

Unterschiede und Einsatzgebiete

- while-Schleife: Diese Schleife ist flexibler, da sie nur eine Bedingung benötigt und keine feste Struktur für Initialisierung und Aktualisierung vorgibt. Sie eignet sich, wenn die Anzahl der Durchläufe im Voraus nicht feststeht oder dynamisch ist.
- for-Schleife: Diese Schleife ist oft übersichtlicher, wenn du eine feste Anzahl von Iterationen benötigst oder über eine Reihe von Werten (wie in einem Array) iterierst. Die kompakte Form macht es einfach, Schleifenvariablen zu verwalten.

Darstellung Struktogramm und Javasyntax

10. Benutzereingaben (Scanner-Klasse)

Benutzereingaben in Java mit der Scanner-Klasse

In Java kannst du Programme erstellen, die auf Benutzereingaben reagieren. Eine der einfachsten Möglichkeiten, dies zu tun, ist die Verwendung der **Scanner-Klasse**. Mit dieser Klasse kannst du Eingaben von der Tastatur einlesen und in deinem Programm weiterverarbeiten. Das ist besonders nützlich, wenn du interaktive Programme erstellen möchtest, die dynamisch auf die Eingaben der Nutzer reagieren.

Wie funktioniert die Scanner-Klasse?

Die Scanner-Klasse gehört zum Paket `java.util`, und du musst sie in deinem Programm importieren, um sie verwenden zu können. Sie bietet verschiedene Methoden, um unterschiedliche Arten von Eingaben wie Zahlen, Zeichenketten (Strings) und mehr einzulesen.

Schritte zur Verwendung der Scanner-Klasse:

1. **Scanner importieren:** Zuerst musst du die Scanner-Klasse importieren.
2. **Scanner-Objekt erstellen:** Dann erstellst du ein Objekt der Scanner-Klasse, das mit dem Eingabegerät (meistens die Tastatur) verbunden ist.
3. **Eingabe einlesen:** Schließlich kannst du verschiedene Methoden des Scanner-Objekts verwenden, um Benutzereingaben zu erfassen.

Beispiel:

Hier ist ein einfaches Beispiel, das zeigt, wie du einen Namen und eine Zahl vom Benutzer einlesen kannst:

```java
import java.util.Scanner;

public class Benutzereingabe {
    public static void main(String[] args) {
        // Scanner-Objekt erstellen
        Scanner scanner = new Scanner(System.in);

        // Name einlesen
        System.out.print("Gib deinen Namen ein: ");
        String name = scanner.nextLine();

        // Alter einlesen
        System.out.print("Gib dein Alter ein: ");
        int alter = scanner.nextInt();

        // Ausgabe
        System.out.println("Hallo " + name + ", du bist " + alter + " Jahre alt.");

        // Scanner schließen
        scanner.close();
    }
}
```

Erklärung des Beispiels:

1. **Import der Scanner-Klasse:** `import java.util.Scanner;`
   - Dieser Befehl sorgt dafür, dass die Scanner-Klasse in deinem Programm verfügbar ist.
2. **Erstellen eines Scanner-Objekts:** `Scanner scanner = new Scanner(System.in);`
   - Hier wird ein Scanner-Objekt erstellt, das mit der Standard-Eingabequelle (`System.in`, also der Tastatur) verbunden ist.
3. **Einlesen eines Strings:** `String name = scanner.nextLine();`
   - Die Methode `nextLine()` liest eine gesamte Textzeile (String) vom Benutzer ein, bis die Enter-Taste gedrückt wird.
4. **Einlesen einer Zahl:** `int alter = scanner.nextInt();`
   - Die Methode `nextInt()` liest eine ganze Zahl (int) vom Benutzer ein.
5. **Ausgabe der Eingaben:** `System.out.println("Hallo " + name + ", du bist " + alter + " Jahre alt.");`

- Hier wird die Kombination der eingegebenen Daten auf der Konsole ausgegeben.

6. **Scanner schließen:** `scanner.close();`
   - Um Ressourcen zu sparen, sollte der Scanner nach der Benutzung geschlossen werden.

Wichtige Methoden der Scanner-Klasse:

- `nextLine()`: Liest eine komplette Zeile als String ein.
- `nextInt()`: Liest eine ganze Zahl (int) ein.
- `nextDouble()`: Liest eine Gleitkommazahl (double) ein.
- `nextBoolean()`: Liest einen boolean-Wert (true/false) ein.
- `next()`: Liest das nächste Wort als String ein (endet bei einem Leerzeichen).

11. Modularisierung: Funktionen und Prozeduren

In Java sind **Funktionen** und **Prozeduren** zentrale Bestandteile der Programmierung, die es ermöglichen, Code zu modularisieren und wiederzuverwenden. Diese Konzepte fördern die Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit von Code und tragen zur Vermeidung von Redundanz bei.

Vorteile von Funktionen und Prozeduren

1. Wiederverwendbarkeit: Einmal definierte Funktionen oder Prozeduren können an verschiedenen Stellen im Programm wiederverwendet werden, ohne den Code erneut schreiben zu müssen.
2. Modularität: Der Code wird in kleinere, übersichtliche Einheiten unterteilt. Jede Funktion oder Prozedur übernimmt eine spezifische Aufgabe, was die Struktur des Programms klarer und leichter verständlich macht.
3. Wartbarkeit: Änderungen müssen nur an einer Stelle im Code vorgenommen werden, was die Wartung und Aktualisierung des Programms vereinfacht.
4. Lesbarkeit: Durch das Zerlegen komplexer Aufgaben in kleinere Funktionen oder Prozeduren wird der Code leichter lesbar und verständlicher.

Funktionen in Java
In Java werden Funktionen als **Methoden** bezeichnet. Eine Methode ist ein Block von Code, der eine bestimmte Aufgabe ausführt und ein Ergebnis zurückgibt. Methoden werden innerhalb von Klassen definiert und können aufgerufen werden, um ihre Aufgaben auszuführen.

Syntax einer Methode:
Rückgabedatentyp MethodeName(Parameterliste) {
* // Code der Methode*
* return Rückgabewert;*
}

****Beispiel:****

```
            public class MatheOperationen{
               //Methode zur Addition von zwei Zahlen
               public int addiere(int a, int b){
               return a + b
               }

               public static void main(String[] args){
                               MatheOperationen mo = new MatheOperationen();
                               int summe = mo.addiere(5, 3);
                               System.out.println("Die Summe ist: " + summe);
                               }
               }
```

In diesem Beispiel wird die Methode `addiere` definiert, die zwei `int`-Werte addiert und das Ergebnis zurückgibt. Die Methode wird im `main`-Programm aufgerufen, um die Addition durchzuführen.

Prozeduren in Java
In Java gibt es keinen speziellen Begriff für Prozeduren, da alle Methoden sowohl Funktionen (die einen Wert zurückgeben) als auch Prozeduren (die keinen Wert zurückgeben) darstellen können. Eine Methode, die keinen Wert zurückgibt, verwendet den Rückgabedatentyp `void`.

****Syntax einer Prozedur:****
void ProzedurName(Parameterliste) {
// Code der Prozedur
}

****Beispiel:****

```
public class NachrichtSender{
            // Methode zum Drucken einer Nachricht
             public void druckeNachricht(String nachricht) {
                          System.out.println(nachricht);
}
public static void main(String[] args) {
            NachrichtSender ns = new NachrichtSender();
            ns.druckeNachricht("Hallo, Welt!");
     }
}
```

In diesem Beispiel wird die Methode `druckeNachricht` als Prozedur definiert, da sie keinen Wert zurückgibt. Sie druckt einfach eine Nachricht auf der Konsole aus.

Zusammengefasst:

Funktionen müssen immer einen Rückgabewert liefern.
static Datentyp Name (Parameterliste){
Anweisungen
return rueckgabewert;
}

Prozeduren haben keinen Rückgabewert (void = leer).
static void Name (Parameterliste){
Anweisungen
}

Begründen Sie, ob es sich bei dem gegebenen Beispiel um eine Funktion oder eine Prozedur handelt.

```
1  static int ggT(int a, int b) {
2    while (a!=b) {
3      if (a > b) {
4        a = a - b;
5      } else {
6        b = b - a;
7      }
8    }
9    return a;
10 }
```

12. Parameter und Rückgabewerte

Parameter sind wichtige Bestandteile von Methoden in Java. Sie ermöglichen es, Informationen an eine Methode zu übergeben, damit diese auf Basis dieser Daten arbeiten kann. Parameter tragen zur Flexibilität und Wiederverwendbarkeit von Methoden bei und spielen eine zentrale Rolle beim Design von Programmen.

Was sind Parameter?
Parameter sind Variablen, die in der Methodendefinition aufgeführt werden und beim Aufruf der Methode Werte entgegennehmen. Sie ermöglichen es der Methode, mit verschiedenen Daten zu arbeiten, ohne den Methodenkörper selbst ändern zu müssen.

**Syntax von Parametern in einer Methode:
Rückgabedatentyp MethodeName(Typ parameter1, Typ parameter2, ...) {

```
    // Code der Methode
    // Nutzung der Parameter
    return Rückgabewert;
}
```

Beispiel:

```
public class Rechner {

    // Methode zur Berechnung des Produkts von zwei Zahlen
    public int multipliziere(int a, int b) {
        return a * b;
    }

    public static void main(String[] args) {
        Rechner rechner = new Rechner();
        int ergebnis = rechner.multipliziere(4, 5);
        System.out.println("Das Ergebnis ist: " + ergebnis);
    }
}
```

In diesem Beispiel hat die Methode `multipliziere` zwei Parameter (`a` und `b`), die beim Aufruf der Methode Werte entgegennehmen. Diese Werte werden innerhalb der Methode verwendet, um das Produkt zu berechnen.

Typen von Parametern

1. Eingabeparameter (Input Parameters): Diese Parameter werden verwendet, um Daten an eine Methode zu übergeben. Im obigen Beispiel sind `a` und `b` Eingabeparameter, die die Werte 4 und 5 erhalten. Der Gültigkeitsbereich des Parameters beschränkt sich auf die jeweilige Methode. Eine Methode kann mehrere Parameter haben.
2. Ausgabeparameter (Output Parameters): Diese Parameter sind für die Rückgabe von Werten aus einer Methode zuständig. Der Typ des Rückgabewertes muss im Methodenkopf angegeben werden. Methoden ohne Rückgabewert haben an dieser Stelle das Schlüsselwort void. Eine Methode kann höchstens einen Rückgabewert haben.
3. Eingabe- und Ausgabeparameter (Input/Output Parameters): Manchmal kann eine Methode sowohl Daten empfangen als auch Daten zurückgeben. In solchen Fällen können Parameter sowohl zur Eingabe als auch zur Ausgabe verwendet werden.

13. Umgang mit Fehlern

Ganz wichtig beim Programmieren, dass man beim Schreiben eines Codes eigentlich immer irgendwelche kleinen oder groÿen Fehler einbaut. Es kann somit passieren, dass man einen Code in einer halben Stunde schreibt, aber mehrere Stunden dann mit der Fehlerbehebung beschäftigt ist. Man nennt das auch Debuggen. Das ist völlig normal!
Wichtig ist nur, dass man sich nicht entmutigen lassen darf. Das Gefühl, wenn man einen versteckten Fehler gefunden hat, ist umso schöner.

14. Arrays

Was ist ein Array?

Ein Array ist eine Datenstruktur, die es ermöglicht, mehrere Werte desselben Datentyps unter einem gemeinsamen Namen zu speichern. Im Gegensatz zu normalen Variablen, die nur einen einzelnen Wert speichern können, bietet ein Array die Möglichkeit, eine Sammlung von Werten zu speichern, die über Indizes zugänglich sind.

Ein Array in Java:

- Ist **statisch**: Die Größe eines Arrays wird bei der Erstellung festgelegt und kann nicht mehr verändert werden.
- Enthält **Elemente desselben Datentyps**: Alle Werte in einem Array müssen den gleichen Datentyp haben (z.B. `int`, `double`, `String`).

Aufbau eines Arrays

Ein Array besteht aus mehreren Elementen, die in aufeinanderfolgenden Speicherplätzen abgelegt sind. Jedes Element in einem Array wird durch einen **Index** angesprochen, der bei 0 beginnt.

Das heißt, das erste Element eines Arrays hat den Index 0, das zweite den Index 1 und so weiter.

Beispiel: Ein Array mit 5 Integer-Werten könnte wie folgt aussehen:

Index	**0**	**1**	**2**	**3**	**4**
Wert	10	20	30	40	50

In diesem Array ist das Element an Index 0 gleich 10, das Element an Index 1 gleich 20 usw.

Deklaration und Initialisierung eines Arrays in Java

Deklaration eines Arrays

Um ein Array in Java zu deklarieren, gibt man den Datentyp der Elemente an, gefolgt von einer eckigen Klammer `[]`, die anzeigt, dass es sich um ein Array handelt, und dann den Namen des Arrays.

```
int[] zahlen;
```

Initialisierung eines Arrays

Nach der Deklaration muss das Array mit einer festen Größe initialisiert werden. Dies geschieht mit dem `new`-Operator.

```
zahlen = new int[5];
```

Dies erstellt ein Array mit Platz für 5 Integer-Werte. Die Standardwerte sind dabei 0 (für numerische Datentypen).

Alternativ kann man Deklaration und Initialisierung in einem Schritt kombinieren:

```
int[] zahlen = new int[5];
```

Direkte Initialisierung mit Werten

Ein Array kann auch direkt mit bestimmten Werten initialisiert werden:

```
int zahlen = {1, 1, 2, 3, 5};
```

Hier wird ein Array von 5 Elementen erstellt und gleichzeitig mit den angegebenen Werten belegt.

Einzelne Zuweisung von Werten

Einem initialisierten Array kann man nun auch einzeln die Werte zuweisen:

```
zahlen[0] = 1;
zahlen[1] = 1;
zahlen[2] = 2;
zahlen[3] = 3;
zahlen[4] = 5;
```

Zugriff auf Array-Elemente

Der Zugriff auf einzelne Elemente eines Arrays erfolgt über den Index. Um ein bestimmtes Element zu lesen oder zu verändern, verwendet man den Namen des Arrays, gefolgt von dem Index in eckigen Klammern `[]`.

Beispiel – Zugriff auf ein Array-Element:

```
int wert = zahlen[2];
```

In diesem Fall ist wert gleich 2, da das Element an Index 2 den Wert 2 hat.

Beispiel – Veränderung eines Array-Elements:

```
zahlen[1] = 100;
```

Das Element an Index 1 wird durch den Wert 100 ersetzt.

14.1. Arrays und Schleifen

Arrays eignen sich perfekt für die Verwendung in Schleifen. Da jedes Element über einen Index zugänglich ist, können Sie eine Schleife verwenden, um jedes Element im Array durchzugehen.

Hierbei spielt das öffentliche Attribut (public) *length* eine wichtige Rolle, welches jedes Array besitzt. Dort wird die Länge des Arrays gespeichert.

Beispiel – Durchlaufen eines Arrays mit einer `for`-Schleife:

```
for (int i = 0; i < zahlen.lenghth; i++){
   System.out.println(zahlen[i]);
 }
```

In diesem Beispiel wird die `for`-Schleife verwendet, um alle Elemente des Arrays `zahlen` auszugeben. Die Schleife beginnt bei 0 und endet bei `zahlen.length - 1`, wobei `zahlen.length` die Anzahl der Elemente im Array zurückgibt.

14.2. Zweidimensionale Arrays

Was ist ein zweidimensionales Array?

Ein **zweidimensionales Array** ist ein Array von Arrays. Man kann es sich wie eine Tabelle oder ein Gitter vorstellen, das aus **Zeilen** und **Spalten** besteht. Es ermöglicht, Daten in einer Matrixstruktur zu speichern, bei der jedes Element durch zwei Indizes angesprochen wird: einen für die Zeile und einen für die Spalte.

Man kann sich das wie eine Tabelle vorstellen:

Zeile \ Spalte	0	1	2
0	10	20	30
1	40	50	60
2	70	80	90

In diesem Beispiel ist das Element in Zeile 0 und Spalte 1 gleich 20, während das Element in Zeile 2 und Spalte 0 den Wert 70 hat.

Deklaration und Initialisierung eines zweidimensionalen Arrays

Die Deklaration eines zweidimensionalen Arrays erfolgt ähnlich wie bei eindimensionalen Arrays, jedoch verwendet man zwei Paare von eckigen Klammern `[][]`, um die Dimensionen anzugeben.

Deklaration eines 2D-Arrays:

```
int[][] matrix;
```

Dies deklariert ein zweidimensionales Array namens `matrix`, das `int`-Werte speichert.

Initialisierung eines 2D-Arrays:

Die Initialisierung legt die Größe des Arrays in beiden Dimensionen fest (Zeilen und Spalten). Dies geschieht ebenfalls mit dem `new`-Operator:

```
matrix = new int[3][3];
```

Initialisierung mit Werten:

Man kann ein zweidimensionales Array auch direkt mit Werten befüllen:

```
int[][] matrix = {
     {10, 20, 30}, {40, 50, 60}, {70, 80, 90}
};
```

Dies erstellt ein 3x3-Array und füllt es sofort mit den angegebenen Werten.

9.3 Zugriff auf Elemente eines 2D-Arrays

Der Zugriff auf ein bestimmtes Element in einem zweidimensionalen Array erfolgt über die Kombination von zwei Indizes: **der erste Index** gibt die Zeile an und **der zweite Index** die Spalte.

Beispiel – Zugriff auf ein Element:

```
int wert = matrix[1][2];
```

In diesem Beispiel greift der Code auf das Element in der zweiten Zeile (Index 1) und dritten Spalte (Index 2) zu, welches den Wert 60 enthält.

14.3. Objekte in Arrays speichern (Ausblick)

In Java können Arrays nicht nur primitive Datentypen wie `int`, `double` oder `char` speichern, sondern auch **Objekte**. Ein Array kann also verwendet werden, um mehrere Instanzen einer Klasse zu speichern. Dies ist besonders nützlich, wenn man eine Sammlung von Objekten benötigt, die man über Indizes ansprechen möchte.

Zum Beispiel könnte man ein Array von `Person`-Objekten erstellen, wobei jede Person ein Objekt mit eigenen Attributen und Methoden ist.

```
Person[] klasse = new Person[30];
```