Einführung in Java

M. Zirbes, Ver2026-02-24.1

Prolog

Das vorliegende Skriptum dient als Einstieg in die Programmiersprache Java. Es richtet sich an Lernende, die erste praktische Programmiererfahrungen sammeln möchten und vermittelt die grundlegenden Konzepte sowie zentrale Syntaxelemente der Sprache. Die Inhalte werden praxisnah anhand von Codebeispielen und schrittweisen Erklärungen eingeführt.

Das Skriptum erhebt keinen Anspruch auf Vollständigkeit. Themen der objektorientierten Programmierung werden in dieser Einführung weitgehend ausgespart und nur punktuell angesprochen.

Das Skriptum gliedert sich in folgende Kapitel:

A) Java-Technologie
   Überblick über die technologischen Bestandteile der Java-Plattform

B) Java installieren und erstes Java-Programm
   Einstieg („getting started“) mit Schwerpunkt auf der Installation unter Windows

C) Befehlsübersicht
   Referenzkapitel zum Nachschlagen zentraler Sprachkonstrukte

D) Programmbeispiele
   Zum sukzessiven Durcharbeiten und Erlernen grundlegender Programmierkonzepte in Java

E) Übungen
   Zum Üben und zur Vertiefung der Inhalte anhand praxisnaher Aufgaben auf Basis der Programmbeispiele

A) Java-Technologie

1. Grundidee von Java

- Java ist eine objektorientierte Programmiersprache mit dem Ziel der Plattformunabhängigkeit.
  - Objektorientierung: Programme werden aus Objekten aufgebaut, die Daten (Eigenschaften) und Funktionen (Methoden) bündeln und miteinander interagieren.
  - Plattformunabhängigkeit: Ein Java-Programm kann auf unterschiedlichen Betriebssystemen (z. B. Android, Linux, macOS, Windows) ausgeführt werden, ohne verändert zu werden.
- Leitsatz: „Write once, run anywhere“.

2. Java-Quellcode und Bytecode

- Java-Programme werden zunächst als Quellcode geschrieben.
  - Java-Quellcode: vom Menschen lesbarer Programmtext mit Programmierbefehlen (Dateiendung: .java).
- Quellcode wird anschließend in Java-Bytecode übersetzt (Dateiendung: .class).
  - Java-Bytecode: plattformunabhängiger „Zwischencode“ zwischen Quellcode und ausführbarer Maschinensprache.
  - Bytecode bildet die Grundlage für die spätere Programmausführung.

3. Java Development Kit (JDK)

- Zentrales Softwarepaket der Java-Technologie.
- Enthält alle notwendigen Komponenten, um Java-Programme zu entwickeln und auszuführen.
- Bestandteile:
  - Java-Compiler (javac): übersetzt Quellcode in Bytecode
  - Java-Laufzeitumgebung (JRE): s. u.
  - weitere Entwicklungswerkzeuge

4. Java-Laufzeitumgebung (JRE)

- Laufzeitumgebung, in der Java-Programme ausgeführt werden.
- Bestandteil des JDK.
- Bestandteile:
  - Java Virtual Machine (JVM): s. u.
  - Java-Standardbibliotheken

5. Java Virtual Machine (JVM)

- Zentrales Ausführungselement der Java-Technologie.
- Virtuelle Maschine, die Bytecode ausführt; wandelt häufig genutzte Teile zur Laufzeit mit dem Just-in-Time-Compiler (JIT) in Maschinensprache um.
- Ausführung mit Hilfe plattformspezifischer JVM für einzelne Betriebssysteme.
- So laufen Java-Programme plattformunabhängig.

6. Ablauf: vom Programm zur Ausführung

- Technischer Ablauf eines Java-Programms:
  1. Java-Quellcode (.java)
  2. Compiler (javac, Teil des JDK)
     → Übersetzung in Java-Bytecode (.class)
  3. Java Virtual Machine (JVM, Teil der JRE im JDK)
     → Ausführung des Bytecodes
- Java-Programm kann auf unterschiedlichen Betriebssystemen ausgeführt werden.

7. Zusammenhang der Komponenten

- Hierarchischer Aufbau der Java-Technologie:
  - Java Development Kit (JDK)
    enthält
  - Java-Laufzeitumgebung (JRE),
    diese enthält
  - Java Virtual Machine (JVM)
- JDK enthält alle Komponenten, die zur Entwicklung und Ausführung von Java-Programmen notwendig sind.

8. Integrierte Entwicklungsumgebung (IDE)

- Software, die die Programmierarbeit erleichtert.
- Aufgaben einer IDE:
  - Schreiben von Quellcode
  - Übersetzen (Kompilieren) von Programmen
  - Starten und Testen von Programmen
  - Fehlersuche (Debugging)
- Beispiele (Java): Eclipse, NetBeans, Java-Editor, ...
- Einordnung:
  - Nicht Teil der Java-Technologie selbst.
  - Nutzt das JDK (Compiler und JRE).
  - Ohne installiertes JDK kann keine IDE Java-Programme ausführen.

9. Zusammenfassung

- Java ist eine objektorientierte und plattformunabhängige Programmiersprache.
- Das JDK enthält alle Entwicklungs- und Laufzeitkomponenten, inkl. JRE und JVM.
- Java-Quellcode wird vom Java-Compiler des JDK in plattformunabhängigen Bytecode übersetzt.
- Die JVM führt als Teil der JRE den Bytecode aus und ist betriebssystemspezifisch.
- IDEs erleichtern die Programmentwicklung, nutzen das JDK, sind aber nicht Teil der Java-Technologie.

B) Java installieren und erstes Java-Programm

Die folgende Darstellung bezieht sich konkret auf das Betriebssystem **Windows 11**.

1. Java Development Kit (JDK) herunterladen

Um Java-Programme lokal auf einem Computersystem entwickeln, kompilieren und ausführen zu können, ist die Installation eines (kostenfreien) Java Development Kit (JDK) erforderlich.

Beispiele

- Oracle JDK
  - Originale von Oracle bereitgestellte Java-Distribution, basierend auf OpenJDK.
  - Nutzung unterliegt einer kommerziellen Lizenz; je nach Einsatzzweck kostenpflichtig.
  - Download: https://www.oracle.com/de/java/technologies/downloads
- OpenJDK
  - Open-Source-Referenzimplementierung der Programmiersprache Java.
  - Funktional gleichwertig zum Oracle JDK; vollständig Java-kompatibel.
  - Download: https://openjdk.org

→ OpenJDK als Empfehlung.

Installation

- OpenJDK herunterladen und installieren.
- Installationsanleitung:
  - Für Windows die 64-Bit-Version (x64) herunterladen.
  - Im Ordner C:\Program Files einen Unterordner openjdk erstellen.
  - Inhalt der Zip-Datei nach C:\Program Files\openjdk entpacken.
  - Im Ergebnis hat man C:\Program Files\openjdk\jdk-xx.x.x (x'e stehen für die Java-Versionsnummer) und darin die Java-Dateien (z. B. Unterordner bin).

2. Systemvariablen setzen

Nach der Installation muss Java dem Betriebssystem bekannt gemacht werden, damit Java-Programme sowie die zugehörigen Werkzeuge (z. B. javac) systemweit über die Kommandozeile (s. u.) aufgerufen werden können. Dazu muss folgende technische Konfiguration vorgenommen werden.

Dialog für Umgebungsvariablen öffnen

- In der Windows-Suche „Erweiterte Systemeinstellungen anzeigen“ öffnen
- Button „Umgebungsvariablen...“ wählen

Neue Systemvariable anlegen

- Im unteren Bereich „Systemvariablen“ die bereits existierende Variable Path auswählen und Button „Bearbeiten…“ klicken
- Neuen Eintrag mit Klick auf Button „Neu“ hinzufügen
- Wert des neuen Eintrags (Pfad):
  C:\Program Files\openjdk\jdk-xx.x.x\bin

  Anm.: Pfad an den tatsächlichen Installationspfad des JDK (s. o.) anpassen.

- Alle Änderungen jeweils mit OK bestätigen.

Die Einstellungen bleiben auch nach einem Neustart des Betriebssystems erhalten. Erst wenn eine neue Java-Version installiert wird, müssen erneut Anpassungen vorgenommen werden (→ Aktualisierung der Dateipfade).

3. Installation testen

Kommandozeile

- Die Kommandozeile ermöglicht es, Programme und Werkzeuge direkt über Textbefehle aufzurufen.
- Sie macht die einzelnen Schritte beim Kompilieren und Ausführen von Java-Programmen sichtbar und nachvollziehbar.
- Gerade zum Einstieg hilft dies, die technischen Abläufe bewusst kennenzulernen, bevor spätere Entwicklungsumgebungen (s. u.) diese Schritte automatisieren.

Kommandozeile öffnen

- Unter Windows stehen hierfür verschiedene Programme zur Verfügung, die eine Kommandozeile bereitstellen:
  - PowerShell
  - Eingabeaufforderung (cmd.exe, Pfad: C:\Windows\System32\cmd.exe)
- Der Aufruf der Eingabeaufforderung erfolgt z. B. über die Windows-Suche mit dem Suchbegriff cmd.

Testbefehl

- In der Kommandozeile eingeben:
  javac -version
- Der Befehl prüft, ob der Java-Compiler (javac) systemweit verfügbar ist.
- Wird eine Java-Versionsnummer angezeigt, ist das JDK korrekt installiert und einsatzbereit.

4. Exkurs: Wichtige CMD-Kommandozeilen-Befehle für die Java-Arbeit

4.1 Navigation im Dateisystem

Befehl	Beschreibung	Beispiel / Anm.
`dir`	Inhalt des aktuellen Verzeichnisses anzeigen	
`cd <Verzeichnis>`	eine Verzeichnis-Ebene tiefer (= in ein Unterverzeichnis) wechseln	`cd Projekte`
`cd ..`	eine Verzeichnis-Ebene höher wechseln	
`cd \`	zum Hauptverzeichnis des Laufwerks wechseln	
`cd <Pfad>`	mehrere Verzeichnis-Ebenen direkt wechseln; bei Laufwerkswechsel zusätzlich /d erforderlich	`cd \Projekte\Java\HalloWelt` `cd /d D:\Projekte\Java\HalloWelt`
`<Laufwerk>:`	Laufwerk wechseln	`D:`
`cls`	Bildschirm der Kommandozeile leeren	

4.2 Dateien und Ordner verwalten

Befehl	Beschreibung	Beispiel / Anm.
`mkdir <Verzeichnis>`	neues Verzeichnis erstellen	`mkdir HalloWelt`
`rmdir <Verzeichnis>`	Verzeichnis löschen	`rmdir HalloWelt`
`copy <Quelle> <Ziel>`	Datei kopieren	`copy ABC.java XYZ.java`
`move <Quelle> <Ziel>`	Datei verschieben oder umbenennen	`move XYZ.java Projekte\XYZ.java`
`del <Datei>`	Datei löschen	`del HalloWelt.class`

4.3 Java im System finden und überprüfen

Befehl	Beschreibung	Beispiel / Anm.
`java -version`	installierte Java-Version anzeigen	*sollte aktuell sein und zur angezeigten javac-Version passen*
`javac -version`	installierte Java-Compiler-Version anzeigen	*sollte aktuell sein und zur angezeigten java-Version passen*
`where java`	Speicherort der verwendeten Java-Laufzeit anzeigen	*erste Treffer von java und javac sollten im selben JDK-Verzeichnis liegen*
`where javac`	Speicherort des verwendeten Java-Compilers anzeigen	*erste Treffer von java und javac sollten im selben JDK-Verzeichnis liegen*
`echo %PATH%`	PATH als eine lange Zeile anzeigen	*idealerweise nur wenige Java-Einträge; in der Praxis oft mehrere; Reihenfolge entscheidet, welcher verwendet wird*
`echo %PATH:;=&echo.%`	PATH mit einem Eintrag pro Zeile formatiert anzeigen	*erleichtert das Erkennen der Reihenfolge und möglicher Doppelungen*

4.4 Java kompilieren und starten (ausführliches Beispiel siehe Kapitel 7)

Befehl	Beschreibung	Beispiel / Anm.
`javac <Datei.java>`	Quelltext in Bytecode übersetzen (.java → .class)	`javac HalloWelt.java`
`java <Klasse>`	kompilierte Klasse starten (ohne Dateiendung!)	`java HalloWelt`
`java <Datei.java>`	Quelltext direkt starten, kompiliert intern, erzeugt keine .class-Bytecode-Datei (ab Java SE 11)	`java HalloWelt.java`

5. Texteditor als einfache Entwicklungsumgebung

Um erste Java-Programme zu schreiben, genügt ein einfacher Texteditor.

Beispiele

- Windows-Editor
  - bereits installiert
  - funktional eingeschränkt
- Notepad++
  - erweiterte Funktionen (z. B. Syntax-Hervorhebung, Zeilennummern)
  - kostenfrei, Open Source
  - Download: https://notepad-plus-plus.org

→ Notepad++ als Empfehlung.

Installation

- Notepad++ herunterladen und installieren.

6. Integrierte Entwicklungsumgebung (IDE)

Alternativ zum Texteditor kann eine integrierte Entwicklungsumgebung (IDE) verwendet werden. Diese bietet mehr Möglichkeiten, ist jedoch komplexer in der Bedienung.

Beispiele

- Eclipse
  - ursprünglich von IBM, heute Eclipse Foundation
  - kostenfrei, Open Source
  - sehr flexibel, etwas komplexer
  - Download: https://www.eclipse.org
- NetBeans
  - ursprünglich von Sun, dann Oracle, heute Apache Software Foundation
  - kostenfrei, Open Source
  - gute Standardkonfiguration, einfacher Einstieg
  - speziell für Java entwickelt
  - Download: https://netbeans.apache.org/download
- Java-Editor
  - entwickelt vom Informatik-Lehrer Gerhard Röhner
  - kostenfrei, nicht Open Source
  - speziell entwickelt für den Informatik-Unterricht unter didaktischen Gesichtspunkten
  - Download: https://javaeditor.org

→ Java-Editor als Empfehlung (NetBeans als Alternative).

Installation

- Java-Editor herunterladen und installieren.
- Voraussetzung: OpenJDK muss bereits installiert sein.

7. Erstes Java-Programm erstellen und ausführen

Arbeitsweise

- Zum Einstieg und für einfache Java-Programme:
  - Texteditor in Verbindung mit
  - Kommandozeile
- Für größere und weiterführende Projekte:
  - IDE, die viele Arbeitsschritte automatisiert

Zunächst wird die Nutzung von Texteditor und Kommandozeile betrachtet.

Programm erstellen

Anm.: Details zum Programmcode folgen später.

- In Notepad++ eine neue Datei anlegen.
- Folgendes Programm eingeben:

```
public class HalloWelt {

  public static void main(String[] args) {

    System.out.println("Hallo, Welt!");
  }

}
```

- Datei speichern als HalloWelt.java.

  Anm.: Als Dateiname sollte stets der Klassenname (hier: HalloWelt) gewählt werden, ergänzt um die Dateiendung .java. Dabei stets auf Groß-/Kleinschreibung achten, auch bei den Kommandozeilenbefehlen (s. u.). Jedes Java-Programm in einem eigenen Ordner speichern (z. B. Hallo-Welt-Programm).

Programm kompilieren

- Kommandozeile öffnen und in den Ordner mit der Datei HalloWelt.java wechseln (vgl. Kapitel 4).

  Anm.: Im Explorer mit Shift + Rechtsklick auf den Ordner (z. B. Hallo-Welt-Programm) klicken und „Eingabeaufforderung hier öffnen“ bzw. „Terminal hier öffnen“ auswählen.

- Kompilieren mit:
  javac HalloWelt.java
- Es entsteht die Datei HalloWelt.class (Java-Bytecode) im selben Ordner.

Programm ausführen

- In der Kommandozeile eingeben (ohne Dateiendung):
  java HalloWelt
- Es erscheint folgende Programm-Ausgabe:

```
Hallo, Welt!
```

8. Alternative zur Installation: Java direkt im Browser

Webbasierte Werkzeuge

- Neben der lokalen Installation von Java existieren webbasierte Werkzeuge, mit denen Java-Programme direkt im Browser geschrieben und ausgeführt werden können.
- Ein Beispiel ist der Online-Java-Compiler von Programiz:
  https://www.programiz.com/java-programming/online-compiler

Vorteile

- Kein lokales JDK und keine Systemkonfiguration (z. B. Systemvariablen) erforderlich.
- Geeignet für schnelles Ausprobieren einzelner Code-Fragmente und kurze Tests einfacher Programme.

Nachteile

- Keine realistische Arbeitsumgebung für weiterführende Java-Entwicklung.
- Technische Abläufe (Kompilieren, Ausführen, Dateistruktur) werden nur vereinfacht dargestellt.
- Nicht geeignet für größere Programme oder Projekte mit mehreren Quelltexten.

→ Für eine systematische und vertiefte Arbeit mit Java ist die lokale Installation mit JDK sowie die Nutzung der Kommandozeile bzw. einer IDE erforderlich und wird daher in diesem Skript bewusst verwendet.

9. Zusammenfassung

- Für die Entwicklung von Java-Programmen wird ein JDK benötigt.
- OpenJDK ist eine freie und empfohlene Variante.
- Nach der Installation müssen Systemvariablen (PATH) korrekt gesetzt werden.
- Die Kommandozeile dient zur Überprüfung der Installation und hilft mit grundlegenden Befehlen bei Navigation und Fehlersuche.
- Java-Programme können zunächst mit einem Texteditor erstellt und in der Kommandozeile ausgeführt werden (→ Sichtbarmachen der Abläufe).
- IDEs wie NetBeans bieten mehr Komfort und Automatisierung (→ Verbergen der Abläufe).
- Ein Java-Programm wird:
- geschrieben (.java),
- kompiliert (.class),
- ausgeführt (über die JVM).
- Online-Compiler eignen sich für kurze Tests, ersetzen jedoch keine lokale Entwicklungsumgebung.

C) Befehlsübersicht

1. Programmaufbau und Ablauf

Code	Beschreibung	Beispiel
`class`	definiert eine Klasse; in Java muss jeder Code in einer Klasse stehen; Dateiname sollte (bei public-Klassen: muss) mit Klassennamen übereinstimmen	`public class Hallo { ... }`
`private`	Zugriffsmodifikator; erlaubt den Zugriff nur von innerhalb der eigenen Klasse aus	`private int x;`
`public`	Zugriffsmodifikator; erlaubt den Zugriff auf Klasse oder Methode von überall aus	`public class Hallo { ... }`
`public static void main(String[] args)`	**Startpunkt jedes Java-Programms**; die JVM beginnt hier mit der Ausführung	`public static void main(String[] args) { ... }`
`static`	Kennzeichnet Klassenvariablen oder -methoden; sie gehören zur Klasse selbst und nicht zu einem einzelnen Objekt	`public static void main(String[] args)`
`void`	Rückgabetyp; bedeutet, dass eine Methode keinen Wert zurückliefert	`public static void main(String[] args)`
`{ }`	Blockklammern; fassen zusammen, was logisch zusammengehört (Klassen, Methoden, Schleifen, Bedingungen); Blöcke können verschachtelt sein	`if (x > 0) { ... }`
`;`	beendet eine einzelne Anweisung; Pflicht in Java	`int x = 5;`

2. Kommentare

Code	Beschreibung	Beispiel
`// ...`	einzeiliger Kommentar; alles nach // innerhalb einer Codezeile wird ignoriert	`// Dies ist ein Kommentar`
`/* ... */`	mehrzeiliger Kommentar; alles zwischen /* und */ wird ignoriert	`/* Dies ist ein \n mehrzeiliger Kommentar */`
`/** ... */`	JavaDoc-Kommentar; dient zur automatischen Dokumentation von Klassen und Methoden	`/** Diese Methode berechnet ... */`

3. Ein- und Ausgaben (in der Konsole)

3.1 (Programm-)Ausgaben

Code	Beschreibung	Beispiel
`System.out.print()`	gibt Text aus ohne Zeilenumbruch	`System.out.print("Hallo");`

`System.out.printf()`	Formatierte Ausgabe ohne Zeilenumbruch, mit Platzhaltern; Platzhalter:
	%d → ganze Zahl,
	%f → Kommazahl,
	%s → String,
	%c → einzelnes Zeichen
		`int alter = 25; String name = "Bob";`
		
		`System.out.printf("Name: %s, Alter: %d", name, alter);`
		
		`Ausgabe: Name: Bob, Alter: 25`

`System.out.println()`	gibt Text aus mit Zeilenumbruch	`System.out.println("Welt");`

3.2 Escape-Sequenzen

Code	Beschreibung	Beispiel

`\n`	Zeilenumbruch innerhalb eines Strings (newline); bewirkt, dass die Ausgabe in der Konsole in die nächste Zeile springt	`"Hallo\nWelt"`
		
		`Ausgabe:`
		`Hallo`
		`Welt`

`\t`	Tabulatorzeichen innerhalb eines Strings; dient zur Einrückung oder Ausrichtung von Text	`"Hallo\tWelt"`
		
		`Ausgabe: Hallo  Welt`

`\"`	Ein Anführungszeichen innerhalb eines Strings, ohne den String zu beenden	`"Hallo \"Welt\""`
		
		`Ausgabe: Hallo "Welt"`

`\\`	Ein Backslash innerhalb eines Strings	`"C:\\Programme"`
		
		`Ausgabe: C:\Programme`

3.3 (Benutzer-)Eingaben

Code	Beschreibung	Beispiel
`close()`	beendet Scanner und gibt Ressourcen frei; der Scanner kann danach nicht mehr verwendet werden	`sc.close();`
`import java.util.Scanner;`	bindet die Scanner-Klasse aus der Java-Standardbibliothek ein	`import java.util.Scanner;`
`new Scanner(...)`	erzeugt ein neues Scanner-Objekt	`Scanner sc = new Scanner(System.in);`
`next()`	liest ein einzelnes Wort (String) bis zum nächsten Leerzeichen	`String wort = sc.next();`

`nextBoolean()`	liest einen Wahrheitswert (boolean) ein (true oder false)	`boolean wahr = sc.nextBoolean();`
`nextDouble()`	liest eine Kommazahl (double) ein	`double kommazahl = sc.nextDouble();`
`nextInt()`	liest eine ganze Zahl (int) ein	`int zahl = sc.nextInt();`
`nextLine()`	liest eine komplette Textzeile (String) ein	`String text = sc.nextLine();`
`Scanner`	Klasse zum Einlesen von Benutzereingaben von der Tastatur	`Scanner sc = new Scanner(System.in);`
`System.in`	steht für die Tastatureingabe	`new Scanner(System.in)`

4. Variablen und Datentypen

4.1 Allgemeines

Code	Beschreibung	Beispiel
Zuweisung =	speichert einen Wert in einer Variable; nicht verwechseln mit Vergleich == (s.u.)!	`x = 42;`

4.2 Primitive Datentypen (→ feste Sprachbestandteile; Datentyp-Bezeichner klein schreiben)

Code	Beschreibung	Beispiel
`boolean`	Wahrheitswert (true / false), keine festgelegte Speichergröße (variiert)	`boolean fertig = false;`
`byte`	ganze Zahl, 1 Byte, Wertebereich: -128 bis +127	`byte a = 100;`
`char`	einzelnes Unicode-Zeichen (Buchstaben, Ziffern, Sonderzeichen), 2 Byte	`char note = 'A';`
`double`	Kommazahl, 8 Byte, doppelte Genauigkeit (ca. 15-16 Dezimalen); Standard-Gleitkommatyp	`double preis = 2.99;`
`float`	Kommazahl, 4 Byte, einfache Genauigkeit (ca. 7 Dezimalen), Zahlenliterale mit f am Ende kennzeichnen	`float pi = 3.14f;`
`int`	ganze Zahl, 4 Byte, Wertebereich: -2.147.483.648 bis +2.147.483.647; Standard-Ganzzahltyp	`int alter = 42;`
`long`	ganze Zahl, 8 Byte, große Werte, Zahlenliterale mit L am Ende kennzeichnen	`long million = 1000000L;`
`short`	ganze Zahl, 2 Byte, Wertebereich: -32.768 bis +32.767	`short b = 2000;`

4.3 Referenztypen (→ speichern keine Werte direkt, sondern eine Referenz auf ein Objekt; Datentyp-Bezeichner groß schreiben)

Code	Beschreibung	Beispiel

Array	Sammlung von Werten gleichen Typs („Felder“); Objekt, speichert Referenz auf das Objekt; Größe variabel, abhängig von der Länge; Details s. u.	`int[] zahlen;`
		`zahlen = new int[5];`
		
		oder kürzer:
		`int[] zahlen = new int[5];`

`String`	Zeichenkette (Text); Objekt, speichert Referenz auf das Objekt; Größe variabel, abhängig von der Länge; Details s. u.	`String name = "User";`

4.4 Strings und wichtige Methoden

Code	Beschreibung	Beispiel
`charAt(int index)`	liefert das Zeichen an der angegebenen Position (0-basiert)	`"Hallo".charAt(1)` `→ 'a'`
`contains(String s)`	prüft, ob String den Teilstring enthält; liefert einen boolean	`"Hallo Welt".contains("Welt")` `→ true`
`equals(String s)`	vergleicht Inhalt von Strings (nicht Referenz!); liefert einen boolean	`"abc".equals("abc")` `→ true`
`equalsIgnoreCase (String s)`	vergleicht wie equals, aber ohne Berücksichtigung der Groß-/ Kleinschreibung; liefert einen boolean	`"ABC".equalsIgnoreCase("abc")` `→ true`
`indexOf(String s)`	liefert Position des ersten Vorkommens zurück; -1 wenn nicht gefunden	`"Hallo Welt".indexOf("l")` `→ 2`
`length()`	liefert die Anzahl der Zeichen im String; beachte: mit () (anders bei Arrays, s. u.)	`"Hallo".length()` `→ 5`
`replace(CharSequence old, CharSequence new)`	ersetzt alle Vorkommen eines Zeichens oder Teilstrings (unter Beachtung der Groß-/Kleinschreibung); liefert neuen String	`"abcabcA".replace("a","x")` `→ "xbcxbcA"`
`substring(int start, int end)`	liefert einen neuen (Teil-)String von start bis end-1	`"Hallo".substring(1,4)` `→ "all"`
`toUpperCase() / toLowerCase()`	wandelt alle Zeichen in Groß- bzw. Kleinbuchstaben um; liefert neuen String	`"abc".toUpperCase()` `→ "ABC"`
`trim()`	entfernt führende und nachgestellte Leerzeichen, Zeilenumbrüche, Tabulatoren – nicht aber innerhalb des Textes; liefert neuen String	`" Hallo Welt  ".trim()` `→ "Hallo Welt"`

4.5 Arrays

Code	Beschreibung	Beispiel

Indexierung	Zugriff auf Array-Element über Index in eckigen Klammern [ ]; Zählung ab 0	(bzgl. Bsp. unter „Initialisierung“)
		
		`zahlen[0]`
		`→ 6`

Initialisierung	direkte Befüllung eines Arrays mit Start-werten bei der Deklaration; Länge ergibt sich automatisch aus der Anzahl der Werte	`int[] zahlen = {6,7,8,9};`
		
		`→ Index  0  1  2  3`
		`  Wert  6  7  8  9`

`length`	liefert die Anzahl der Elemente eines Arrays (Array-Länge); beachte: ohne () (anders bei Strings, s. o.)	(bzgl. Bsp. unter „Initialisierung“)
		
		`zahlen.length`
		`→ 4`

`new` *Typ*`[]`	reserviert Speicherplatz und erzeugt ein Array-Objekt mit fester Länge; der konkrete Typ bestimmt die Art der gespeicherten Werte; Standardwerte:
	int → 0,
	double → 0.0,
	boolean → false,
	String → null (Wert für „kein Objekt“)
		(bzgl. Bsp. unter „Initialisierung“)
		
		`zahlen = new int[4];`
		
		`→ Index  0  1  2  3`
		`  Wert  0  0  0  0`

Typ`[]`	Datentyp für ein Array eines bestimmten Typs; Typ steht hier als Platzhalter für einen konkreten Datentyp wie int, double, boolean oder String; Arrays sind Objekte, können null sein	`int[] zahlen;`

4.6 Zweidimensionale Arrays

Code	Beschreibung	Beispiel
Indexierung	Zugriff auf Array-Element über zwei Indizes in eckigen Klammern [zeile][spalte]; Zählung jeweils ab 0	(bzgl. Bsp. unter „Initialisierung“) matrix[0][0] → 5 matrix[0][1] → 6 matrix[1][0] → 7 matrix[1][2] → 9
Initialisierung	direkte Befüllung eines 2D-Arrays mit Startwerten bei der Deklaration (s. o.); äußere Klammer = Zeilen, innere Klammer = Spalten; unregelmäßige Zeilen („jagged arrays“) sind erlaubt	int[][] matrix = {{5,6},{7,8,9}}; → Spalte 0 Spalte 1 Spalte 2 Zeile 0 5 6 - Zeile 1 7 8 9
length	matrix.length liefert die Anzahl der Zeilen, matrix[i].length liefert die Anzahl der Spalten der i-ten Zeile	(bzgl. Bsp. unter „Initialisierung“) matrix.length → 2 matrix[0].length → 2 matrix[1].length → 3
new *Typ*[zeilen] [spalten]	reserviert Speicherplatz und erzeugt ein 2D-Array-Objekt mit fester Länge (s. o.)	(bzgl. Bsp. unter „Initialisierung“) matrix = new int[2][3]; → Spalte 0 Spalte 1 Spalte 2 Zeile 0 0 0 0 Zeile 1 0 0 0
Typ[][]	Datentyp für ein zweidimensionales Array (= Array, dessen Elemente selbst wieder Arrays sind) eines bestimmten Typs (s. o.)	int[][] matrix;

4.7 Typumwandlung

Umwandlung	Beschreibung	Beispiel

int → double	numerische Typumwandlung (ohne Cast): Ganzzahl wird in Kommazahl umgewandelt; Nachkommastelle wird hinzugefügt (Typerweiterung)	int i = 5;
		double d = i;
		
		→ 5.0

double → int	numerische Typumwandlung (mit Cast): Kommazahl wird in Ganzzahl umgewandelt; Nachkommastellen werden abgeschnitten (Typverengung)	double d = 5.7;
		int i = (int) d;
		
		→ 5

char → int	numerische Typumwandlung (mit Cast) über Zeichencode (Zeichen → Zahl): liefert den Unicode-Code eines Zeichens	char c = 'Z'; int x = (int) c; → 90

int → char	numerische Typumwandlung (mit Cast) über Zeichencode (Zahl → Zeichen): liefert das Zeichen eines Unicode-Codes	int x = 90;
		char c = (char) x;
		
		→ Z

String → int	Parsing (String → Ganzzahl): wandelt den Inhalt eines Strings in eine int-Zahl um; Text muss Ganzzahl abbilden	String s = "13"; int x = Integer.parseInt(s); → 13
String → double	Parsing (String → Kommazahl): wandelt den Inhalt eines Strings in eine double-Zahl um; Text muss Ganz- od. Kommazahl abbilden	String s = "13.42"; double d = Double.parseDouble(s); → 13.42
Zahl → String	Typumwandlung (Zahl → String): erzeugt einen neuen String mit der textuellen Darstellung eines numerischen Wertes (z.B. mit String.valueOf)	int x = 18; String s = String.valueOf(x); → "18"
	Typumwandlung (Zahl → String): erzeugt einen neuen String mit der textuellen Darstellung eines numerischen Wertes (durch String-Verkettung)	int x = 18; String s = "" + x; → "18"

5. Operatoren und Ausdrücke

5.1 Arithmetik

Code	Beschreibung	Beispiel
`+`	Addition / Textverkettung	`a + b` `"Hi " + name`
`-`	Subtraktion	`a - b`
`*`	Multiplikation	`a * b`
`/`	Division	`a / b`
`%`	Rest der Division (Modulo)	`x % 2`

5.2 Vergleich und Logik

Code	Beschreibung	Beispiel
`==`	gleich; nicht verwechseln mit Zuweisung (s.o.)!	`x == 5`
`!=`	ungleich	`x != 0`
`<  >  <=  >=`	vergleiche zwischen Zahlen	`alter >= 18`
`&&`	logisches UND: liefert true nur, wenn alle Bedingungen wahr sind	`a > 0 && b > 0`
`||`	logisches ODER: liefert true, wenn mindestens eine Bedingung wahr ist	`a > 0 || b > 0`
`!`	NICHT: liefert das Gegenteil des aktuellen Wahrheitswerts	`!fertig`

6. Rundung, mathematische Funktionen und Zufall

6.1 Rundung

Code	Beschreibung	Beispiel
`Math.ceil(x)`	rundet auf die nächstgrößere ganze Zahl; liefert double	`Math.ceil(4.2)` `→ 5.0`
`Math.floor(x)`	rundet ab auf die nächstkleinere ganze Zahl; liefert double	`Math.floor(4.2)` `→ 4.0`
`Math.round(x)`	rundet eine Kommazahl auf die nächstgelegene ganze Zahl (kaufmännische Rundung, ≥0.5 wird aufgerundet); liefert long	`Math.round(4.2)` `→ 4`

6.2 Mathematische Konstanten und Funktionen

Code	Beschreibung	Beispiel
`Math`	Klasse der Java-Standardbibliothek mit mathematischen Konstanten und Funktionen; alle Elemente sind statisch und werden über den Klassennamen aufgerufen	`Math.sqrt(64)`
`Math.abs(x)`	berechnet den Betrag eines Wertes; liefert double	`Math.abs(-42)` `→ 42`
`Math.cbrt(x)`	Kubikwurzel; funktioniert auch für negative Werte; liefert double	`Math.cbrt(64)` `→ 4.0`
`Math.cos(x)`	Kosinus eines Winkels im Bogenmaß (Radiant); liefert double	`Math.cos(Math.PI / 3)` `→ 0.5`
`Math.E`	Konstante für die Eulersche Zahl e; liefert double	`Math.E` `→ 2.718281828459045`
`Math.exp(x)`	berechnet e hoch x; liefert double	`Math.exp(1)` `→ 2.71828…`
`Math.log(x)`	natürlicher Logarithmus (Basis e); liefert double	`Math.log(Math.E)` `→ 1.0`
`Math.max(a, b)`	liefert den größeren von zwei Werten	`Math.max(42, 7)` `→ 42`
`Math.min(a, b)`	liefert den kleineren von zwei Werten	`Math.min(42, 7)` `→ 7`

`Math.PI`	Konstante für die Kreiszahl π; liefert double	`Math.PI` → 3.141592653589793
`Math.pow(a, b)`	berechnet a hoch b; liefert double	`Math.pow(2, 6)` → 64.0
`Math.signum(x)`	gibt das Vorzeichen eines Wertes aus: -1.0 (negativ), 0.0, 1.0 (positiv); liefert double	`Math.signum(-13)` → -1.0
`Math.sin(x)`	Sinus eines Winkels im Bogenmaß (Radiant); liefert double	`Math.sin(Math.PI / 6)` → 0.5
`Math.sqrt(x)`	Quadratwurzel; sinnvoll nur für x ≥ 0; liefert double	`Math.sqrt(64)` → 8.0
`Math.tan(x)`	Tangens eines Winkels im Bogenmaß (Radiant); liefert double	`Math.tan(Math.PI / 4)` → 1.0

6.3 Zufall

Code	Beschreibung	Beispiel
`(int)(Math.random() * n)`	liefert Pseudo-Zufallszahl vom Typ int (durch Cast) im Bereich 0 bis n−1	`(int)(Math.random() * 10)` → 0–9
`Math.random()`	liefert Pseudo-Zufallszahl vom Typ double im Intervall [0.0, 1.0)	`double zuf = Math.random();`
`Random`	Klasse aus java.util; liefert Pseudo-Zufallszahlen verschiedener Typen (int, double, boolean); ohne Seed	`Random rand = new Random();`
`Random(seed)`	wie oben, aber mit Seed (fester Startwert) für reproduzierbare Folge von Pseudo-Zufallszahlen bei jeder Ausführung	`Random rand = new Random(42);`
`rand.nextBoolean()`	liefert Pseudo-Zufallszahl vom Typ boolean im Bereich true oder false	`boolean zuf = rand.nextBoolean();`
`rand.nextDouble()`	liefert Pseudo-Zufallszahl vom Typ double im Bereich [0.0, 1.0)	`double zuf = rand.nextDouble();`
`rand.nextInt(n)`	liefert Pseudo-Zufallszahl vom Typ int im Bereich 0 bis n-1; Obergrenze exklusiv	`int zuf = rand.nextInt(6);` → 0–5
`SecureRandom`	Klasse für sicherheitskritische Anwendungen; liefert kryptografisch sichere Pseudo-Zufallszahlen	`SecureRandom srand = new SecureRandom();`

7. Kontrollstrukturen

7.1 Bedingungen

Code	Beschreibung	Beispiel
`default`	wird ausgeführt, wenn keiner der case-Werte zutrifft	`switch(note) { case 1: ... break; default: System.out.println("Un-gültige Note"); }`
`else`	wird ausgeführt, wenn die vorherige if-Bedingung nicht zutrifft	`if (x > 0) { ... } else { System.out.println("x nicht positiv"); }`
`else if`	prüft zusätzliche Bedingungen, wenn die vorherige if oder else if nicht zutrifft	`if (x > 0) { ... } else if (x == 0) { System.out.println("x ist 0"); }`
`if`	führt den Code nur aus, wenn die Bedingung wahr ist (true)	`if (x > 0) { System.out.println("x positiv"); }`
`switch` (klassische Variante)	Auswahl anhand eines Werts (z. B. Zahl, String); jeder case prüft auf Gleichheit; wichtig: ohne break wird der Code in allen folgenden case-Blöcken ausgeführt	`switch(note) { case 1: System.out.println("1"); break; case 2: System.out.println("2"); break; }`

7.2 Schleifen

Code	Beschreibung	Beispiel
`break`	beendet die aktuelle Schleife sofort	`break;`
`continue`	überspringt den aktuellen Schleifen-durchlauf und fährt mit dem nächsten fort	`continue;`
`do-while`	Schleife mit Nachbedingung; wird mindestens einmal ausgeführt	`do { ... } while (x > 0);`
`for`	Zählschleife mit Initialisierung, Bedingung und Schrittweite; geeignet, wenn die Anzahl der Durchläufe bekannt ist	`for (int i = 0; i < 5; i++)`
`for-each`	erweiterte for-Schleife; durchläuft ohne expliziten Index alle Elemente eines Arrays	`for (int z : zahlen)`
`while`	Schleife mit Vorbedingung; wird nur ausgeführt, solange die Bedingung wahr ist	`while (x > 0)`

8. **Methoden** (Funktionen, ohne OOP)

Code	Beschreibung	Beispiel
Methodenkopf	Name der Methode, Rückgabetyp und Parameter	`static int add(int a, int b)`
Parameter	Übergabewerte an die Methode	`(int x, int y)`
`return`	beendet die Ausführung der Methode sofort und gibt einen Wert zurück (bei void ohne Wert); Code nach return wird nicht mehr ausgeführt	`return a + b;`
`static`	gehört zur Klasse, nicht zu einem Objekt (für main erforderlich)	`static int x;`
`void`	Methode ohne Rückgabewert	`static void ausgabe()`

9. **Fehler und Debugging** (Grundlagen)

Begriff	Beschreibung	Beispiel
Compilerfehler	Fehler, die bereits beim Übersetzen des Programms auftreten; Programm kann nicht gestartet werden	*Fehlendes ;*
Debug-Ausgabe	println als simple Technik zur Fehlersuche durch zusätzliche Ausgaben im Code, um Werte und Abläufe zu prüfen	`System.out.println(x);`
Laufzeitfehler	Fehler, die erst während der Programmausführung auftreten; Programm startet, bricht aber ab	*Division durch 0*
Logikfehler	Programm läuft ohne Absturz, liefert aber ein falsches oder unerwartetes Ergebnis	*Falsche Bedingung*

D) Programmbeispiele

01. Grundstruktur eines Java-Programms

Code

```
public class Grundstruktur {

  public static void main(String[] args) {

    // Hier kommen Programmbefehle hin
  }

}
```

Ausgabe

```
<keine Ausgabe>
```

Erläuterung

Anm.: Näheres zu Klassen, Objekten und Methoden folgt später.

- Dieser Programmcode zeigt die minimale, zwingend notwendige Struktur, damit ein Java-Programm ausgeführt werden kann.

- public class Grundstruktur
  - Hauptklasse des Java-Programms.
  - public
    - Zugriffsmodifikator; bedeutet: die Klasse ist von überall im Programm sichtbar.
    - Java Virtual Machine (JVM) muss diese Klasse von außen laden können, sonst kann das Programm nicht starten.
  - class
    - Java ist strikt klassenbasiert.
    - Es gibt in Java keinen ausführbaren Code außerhalb einer Klasse.
  - Grundstruktur
    - Weitgehend frei wählbarer Name der Klasse.
    - Darf Buchstaben, Ziffern, Unterstrich _, Dollarzeichen $ enthalten; keine Leerzeichen, keine reservierten Schlüsselbegriffe (z. B. public), nicht mit Ziffer beginnen.
    - Muss exakt dem Dateinamen entsprechen, hier: Grundstruktur.java.

- public static void main(String[] args)
  - Startmethode des Programms (erkennbar an main); Java sucht beim Programmstart nach *exakt* dieser Signatur.
  - public
    - Siehe oben; die JVM ruft diese Methode von außen auf; sie muss öffentlich sein.
  - static
    - Die Methode gehört zur Klasse selbst, nicht zu einem Objekt.
    - Beim Programmstart existiert noch kein Objekt, daher ist static zwingend erforderlich.
  - void
    - Bedeutet: kein Rückgabewert.
    - Die main-Methode liefert kein Ergebnis zurück, sie startet lediglich das Programm.
  - main
    - Der Name der Startmethode ist mit main fest vorgegeben und darf nur einmal in einem Java-Programm vorkommen.
  - String[] args
    - Ein Array von Zeichenketten (String). Hier können beim Programmstart Parameter von außen an main übergeben werden (z. B. über die Kommandozeile).
    - String[] ist ein Datentyp, konkret: ein Array von Strings. args ist ein Variablenname und frei wählbar (üblich ist hier args als Abkürzung für arguments). Damit handelt es sich bei args um eine Variable, die ein Array von Texten enthält.
    - Kurzum: args kann beim Programmstart Parameter aufnehmen; in diesem Beispiel bleibt es leer. Ausführliche Erläuterungen zu Arrays s. u.

- { … }
  - Geschweifte Klammern definieren Codeblöcke.

- Jede Klasse und jede Methode wird durch ein eigenes Paar geschweifter Klammern als Codeblock abgegrenzt. Ohne korrekt gesetzte Klammern ist Java-Code ungültig (Syntaxfehler).
- Code innerhalb der geschweiften Klammern wird eingerückt, um die Struktur sichtbar zu machen (s. u.).

02. Hallo Welt: Ausgabe, Einrückung, Groß-/Kleinschreibung, Kommentar

Code

```
public class HalloWelt {

  public static void main(String[] args) {

    System.out.print("Hallo, ");
    System.out.println("Welt!");

    /*
    Oder kürzer/kombiniert:
    System.out.println("Hallo, Welt!");
    */
  }

}
```

Ausgabe

```
Hallo, Welt!
```

Erläuterung

- Ausgabe
  - print gibt Text ohne Zeilenumbruch aus, println mit Zeilenumbruch.
  - Durch die Kombination beider Befehle entsteht im Beispiel eine einzelne Ausgabezeile „Hallo, Welt!“.

- Code-Einrückungen
  - Sind in Java technisch nicht verpflichtend, der Compiler ignoriert sie vollständig (ebenso wie Kommentare, s. u.).
  - Für Menschen sind Code-Einrückungen hingegen extrem wichtig: Sie machen Codeblöcke sichtbar, erhöhen die Lesbarkeit und helfen, Logikfehler zu vermeiden.

- Groß-/Kleinschreibung
  - Java ist case-sensitiv (groß-/kleinschreibungssensitiv) – Klassen-, Methoden- und Variablennamen müssen exakt stimmen.
  - Zum Beispiel sind Schreibweisen wie system.out.println oder System.Out.Println falsch.

- Kommentare
  - Bei /* … */ handelt es sich um einen mehrzeiligen Kommentar.
  - Mit Kommentaren kann man Anmerkungen im Code platzieren (Dokumentation) oder Code-Blöcke „ausklammern“, so dass diese im Quelltext erhalten bleiben, aber bei der Ausführung des Programms ignoriert werden (wie im Beispiel oben).
  - Einzeilige Kommentare werden mit // geschrieben.

03. Variable I: Deklaration, Initialisierung

Code

```
public class VariableDeklarationInitialisierung {

  public static void main(String[] args) {

    int alter;
    alter = 16;

    /*
    Oder kürzer/kombiniert:
    int alter = 16;
    */

    System.out.println(alter);
  }

}
```

Ausgabe

```
16
```

Erläuterung

- Variablen und Datentypen
  - Variablen sind benannte Speicherplätze und dienen dazu, Werte zu speichern. Im Gegensatz zur Mathematik können in Java Variablennamen aus mehreren Zeichen bestehen, z. B. alter anstelle nur x.
  - In Java muss jede Variable einen Datentyp haben, der festlegt, wie ein gespeicherter Wert zu interpretieren ist (z. B. int für Ganzzahlen, double für Kommazahlen, char für einzelne Zeichen, String für Text).

- int alter;
  - Deklaration: Java reserviert Speicherplatz für eine Variable namens alter vom Datentyp int (Ganzzahl).
  - Mit Hilfe des Variablennamens kann auf diesen Speicherplatz zugegriffen werden.
  - Variablen müssen vor ihrer Verwendung deklariert werden. Deklaration können an verschiedenen Stellen innerhalb eines Codeblocks vorgenommen werden.
  - Gültigkeitsbereich (Scope): Eine Variable kann nur innerhalb des Codeblocks verwendet werden („ist gültig“), in dem sie deklariert wurde. Näheres dazu s. u.
  - Java ist case-sensitiv (s. o.): alter und Alter sind unterschiedliche Variablen. Zwar erlaubt Java int alter; und int Alter;, jedoch sollte der Vermeidung von Verwirrung wegen darauf verzichtet werden.

- alter = 16;
  - Zuweisung: Der Variablen alter wird der Wert 16 zugewiesen.
  - Eine Zuweisung hat immer die Form: Variable = Ausdruck;.
  - In Java erfolgen Zuweisungen mit = (Zuweisungsoperator), nicht verwechseln mit == (Vergleichsoperator)!
  - Bevor man mit Variablen arbeitet (z. B. rechnet), sollten sie immer einen definierten Wert zugewiesen bekommen.
  - Die erste Zuweisung einer Variablen nennt man Initialisierung.

- int alter = 16;
  - Eine kombinierte Kurzschreibweise für Deklaration und Initialisierung.

04. Variable II: Konkatenation, Escape-Sequenzen

Code

```java
public class VariableKonkatenationEscapeSequenzen {

  public static void main(String[] args) {

    String name = "User";
    char satzzeichen = '!';

    // Ausgabe normal
    System.out.println("Hallo, " + name + satzzeichen);

    System.out.println();  // Zur optischen Trennung der Ausgaben

    // Ausgabe mit Escape-Sequenz für Zeilenumbruch
    System.out.println("Hallo,\n" + name + satzzeichen);
  }

}
```

Ausgabe

```
Hallo, User!

Hallo,
User!
```

Erläuterung

Anm.: Technische Details zu Strings (Objekte, Referenzen etc.) folgen später.

- String name = "User";
  - String ist der Datentyp für Texte (Zeichenketten).
  - Strings können beliebig lang sein; die Länge ist nicht fest vorgegeben.
  - In Java muss der Datentyp String großgeschrieben werden (case-sensitiv), d. h. mit großem „S“.
  - Strings werden in doppelte Anführungszeichen " " gesetzt.
  - Im Beispiel wird der Text "User" in die String-Variable gespeichert.

- char satzzeichen = '!';
  - Char ist ein primitiver Datentyp und enthält – anders als ein String – stets nur ein einzelnes Zeichen (Buchstabe, Ziffer oder Sonderzeichen).
  - Chars werden in einfache Anführungszeichen ' ' gesetzt.
  - Im Beispiel wird das Zeichen „!“ in die Char-Variable gespeichert.

- System.out.println("Hallo, " + name + satzzeichen);
  - Das Verknüpfen von Texten und (Text-)Variablen nennt man Konkatenation.
  - Der Operator + verbindet Strings mit anderen Strings oder Datentypen.
  - Nicht-String-Werte wie char und int werden dabei automatisch in einen String umgewandelt.
  - Im Beispiel werden der String „Hallo, “, die String-Variable name und die (automatisch in einen String umgewandelte) Char-Variable satzzeichen zu einem String zusammengesetzt.
  - Ergebnis: „Hallo, User!“.

- System.out.println("Hallo,\n" + name + satzzeichen);
  - Die Escape-Sequenz \n erzeugt einen Zeilenumbruch innerhalb des Strings.
  - Ergebnis: „Hallo,“ in der ersten Zeile, „User!“ in der zweiten Zeile.
  - Escape-Sequenzen wie \n können auch direkt in String-Variablen gespeichert und später ausgegeben werden, z. B. String text = "Zeile 1\nZeile 2";.

05. Rechnen & Ausgabe I: Addition, Ergebnisvariable, Klammern

Code

```java
public class RechnenAusgabeAdditionErgebnisvariableKlammern {

  public static void main(String[] args) {

    int zahl1 = 5;
    int zahl2 = 2;

    int ergebnis = zahl1 + zahl2;

    // Ausgaben mit Ergebnisvariable
    System.out.println(ergebnis);
    System.out.println("Das Ergebnis ist " + ergebnis + ".");

    System.out.println();  // Zur optischen Trennung der Ausgaben

    // Ausgaben ohne Ergebnisvariable (Berechnung in der Ausgabe)
    System.out.println(zahl1 + zahl2);
    System.out.println("Das Ergebnis ist " + (zahl1 + zahl2) + ".");
    System.out.println("Das Ergebnis ist " + zahl1 + zahl2 + ".");  // Vorsicht!
  }

}
```

Ausgabe

```
7
Das Ergebnis ist 7.

7
Das Ergebnis ist 7.
Das Ergebnis ist 52.
```

Erläuterung

*Anm.: Die Subtraktion (zahl1 - zahl2) und Multiplikation (zahl1 * zahl2) funktionieren analog. Näheres zur Division als vierte Grundrechenart s. u.*

- Grundidee der Aufgabe
  - In diesem Beispiel werden Rechnen und Textausgabe kombiniert.
  - Es wird gezeigt, dass der Operator + in Java zwei unterschiedliche Bedeutungen haben kann:
    - ~ Addition, wenn beide Operanden Zahlen sind,
    - ~ Konkatenation, sobald ein String beteiligt ist.
  - Die Aufgabe verdeutlicht außerdem die Bedeutung von Klammern bei Ausgaben.

- int ergebnis = zahl1 + zahl2;
  - zahl1 + zahl2 ist ein arithmetischer Ausdruck (Addition).
  - Der Ausdruck wird zuerst berechnet.
  - Das Ergebnis (hier 7) wird in die Ganzzahl-Variable ergebnis gespeichert.
  - Diese Variante mit Ergebnis-Variable ist übersichtlich und gut geeignet für:
    - ~ längere Rechnungen,
    - ~ spätere Weiterverarbeitung,
    - ~ mehrfach verwendete Ergebnisse.

Ausgaben mit Ergebnisvariable

- System.out.println(ergebnis);
  - Gibt den reinen Zahlenwert aus.

- System.out.println("Das Ergebnis ist " + ergebnis + ".");
  - Verknüpft Text mit einer Zahl (Konkatenation).
  - Der Zahlenwert von ergebnis wird dabei automatisch in einen String umgewandelt.

Ausgaben ohne Ergebnisvariable (Berechnung in der Ausgabe)

- System.out.println(zahl1 + zahl2);
  - Die Addition wird direkt berechnet und ausgegeben.

- System.out.println("Das Ergebnis ist " + (zahl1 + zahl2) + ".");
  - Die Klammern legen fest, dass der Ausdruck zuerst ausgerechnet werden soll.
  - Erst danach wird das Ergebnis mit dem Text verknüpft.

- System.out.println("Das Ergebnis ist " + zahl1 + zahl2 + ".");
  - Der +-Operator wird von links nach rechts ausgewertet.
  - Sobald ein String beteiligt ist, führt + keine Addition mehr aus, sondern Konkatenation.
  - Ablauf:
    1. "Das Ergebnis ist " + zahl1 → "Das Ergebnis ist 5"
    2. "5" + zahl2 → "52"
  - Es findet keine Rechnung, sondern nur Textverknüpfung statt.
  - Ausgabe: „Das Ergebnis ist 52.“

- Wichtige Erkenntnisse
  - Die Bedeutung des Operators + hängt davon ab, welche Datentypen beteiligt sind (kontextabhängig).
  - Klammern legen fest, welche Teile eines Ausdrucks zuerst berechnet werden.
  - Ohne Klammern kann es zu unerwarteten Ergebnissen kommen.

06. Rechnen & Ausgabe II: Konstanten, Rechenregeln, Formatierung

Code

```
public class RechnenAusgabeKonstantenRechenregelnFormatierung {

  public static void main(String[] args) {

    final double MWST = 0.19;  // Konstante: Mehrwertsteuer (19 %)
    double nettoPreis = 100.0;

    // Berechnungen
    double bruttoOhneKlammern = 1 + MWST * nettoPreis;    // Falscher Brutto-Wert
    double bruttoMitKlammern  = (1 + MWST) * nettoPreis;  // Richtiger Brutto-Wert

    // Ausgabe mit println
    System.out.println("Bruttoberechnung falsch:  " + bruttoOhneKlammern);
    System.out.println("Bruttoberechnung richtig: " + bruttoMitKlammern);

    System.out.println();

    // Formatierte Ausgabe mit printf
    System.out.printf("Bruttoberechnung richtig: %6.2f €%n", bruttoMitKlammern);
  }

}
```

Ausgabe (Beispiel)

```
Bruttoberechnung falsch:  20.0
Bruttoberechnung richtig: 119.0

Bruttoberechnung richtig: 119,00 €
```

Erläuterung

- Grundidee der Aufgabe
  - Java wertet Rechenausdrücke nach festen Regeln aus (Klammern zuerst, Punkt vor Strich).
  - Diese Regeln gelten immer, unabhängig davon, ob der Ausdruck fachlich sinnvoll ist.
  - In dieser Aufgabe wird gezeigt, dass:
    - ~ ein Ausdruck syntaktisch korrekt, aber fachlich falsch sein kann,
    - ~ Klammern notwendig sind, um die beabsichtigte Rechnung eindeutig festzulegen,
    - ~ die Ausgabe von Zahlwerten formatiert werden kann.

Konstanten mit final

- final double MWST = 0.19;
  - final kennzeichnet eine Konstante.
  - Der Wert darf nach der Initialisierung nicht mehr verändert werden.
  - Typische Anwendungsfälle: feste Faktoren (z. B. Steuersätze), physikalische Konstanten.
  - Konvention [= Übereinkunft] für die Schreibweise von Konstanten:
    - ~ Großbuchstaben (z. B. MWST)
    - ~ Einzelne Wörter mit Unterstrich trennen (z. B. MWST_SATZ)
  - Versucht man nach der Initialisierung einer Konstanten ihren Wert zu verändern, so

    - entsteht ein Compilerfehler:
        - MWST = 0.20; // Fehler: cannot assign a value to final variable MWST

Punkt vor Strich ohne Klammern

- double bruttoOhneKlammern = 1 + MWST * nettoPreis;
    - Java rechnet zuerst Punkt vor Strich:
      MWST * nettoPreis
    - anschließend wird 1 addiert.

- Mathematisch korrekt ausgewertet, aber fachlich falsch, da der Brutto-Wert so nicht berechnet wird.

Klammerrechnung als bewusste Steuerung

- double bruttoMitKlammern = (1 + MWST) * nettoPreis;
    - Die Klammern legen fest:
        1. Addition: 1 + MWST
        2. danach Multiplikation mit nettoPreis

- Das Ergebnis entspricht der fachlich korrekten Bruttoberechnung.
  → Klammern sind hier nicht optional, sondern notwendig.

Formatierte Ausgabe: println vs. printf

- println
    - Schnelle, einfache Standardausgabe.
    - Bei double: Ausgabe von so vielen Nachkommastellen, wie zur Darstellung des Wertes nötig sind (ohne Rundung, Formatierung, Ausrichtung).
    - Deshalb erscheint im Beispiel nur eine Nachkommastelle (20.0 bzw. 119.0).

- printf
    - Formatierte Ausgabe mit kontrollierter und einheitlicher Darstellung.
    - Besonders hilfreich bei Tabellen, ausgerichteten Zahlen, Geldbeträgen.

    - Bedeutung von z. B. %6.2f:
        - f → Ausgabe als Gleitkommazahl (double);
        - .2 → genau zwei Nachkommastellen (deshalb Ausgabe als 119,00 anstelle 119,0);
        - 6 → mindestens sechs Zeichen Gesamtbreite (Breitenangabe ist optional, z. B. ist %.2f möglich).

    - Dezimaltrennzeichen (Locale):
        - Die Darstellung des Dezimaltrennzeichens (Punkt oder Komma) hängt vom verwendeten Locale der Java Virtual Machine ab.
        - In einem deutschsprachigen Locale (z. B. Deutschland) wird ein Komma als Dezimaltrennzeichen ausgegeben.
        - In einem englischsprachigen Locale (z. B. USA) wird stattdessen ein Punkt verwendet.
        - In diesem Skript wird von einem deutschen Locale ausgegangen, daher erscheint in den Beispielen ein Komma (z. B. 119,00 €).
        - Das Locale kann bei printf auch explizit festgelegt werden, z. B.:
          System.out.printf(Locale.GERMANY, "%.2f%n", zahl);
          für ein deutsches Locale oder
          System.out.printf(Locale.US, "%.2f%n", zahl);
          für ein US-englisches Locale.

    - Nachkommastellen und Rundung:
        - Wenn der gespeicherte Wert mehr als zwei Nachkommastellen besitzt, wird er für die Ausgabe gerundet.
        - Beispiel:
          double zahl = 2.3456;
          System.out.printf("%.2f%n", zahl);
        - Ausgabe:
          2,35
        - Wichtig: Der Wert der Variable bleibt unverändert (wird nicht gerundet) – nur die Darstellung wird angepasst.
          Anm.: Zur Rundung von Variableninhalten siehe Aufgabenbeispiele unten.
        - %n erzeugt einen Zeilenumbruch; es ist keine Escape-Sequenz wie \n, sondern eine Formatangabe, die von printf plattformunabhängig als Zeilenumbruch interpretiert wird.

    - Gesamtbreite:
        - Umfasst die Ziffern vor und nach dem Dezimaltrennzeichen, das Dezimaltrennzeichen selbst sowie ggf. ein Vorzeichen.
        - Text vor und nach der Kommazahl zählt nicht zur Gesamtbreite, auch keine Währungszeichen.
        - Ist die durch printf auszugebende Zahl kürzer als die angegebene Gesamtbreite, wird in der Ausgabe links mit Leerzeichen aufgefüllt.

- Wichtige Erkenntnisse
    - Java kennt feste Operatorprioritäten (z. B. Punkt vor Strich).
    - Klammern steuern die Berechnung explizit.
    - Ein Ausdruck kann korrekt ausgewertet, aber fachlich falsch sein (→ logischer Programmierfehler; Näheres dazu folgt in den Programmbeispielen zum Thema Debugging unten).
    - Berechnungen sollten in separaten Ergebnis-Variablen gespeichert werden.
    - Variablen sind veränderliche Werte, Konstanten (mit final) sind feste Werte.
    - printf ermöglicht kontrolliert formatierte Ausgaben.

07. Scanner I: Einfache Benutzereingabe (Zahl)

Code

```
import java.util.Scanner;

public class ScannerZahl {

  public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);

    System.out.print("Geben Sie eine Zahl ein: ");
    int zahl = sc.nextInt();

    System.out.println("Sie haben eingegeben: " + zahl);
    sc.close();
  }

}
```

Ausgabe (Beispiel)

```
Geben Sie eine Zahl ein: 7                <7: Benutzer-Eingabe über die Tastatur>
Sie haben eingegeben: 7                   <Zeile komplett vom Programm ausgegeben>
```

Erläuterung

Anm.: Näheres zu Bibliotheken, Klassen und Objekten folgt später.

Einbinden der Scanner-Klasse

- import java.util.Scanner;
    - Bindet die Klasse Scanner aus der Java-Standardbibliothek ein.
    - Ohne diesen Import kann der Scanner nicht verwendet werden.

Erzeugen eines Scanner-Objekts

- Scanner sc = new Scanner(System.in);
    - Erstellt ein Scanner-Objekt mit dem Namen sc (der Name ist frei wählbar).
    - System.in steht für die Standardeingabe (Tastatur).
    - Über dieses Objekt können Eingaben des Benutzers gelesen werden.

Einlesen einer Zahl

- int zahl = sc.nextInt();
    - int ist der Datentyp für ganze Zahlen.
    - zahl speichert den vom Benutzer eingegebenen Wert.
    - sc.nextInt() liest eine ganze Zahl von der Tastatur ein.
    - Die Programmausführung wird an dieser Stelle angehalten, bis der Benutzer eine gültige Zahl eingibt und mit Enter bestätigt. Danach wird der gelesene Wert in der Variablen zahl gespeichert.

Ausgabe

- System.out.println("Sie haben eingegeben: " + zahl);
    - Text und Zahlenwert werden mit + zu einer gemeinsamen Ausgabe verknüpft.
    - Der Zahlenwert wird automatisch in einen String umgewandelt.

Schließen des Scanners

- sc.close();
    - Schließt den Scanner, da hier keine weiteren Eingaben benötigt werden.
    - Nach dem Schließen können über dieses Scanner-Objekt keine Eingaben mehr gelesen werden.
    - In einfachen Programmen ist das Schließen nicht zwingend erforderlich, gehört jedoch zu sauberem Programmierstil.

08. Scanner II: Zwei Benutzereingaben (Zahl und Text)

Code

```
import java.util.Scanner;

public class ScannerZahlText {

  public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);

    System.out.print("Bitte gib deinen Namen ein: ");
    String name = sc.nextLine();

    System.out.print("Bitte gib dein Alter ein: ");
    int alter = sc.nextInt();

    System.out.println();

    System.out.println("Hallo " + name + ", du bist " + alter + " Jahre alt.");
    sc.close();
  }

}
```

Ausgabe (Beispiel)

```
Bitte gib deinen Namen ein: Bob          <Bob: Benutzer-Eingabe über die Tastatur>
Bitte gib dein Alter ein: 42             <42: Benutzer-Eingabe über die Tastatur>

Hallo Bob, du bist 42 Jahre alt.         <Zeile komplett vom Programm ausgegeben>
```

Erläuterung

Einlesen eines Textes

- String name = sc.nextLine();
    - String ist der Datentyp für Texte.
    - name ist die Variable, in der die Benutzereingabe gespeichert wird.
    - sc.nextLine() liest eine komplette Textzeile von der Tastatur ein (ohne den Zeilenumbruch, der durch das Drücken der Enter-Taste zum Abschluss der Eingabe entsteht).

Einlesen einer Zahl

- int alter = sc.nextInt();
    - int ist der Datentyp für ganze Zahlen.
    - alter ist die Variable, in der die Benutzereingabe gespeichert wird.
    - nextInt() liest eine ganze Zahl ein.

Unterschiedliche Methoden für unterschiedliche Datentypen

- In diesem Beispiel werden zwei verschiedene Datentypen eingelesen:
    - ein Text (String) für den Namen,
    - eine Zahl (int) für das Alter.
- Der Scanner stellt dafür unterschiedliche Methoden bereit, z. B.
    - nextLine() für eine ganze Textzeile,
    - nextInt() für eine ganze Zahl.

Besonderheit bei der Kombination von nextInt() und nextLine()

- Wird in einem Programm zuerst nextInt() und danach nextLine() verwendet, kann es passieren, dass nextLine() keine neue Benutzereingabe einliest, sondern eine leere Zeichenkette liefert.
- Grund: nextInt() liest nur die Zahl, nicht aber den abschließenden Zeilenumbruch (Enter). Dieser verbleibt im Eingabepuffer.
- Ein anschließendes nextLine() liest dann lediglich diesen verbleibenden Zeilenumbruch ein – und nicht die eigentlich erwartete Texteingabe.
- Typische Abhilfe: Vor einem weiteren nextLine()-Aufruf wird häufig eine zusätzliche nextLine()-Anweisung eingefügt, um den verbliebenen Zeilenumbruch zu „leeren".

Ausgabe

- System.out.println("Hallo " + name + ", du bist " + alter + " Jahre alt.");
    - Mehrere Texte und Variablen werden mit + zu einer gemeinsamen Ausgabe verknüpft.
    - Der Zahlenwert wird automatisch in Text umgewandelt.

Schließen des Scanners

- sc.close();
    - Der Scanner wird geschlossen, da keine weiteren Eingaben benötigt werden.
    - Nach dem Schließen können über diesen Scanner keine weiteren Eingaben mehr gelesen werden.

09. Division I: Ganzzahl- und Kommadivision, Typstrenge

Code

```
public class DivisionGanzzahlKommadivisionTypstrenge {

  public static void main(String[] args) {

    int intErgebnis;
    double doubleErgebnis1;
    double doubleErgebnis2;

    intErgebnis     = 5 / 2;    // = 2
    doubleErgebnis1 = 5 / 2;    // = 2.0 (!)
    doubleErgebnis2 = 5 / 2.0;  // = 2.5

    System.out.println(intErgebnis);
    System.out.println(doubleErgebnis1);
    System.out.println(doubleErgebnis2);
  }

}
```

Ausgabe

```
2
2.0
2.5
```

Erläuterung

- Division in Java
    - Kommazahlen werden mit Punkt (.) geschrieben, nicht mit Komma.
    - Für Ganzzahl- und Kommadivision wird mit / derselbe Operator verwendet.
    - Java unterscheidet zwischen der Art der Division und der Speicherung/Darstellung des Ergebnisses.

Beispiel 1: Ganzzahldivision mit int-Zielvariable

- intZahl = 5 / 2;
    - Beide Operanden (5 und 2) sind vom Typ int.
    - Es wird eine Ganzzahldivision durchgeführt.
    - Nachkommastellen werden abgeschnitten.
    → Ergebnis: 2 (und nicht 2.0)

Beispiel 2: Ganzzahldivision mit double-Zielvariable

- doubleZahl1 = 5 / 2;
    - Auch hier sind beide Operanden int.
    - Auch hier wird eine Ganzzahldivision durchgeführt.
    - Das Ergebnis der Rechnung ist zunächst 2.
    - Erst danach wird das Ergebnis in den Datentyp der Ergebnisvariable doubleZahl1 umgewandelt.
    → Ergebnis: 2.0 (und nicht 2.5)

Beispiel 3: Kommadivision mit double-Zielvariable

- doubleZahl2 = 5 / 2.0;
    - Mindestens ein Operand ist vom Typ double.
    - Es wird eine Kommadivision durchgeführt.
    - Die Reihenfolge der Operandentypen spielt keine Rolle (5 / 2.0 oder 5.0 / 2).
    → Ergebnis: 2.5

- Java ist eine streng typisierte Programmiersprache.
- Der Datentyp der Operanden entscheidet über die Art der Division.
    - int / int ergibt immer int, z. B. 1 / 2 → 0
    - Min. ein double für eine Kommadivision, z. B.
        - 1 / 2.0 → 0.5
        - 1.0 / 2 → 0.5
        - 1.0 / 2.0 → 0.5
- Erst nach der Berechnung erfolgt ggf. eine Typumwandlung.
- Der Datentyp der Ergebnisvariable beeinflusst nur die Speicherung/Darstellung des Ergebnisses, nicht die Rechnung.

10. Division II: Explizite Typumwandlung (Cast)

Code

```
public class DivisionExpliziteTypumwandlungCast {

  public static void main(String[] args) {

    double doubleErgebnis1;
    double doubleErgebnis2;
    double doubleErgebnis3;

    doubleErgebnis1 = 5 / 2;           // = 2.0  (Ganzzahldivision, vgl. Div. I)
    doubleErgebnis2 = (double) 5 / 2;  // = 2.5  (Cast greift vor der Division)
    doubleErgebnis3 = 5 / (double) 2;  // = 2.5  (gl. Ergebnis, anderer Operand)

    System.out.println(doubleErgebnis1);
    System.out.println(doubleErgebnis2);
    System.out.println(doubleErgebnis3);
  }

}
```

Ausgabe

```
2.0
2.5
2.5
```

Erläuterung

- Cast [engl. „to cast" = (um-)formen, gießen)
    - Ein Cast ist eine explizite Typumwandlung, die vom Programmierer bewusst vorgenommen wird.
    - Mit einem Cast wird ein Wert *vor der Berechnung* in einen anderen Datentyp umgewandelt.
    - Syntax:
        - (zieltyp) ausdruck/wert
        - z. B. (double) 5

- Warum braucht man Casts?
    - Java ist streng typisiert und führt Rechnungen strikt nach den Typen der Operanden aus.
    - In manchen Situationen liefert Java daher ein formal korrektes, aber fachlich unerwünschtes Ergebnis.
    - Casts ermöglichen es, gezielt in eine Rechnung einzugreifen, ohne die beteiligten Variablen zu ändern.

Beispiel 1: Keine explizite Typumwandlung

- doubleErgebnis1 = 5 / 2;   // = 2.0
    - Beide Operanden sind int
    - Ganzzahldivision → Ergebnis der Rechnung: 2
    - Speicherung in double erfolgt erst danach
      → Ausgabe: 2.0

Beispiel 2: Cast vor der Division (linker Operand)

- doubleErgebnis2 = (double) 5 / 2;   // = 2.5
    - Der Wert 5 wird *vor der Division* in double umgewandelt
    - Die Rechnung lautet somit: 5.0 / 2
    - Mindestens ein Operand ist double
    - Damit wird eine Kommadivision durchgeführt
    → Ergebnis: 2.5

Beispiel 3: Cast vor der Division (rechter Operand)

- doubleErgebnis3 = 5 / (double) 2;   // = 2.5
    - Der Wert 2 wird vor der Division in double umgewandelt
    - Die Rechnung lautet somit: 5 / 2.0
    - Mindestens ein Operand ist double
    - Damit wird auch hier eine Kommadivision durchgeführt
      → Ergebnis: 2.5

- Wichtige Erkenntnisse
    - Entscheidend für die Art der Division ist der Datentyp der Operanden zum Zeitpunkt der Rechnung.
    - Ein Cast verändert gezielt den Datentyp eines Operanden *vor der Rechnung* und kann dadurch die Berechnung beeinflussen.
    - Es reicht aus, einen der Operanden zu casten:
        - Sobald mindestens ein Operand vom Typ double ist, wird eine Kommadivision durchgeführt.
        - Die Reihenfolge der Operanden spielt dabei keine Rolle ((double) 5 / 2 und 5 / (double) 2 liefern dasselbe Ergebnis).
    - Ein Cast auf die Ergebnisvariable erfolgt erst nach der Rechnung und beeinflusst nur die Speicherung bzw. Darstellung des Ergebnisses, nicht die Berechnung.
    - Casts sind ein bewusstes Eingreifen des Programmierers, um das Rechenverhalten gezielt zu steuern.

Typische Fehler beim Einsatz von Casts

Fehler 1: Cast auf die Ergebnisvariable statt auf die Rechnung

```
double x = (double) (5 / 2);  // Ergebnis: 2.0
```

- Sollte hier intendiert sein, dass eine Kommadivision durchgeführt wird (5 : 2 = 2,5), so ist der Code falsch.
- Die Division a / b wird zuerst als Ganzzahldivision ausgeführt → Ergebnis: 2
- Der Cast erfolgt auf dieses Ergebnis – und damit zu spät → Ergebnis: 2.0
- Lösung: Cast vor der Division durchführen:

```
double x = (double) 5 / 2;  // Ergebnis: 2.5
```

Fehler 2: Erwartung, dass der Cast „automatisch mitrechnet", z. B. rundet

```
int z = (int) 3.9; // Ergebnis: 3
```

- Sollte hier intendiert sein, dass durch die int-Typumwandlung die Kommazahl 3,9 in eine (kaufmännisch gerundete) 4 überführt wird (4 ist die näher an 3,9 liegende Ganzzahl als 3), so ist der Code falsch.
- Beim Cast von double zu int werden Nachkommastellen abgeschnitten, nicht gerundet.
- Runden ist kein Cast, sondern erfordert eigene Methoden (z. B. Math.round, s. u.).

11. Rundung I: Kaufmännisches Runden

Code

```
public class RundungRound {

  public static void main(String[] args) {

    double zahl1 = 10.0 / 3.0;
    System.out.println(zahl1);          // ohne Rundung

    double zahl2 = 3.6;
    long zahl2Gerundet = Math.round(zahl2);
    System.out.println(zahl2Gerundet);  // Rundung auf Ganzzahl

    double zahl3 = 3.14159;
    double zahl3Gerundet = Math.round(zahl3 * 100.0) / 100.0;
    System.out.println(zahl3Gerundet);  // Rundung auf 2 Nachkommastellen

    /*
    Oder kürzer/kombiniert:
    System.out.println(Math.round(zahl3 * 100.0) / 100.0);
    */
  }

}
```

Ausgabe

```
3.3333333333333335
4
3.14
```

Erläuterung

- Allgemeines zur Klasse Math
    - Math ist eine wichtige Klasse der Java-Standardbibliothek.
    - Sie enthält statische Methoden und Konstanten für mathematische Berechnungen, u. a. zu Rundung und Zufall (siehe dazu entsprechende Programmbeispiele an anderen Stellen).
    - Der Zugriff erfolgt immer über:
      Math.<name>(…)
    - Die meisten Math-Methoden liefern einen Wert vom Typ double.

- Rundung in Java
    - Java rundet Kommazahlen nicht automatisch.
    - Werte vom Typ double können sehr viele Nachkommastellen besitzen.
    - Für gezieltes Runden stellt Java verschiedene Methoden bereit.

Beispiel 1: Darstellung von Kommazahlen (ohne Rundung)

- double zahl1 = 10.0 / 3.0;
    - 10.0 / 3.0 ergibt eine periodische Dezimalzahl.
    - Diese kann nicht exakt dargestellt werden.
    - Java gibt daher eine Näherung aus: 3.3333333333333335.
    - Dies ist kein Fehler, sondern eine Eigenschaft des Datentyps double.

Beispiel 2: Rundung auf eine Ganzzahl

- long zahl2Gerundet = Math.round(zahl2);
    - Math.round(double) rundet eine Kommazahl auf die nächstgelegene ganze Zahl (kaufmännisches Runden).
    - Das Ergebnis ist vom Typ long. Damit ist sichergestellt, dass auch große gerundete double-Werte korrekt gespeichert werden können.
    - Im Beispiel:
      → 3.6 wird auf 4 gerundet.

Beispiel 3: Rundung auf zwei Nachkommastellen

- Math.round rundet immer auf eine ganze Zahl. Um auf eine bestimmte Anzahl von Nachkommastellen zu runden (hier: 2), wird eine Rechenhilfe verwendet.

- Schrittweise Vorgehensweise:
    1. Ausgangszahl:
        - double zahl3 = 3.14159;
    2. Verschieben der Nachkommastellen um zwei Stellen nach rechts (Multiplikation)
        - zahl3 * 100.0
        - 3.14159 → 314.159
    3. Runden auf die nächstgelegene ganze Zahl
        - Math.round(…)
        - 314.159 → 314
    4. Zurückverschieben der Nachkommastellen
        - / 100.0
        - 314 → 3.14

- Wichtige Erkenntnisse
    - Math.round rundet auf eine ganze Zahl.
    - Das Runden auf eine bestimmte Anzahl von Nachkommastellen erfolgt durch Multiplikation → Rundung → Division.
    - Diese Technik ist weit verbreitet und kommt ohne zusätzliche Java-Klassen aus.
    - Math.round ersetzt keinen Cast und umgekehrt: Casts verändern nur den Datentyp, Math.round verändert den Zahlenwert.

12. Rundung II: Abrunden, Aufrunden

Code

```
public class RundungFloorCeil {

  public static void main(String[] args) {

    double zahl1 = 3.7;
    double zahl2 = -3.7;

    System.out.println("Ausgangswerte:");
    System.out.println("zahl1 = " + zahl1);
    System.out.println("zahl2 = " + zahl2);

    System.out.println();

    // Abrunden (floor)
    System.out.println("Math.floor(3.7) = " + Math.floor(zahl1));
    System.out.println("Math.floor(-3.7) = " + Math.floor(zahl2));

    System.out.println();

    // Aufrunden (ceil)
    System.out.println("Math.ceil(3.7) = " + Math.ceil(zahl1));
    System.out.println("Math.ceil(-3.7) = " + Math.ceil(zahl2));

    System.out.println();

    // Vergleich mit round
    System.out.println("Math.round(3.7) = " + Math.round(zahl1));
    System.out.println("Math.round(-3.7) = " + Math.round(zahl2));
  }

}
```

Ausgabe

```
Ausgangswerte:
zahl1 = 3.7
zahl2 = -3.7

Math.floor(3.7) = 3.0
Math.floor(-3.7) = -4.0

Math.ceil(3.7) = 4.0
Math.ceil(-3.7) = -3.0

Math.round(3.7) = 4
Math.round(-3.7) = -4
```

Erläuterung

- Math.floor(x)
    - Rundet immer ab zur nächstkleineren ganzen Zahl.
    - „Kleiner" bezieht sich auf den Zahlenstrahl, nicht auf den Betrag.
    - Das Ergebnis ist vom Typ double.
    - Beispiele:
        - 3.7 → 3.0
        - −3.7 → −4.0

- Math.ceil(x)
    - Rundet immer auf zur nächstgrößeren ganzen Zahl.
    - Beispiele:
        - 3.7 → 4.0
        - −3.7 → −3.0

- Typische Anwendungsfälle
    - Raster- und Indexberechnungen
    - Zeitfenster, Seitenzahlen, Kachelberechnung
    - Begrenzen von Werten nach oben oder unten

Unterschied zwischen Math.round ↔ Math.floor / Math.ceil

- Math.round nimmt eine kaufmännische Rundung zur nächstgelegenen ganzen Zahl vor; der Ergebnistyp ist eine Ganzzahl (long).

- Math.floor und Math.ceil runden gerichtet, unabhängig von der Nachkommastelle; der Ergebnistyp ist eine Kommazahl (double) – auch wenn das Ergebnis wie eine ganze Zahl aussieht (z. B. 3.0).

13. Mathematische Konstanten und Funktionen

Code

```
public class MathematischeKonstantenFunktionen {

  public static void main(String[] args) {

    // Konstanten
    System.out.println("Math.PI = " + Math.PI);
    System.out.println("Math.E = " + Math.E);

    System.out.println();

    // Betrag und Vorzeichen
    System.out.println("Math.abs(-42)    = " + Math.abs(-42));
    System.out.println("Math.signum(-13) = " + Math.signum(-13));
    System.out.println("Math.signum(0)   = " + Math.signum(0));
    System.out.println("Math.signum(13)  = " + Math.signum(13));

    System.out.println();

    // Maximum / Minimum
    System.out.println("Math.max(42, 7) = " + Math.max(42, 7));
    System.out.println("Math.min(42, 7) = " + Math.min(42, 7));

    System.out.println();

    // Potenzen
    System.out.println("Math.pow(2, 6) = " + Math.pow(2, 6));

    System.out.println();

    // Wurzeln
    System.out.println("Math.sqrt(64) = " + Math.sqrt(64));
    System.out.println("Math.cbrt(64) = " + Math.cbrt(64));

    System.out.println();

    // Exponential- und Logarithmusfunktion
    System.out.println("Math.exp(1)       = " + Math.exp(1));
    System.out.println("Math.log(Math.E) = " + Math.log(Math.E));

    System.out.println();

    // Trigonometrische Funktionen (Winkel im Bogenmaß)
    System.out.println("Math.sin(Math.PI / 6) = " + Math.sin(Math.PI / 6));
    System.out.println("Math.cos(Math.PI / 3) = " + Math.cos(Math.PI / 3));
    System.out.println("Math.tan(Math.PI / 4) = " + Math.tan(Math.PI / 4));
  }

}
```

Ausgabe

```
Math.PI = 3.141592653589793
Math.E = 2.718281828459045

Math.abs(-42)    = 42
Math.signum(-13) = -1.0
Math.signum(0)   = 0.0
Math.signum(13)  = 1.0

Math.max(42, 7) = 42
Math.min(42, 7) = 7

Math.pow(2, 6) = 64.0

Math.sqrt(64) = 8.0
Math.cbrt(64) = 4.0

Math.exp(1)       = 2.718281828459045
Math.log(Math.E) = 1.0

Math.sin(Math.PI / 6) = 0.49999999999999994
Math.cos(Math.PI / 3) = 0.5000000000000001
Math.tan(Math.PI / 4) = 0.9999999999999999
```

Erläuterung

- Math.PI
    - Liefert die Kreiszahl π als double.
    - Der Wert ist konstant und mit 15 Dezimalstellen sehr genau gespeichert.
    - Verwendung z. B. in Kreisformeln oder trigonometrischen Berechnungen.

- Math.E
    - Liefert die Eulersche Zahl e als double.
    - Der Wert ist konstant und mit 15 Dezimalstellen sehr genau gespeichert.
    - Verwendung z. B. in natürlichen Logarithmen, Exponentialfunktionen oder Wachstums-/Zerfallsberechnungen.

- Math.abs(x)
    - Liefert den Betrag eines Wertes.
    - Das Ergebnis ist immer nicht negativ.
    - Geeignet z. B. zur Abstandsmessung oder Fehlerberechnung.

- Math.signum(x)
    - Liefert das Vorzeichen eines Wertes:
        - -1.0 bei negativem Wert
        - 0.0 bei 0
        - 1.0 bei positivem Wert
    - Nützlich zur Richtungs- oder Vorzeichenanalyse.

- Math.max(a, b) / Math.min(a, b)
    - Liefert den größeren bzw. kleineren von zwei Werten.
    - Typisch für Grenzwert- oder Vergleichsoperationen.

- Math.pow(a, b)
    - Berechnet Potenzen a^b.
    - Ergebnis ist immer double.
    - Für einfache Quadrate ist oft lesbarer und effizienter im Code zu schreiben: x * x.

- Math.sqrt(x) und Math.cbrt(x)
    - Quadratwurzel bzw. Kubikwurzel.
    - Math.sqrt(x) ist nur für x ≥ 0 sinnvoll.
    - Math.cbrt(x) funktioniert auch für negative Werte.

- Math.exp(x) und Math.log(x)
    - Math.exp(x) berechnet e^x.
    - Math.log(x) ist der natürliche Logarithmus (Basis e).
    - Math.log(Math.E) ergibt exakt 1.

- Trigonometrische Funktionen
    - Math.sin, Math.cos, Math.tan
    - Winkelangaben erfolgen im Bogenmaß (Radiant), nicht in Grad.
    - Typische Referenzwerte:
        - π / 2 → 90°
        - π / 4 → 45°
        - 0 → 0°
    - Hinweis: Kleine Rundungsabweichungen (z. B. bei tan) sind normal und kein Fehler.

14. Typumwandlung I: Numerische Typumwandlung (inkl. Cast)

Code

```
public class TypumwandlungZahlZuZahl {

  public static void main(String[] args) {

    int intZahl = 7;

    // Typumwandlung int -> double (ohne Cast)
    double doubleErgebnis = intZahl;

    // Typumwandlung double -> int (mit Cast)
    int intErgebnis = (int) doubleErgebnis;

    System.out.println("Originale Integer-Zahl:  " + intZahl);
    System.out.println("Typumwandlung ohne Cast: " + doubleErgebnis);
    System.out.println("Typumwandlung mit Cast:  " + intErgebnis);
  }

}
```

Ausgabe

```
Originale Integer-Zahl:  7
Typumwandlung ohne Cast: 7.0
Typumwandlung mit Cast:  7
```

Erläuterung

Anm.: Casts wurden bereits bei den Divisionsbeispielen verwendet (s. o.). Dieses Beispiel ordnet sie systematisch ein.

Grundidee

- In der Programmierung existiert ein grundsätzlicher Unterschied zwischen der Zahl 42 und dem Text „42":
    - Die Zahl 42 ist ein numerischer Wert,
    - der Text „42" eine bloße Zeichenfolge wie bspw. „xy" oder „?!".
  Für beide gibt es separate Datentypen.
- Von besonderem Interesse ist die Technik der Typumwandlung, d. h. das Überführen eines Wertes von einem Datentyp in einen anderen.
- Java kennt numerische Typumwandlungen zwischen primitiven Zahlentypen, die kompatibel sind (z. B. int → double, double → int).
- Dabei ändert sich nur der numerische Datentyp bzw. die Darstellung des Wertes (z. B. 42 → 42.0, 42.0 → 42), es entsteht kein Text (42 ≠ „42").
- Dieses Beispiel konzentriert sich auf die numerische Typumwandlung Zahl → Zahl.

Typumwandlung ohne Cast

- double doubleErgebnis = intZahl;
    - Automatische Typumwandlung zwischen zwei numerischen Datentypen (int nach double).
    - Ohne Cast, da keine explizite Typangabe im Programmcode verwendet wird.
    - Nur möglich bei sog. Typerweiterungen (z. B. int → double, char → int), also wenn kein Informationsverlust auftreten kann.
    - Die Zahl bleibt hier eine Zahl, lediglich die Nachkommastellen kommen hinzu.
    - Ergebnis: 7 → 7.0.

Typumwandlung mit Cast

- int intErgebnis = (int) doubleErgebnis;
    - Explizite Typumwandlung zwischen zwei numerischen Datentypen (double nach int).
    - Der Cast „(int)" im Programmcode erzwingt das Überführen des Wertes in einen kleineren Typ.
    - Sog. Typverengungen (z. B. double → int) erfordern stets einen expliziten Cast.
    - Auch hier bleibt die Zahl eine Zahl – lediglich die Nachkommastellen werden abgeschnitten.
    - Ergebnis: 7.0 → 7.

Wichtige Abgrenzung

- Numerische Typumwandlungen (inkl. Cast, als eine Spezialform davon) betreffen nur Zahlentypen.
- Umwandlungen von:
    - String → Zahl und
    - Zahl → String
  sind keine numerischen Typumwandlungen und kein Cast.
- Solche Umwandlungen erfolgen über Methoden (z. B. Integer.parseInt, valueOf) und werden in den folgenden Aufgaben behandelt.

15. Typumwandlung II: String zu Zahl (Parsing)

Code

```
public class TypumwandlungStringZuZahl {

  public static void main(String[] args) {

    String str = "42";

    // Typumwandlung String -> int
    int intErgebnis = Integer.parseInt(str);

    // Typumwandlung String -> double
    double doubleErgebnis = Double.parseDouble(str);

    System.out.println("Originaler String: " + str);
    System.out.println("Typumwandlung mit Integer.parseInt:    " + intErgebnis);
    System.out.println("Typumwandlung mit Double.parseDouble: " + doubleErgebnis);
  }

}
```

Ausgabe

```
Originaler String: 42
Typumwandlung mit Integer.parseInt:    42
Typumwandlung mit Double.parseDouble: 42.0
```

Erläuterung

- Strings können nicht direkt gecastet werden:
    - (int) "42" ist nicht erlaubt.
    - Strings sind Referenztypen auf Objekte; sie enthalten Text, keine Zahlen.
    - Lösung: Parsing

Typumwandlung durch Parsing

- Integer.parseInt(str)
    - Wandelt den Inhalt eines Strings in eine Ganzzahl um und liefert diese zurück.
    - Ist der String keine gültige Ganzzahl, erzeugt Java eine Fehlermeldung (Exception).
- Double.parseDouble(str)
    - Wandelt den Inhalt eines Strings in eine Kommazahl um und liefert diese zurück.
    - Ist der String keine gültige Ganz- oder Kommazahl, erzeugt Java eine Fehlermeldung (Exception).

Unterschied:
Numerische Typumwandlung (inkl. Cast) vs. Typumwandlung String → Zahl

- Numerische Typumwandlungen (inkl. Cast) ändern nur den Zahlentyp – es sind Zahlen und es bleiben Zahlen.
- Parsing bezeichnet die Umwandlung eines String (Text) in eine Zahl. Dabei handelt es sich nicht um bloßes „Umschreiben", sondern um eine inhaltliche Neuinterpretation.
- Die Umwandlung von Zahl → String ist ebenfalls eine Typumwandlung, aber kein Parsing, sondern eine gezielte Erzeugung einer Zeichenfolge (z. B. mit String.valueOf). Dies wird in der folgenden Aufgabe behandelt.

16. Typumwandlung III: Zahl zu String

Code

```
public class TypumwandlungZahlZuString {

  public static void main(String[] args) {

    double zahl = 13.5;

    // Typumwandlung Zahl -> String (durch String-Verkettung)
    String strErgebnis1 = "" + zahl;

    // Typumwandlung Zahl -> String (mit String.valueOf)
    String strErgebnis2 = String.valueOf(zahl);

    System.out.println("Originale Zahl: " + zahl);
    System.out.println("Typumwandlung durch String-Verkettung: " + strErgebnis1);
    System.out.println("Typumwandlung mit String.valueOf:      " + strErgebnis2);
  }

}
```

Ausgabe

```
Originale Zahl: 13.5
Typumwandlung durch String-Verkettung: 13.5
Typumwandlung mit String.valueOf:      13.5
```

Erläuterung

- Zahlen werden häufig als Text benötigt, was eine Umwandlung von Zahl → String erforderlich macht:
    - für Ausgaben,
    - für Verkettung mit anderen Strings,
    - für Weiterverarbeitung als Text.

Typumwandlung durch String-Verkettung

- "" + zahl
    - Nutzt die automatische String-Verkettung.
    - Automatisch bedeutet hier: ohne expliziten Methodenaufruf im Code – der Mechanismus ist jedoch ein anderer als bei der automatischen Typumwandlung numerischer Datentypen.
    - Der numerische Wert (Ganz- oder Kommazahl) wird implizit in String umgewandelt und an einen Leerstring "" angehängt.

Typumwandlung mit String.valueOf

- String.valueOf(zahl)
    - Wandelt eine Ganz- oder Kommazahl zuverlässig in einen String um.
    - Im Vergleich zur String-Verkettung ist dies die *empfohlene*, explizite Variante.
- Für Objekte existiert auch die Methode toString(), die ebenfalls eine textuelle Darstellung erzeugt. Da toString jedoch ein „Teil" von valueOf ist, wird hier nicht näher darauf eingegangen.

Wichtige Abgrenzung

- Bei der Umwandlung Zahl → Text handelt es sich weder um einen Cast, noch um Parsing.
- Es wird ein neuer String mit textueller Darstellung dieses Wertes erzeugt.

17. Typumwandlung IV: Zeichen / Zahl (inkl. Unicode)

Code

```
public class TypumwandlungZeichenZahl {

  public static void main(String[] args) {

    char unicodeZeichen = 'A';
    // Typumwandlung Zeichen -> Zahl (Code)
    int unicodeCode = (int) unicodeZeichen;

    System.out.println("Unicode-Zeichen: " + unicodeZeichen);
    System.out.println("Unicode-Code:    " + unicodeCode);

    System.out.println();

    unicodeCode = 64;
    // Typumwandlung Zahl (Code) -> Zeichen
    unicodeZeichen = (char) unicodeCode;

    System.out.println("Unicode-Code:    " + unicodeCode);
    System.out.println("Unicode-Zeichen: " + unicodeZeichen);
  }

}
```

Ausgabe

```
Unicode-Zeichen: A
Unicode-Code:    65

Unicode-Code:    64
Unicode-Zeichen: @
```

Erläuterung

Anm.: Diese Aufgabe knüpft an die Cast-Beispiele der numerischen Typumwandlung oben an und zeigt, dass auch char intern als numerischer Typ behandelt wird.

Grundidee

- Char speichert – wie bereits bekannt – einzelne Zeichen (Buchstaben, Ziffern, Sonderzeichen).
- In Java erfolgt die interne Darstellung eines char jedoch nicht textuell, sondern als numerischer Datentyp – ähnlich zu int und double.
- Das bedeutet: Jedes Zeichen wird intern als Zahl gespeichert.
- Diese Zahl entspricht einem fest definierten Zeichencode gemäß der Zeichentabelle (früher: ASCII-Tabelle, heute: modernere Unicode-Tabelle; Näheres siehe Exkurs unten).
- Deshalb sind Cast-Umwandlungen zwischen char und int möglich und sinnvoll.

Typumwandlung Zeichen → Zahl (Code)

- int unicodeCode = (int) unicodeZeichen;
    - Über Unicode ist allgemein festgelegt, dass das Unicode-Zeichen „A" durch den Unicode-Code 65 repräsentiert wird.
    - Durch den Cast (int) wird nicht gerundet und kein Inhalt interpretiert, sondern lediglich der numerische Code des Zeichens übernommen.
    - Ergebnis:
      „A" → 65

Typumwandlung Zahl (Code) → Zeichen

- unicodeZeichen = (char) unicodeCode;
    - Der Unicode-Code 64 entspricht dem Unicode-Zeichen „@".
    - Der Cast (char) interpretiert die Zahl als Zeichencode und liefert das zugehörige Zeichen.
    - Ergebnis:
      64 → „@"

Zusammenfassung

Umwandlung	Technik	Art der Umwandlung	Ergebnis
int → double	automatisch (ohne Cast)	numerische Typumwandlung (Typerweiterung)	Kommazahl
double → int	Cast	numerische Typumwandlung (Typverengung)	Ganzzahl
char → int	Cast	Typumwandlung über Zeichencode (numerisch)	Ganzzahl
int → char	Cast	Typumwandlung über Zeichencode (symbolisch)	Zeichen
String → Zahl	Parsing (Integer.parseInt / Double.parseDouble)	Typumwandlung mit inhaltlicher Interpretation	Ganz- oder Kommazahl
Zahl → String	automatisch (String-Verkettung) od. String.valueOf	Typumwandlung mit textueller Darstellung	Text

→ Cast, Parsing und String-Umwandlung sind unterschiedliche Techniken der Typumwandlung mit jeweils unterschiedlicher Bedeutung.

Exkurs: Zeichencodierung mit ASCII und Unicode

Grundproblem: Text im Computer

- Computer können intern nur Zahlen verarbeiten und speichern.
- Damit Texte dargestellt, gespeichert und übertragen werden können, müssen Zeichen eindeutig Zahlen zugeordnet werden.
- Diese Zuordnung nennt man Zeichencodierung – formal festgelegt und geordnet in einer Zeichentabelle.
- Ohne eine einheitliche Zeichencodierung gäbe es:
    - keine verlässliche Textdarstellung,
    - fehlerhafte Speicherung von Texten,
    - Probleme beim systemübergreifenden Austausch von Daten

ASCII – der historische Ursprung

- ASCII (American Standard Code for Information Interchange) ist eine der ältesten Zeichencodierungen.
- Umfasst 128 Zeichen (Codes 0–127), darunter lateinische Buchstaben, Ziffern, grundlegende Satz- und Steuerzeichen.
- Beispiele:
  „0" → 48, „@" → 64, „A" → 65, „a" → 97
- ASCII enthält aber keine Umlaute (ä, ö, ü), keine Akzente (é, ñ), keine nicht-lateinischen Schriften, keine mathematischen oder typografischen Symbole etc.
- Verschiedene Länder begannen, eigene Erweiterungen zu definieren.
- Folge: Durch die fehlende Standardisierung erzeugte derselbe Zeichencode auf verschiedenen Systemen unterschiedliche Zeichen („kaputte" Umlaute, unlesbare Textdateien etc.).

Unicode – eine einheitliche Lösung

- Unicode weist jedem Zeichen weltweit eine eindeutige Nummer zu.
- Enthalten sind: verschiedenste Alphabete, mathematische Zeichen, Sonderzeichen, Grafikzeichen, Emojis etc.
- ASCII ist vollständig in Unicode enthalten (Codes 0–127).
- Unicode wird als internationaler Standard gepflegt und beständig erweitert.
- Java verwendet intern Unicode: ein char repräsentiert einen Unicode-Zeichenwert, technisch als numerischer Wert gespeichert.
- Ein Cast zwischen char und int bedeutet daher weder Rechnen, noch Parsen, noch String-Umwandlung, sondern Zugriff auf den zugrunde liegenden Zeichencode oder dessen Interpretation als Zeichen.

Fazit

- Zeichen sind keine Texte, sondern numerisch codierte Einheiten.
- Zeichen und Zahlen wirken unterschiedlich, obwohl beide intern Zahlen sind.
- Eine einheitliche Codierung (z. B. Unicode) bildet die unsichtbare Grundlage jeder Textverarbeitung.
- Daraus ergibt sich auch die Möglichkeit, Zeichen miteinander zu vergleichen (z. B. „A" < „a", weil der ASCII-Code 65 < 97 ist).

18. String I: Länge, Inhaltsprüfung, Teilstring

Code

```
public class StringLaengeInhaltspruefungTeilstring {

  public static void main(String[] args) {

    String wort = "Programmieren";

    System.out.println("Länge:          " + wort.length());
    System.out.println("Enthält 'gram': " + wort.contains("gram"));
    System.out.println("Teilstring:     " + wort.substring(3, 7));
  }

}
```

Ausgabe

```
Länge:          13
Enthält 'gram': true
Teilstring:     gram
```

Erläuterung

Anm.: Näheres zu Wahrheitswerten (true/false) und Boolean folgt später.

- String wort = "Programmieren";
    - Deklariert eine String-Variable wort mit dem Text „Programmieren".

Beispiel 1: Länge eines Strings

- wort.length()
    - Liefert die Anzahl der Zeichen im String
    - Leerzeichen und Sonderzeichen zählen mit.
    - Ergebnis hier: 13 Zeichen.

Beispiel 2: Inhalt prüfen

- wort.contains("gram")
    - Prüft, ob wort den Teilstring „gram" enthält.
    - Liefert als Ergebnis einen Wahrheitswert (Boolean): true (= wahr) oder false (= falsch).
    - Hier ist „gram" Bestandteil von „Programmieren" → true.
    - Typische Verwendung:
        - Benutzereingaben prüfen,
        - Schlüsselwörter erkennen,
        - Vorbedingungen für weitere Verarbeitung prüfen.

Beispiel 3: Teilstring erzeugen

- wort.substring(3, 7)
    - Liefert einen neuen String aus einem Teil des ursprünglichen Strings.
    - Startindex: 3 (inklusive)
    - Endindex: 7 (exklusive, d. h. das Zeichen an Position 7 wird nicht mehr übernommen)
    - Java zählt Zeichen ab Index 0.
    - Ergebnis hier: "gram".

    ```
    String:  P r o g r a m m i e r e n
    Index:   0 1 2 3 4 5 6 7 ...
                   ↑       ↑
    ```

    - Typische Verwendung:
        - Wortteile extrahieren,
        - Texte weiterverarbeiten.

Fazit

- String-Methoden wie length(), contains() und substring() verändern den ursprünglichen String nicht.
- Sie liefern entweder Informationen (int, boolean) oder neue Strings zurück.

19. String II: Vergleich von Texten

Code

```
public class StringVergleich {

  public static void main(String[] args) {

    String str1 = "Test";
    String str2 = "test";

    System.out.println(str1.equals(str2));
    System.out.println(str1.equalsIgnoreCase(str2));
  }

}
```

Ausgabe

```
false
true
```

Erläuterung

- str1.equals(str2)
    - Vergleicht den Inhalt von String str1 mit str2.
    - Berücksichtigt Groß-/Kleinschreibung.
    - "Test" und "test" sind daher nicht gleich → false.

- str1.equalsIgnoreCase(str2)
    - Vergleicht den Inhalt von String str1 mit str2, aber ohne Berücksichtigung der Groß-/Kleinschreibung.
    - "Test" und "test" sind insofern gleich → true.

- Typische Verwendung:
    - Benutzereingaben auf bestimmte Texte prüfen.
    - Befehle erkennen, unabhängig von Groß-/Kleinschreibung ("start", "Start", "START").
    - Antworten wie Ja/Nein auswerten.

Unterschiede beim Vergleich von primitiven Datentypen und Strings

- Primitive Datentypen (int, double, boolean u. a.) speichern Werte (Inhalte) direkt. Man vergleicht sie mit ==.

```
int a = 5;
int b = 5;

System.out.println(a == b);  // liefert true
```

- Ein String ist ein sogenannter Referenztyp, d. h. die Variable speichert nicht den Text selbst, sondern einen Verweis („Referenz") auf den Text im Speicher.
- Daher prüft == bei String nicht den Textinhalt, sondern nur, ob zwei Variablen auf denselben Text im Speicher verweisen.

```
String str1 = new String("Hallo");
String str2 = new String("Hallo");

System.out.println(str1 == str2);  // liefert nicht true, sondern false
```

- Zum Vergleichen des Textinhalts von Strings muss equals() verwendet werden.

```
System.out.println(str1.equals(str2));  // liefert true
```

20. String III: Text aufbereiten und manipulieren

Code

```
public class StringAufbereitenManipulieren {

  public static void main(String[] args) {

    String text = "  Java ist toll  ";  // mit je 2 Leerzeichen vor und nach Text

    System.out.println("Original:        '" + text + "'");

    String bereinigt = text.trim();
    System.out.println("Nach trim:       '" + bereinigt + "'");

    String ersetzt = bereinigt.replace("toll", "spannend");
    System.out.println("Nach replace:    '" + ersetzt + "'");

    String gross = ersetzt.toUpperCase();
    System.out.println("Großbuchstaben:  '" + gross + "'");

    String klein = ersetzt.toLowerCase();
    System.out.println("Kleinbuchstaben: '" + klein + "'");

    /*
    Oder kürzer/kombiniert (nur letzte Ausgabe):
    System.out.println("Kleinbuchstaben: '" + ersetzt.toLowerCase() + "'");
    */
  }

}
```

Ausgabe

```
Original:        '  Java ist toll  '
Nach trim:       'Java ist toll'
Nach replace:    'Java ist spannend'
Großbuchstaben : 'JAVA IST SPANNEND'
Kleinbuchstaben: 'java ist spannend'
```

Erläuterung

- String text = " Java ist toll ";
    - Enthält absichtlich führende und nachgestellte Leerzeichen, wie sie z. B. bei Benutzereingaben häufig vorkommen.
    - Hochkommas in der Ausgabe machen Leerzeichen sichtbar.

Beispiel 1: Text bereinigen (trim)

- String bereinigt = text.trim();
    - Entfernt führende und nachgestellte Leerzeichen, Tabulatoren und Zeilenumbrüche.
    - Leerzeichen innerhalb des Textes bleiben erhalten.
    - Das Ergebnis ist ein neuer String und wird in bereinigt gespeichert.

Beispiel 2: Text ersetzen (replace)

- String ersetzt = bereinigt.replace("toll", "spannend");
    - Ersetzt alle Vorkommen des Teilstrings "toll" durch "spannend".
    - Das Ergebnis ist ein neuer String und wird in ersetzt gespeichert.

Beispiel 3: Groß-/Kleinschreibung (toUpperCase / toLowerCase)

- String gross = ersetzt.toUpperCase();
    - Wandelt alle Buchstaben in Großbuchstaben um.
    - Das Ergebnis ist ein neuer String und wird in gross gespeichert.

- String klein = ersetzt.toLowerCase();
    - Wandelt alle Buchstaben in Kleinbuchstaben um.
    - Das Ergebnis ist ein neuer String und wird in klein gespeichert.

- System.out.println("Kleinbuchstaben: '" + ersetzt.toLowerCase() + "'");
    - Exemplarisch eine verkürzte Schreibweise, die die String-Variable klein obsolet macht.

- Typische Verwendung:
    - Vereinheitlichung von Benutzereingaben,
    - Vorbereitung für Vergleiche (equalsIgnoreCase),
    - Normalisierung von Texten.

Fazit

- `Strings sind unveränderlich:`
    - Keine String-Methode verändert den ursprünglichen String.

  - Jede Manipulation erzeugt einen neuen String, der explizit gespeichert werden muss.
- String-Methoden lassen sich schrittweise kombinieren, z. B. erst bereinigen, dann prüfen, dann manipulieren.

21. String IV: Textteile entfernen („löschen“)

Code

```
public class StringTextteileEntfernen {

  public static void main(String[] args) {

    String meldung = "Fehler: Datei nicht gefunden!";

    System.out.println("Original: " + meldung);

    System.out.println();

    // Entfernen über Positionen (substring)
    String ohnePraefix1 = meldung.substring(8);
    System.out.println("Ohne Präfix (substring): " + ohnePraefix1);

    // Entfernen über Inhalt (replace)
    String ohnePraefix2 = meldung.replace("Fehler: ", "");
    System.out.println("Ohne Präfix (replace):   " + ohnePraefix2);
  }

}
```

Ausgabe

```
Original: Fehler: Datei nicht gefunden!

Ohne Präfix (substring): Datei nicht gefunden!
Ohne Präfix (replace):   Datei nicht gefunden!
```

Erläuterung

- Grundidee: „Löschen“ bei Strings
  - In Java gibt es keine eigene Methode zum „Löschen“ von Zeichen in einem String.
  - Strings sind unveränderlich: Der ursprüngliche Text bleibt immer erhalten.
  - „Löschen“ (besser: Entfernen) bedeutet daher immer:
    - ~ bestimmte Zeichen nicht übernehmen oder
    - ~ sie durch einen leeren String (sog. Leerstring "") ersetzen.
    - ~ Das Ergebnis ist ein neuer String.
- Hier werden zwei bereits bekannte String-Methoden in einem neuen Kontext verwendet:
  - substring und replace,
  - jedoch nicht zum klassischen Extrahieren oder Ersetzen,
  - sondern gezielt zum Entfernen von Textteilen.
- String meldung = "Fehler: Datei nicht gefunden!";
  - Enthält eine typische Status- oder Fehlermeldung mit festem Präfix („Fehler: “) und eigentlicher Information.
  - Solche Texte kommen häufig in Konsolenprogrammen, Log-Ausgaben oder Fehlermeldungen vor.

Beispiel 1: Textteil über Position entfernen (substring)

- meldung.substring(8);
  - Liefert einen neuen String *ab* der angegebenen Startposition 8.
  - Der Index 8 entspricht der Position direkt *nach* dem Präfix "Fehler: ".
  - Der ursprüngliche String bleibt unverändert.

```
String:  F e h l e r :   D a t e i ...
Index:   0 1 2 3 4 5 6 7 8 ...
                         ↑
```

  - Ergebnis:
    Datei nicht gefunden!
- Diese Variante ist besonders geeignet, wenn:
  - der Aufbau des Textes fest bekannt ist,
  - bestimmte Informationen immer an derselben Position beginnen.

Beispiel 2: Textteil über Inhalt entfernen (replace)

- meldung.replace("Fehler: ", "")
  - Entfernt *alle* Vorkommen des angegebenen Teilstrings.
  - Hier wird das Präfix „Fehler: “ gezielt durch einen leeren String ersetzt und faktisch „gelöscht“ (entfernt).
  - Auch hier entsteht ein neuer String, der explizit gespeichert werden muss.
  - Ergebnis:
    Datei nicht gefunden!
- Diese Variante ist besonders geeignet, wenn:
  - der zu entfernende Text inhaltlich bekannt ist,
  - die genaue Position keine Rolle spielen soll,
  - alle Vorkommen des Teilstrings entfernt werden sollen.

22. Variablen tauschen

Code

```
public class VariablenTauschen {

  public static void main(String[] args) {

    // Tausch von int-Werten
    int a = 42;
    int b = 13;

    int tempInt = a;
    a = b;
    b = tempInt;

    System.out.println("a = " + a);
    System.out.println("b = " + b);

    System.out.println();

    // Tausch von String-Werten
    String str1 = "Hallo";
    String str2 = "Welt";

    String tempString = str1;
    str1 = str2;
    str2 = tempString;

    System.out.println("str1 = " + str1);
    System.out.println("str2 = " + str2);
  }

}
```

Ausgabe

```
a = 13
b = 42

str1 = Welt
str2 = Hallo
```

Erläuterung

- Grundidee der Aufgabe
  - Zwei Variablen sollen ihre Inhalte miteinander tauschen.
  - Nach dem Tausch besitzt jede Variable den Wert, der zuvor in der jeweils anderen gespeichert war.
  - Da Zuweisungen nacheinander ausgeführt werden, benötigt man eine zusätzliche Variable zur Zwischenspeicherung.
- Beteiligte Datentypen
  - Im Beispiel werden sowohl primitive Ganzzahlen (int) als auch Referenztypen (String) vertauscht.
  - Das zugrunde liegende Vorgehen ist in beiden Fällen identisch.
- Prinzip des Tauschens
  - Der ursprüngliche Wert der ersten Variable wird vorübergehend gespeichert.
  - Danach erhält die erste Variable den Wert der zweiten.
  - Anschließend bekommt die zweite Variable den zuvor gesicherten ursprünglichen Wert der ersten.
  - Ohne Zwischenspeicher würde eine Information verloren gehen.
- Tausch der int-Werte im Detail
  - int tempInt = a;
    Der aktuelle Wert von a wird in tempInt gesichert.
  - a = b;
    a erhält nun den bisherigen Wert von b.
  - b = tempInt;
    b bekommt den ursprünglich in a gespeicherten Wert.
- Variablenbelegungstabelle
  - Um Programme besser zu verstehen, kann man nach jeder Anweisung festhalten, welchen Wert jede Variable besitzt.
  - Dadurch wird sichtbar, wie sich der Zustand des Programms Schritt für Schritt verändert.
  - Diese Methode ist besonders hilfreich zur Fehlersuche, wenn mehrere Zuweisungen aufeinander folgen oder Werte überschrieben werden.
  - Ablauf im Beispiel Schritt für Schritt:

Schritt	a	b	tempInt
Start	42	13	–
int tempInt = a;	42	13	42
a = b;	13	13	42
b = tempInt;	13	42	42

  - Man erkennt: Der ursprüngliche Wert von a bleibt nur deshalb erhalten, weil er zuvor zwischengespeichert wurde.
- Tausch der String-Werte
  - Obwohl int und String unterschiedliche Datentypen sind, funktioniert das Tauschen nach demselben Prinzip.
  - Entscheidend ist die Reihenfolge der Zuweisungen.
  - Bei String werden dabei keine Texte kopiert, sondern Referenzen neu gesetzt.
- Typische Fehlerquelle
  - Eine scheinbar einfache Lösung wie
    a = b;
    b = a;
    führt nicht zum gewünschten Ergebnis.
  - Nach der ersten Zuweisung ist der ursprüngliche Wert von a bereits überschrieben und kann nicht mehr wiederhergestellt werden.

23. If-Anweisung I: Einfache Bedingung

Code

```
public class IfEinfach {

  public static void main(String[] args) {

    int alter = 17;

    if (alter >= 18) {
      System.out.println("Volljährig");
    }
  }

}
```

Ausgabe

```
<keine Ausgabe>
```

Erläuterung

- Die if-Anweisung prüft eine Bedingung und führt den Code im if-Block nur dann aus, wenn diese Bedingung wahr ist.
- if (alter >= 18)
  - Die Bedingung prüft, ob das Alter mindestens 18 Jahre beträgt.
  - Der Vergleich ergibt entweder true oder false.
- In diesem Beispiel:
  - alter = 17
  - 17 >= 18 ist falsch
  → der Code im if-Block wird übersprungen, daher keine Ausgabe.
- Eine einfache if-Anweisung steuert also, ob Code ausgeführt wird – nicht, was alternativ passieren soll.

24. If-Anweisung II: Verzweigte Bedingung

Code

```
public class IfVerzweigt {

  public static void main(String[] args) {

    int alter = 17;

    if (alter >= 18) {
      System.out.println("Volljährig");
    } else {
      System.out.println("Nicht volljährig");
    }
  }

}
```

Ausgabe

```
Nicht volljährig
```

Erläuterung

- Die Aufgabe erweitert die einfache if-Anweisung um einen else-Block.
- Eine if-else-Anweisung modelliert eine zweiwertige Entscheidung:
  - entweder wird der if- Block ausgeführt
  - oder (als Gegenfall) der else- Block.
- if (alter >= 18)
  - Ist die Bedingung wahr, wird der if-Block ausgeführt.
- else
  - Der else-Block wird ausgeführt, wenn die Bedingung falsch ist.
  - Damit ist sichergestellt, dass immer nur einer der beiden Codeblöcke ausgeführt wird.
- In diesem Beispiel:
  - alter = 17
  - 17 >= 18 ist falsch
  → der Code im else-Block wird ausgeführt („Nicht volljährig“).

Code-Strukturierung: Klammerung und Einrückung (Teil 1)

- Geschweifte Klammern { } fassen zusammengehörige Anweisungen zu Codeblöcken zusammen. Sie definieren die Struktur des Programms (Klasse, Methode, if, else).
- In diesem Code sind mehrere Codeblöcke ineinander verschachtelt. Insgesamt gibt es vier geöffnete { und vier geschlossene }, mit folgender hierarchischer Struktur:
  1. äußerster Block: die Klasse IfVerzweigt
  2. darin enthalten: die Methode main
  3. darin enthalten: der if-Block
  4. auf derselben Ebene wie der if-Block: der else-Block
- Einrückung
  - Die Einrückung von Code-Blöcken (häufig um 2 Leerzeichen oder einen Tabulator) macht diese Struktur für Menschen besser lesbar. Sie ist ein wesentlicher Bestandteil sauber notierten Programmcodes.
  - Für den Compiler zählen jedoch ausschließlich die Klammern, nicht die Einrückung.

25. If-Anweisung III: Verschachtelte Bedingungen

Code

```
public class IfVerschachtelt {

  public static void main(String[] args) {

    int alter = 17;

    if (alter >= 18) {
      System.out.println("Volljährig");
    } else {
      if (alter >= 16) {
        System.out.println("Teilweise erlaubt");
      } else {
        System.out.println("Nicht erlaubt");
      }
    }
  }

}
```

Ausgabe

```
Teilweise erlaubt
```

Erläuterung

- In diesem Beispiel steht eine if-Anweisung innerhalb eines else-Blocks, wodurch sich mehrere if- und else-Befehle ergeben.
- Man spricht von verschachtelten Bedingungen, wenn Entscheidungen stufenweise getroffen werden.
- Das Programm prüft das Alter in mehreren Schritten:
  - Erste Bedingung:
    - ~ if (alter >= 18)
    - ~ Ist die Bedingung wahr, wird „Volljährig“ ausgegeben und das Programm endet.
  - Zweite Bedingung (nur wenn die erste Bedingung falsch ist):
    - ~ else → if (alter >= 16)
    - ~ Ist diese Bedingung wahr, wird „Teilweise erlaubt“ ausgegeben.
    - ~ Andernfalls wird „Nicht erlaubt“ ausgegeben.
- In diesem Beispiel:
  - alter = 17
  - 17 >= 18 ist falsch → else-Block
  - 17 >= 16 ist wahr
  → Ausgabe „Teilweise erlaubt“.

Code-Strukturierung: Klammerung und Einrückung (Teil 2)

- Ein else-Block gehört immer genau zu dem nächststehenden if, das noch kein else besitzt.
- Im Beispiel gehört das erste else zur ersten If-Anweisung und das zweite else zur zweiten if-Anweisung.
- Korrekte Einrückung ist hier besonders wichtig, um die logische Struktur sichtbar zu machen.
- Falsch gesetzte oder fehlende Klammern können zu schwer erkennbaren Logikfehlern führen, obwohl der Code syntaktisch korrekt ist – z. B. dass ein else einem anderen if zugeordnet wird als beabsichtigt.
- Empfehlung:
  - Korrekte Einrückungen und Klammerung mit { } immer verwenden – auch dann, wenn ein Block nur eine einzige Anweisung enthält.
  - Dadurch wird der Code leichter zu verstehen und kann mit geringerer Fehleranfälligkeit erweitert werden.

26. Switch-Anweisung I: Klassische Variante mit case und break

Code

```
public class SwitchKlassisch {

  public static void main(String[] args) {

    int note = 2;

    switch (note) {
      case 1:
        System.out.println("Sehr gut");
        break;
      case 2:
        System.out.println("Gut");
        break;
      case 3:
      case 4:
        System.out.println("Mittlerer Notenbereich");
        break;
      default:
        System.out.println("Minderleistung");
    }
  }

}
```

Ausgabe

```
Gut
```

Erläuterung

- switch (note)
  - Prüft den Wert der Variablen note und vergleicht ihn mit den definierten case-Werten.
- case 1: / case 2:
  - Jeder case entspricht einem möglichen Wert von note.
  - Trifft ein Wert zu, werden die zugehörigen Anweisungen ausgeführt.
- break
  - Beendet die switch-Anweisung nach dem aktuell ausgeführten case.
  - Ohne break würden auch die folgenden case-Blöcke ausgeführt werden (Fall-Through).
- case 3: / case 4:
  - Mehrere case-Anweisungen können für einen Programmzweig zusammengefasst werden.
  - Treffen note == 3 oder note == 4 zu, wird der gemeinsame Codeblock für case 3 und case 4 ausgeführt.
  - Fall-Through
    - ~ Da zwischen case 3 und case 4 kein break steht, „fällt“ die Ausführung zum nächsten case weiter.
    - ~ Dieses Verhalten ist hier gewollt, um mehrere Werte gleich zu behandeln.
- default
  - Wird ausgeführt, wenn keiner der case-Werte zutrifft.
  - Dieser Block ist optional, aber sinnvoll, um alle anderen Werte abzudecken.
- Vergleich zu if/else:
  - switch ist eine Alternative zu mehreren if-else-Verzweigungen.
  - if/else kann komplexe Bedingungen prüfen (z. B. >, <, >=, &&).
  - switch prüft meist exakte Gleichheit auf einen bestimmten Wert (int, String, char usw.).
  - switch ist übersichtlicher, wenn viele Werte nur auf Gleichheit geprüft werden. Bei logischen Verknüpfungen oder nur zwei bis drei einfachen Bedingungen ist if/else sinnvoller.

27. Switch-Anweisung II: Moderne Variante mit Pfeilsyntax

Code

```
public class SwitchModern {

  public static void main(String[] args) {

    int note = 2;

    String text = switch (note) {
      case 1 -> "Sehr gut";
      case 2 -> "Gut";
      case 3, 4 -> "Mittlerer Notenbereich";
      default -> "Minderleistung";
    };  // Abschluss der Zuweisung mit Semikolon!

    System.out.println(text);
  }

}
```

Ausgabe

```
Gut
```

Erläuterung

- switch (note)
  - Auch hier wird der Wert von note geprüft und einem passenden Fall zugeordnet.
- Pfeilsyntax (->)
  - Ersetzt die klassische Variante mit case: und break.
  - Im Falle von case 3, 4 werden hier mehrere Werte explizit in einem case zusammengefasst – und nicht über Fall-Through wie im klassischen switch.
  - Jeder case liefert genau einen Wert zurück.
- switch als Ausdruck
  - Der switch wird hier nicht nur zur Steuerung des Programmablaufs verwendet, sondern als Ausdruck, der einen Wert zurückgibt.
  - Dieser Wert wird der Variablen text zugewiesen.
  - Im Gegensatz zum klassischen switch muss der moderne daher – als gesamte Zuweisung – mit einem Semikolon ; abgeschlossen werden.
- default
  - Liefert einen Rückgabewert für alle nicht explizit behandelten Fälle.
  - Erforderlich, wenn in den cases zuvor nicht alle möglichen Fälle abgedeckt sind.

Vergleich: moderner switch vs. klassischer switch

- Vorteile des modernen switch:
  - Kein break notwendig
  - Kein Fall-Through möglich
  - Genau ein Rückgabewert pro case
  - Kürzer, sicherer und besser lesbar
- Der moderne switch ist nicht bloß eine andere Schreibweise, denn:
  - klassischer switch:
    → steuert Kontrollfluss (führt Anweisungen aus)
  - moderner switch:
    → ist ein Ausdruck und liefert einen Wert zurück
- Der moderne switch eignet sich besonders dann, wenn aus einer Auswahl ein konkreter Wert ermittelt werden soll.
- Empfohlene Variante für neuen Java-Code (ab Java 14).

28. Boolean I: Wahrheitswerte, Negation

Code

```
public class BooleanWahrheitswerteNegation {

  public static void main(String[] args) {

    boolean istAktiv = true;

    if (istAktiv) {
      System.out.println("Ist aktiv");
    }

    istAktiv = false;

    if (istAktiv) {
      System.out.println("Ist aktiv");
    }

    boolean istFertig = false;

    if (!istFertig) {
      System.out.println("Noch nicht fertig");
    }
  }

}
```

Ausgabe

```
Ist aktiv
<keine Ausgabe>
Noch nicht fertig
```

Erläuterung

- Boolean
  - Der Datentyp boolean kann genau zwei Werte annehmen:
    true („wahr“) oder false („falsch“).
- boolean istAktiv = true;
  - Die boolean-Variable istAktiv wird mit dem Wahrheitswert true initialisiert.
- if (istAktiv)
  - Eine if-Anweisung erwartet als Bedingung einen boolean-Ausdruck.
  - Der Ausdruck istAktiv wird direkt ausgewertet:
    - ~ ist istAktiv == true, wird der Codeblock ausgeführt,
    - ~ ist istAktiv == false, wird er übersprungen.
  - In Java ist es üblich, boolean-Variablen direkt als Bedingung zu verwenden und nicht explizit mit == true zu vergleichen. Schreibweisen wie if (istAktiv == true) sind daher zu vermeiden.
  - Da istAktiv hier den Wert true hat, wird „Ist aktiv“ ausgegeben.
- `istAktiv = false; und zweite Abfrage`
  - Die Variable istAktiv wird auf false gesetzt.
  - Die anschließende if (istAktiv)-Abfrage ist damit falsch.
  - Der zugehörige Codeblock wird *nicht* ausgeführt, es erfolgt keine Ausgabe.
- if (!istFertig)
  - Der Operator ! negiert einen boolean-Ausdruck:
    - ~ aus true wird false,
    - ~ aus false wird true.
  - Da istFertig den Wert false hat, wird durch !istFertig der Wert true.
  - Die Bedingung ist damit erfüllt und „Noch nicht fertig“ wird ausgegeben.
- Hinweis zur Lesbarkeit
  - Bedingungen sollten möglichst ohne unnötige Negationen formuliert werden.
  - Insbesondere Konstruktionen mit doppelter Verneinung (z. B. !istFertig == false) sind zwar korrekt, erschweren aber das Verständnis.
  - Gut gewählte Variablennamen und einfache Bedingungen verbessern die Lesbarkeit des Programmcodes.

29. Boolean II: Logisches UND (&&)

Code

```
public class BooleanLogischesUnd {

  public static void main(String[] args) {

    String land = "Deutschland";
    double bestellwert = 75.0;

    if (land.equals("Deutschland") && bestellwert >= 50) {
      System.out.println("Versandkostenfrei");
    } else {
      System.out.println("Versandkosten fallen an");
    }
  }

}
```

Ausgabe

```
Versandkostenfrei
```

Erläuterung

- Logischer Operator && (UND)
  - Der logische Operator && steht für UND.
  - Ein UND-Ausdruck ist nur dann true (wahr), wenn alle beteiligten Bedingungen true sind.
  - Logische Operatoren wie && ermöglichen es, mehrere Bedingungen übersichtlich in einer einzigen if-Anweisung zu kombinieren.
- if (land.equals("Deutschland") && bestellwert >= 50)
  - In dieser Bedingung werden zwei einzelne Ausdrücke miteinander verknüpft.
  - land.equals("Deutschland")
    - ~ Prüft, ob der Inhalt der Zeichenkette land genau „Deutschland“ ist.
    - ~ Der Vergleich erfolgt mit equals(...), da Strings in Java nicht mit == verglichen werden.
  - bestellwert >= 50
    - ~ Prüft, ob der Bestellwert mindestens 50 beträgt.
  - Beide Bedingungen des UND-Ausdrucks werden ausgewertet:
    - ~ land.equals("Deutschland") → true
    - ~ bestellwert >= 50 → true
    - ~ Da beide Teilbedingungen true sind, ist auch der gesamte UND-Ausdruck true.
    - ~ Der if-Block wird ausgeführt und „Versandkostenfrei“ ausgegeben.
- Alternativer Fall
  - Ist mindestens eine der beiden Bedingungen false (falsch), ist der gesamte UND-Ausdruck false.
  - In diesem Fall wird der else-Block ausgeführt und „Versandkosten fallen an“ ausgegeben.

30. Boolean III: Logisches ODER (||)

Code

```
public class BooleanLogischesOder {

  public static void main(String[] args) {

    int alter = 6;
    boolean ferien = true;

    if (alter < 7 || ferien) {
      System.out.println("Eintritt kostenlos");
    } else {
      System.out.println("Eintritt kostenpflichtig");
    }
  }

}
```

Ausgabe

```
Eintritt kostenlos
```

Erläuterung

- Logischer Operator || (ODER)
  - Der logische Operator || steht für ODER.
  - Ein ODER-Ausdruck ist true (wahr), wenn mindestens eine der beteiligten Bedingungen true ist.
- if (alter < 7 || ferien)
  - In dieser Bedingung werden zwei einzelne Ausdrücke miteinander verknüpft.
  - alter < 7
    - ~ Prüft, ob das Alter kleiner als 7 ist.
  - ferien
    - ~ Eine boolean-Variable, die direkt als Bedingung ausgewertet wird.
  - Beide Bedingungen des ODER-Ausdrucks werden ausgewertet:
    - ~ alter < 7 → true
    - ~ ferien → true
    - ~ Da mindestens eine der Bedingungen true ist, ist der gesamte ODER-Ausdruck true.
    - ~ Der if-Block wird ausgeführt und „Eintritt kostenlos“ ausgegeben.
- Alternativer Fall
  - Nur wenn beide Bedingungen falsch (false) sind, ist der ODER-Ausdruck insgesamt falsch.
  - In diesem Fall wird der else-Block ausgeführt und „Eintritt kostenpflichtig“ ausgegeben.

Exkurs: Wahrheitstabellen zu logischen Operatoren (!, &&, ||)

- Allgemeines
  - Logische Operatoren verknüpfen oder verändern Wahrheitswerte (true / false).
  - Sie bilden die Grundlage für Entscheidungen in if-Anweisungen, Schleifen und komplexeren Bedingungen.
  - Die folgenden Wahrheitstabellen zeigen das Verhalten der wichtigsten logischen Operatoren systematisch.
- Logische Negation (!)
  - Der Negationsoperator kehrt den Wahrheitswert eines Ausdrucks um.

Ausdruck	!Ausdruck
false	**true**
true	**false**

- Logisches UND (&&)
  - Der UND-Operator verknüpft zwei Bedingungen.
  - Der gesamte Ausdruck ist nur dann wahr, wenn beide Teilbedingungen wahr sind.

Bedingung A	Bedingung B	A && B
false	false	**false**
false	true	**false**
true	false	**false**
true	true	**true**

- Logisches ODER (||)
  - Der ODER-Operator verknüpft ebenfalls zwei Bedingungen.
  - Der gesamte Ausdruck ist wahr, wenn mindestens eine der Bedingungen wahr ist.

Bedingung A	Bedingung B	A || B
false	false	**false**
false	true	**true**
true	false	**true**
true	true	**true**

  - ODER vs. XOR
    - ~ Das logische ODER (||) ist nicht exklusiv: Wenn beide Bedingungen wahr sind, ist der Ausdruck trotzdem wahr.
    - ~ Es handelt sich also nicht um ein entweder-oder.
    - ~ In der Logik existiert zwar ein sog. exklusives ODER (XOR), das nur dann wahr ist, wenn genau eine Bedingung wahr ist. Dieses spielt in den Java-Grundlagen und in typischen `if`-Bedingungen jedoch kaum eine Rolle.
- Hinweis zur Lesbarkeit
  - Kombinierte logische Operatoren können die Lesbarkeit stark verringern.
  - Klammern helfen, komplexe Bedingungen eindeutig zu strukturieren.
  - Im Zweifel ist es besser, komplexe Bedingungen in mehrere einfache, gut lesbare Abfragen aufzuteilen.

31. Modulo I: Gerade und ungerade Zahlen

Code

```
public class ModuloGeradeUngerade {

  public static void main(String[] args) {

    int zahl = 7;

    if (zahl % 2 == 0) {
      System.out.println("Zahl ist gerade");
    } else {
      System.out.println("Zahl ist ungerade");
    }
  }

}
```

Ausgabe

```
Zahl ist ungerade
```

Erläuterung

- Modulo-Operator %
  - Der Operator % liefert den Rest einer ganzzahligen Division (auch wenn der Dividend negativ ist).
  - Beispiele:
    - ~ 10 % 5 = 0 (10 : 5 = 2, Rest 0)
    - ~ 10 % 6 = 4 (10 : 6 = 1, Rest 4)
    - ~ -10 % 6 = -4 (-10 : 6 = 1, Rest -4)
  - Er wird häufig verwendet, um Teilbarkeiten oder wiederkehrende Muster zu überprüfen.
- zahl % 2 == 0
  - Prüft, ob der Rest der Division durch 2 gleich 0 ist.
    - ~ Rest 0 → Zahl ist durch 2 teilbar und damit gerade.
    - ~ Rest 1 → Zahl ist nicht durch 2 teilbar und damit ungerade.
  - Im Beispiel:
    - ~ 7 % 2 = 1
    - ~ Die Bedingung zahl % 2 == 0 ist false (falsch).
    → der else-Block wird ausgeführt
    → Ausgabe: „Zahl ist ungerade“

32. Modulo II: Teilbarkeitsprüfung durch 6

Code

```
public class ModuloTeilbarkeitspruefungDurchSechs {

  public static void main(String[] args) {

    int zahl = 18;
    boolean istDurch6Teilbar;

    istDurch6Teilbar = (zahl % 2 == 0) && (zahl % 3 == 0);

    if (istDurch6Teilbar) {
      System.out.println("Zahl ist durch 6 teilbar");
    } else {
      System.out.println("Zahl ist nicht durch 6 teilbar");
    }

    /*
    Oder kürzer/kombiniert:
    if ((zahl % 2 == 0) && (zahl % 3 == 0)) { …
    */
  }

}
```

Ausgabe

```
Zahl ist durch 6 teilbar
```

Erläuterung

- Das Programm prüft, ob eine Zahl durch 6 teilbar ist.
- Teilbarkeitsregel für 6: Eine Zahl ist genau dann durch 6 teilbar, wenn sie
  - durch 2 teilbar ist und
  - durch 3 teilbar ist.
- istDurch6Teilbar = (zahl % 2 == 0) && (zahl % 3 == 0);
  - Modulo-Prüfungen
    - ~ zahl % 2 == 0 prüft die Teilbarkeit durch 2
    - ~ zahl % 3 == 0 prüft die Teilbarkeit durch 3
  - Beide Bedingungen werden mit dem logischen UND (&&) verknüpft.
  - Das Ergebnis der Teilbarkeitsprüfung wird in der boolean-Variable istDurch6Teilbar gespeichert.
  - Eine Initialisierung von istDurch6Teilbar ist nicht nötig, da ihr vor der Verwendung in der if-Anweisung in jedem Fall ein Wert zugewiesen wird.
  - Im Beispiel:
    - ~ 18 % 2 == 0 → true (wahr)
    - ~ 18 % 3 == 0 → true
    - ~ Beide Bedingungen sind erfüllt.
    - ~ istDurch6Teilbar erhält den Wert true.
    → Der if-Block wird ausgeführt
    → Ausgabe: „Zahl ist durch 6 teilbar“
- Hinweis zur Struktur
  - Berechnung (logische Prüfung) und Ausgabe sind klar voneinander getrennt.
  - Dadurch bleibt der Code übersichtlich und leicht erweiterbar.

33. Schleifen I: for-Schleife

Code

```
public class SchleifeFor {

  public static void main(String[] args) {

    for (int i = 1; i <= 3; i++) {
      System.out.println(i);
    }
  }

}
```

Ausgabe

```
1
2
3
```

Erläuterung

- Grundidee der Aufgabe
  - Eine Schleife ermöglicht es, einen Programmblock mehrfach auszuführen.
  - Die for-Schleife wird besonders dann eingesetzt, wenn bereits vor dem Start klar ist, wie viele Durchläufe stattfinden sollen, z. B.:
    - ~ Gib die Zahlen von 1 bis 100 aus;
    - ~ Erzeuge 42 Zufallszahlen;
    - ~ Durchlaufe alle Elemente eines Arrays über ihre Indizes von 0 bis zum letzten gültigen Index (Näheres s. u.).
  - Im Beispiel wird eine Zählvariable verwendet, die die Werte 1 bis 3 annimmt und bei jedem Durchlauf ausgegeben wird.
- Aufbau der for-Schleife
  - for (<Initialisierung>; <Bedingung>; <Veränderung>) {
      <Schleifenrumpf>
    }
  - Im Beispiel (nur Schleifenkopf):
    for (int i = 1; i <= 3; i++)
  - Lesbare Übersetzung: „Führe die Befehle innerhalb der for-Schleife aus, solange i kleiner oder gleich 3 ist und erhöhe i nach jedem Schleifendurchlauf um 1.“
- int i = 1
  - Hier wird die sog. Zählvariable erzeugt und initialisiert.
  - Sie erhält ihren Startwert i = 1, bevor die Schleife zum ersten Mal betreten wird.
  - Die Variable existiert nur innerhalb der Schleife (lokaler Gültigkeitsbereich).
- i <= 3
  - Vor jedem Schleifendurchlauf prüft Java die Schleifenbedingung.
  - Nur wenn sie den Wahrheitswert true liefert, wird der Schleifenblock ausgeführt.
  - Sobald die Bedingung false wird, endet die Schleife sofort.
  - Hier läuft die Schleife also nur, solange i höchstens 3 ist.
- i++
  - Wird nach jedem vollständigen Schleifendurchlauf ausgeführt.
  - i++ ist die Kurzform für:
    i = i + 1;
  - Dadurch erhöht sich der Wert der Zählvariable schrittweise nach jedem Schleifendurchlauf um 1.
- Ablauf Schritt für Schritt (Variablenbelegungstabelle):

Durchlauf	Wert von i vor Prüfung	Bedingung i <= 3	Ausgabe	Wert von i nach i++
1	1	true	1	2
2	2	true	2	3
3	3	true	3	4
4	4	false	–	–

  Beim vierten Prüfzeitpunkt ist die Bedingung falsch → die Schleife endet nach drei Durchläufen.
- Typische Fehlerquellen
  - Falscher Startwert (z. B. 0 statt 1)
  - Falsche Vergleichsoperatoren (< statt <=)
  - Fehlende oder falsche Veränderung der Zählvariable
    → führt häufig zu Endlosschleifen (Näheres s. u.).

34. Schleifen II: while-Schleife

Code

```
public class SchleifeWhile {

  public static void main(String[] args) {

    int i = 3;

    while (i > 0) {
      System.out.println(i);
      i--;
    }
  }

}
```

Ausgabe

```
3
2
1
```

Erläuterung

- Grundidee der Aufgabe
  - Die while-Schleife wird eingesetzt, wenn die Anzahl der Schleifendurchläufe vorher nicht bekannt ist und man nur unter bestimmten Bedingungen überhaupt die Schleife ausführen möchte, z. B.
    - ~ Wiederhole eine Eingabe, bis der Benutzer einen gültigen Wert eingegeben hat;
    - ~ Führe Berechnungen aus, solange noch Daten vorhanden sind.
  - Statt einer festen Zählspanne entscheidet allein eine Bedingung, ob ein weiterer Durchlauf stattfindet.
  - Im Beispiel wird eine Variable verwendet, die rückwärts zählt, solange ihr Wert größer als 0 ist.
- Aufbau der while-Schleife
  - while (<Bedingung>) {
      <Schleifenrumpf>
    }
  - Im Beispiel (nur Schleifenkopf):
    while (i > 0)
  - Lesbare Übersetzung: „Solange i größer als 0 ist, führe die Befehle innerhalb der while-Schleife aus.“
- int i = 3;
  - Anders als bei der for-Schleife erfolgt die Erzeugung und Initialisierung der Zählvariable außerhalb des Schleifenkopfes.
  - Der Startwert (hier: i = 3) wird gesetzt, bevor die erste Prüfung der Bedingung stattfindet.
- i > 0
  - Vor jedem Schleifendurchlauf prüft Java die Schleifenbedingung.
  - Nur wenn sie den Wahrheitswert true liefert, wird der Schleifenblock ausgeführt.
  - Sobald die Bedingung false wird, endet die Schleife sofort.
  - Hier läuft die Schleife also nur, solange i positiv ist.
- i--
  - Die while-Schleife besitzt keine automatische Schrittweite.
  - Deshalb muss die Veränderung der Zählvariable im Schleifenblock selbst erfolgen.
  - i-- ist die Kurzform für:
    i = i - 1;
  - Dadurch wird i nach jedem Durchlauf um 1 verringert.
- Ablauf Schritt für Schritt (Variablenbelegungstabelle):

Durchlauf	Wert von i vor Prüfung	Bedingung i > 0	Ausgabe	Wert von i nach i--
1	3	true	3	2
2	2	true	2	1
3	1	true	1	0
4	0	false	–	–

  Beim vierten Prüfzeitpunkt ist die Bedingung falsch → die Schleife endet nach drei Durchläufen.
- Besonderheit der while-Schleife
  - while ist „kopfgesteuert“: Die Bedingung wird *vor* dem Schleifendurchlauf geprüft.
  - Ist sie bereits zu Beginn false, wird der Schleifenblock kein einziges Mal ausgeführt.
  - Damit sind auch 0 Durchläufe möglich.
  - Allgemein ergibt sich die Anzahl der Durchläufe erst während der Programmausführung.
- Typische Fehlerquellen
  - Startwert und Bedingung passen nicht zusammen
    → möglicherweise 0 Durchläufe.
  - Die Zählvariable wird nicht verändert.
  - Trotz Veränderung der Zählvariable wird die Bedingung immer erfüllt
    → führt zu Endlosschleifen.

35. Schleifen III: do-while-Schleife

Code

```
public class SchleifeDoWhile {

  public static void main(String[] args) {

    int i = 0;

    do {
      System.out.println(i);
    } while (i != 0);  // Semikolon!
  }

}
```

Ausgabe

```
0
```

Erläuterung

- Grundidee der Aufgabe
  - Die do-while-Schleife wird eingesetzt, wenn die Anzahl der Schleifendurchläufe vorher nicht bekannt ist, der Schleifenrumpf jedoch mindestens einmal ausgeführt werden soll, z. B.
    - ~ Menüs, die mindestens einmal angezeigt werden sollen;
    - ~ Eingaben (z. B. Passwörter), die mindestens einmal erfolgen müssen;
    - ~ Führe eine Spielrunde aus und frage danach, ob erneut gespielt werden soll.
  - Wie bei der while-Schleife entscheidet eine Bedingung darüber, ob ein weiterer Durchlauf stattfindet.
  - Der Unterschied besteht im Zeitpunkt der Prüfung: nicht am Anfang der Schleife, sondern am Ende.
  - Im Beispiel wird eine Variable ausgegeben, obwohl die Bedingung bereits zu Beginn nicht erfüllt ist.
- Aufbau der do-while-Schleife
  - do {
      <Schleifenrumpf>
    } while (<Bedingung>);
  - Beachte: Da die Schleife nicht nach der geschweiften Klammer } endet, bedarf es eines Semikolons ; zum Abschluss.
  - Im Beispiel (nur Schleifenkopf, bei do-while am Ende platziert):
    while (i != 0);
  - Lesbare Übersetzung: „Führe die Befehle innerhalb der do-while-Schleife aus, solange i ungleich 0 ist.“
- int i = 0;
  - Die Zählvariable wird – wie bei der while-Schleife – außerhalb des Schleifenkopfes initialisiert.
  - Der Startwert (hier: i = 0) wird gesetzt, bevor der erste Durchlauf beginnt.
- i != 0
  - Diese Bedingung am Schleifenende entscheidet, ob nach einem Durchlauf ein weiterer folgt.
  - Liefert sie true, wird der Schleifenrumpf erneut ausgeführt.
  - Liefert sie false, endet die Schleife.
  - Im Beispiel läuft die Schleife nur weiter, solange i nicht 0 ist.
  - Die Zählvariable wird hier nicht verändert. Grundsätzlich könnte innerhalb des Schleifenrumpfs jedoch z. B. mit i++ oder i-- gearbeitet werden.
- Ablauf Schritt für Schritt (Variablenbelegungstabelle):

Durchlauf	Wert von i vor Prüfung	Ausgabe	Bedingung i != 0
1	0	0	false

  Da die Bedingung bereits nach dem ersten Durchlauf false ist, endet die Schleife sofort.
- Besonderheit der do-while-Schleife
  - do-while ist „fußgesteuert“: Die Bedingung wird *nach* dem Schleifendurchlauf geprüft.
  - Deshalb wird der Schleifenrumpf immer mindestens einmal ausgeführt – unabhängig vom Startwert.
  - Im Gegensatz zur while-Schleife sind 0 Durchläufe *nicht* möglich.
  - Allgemein ergibt sich die Anzahl der Durchläufe wie bei der while-Schleife erst während der Programmausführung.
- Typische Fehlerquellen
  - Man übersieht, dass die Schleife mindestens einmal durchlaufen wird.
  - Siehe Erläuterungen zur while-Schleife.

Gemeinsames Grundprinzip aller Schleifen

- Unabhängig von ihrer konkreten Schreibweise folgt in Java jede Schleife derselben Logik:
  → Die Ausführung wird fortgesetzt, solange die Bedingung den Wert true liefert.
- Der Unterschied zwischen den Schleifen besteht daher nicht in der Bedeutung der Bedingung, sondern im Zeitpunkt ihrer Prüfung (also der Platzierung im Code).

Wann verwendet man welche Schleife?

- Ein Beispiel soll die Unterschiede verdeutlichen: Ein Roboter soll immer wieder eine Stufe erklimmen, solange er oben nicht gegen eine Wand stößt.
- Die auszuführende Aktion lautet: ErklimmeStufe();
- Der Code wird in **Pseudocode** dargestellt. Pseudocode beschreibt Programmabläufe in vereinfachter, sprachunabhängiger Form und dient dazu, die Struktur eines Algorithmus verständlich darzustellen, ohne sich auf konkrete Syntaxregeln einer Programmiersprache festzulegen.

Schleife	Einsatz / Funktionsweise	Beispiel in Pseudocode	Wirkung im Beispiel

for	• zählbare Wiederholungen
	• Startwert, Endwert und Schrittgröße sind vor Beginn bekannt; Prüfung vor dem Durchlauf + Zählmechanismus
	• Anzahl ist von Anfang an festgelegt (z. B. von 1 bis n)
		for (int i = 0; i < anzahlStufen; i++) {
		  ErklimmeStufe();
		}
			der Roboter steigt eine vorher festgelegte Anzahl von Stufen

while	• Wiederholung solange Bedingung true ist
	• kopfgesteuert: Bedingung wird *vor* jedem Durchlauf geprüft (d. h. auch kein Durchlauf möglich)
	• Anzahl ergibt sich während der Ausführung (0 bis n)
		while (keineWand) {
		  ErklimmeStufe();
		}
			der Roboter steigt nur, wenn der Weg frei ist

do-while	• Wiederholung solange Bedingung true ist • fußgesteuert: Bedingung wird *nach* jedem Durchlauf geprüft (d. h. min. 1 Durchlauf) • Anzahl ergibt sich während der Ausführung (1 bis n)	do { ErklimmeStufe(); } while (keineWand);	der Roboter steigt mindestens eine Stufe, selbst wenn direkt eine Wand folgt

- Bewertung für das gezeigte Beispiel
  - Die Anzahl der möglichen Stufen ist im Vorfeld nicht bekannt.
  - Ob ein weiterer Schritt erlaubt ist, hängt allein vom Zustand keineWand ab.
  - Es kann außerdem sein, dass bereits zu Beginn eine Wand vorhanden ist.
  - Für dieses Szenario ist daher die while-Schleife besonders geeignet.
  - Wäre dagegen die Anzahl der Stufen vorher bekannt, würde sich eher eine for-Schleife anbieten.
  - Sollte mindestens ein Schritt garantiert ausgeführt werden (z. B. Start auf einer Stufe mit späterer Prüfung), wäre eine do-while-Schleife passend.
  - Die Wahl der Schleife hängt also stets vom Problemkontext ab.

while als universelle Schleife

- Theoretisch lässt sich jede for- und jede do-while-Schleife auch mit einer while-Schleife formulieren, da diese lediglich eine Bedingungsprüfung benötigt.
- Praktisch kann die Umsetzung jedoch umständlicher und weniger übersichtlich sein, insbesondere bei Konstruktionen, bei denen mindestens ein Durchlauf garantiert sein soll.

for-each-Schleife

Näheres zu einer speziellen Variante der for-Schleife – die for-each-Schleife –, die vor allem zum Durchlaufen von Arrays verwendet wird, folgt weiter unten.

36. Schleifen IV: Verschachtelung for/for

Code

```
public class SchleifeVerschachteltForFor {

  public static void main(String[] args) {

    for (int i = 1; i <= 3; i++) {
      for (int j = 1; j <= 3; j++) {
        System.out.print(i * j + " ");
      }
      System.out.println();
    }
  }

}
```

Ausgabe

```
1 2 3
2 4 6
3 6 9
```

Erläuterung

- Grundidee der Aufgabe
  - Schleifen können innerhalb anderer Schleifen stehen.
  - Dabei wird die innere Schleife für jeden Durchlauf der äußeren Schleife vollständig ausgeführt.
  - Auf diese Weise lassen sich z. B. Tabellen, Matrizen oder Raster erzeugen.
  - Im Beispiel entsteht eine kleine Einmaleins-Tabelle.
- Rollen der Zählvariablen
  - Die äußere Schleife mit der Zählvariable i bestimmt die Zeilen.
  - Die innere Schleife mit der Zählvariable j bestimmt die Spalten.
  - Für jede Zeile werden alle Spaltenwerte neu berechnet.
- Was wird berechnet?
  - In jedem inneren Durchlauf wird das Produkt i × j gebildet.
  - Das Ergebnis wird mit System.out.print() in derselben Zeile ausgegeben.
  - Nach Abschluss der inneren Schleife sorgt System.out.println() für einen Zeilenumbruch.
- Ablauf der Verschachtelung
  - Die äußere Schleife startet mit i = 1.
  - Danach läuft die innere Schleife mit j = 1 bis 3 vollständig durch.
  - Erst dann wird i erhöht.
  - Anschließend beginnt die innere Schleife erneut bei j = 1.
  - Für jeden festen Wert von i nimmt j somit nacheinander alle Werte an.
  - Man kann sich vorstellen: Die äußere Schleife bewegt sich zeilenweise, die innere spaltenweise.
- Ablauf Schritt für Schritt (Variablenbelegungstabelle):

Durchlauf	äußere Schleife (i)	innere Schleife (j)	Berechnung	Ausgabe
1	1	1	1 • 1	1
2	1	2	1 • 2	2
3	1	3	1 • 3	3
4	2	1	2 • 1	2
5	2	2	2 • 2	4
6	2	3	2 • 3	6
7	3	1	3 • 1	3
8	3	2	3 • 2	6
9	3	3	3 • 3	9

37. Schleifen V: Verschachtelung for/while

Code

```
public class SchleifeVerschachteltForWhile {

  public static void main(String[] args) {

    for (int i = 1; i <= 3; i++) {
      int j = 1;

      while (j <= 3) {
        System.out.print(i * j + " ");
        j++;
      }

      System.out.println();
    }
  }

}
```

Ausgabe

```
1 2 3
2 4 6
3 6 9
```

Erläuterung

- Grundidee der Aufgabe
  - Die Struktur entspricht der vorherigen Aufgabe: Es wird erneut eine kleine Einmaleins-Tabelle erzeugt.
  - Der Unterschied liegt diesmal nicht im Ergebnis, sondern in der Art der Steuerung der inneren Schleife.
  - Außen arbeitet weiterhin eine for-Schleife, innen nun eine while-Schleife.
- Rollen der Zählvariablen
  - Die äußere Zählvariable i bestimmt wie zuvor die Zeilen.
  - Die innere Zählvariable j bestimmt die Spalten.
  - Für jede Zeile werden alle Spaltenwerte vollständig neu durchlaufen.
- Wichtiger Unterschied zur for/for-Variante
  - Eine while-Schleife besitzt keine Initialisierung und keine automatische Veränderung im Schleifenkopf.
  - Deshalb muss die Zählvariable vor der Schleife gesetzt werden.
  - Ebenso muss sie im Schleifenrumpf verändert werden.
- int j = 1;
  - Vor jedem Start der inneren Schleife wird j neu auf 1 gesetzt.
  - Das ist notwendig, damit jede Zeile wieder bei der ersten Spalte beginnt.
- j++
  - Nach jedem Durchlauf wird j um 1 erhöht.
  - Ohne diese Änderung würde die Bedingung j <= 3 dauerhaft true bleiben.
  - Die Schleife würde dann nie enden → Endlosschleife.

- Was bleibt gleich?
  - Für jeden festen Wert von i läuft die innere Schleife vollständig durch.
  - Erst dann wird i erhöht.
  - Das Ablaufprinzip der Verschachtelung bleibt also identisch zur vorherigen Aufgabe.

- Warum ist das Beispiel sinnvoll?
  - Es zeigt, dass unterschiedliche Schleifenarten miteinander kombiniert werden können.
  - Das Ergebnis kann gleich bleiben, auch wenn sich die technische Umsetzung ändert.
  - Allerdings steigt mit gemischten Schleifenarten häufig die Fehleranfälligkeit.

Fazit zu verschachtelten Schleifen

- Die innere Schleife wird für jeden Wert der äußeren Schleife vollständig ausgeführt.
- Typische Zählvariablen heißen i, j, k usw.
- Verschachtelte Schleifen eignen sich besonders für zweidimensionale Strukturen wie Tabellen, Matrizen oder Raster.
- Kombinationen verschiedener Schleifenarten sind möglich, aber nicht immer die übersichtlichste Lösung. Für einfache Anwendungsbeispiele wie obige Multiplikationstabelle ist for-in-for meist die bevorzugte Wahl.
- Gesamtzahl der Durchläufe = Anzahl der Durchläufe äußere Schleife × innere Schleife (hier: 3 • 3 = 9 Durchläufe).
- Ein häufiger Fehler sind Endlosschleifen, wenn die Zählvariable nicht korrekt verändert wird.

38. Schleifen VI: break

Code

```
public class SchleifeBreak {

  public static void main(String[] args) {

    for (int i = 0; i < 5; i++) {
      if (i == 3) break;
      System.out.println(i);
    }
  }

}
```

Ausgabe

```
0
1
2
```

Erläuterung

- Grundidee der Aufgabe
  - Normalerweise endet eine Schleife, wenn ihre Bedingung false wird.
  - Mit break kann eine Schleife jedoch sofort verlassen werden – unabhängig davon, wie viele Durchläufe eigentlich noch vorgesehen waren.
  - Der Abbruch erfolgt unmittelbar an der Stelle, an der break ausgeführt wird.

- Position von break im Beispiel
  - Innerhalb der Schleife wird in jedem Durchlauf geprüft, ob i == 3 gilt.
  - Trifft diese Bedingung zu, wird break ausgeführt.

- Auswirkung auf den Ablauf
  - Für i = 0, 1 und 2 wird System.out.println(i) normal ausgeführt.
  - Sobald i den Wert 3 erreicht, wird durch break die Schleife sofort beendet.
  - Obwohl die Schleifenbedingung i < 5 lautet, werden alle weiteren geplanten Werte (3 und 4) nicht mehr ausgegeben.

- Warum ist das Beispiel sinnvoll?
  - Es demonstriert, wie break die reguläre Abbruchlogik übersteuert.
  - Im Java-Codedesign ist dies ein gangbarer Weg; zugleich ist es in einfachen Fällen stilistisch oft sauberer, die Bedingung direkt im Schleifenkopf anzupassen, hier z. B. mit for (int i = 0; i < 3; i++).
  - Typisch ist der Einsatz von break in while-, do-while- oder verschachtelten Schleifen, um bei einer bestimmten Bedingung (z. B. gefundenes Suchergebnis oder Fehlerfall) sofort auszusteigen.

39. Schleifen VII: continue

Code

```
public class SchleifeContinue {

  public static void main(String[] args) {

    for (int i = 0; i < 5; i++) {
      if (i == 2) continue;
      System.out.println(i);
    }
  }

}
```

Ausgabe

```
0
1
3
4
```

Erläuterung

- Grundidee der Aufgabe
  - Normalerweise wird innerhalb eines Schleifendurchlaufs jede Anweisung im Schleifenrumpf nacheinander ausgeführt.
  - Mit continue kann der aktuelle Durchlauf jedoch vorzeitig beendet werden.
  - Die Schleife selbst läuft anschließend mit dem nächsten geplanten Wert weiter.

- Position von continue im Beispiel
  - In jedem Durchlauf wird geprüft, ob i == 2 gilt.
  - Ist dies der Fall, wird continue ausgeführt.

- Auswirkung auf den Ablauf
  - Für i = 0 und i = 1 wird System.out.println(i) normal ausgeführt.
  - Sobald i den Wert 2 erreicht, wird durch continue die Ausgabe übersprungen.
  - Danach setzt die Schleife mit dem nächsten Wert (i = 3) fort.
  - Auch 3 und 4 werden wieder regulär ausgegeben.

- Warum ist das Beispiel sinnvoll?
  - Es demonstriert, dass im Gegensatz zu break die Schleife nicht beendet wird.
  - Stattdessen wird lediglich der aktuelle Durchlauf abgebrochen.
  - continue kann in allen Schleifenarten verwendet werden; in verschachtelten Schleifen wirkt es nur auf die Schleife, in der es direkt steht.
  - Typisch ist der Einsatz von continue, wenn bestimmte Elemente oder Bedingungen (z. B. ungültige bzw. unerwünschte Fälle) übersprungen, die Schleife aber nicht vollständig beendet werden soll.

40. Gültigkeitsbereich von Variablen

Code

```
public class GueltigkeitsbereichVariablen {

  public static void main(String[] args) {

    int a = 1;

    // if-Block
    if (a == 1) {
      int b = 2;
      System.out.println("Ausgabe in if-Block:   a = " + a);
      System.out.println("Ausgabe in if-Block:   b = " + b);
    }

    System.out.println("Ausgabe nach if-Block: a = " + a);
    // System.out.println("Ausgabe nach if-Block: b = " + b);  // Fehler!

    System.out.println();

    // switch-Block
    switch (a) {
      case 1:
        int c = 3;
        System.out.println("Ausgabe in switch-Block:   a = " + a);
        System.out.println("Ausgabe in switch-Block:   c = " + c);
        break;
    }

    System.out.println("Ausgabe nach switch-Block: a = " + a);
    // System.out.println("Ausgabe nach switch-Block: c = " + c);  // Fehler!

    System.out.println();

    // Schleife
    for (int i = 0; i < 1; i++) {
      int d = 4;
      System.out.println("Ausgabe in for-Block:   a = " + a);
      System.out.println("Ausgabe in for-Block:   d = " + d);
    }

    System.out.println("Ausgabe nach for-Block: a = " + a);
    // System.out.println("Ausgabe nach for-Block: d = " + d);  // Fehler!
    // System.out.println("Ausgabe nach for-Block: i = " + i);  // Fehler!
  }

}
```

Ausgabe

```
Ausgabe in if-Block:   a = 1
Ausgabe in if-Block:   b = 2
Ausgabe nach if-Block: a = 1            <Ausgabe von b würde Fehler produzieren>

Ausgabe in switch-Block:   a = 1
Ausgabe in switch-Block:   c = 3
Ausgabe nach switch-Block: a = 1        <Ausgabe von c würde Fehler produzieren>

Ausgabe in for-Block:   a = 1
Ausgabe in for-Block:   d = 4
Ausgabe nach for-Block: a = 1      <Ausgabe von d bzw. i würde Fehler produzieren>
```

Erläuterung

- Grundidee der Aufgabe
  - Wie bereits in einem Programmbeispiel zu Kapitelanfang erklärt, ist der Gültigkeitsbereich (engl. Scope) von Variablen nur innerhalb des Blocks, in dem sie deklariert wurden.
  - Ein Block ist der Bereich zwischen { und }.
  - if, switch und Schleifen enthalten jeweils eigene innere Blöcke.

Variable im äußeren Block

- int a = 1;
  - a wird im Block der main-Methode deklariert.
  - Deshalb kann a überall innerhalb dieser Methode verwendet werden:
    - ~ im if-Block,
    - ~ im switch,
    - ~ in der Schleife,
    - ~ und auch danach verwendet werden.
  - Äußere Variablen sind in inneren Blöcken sichtbar.

Variablen in inneren Blöcken

- Beispiele oben:
  - int b = 2; // im if
  - int c = 3; // im switch
  - int d = 4; // in der Schleife
  - int i = 0; // Schleifenvariable

- Diese Variablen existieren nur innerhalb ihres jeweiligen Blocks. Nach der schließenden Klammer } ist der Gültigkeitsbereich beendet und die Variablen stehen nicht mehr zur Verfügung.

- Die auskommentierten Zeilen würden daher zu zu Compilerfehlern führen – nicht weil die Variable gelöscht wurde, sondern weil sie außerhalb ihres Gültigkeitsbereichs nie existiert hat.

- Wichtige Erkenntnisse
  - Der Ort der Deklaration bestimmt die Sichtbarkeit einer Variable.
  - Innere Blöcke können auf äußere Variablen zugreifen.
  - Äußere Blöcke können nicht auf Variablen aus inneren Blöcken zugreifen.
  - Sobald ein Block endet, enden auch die darin deklarierten Variablen.
  - Viele typische Compilerfehler entstehen durch falsch platzierte Deklarationen. Näheres zu Compilerfehlern folgt in den Programmbeispielen zum Thema Debugging unten.

41. Array I: Array anlegen und initialisieren

Code

```
public class ArrayAnlegenInitialisieren {

  public static void main(String[] args) {

    // Deklaration: Array-Referenz definieren
    int[] zahlen;

    // Erzeugung: Array-Objekt anlegen, Länge = 3
    zahlen = new int[3];

    /*
    Oder kürzer/kombiniert:
    int[] zahlen = new int[3];
    */

    // Wertzuweisung
    zahlen[0] = 1;
    zahlen[1] = 2;
    zahlen[2] = 3;

    /*
    Oder kürzer/kombiniert:
    int[] zahlen = { 1, 2, 3 };
    */
  }

}
```

Ausgabe

```
<keine Ausgabe>
```

Erläuterung

Anm.: Näheres zu Objekten folgt später.

- Arrays
  - Ein Array ist eine Datenstruktur, in der mehrere Werte des gleichen Typs (Elemente) gespeichert werden.
  - Arrays haben eine feste Länge, die nach der Erzeugung nicht mehr verändert werden kann.
  - Ein Array-Element entspricht in seiner Funktion einer Variablen.
  - Auf die einzelnen Elemente wird nicht über eigene Namen, sondern über einen Index in eckigen Klammern [ ] zugegriffen.
  - In Java ist ein Array kein primitiver Datentyp, sondern ein Objekt.

- int[] zahlen;
  - int[] gibt den Datentyp an: ein Array von Ganzzahlen.
  - zahlen ist der Name der Variablen, die auf ein Array verweist (Array-Referenz).
  - An dieser Stelle existiert noch kein Array-Objekt; es wird lediglich die Referenzvariable definiert.
  - Die Variable zahlen speichert daher nicht die Werte selbst, sondern kann später eine Referenz auf ein Array-Objekt aufnehmen.

- zahlen = new int[3];
  - new int[3] erzeugt ein Array-Objekt der Länge drei, also mit drei Elementen (hier drei Ganzzahlwerten).
  - Das erzeugte Array-Objekt wird der Referenzvariable zahlen zugewiesen.
  - Alle Elemente erhalten automatisch einen Standardwert (Initialisierung), hier: 0.
  - Die Standardinitialisierung von Arrays erfolgt abhängig vom Datentyp:
    - ~ int, double, … → 0 bzw. 0.0
    - ~ boolean → false
    - ~ char → '\u0000' (unsichtbares Null-Zeichen)
    - ~ Referenztypen (z. B. String[]) → null
  - Die Länge des Arrays ist nun festgelegt und kann nicht nachträglich verändert werden.

- int[] zahlen = new int[3];
  - Kurzschreibweise: Deklaration der Referenzvariable und Erzeugung des Array-Objekts in einem Schritt.

- zahlen[0] = 1;, zahlen[1] = 2;, zahlen[2] = 3;
  - Jedem Array-Element wird ein Ganzzahlwert zugewiesen.
  - Der Zugriff auf die Elemente erfolgt über den Index in eckigen Klammern [ ].
  - **In Java beginnt die Indizierung bei 0**; ein Array der Länge 3 besitzt daher die Indizes 0, 1 und 2.
  - Beispiel: zahlen[0] = 1; setzt das erste Element des Arrays auf den Wert 1.

- int[] zahlen = {1, 2, 3};
  - Kurzschreibweise: Deklaration, Erzeugung des Array-Objekts und Initialisierung der Elemente in einem Schritt.
  - Diese Schreibweise wird häufig verwendet, wenn die Werte von Anfang an bekannt sind.

- Hinweise
  - Das Programm selbst gibt nichts aus.
  - Arrays werden häufig zusammen mit Schleifen (z. B. for oder for-each) bearbeitet und ausgegeben (System.out.println()).

42. Array II: Array ausgeben

Code

```
public class ArrayAusgeben {

  public static void main(String[] args) {

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

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

}
```

Ausgabe

```
1
2
3
```

Erläuterung

- zahlen.length
  - Gibt die Anzahl der Elemente des Arrays zahlen aus (Array-Länge).
  - Im Beispiel: 3; das Array hat also eine Länge von 3 mit den Indizes 0, 1 und 2.
  - Der höchste gültige Index eines Arrays ist immer length - 1.
  - length wird häufig in Schleifen verwendet, um die Array-Grenzen einzuhalten.

- for (…)
  - Ein Array zu durchlaufen, z. B. um alle Werte auszugeben, lässt sich gut mit einer for-Schleife realisieren.
  - Hier läuft die Zählvariable i der for-Schleife von 0 aus solange, bis i < zahlen.length ist, also i < 3, also bis zum letzten Index i = 2.
  - Der Vergleich mit < verhindert, dass die Schleife die Array-Grenzen überschreitet – denn der höchste (dritte) Index von zahlen hat den Wert length - 1 = 3 - 1 = 2.

- zahlen[i]
  - Zugriff auf das Element mit Index i.
  - Im ersten Schleifendurchlauf also auf zahlen[0] (→ Ausgabe von 1), im zweiten auf zahlen[1] (→ Ausgabe von 2) und im dritten auf zahlen[2] (→ Ausgabe von 3).

43. Array III: Summe aller Arraywerte (Variante mit for-Schleife)

Code

```
public class ArraySummeFor {

  public static void main(String[] args) {

    int[] zahlen = { 1, 2, 3 };
    int summe = 0;

    for (int i = 0; i < zahlen.length; i++) {
      summe = summe + zahlen[i];
    }

    System.out.println(summe);
  }

}
```

Ausgabe

```
6
```

Erläuterung

- int[] zahlen = { 1, 2, 3 };
  - Das Array zahlen enthält drei Ganzzahlen. Zugriff erfolgt über Indizes [0], [1], [2].

- int summe = 0;
  - Deklaration und Initialisierung der Variable summe, die die Gesamtsumme speichern wird.

- for (int i = 0; i < zahlen.length; i++)
  - Die for-Schleife läuft als Zählschleife von Index 0 bis zum letzten Array-Element (zahlen.length - 1).
  - i++ erhöht die Zählvariable nach jedem Durchlauf um 1.

- summe = summe + zahlen[i];
  - Addiert jedes Array-Element zur bisherigen Summe.
  - Nach allen Durchläufen enthält summe die Summe aller Array-Werte.
  - Summe nimmt während des Programmablaufs somit folgende Werte an: 0 (Initialisierung), 1, 3 und 6 (Endergebnis).

- System.out.println(summe);
  - Gibt das Ergebnis aus.

- Vorteile dieser Variante:
  - Flexibel, man kann während der Schleife z. B. bestimmte Elemente überspringen oder verändern.
  - Klarer Zugriff auf die Indizes des Arrays.

44. Array III: Summe aller Arraywerte (Variante mit for-each-Schleife)

Code

```
public class ArraySummeForEach {

  public static void main(String[] args) {

    int[] zahlen = { 1, 2, 3 };
    int summe = 0;

    for (int z : zahlen) {
      summe += z;
    }

    System.out.println(summe);
  }

}
```

Ausgabe

```
6
```

Erläuterung

- for (int z : zahlen)
  - Erweiterte for-Schleife: for-each.
  - Übersetzung: „Nimm jeden Element-Wert aus zahlen, speichere ihn temporär in z und arbeite damit entsprechend den nachfolgenden Befehlen."
  - Vorteil: Keine Index-Variable nötig (Java übergibt automatisch jedes Element einzeln), einfacher zu lesen und weniger fehleranfällig.
  - z ist eine lokale Schleifenvariable, die bei jedem Schleifendurchlauf neu gesetzt wird. Sie existiert nur innerhalb der Schleife und kann danach nicht weiter genutzt werden. Ihre einzige Aufgabe besteht darin, den aktuellen Element-Wert des Arrays zu liefern.

- summe += z;
  - Kurzschreibweise für summe = summe + z;.

Unterschiede zwischen for- und for-each-Schleife

Merkmal	for-Schleife	for-each-Schleife
Idee	Denken in Positionen: „Ich weiß, an welcher Stelle ich bin."	Denken in Werten: „Ich bekomme jedes Element, um damit zu arbeiten – mehr nicht."
Zugriff auf Array	über Index zahlen[i]	direkt auf Element z
Flexibilität	sehr flexibel: Index kann angepasst, Elemente übersprungen oder verändert werden	weniger flexibel: iteriert stur über alle Elemente, keine direkte Kontrolle der Indizes
Fehleranfälligkeit	leicht Indexfehler möglich (z. B. i <= zahlen.length)	weniger fehleranfällig, da keine Indexberechnung nötig
Lesbarkeit	etwas ausführlicher, mehr Standard-Code erforderlich	kompakt, gut lesbar, ideal für einfache Durchläufe
Einsatzgebiet	wenn Index benötigt wird oder nicht alle Elemente bearbeitet werden sollen	wenn alle Elemente bearbeitet werden sollen bzw. der Index keine Rolle spielt

45. Array IV: 2D-Arrays – Multiplikationsmatrix

Code

```
public class Array2DMultiplikationsmatrix {

  public static void main(String[] args) {

    // 2D-Array 3x3 für Multiplikationsmatrix
    int[][] matrix = new int[3][3];

    // Wertzuweisung
    matrix[0][0] = 1 * 1;
    matrix[0][1] = 1 * 2;
    matrix[0][2] = 1 * 3;

    matrix[1][0] = 2 * 1;
    matrix[1][1] = 2 * 2;
    matrix[1][2] = 2 * 3;

    matrix[2][0] = 3 * 1;
    matrix[2][1] = 3 * 2;
    matrix[2][2] = 3 * 3;

    // Beispielausgabe: direkter Zugriff auf den Wert der 3. Zeile, 2. Spalte
    System.out.println("matrix[2][1] = " + matrix[2][1]);
  }

}
```

Ausgabe

```
matrix[2][1] = 6
```

Erläuterung

- int[][] matrix = new int[3][3];
  - Ein zweidimensionales Array ist ein Array, dessen Elemente selbst wieder Arrays sind (Array von Array-Objekten).
  - Hier wird ein zweidimensionales Ganzzahl-Array matrix mit 3 Zeilen und 3 Spalten erzeugt.
  - Alle Elemente werden automatisch mit 0 initialisiert.
  - Die Abbildung des Arrays sieht wie folgt aus:

	Spalte 0	Spalte 1	Spalte 2
Zeile 0	0	0	0
Zeile 1	0	0	0
Zeile 2	0	0	0

  - Wichtiger Hinweis zum Aufbau von 2D-Arrays:
    - ~ In Java ist ein zweidimensionales Array kein echtes rechteckiges Raster.
    - ~ Jede Zeile ist ein eigenständiges Array und kann theoretisch **eine andere Länge** besitzen.
    - ~ In diesem Beispiel haben alle Zeilen die gleiche Länge (3 Spalten), sodass eine tabellenartige Struktur entsteht.

- matrix[0][0] = 1 * 1; etc.
  - Direkte Zuweisung einzelner Werte in Array-Elemente.
  - Jede Zeile ist ein Array, daher [Zeile][Spalte].
  - Die Indizierung beginnt bei 0.
  - Z. B. schreibt matrix[0][2] = 1 * 3; den Wert 3 in das Element von Zeile 0, Spalte 2.
  - Belegung nach allen Zuweisungen (Multiplikationsmatrix):

	Spalte 0	Spalte 1	Spalte 2
Zeile 0	1	2	3
Zeile 1	2	4	6
Zeile 2	3	6	9

- System.out.println("matrix[2][1] = " + matrix[2][1]);
  - Greift direkt auf Zeile 2, Spalte 1 zu und gibt den Wert 6 aus.

- Diese Art der direkten Zuweisung ist sinnvoll für kleine Tabellen. Für größere Arrays eignen sich (verschachtelte) Schleifen (s. u.).

46. Array V: 2D-Arrays – Notenpunkte-Tabelle

Code

```
public class Array2DArrayNotenpunkteTabelle {

  public static void main(String[] args) {

    // 2D-Array: 3 Schüler, 5 Fächer, Werte 0–15
    int[][] noten = new int[3][5];

    // Zuweisung von Noten
    // Schüler 1
    noten[0][0] = 11;
    noten[0][1] = 14;
    noten[0][2] = 12;
    noten[0][3] = 15;
    noten[0][4] = 14;

    // Schüler 2
    noten[1][0] = 10;
    noten[1][1] = 7;
    noten[1][2] = 8;
    noten[1][3] = 8;
    noten[1][4] = 6;

    // Schüler 3
    noten[2][0] = 4;
    noten[2][1] = 5;
    noten[2][2] = 4;
    noten[2][3] = 1;
    noten[2][4] = 3;

    // Ausgabe der Tabelle
    System.out.println("Notenpunkte-Tabelle (3 Schüler mit je 5 Noten):");
    for (int i = 0; i < noten.length; i++) {
      for (int j = 0; j < noten[i].length; j++) {
        System.out.print(noten[i][j] + "\t");
      }
      System.out.println();
    }
  }

}
```

Ausgabe

```
Notenpunkte-Tabelle (3 Schüler mit je 5 Noten):
11      14      12      15      14
10      7       8       8       6
4       5       4       1       3
```

Erläuterung

- int[][] noten = new int[3][5];
  - Erzeugt ein zweidimensionales Ganzzahl-Array noten mit 3 Zeilen und 5 Spalten.
  - Alle Elemente werden automatisch mit 0 initialisiert.

- noten[0][0] = 11; etc.
  - [Zeile][Spalte] → [Schüler][Fach].
  - Werte zwischen 0 und 15, angepasst an Leistungsniveau.

- Ausgabe mit verschachtelten for-Schleifen:
  - for (int i = 0; i < noten.length; i++)
    - ~ Äußere for-Schleife mit Zählvariable i.
    - ~ noten.length liefert die Anzahl der Zeilen des 2D-Arrays.
    - ~ Jede Zeile entspricht genau einem Schüler.
    - ~ Die Schleife läuft daher über die Zeilen mit i = 0 bis 2 → 3 Schüler.
  - for (int j = 0; j < noten[i].length; j++)
    - ~ Innere for-Schleife mit Zählvariable j.
    - ~ noten[i] ist ein 1D-Array mit den Notenpunkten eines Schülers.
    - ~ noten[i].length liefert die Anzahl der Fächer eines Schülers.
    - ~ Die Schleife läuft über die Spalten mit j = 0 bis 4 → 5 Fächer.

- Zugriff auf die einzelnen Werte:
  - noten[i][j] greift auf das aktuelle Element zu: Schüler i, Fach j.
  - Beispiel: noten[1][3] → Schüler 2, Fach 4.

- Verwendung von System.out.print und System.out.println:
  - System.out.print(…) sorgt dafür, dass die Notenpunkte eines Schülers hintereinander ausgegeben werden (ohne Zeilenumbruch).
  - Escape-Sequenz "\t" (horizontaler Tabulator) sorgt für gleichmäßige Abstände zwischen den Notenpunkten.
  - System.out.println();
    - ~ Erzeugt einen Zeilenumbruch nach jeder Ausgabe-Zeile.
    - ~ Steht somit bewusst außerhalb der inneren for-Schleife.
  - Ergebnis ist eine klar strukturierte Notenpunkte-Tabelle mit 3 Zeilen (→ Schüler) und 5 Spalten (→ Fächer):

	Fach 1	Fach 2	Fach 3	Fach 4	Fach 5
Schüler 1	11	14	12	15	14
Schüler 2	10	7	8	8	6
Schüler 3	4	5	4	1	3

- Hinweise:
  - Verschachtelte Schleifen erleichtern das Durchlaufen von 2D-Arrays.
  - Innere Schleife bearbeitet die Spalten einer Zeile, äußere Schleife die Zeilen.
  - Diese Technik ist universell einsetzbar für Matrizen, Tabellen, Raster usw.

47. ArrayList I: Erzeugen u. Befüllen einer Liste mit veränderlicher Länge

Code

```
import java.util.ArrayList;
import java.util.Scanner;

public class ArrayListErzeugenBefuellen {

  public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);

    // Erzeugen einer leeren ArrayList für Besuchernamen
    ArrayList<String> besucher = new ArrayList<>();

    // Erste Eingabe
    System.out.print("Name eingeben (leer für Ende): ");
    String name = sc.nextLine();

    // Solange kein leerer String eingegeben wird, werden Namen gespeichert
    while (!name.equals("")) {
      besucher.add(name);

      System.out.print("Name eingeben (leer für Ende): ");
      name = sc.nextLine();
    }

    System.out.println();
    System.out.println("Anzahl der Besucher: " + besucher.size());
    System.out.println();
    System.out.println("Besucherliste:");

    // Ausgabe aller Namen (for-each)
    for (String b : besucher) {
      System.out.println(b);
    }

    sc.close();
  }

}
```

Ausgabe (Beispiel)

```
Name eingeben (leer für Ende): Kirk
Name eingeben (leer für Ende): Picard
Name eingeben (leer für Ende): Janeway
Name eingeben (leer für Ende):

Anzahl der Besucher: 3

Besucherliste:
Kirk
Picard
Janeway
```

Erläuterung

Bei Arrays muss die Länge beim Anlegen feststehen. In vielen Anwendungen ist jedoch vorher nicht bekannt, wie viele Elemente benötigt werden. In Java werden solche veränderlichen Datenmengen nicht mit Arrays, sondern mit Klassen wie ArrayList realisiert.

- Grundidee
  - Die Anzahl der Besucher ist vorher nicht bekannt.
  - Ein Array wäre hier ungeeignet, da seine Länge festgelegt werden müsste.
  - Eine ArrayList kann dagegen zur Laufzeit beliebig erweitert werden.

- ArrayList<String> besucher = new ArrayList<>();
  - ArrayList ist kein Array, sondern eine Klasse der Java-Standardbibliothek.
  - Objekte dieser Klasse verwalten intern Arrays und ermöglichen so eine Liste von Elementen mit veränderlicher („dynamischer") Länge.
  - String legt fest, dass in der Liste Textwerte (Namen) gespeichert werden.
  - Die Liste wird zunächst leer erzeugt.

- Eingabe mit Scanner und Abbruch durch Leerstring
  - Mit sc.nextLine() wird jeweils eine komplette Textzeile eingelesen.
  - Die erste Eingabe erfolgt vor der Schleife.
  - Die while-Schleife läuft solange der eingegebene Name nicht leer ist.
  - Wird ein Leerstring ("") eingegeben, ist die Schleifenbedingung nicht mehr erfüllt und die Schleife endet.
  - Der Leerstring dient hier bewusst als Endekriterium für die Eingabe.

- besucher.add(name);
  - Fügt den eingegebenen Namen am Ende der Liste hinzu.
  - Die ArrayList wächst dabei automatisch.
  - Es ist keine vorherige Festlegung der Anzahl der Elemente erforderlich.

- besucher.size()
  - Liefert die aktuelle Anzahl der gespeicherten Elemente („Länge" der ArrayList).
  - Entspricht funktional length bei Arrays, ist jedoch eine Methode.

- Ausgabe mit for-each-Schleife
  - Die for-each-Schleife durchläuft alle Elemente der Liste.
  - Die Reihenfolge entspricht der Reihenfolge der Eingaben.

48. ArrayList II: Elemente ausgeben, verändern und entfernen

Code

```
import java.util.ArrayList;

public class ArrayListAusgebenVeraendernEntfernen {

  public static void main(String[] args) {

    // Beispielhafte Besucherliste
    ArrayList<String> besucher = new ArrayList<>();
    besucher.add("Data");
    besucher.add("Luke");
    besucher.add("Ripley");

    // Ausgabe der ursprünglichen Liste
    System.out.println("Ursprüngliche Besucherliste:");
    for (String b : besucher) {
      System.out.println(b);
    }

    System.out.println();

    // Element gezielt auslesen
    System.out.println("Besucher an Position 1: " + besucher.get(1));

    // Element verändern
    besucher.set(1, "Mr. Spock");

    // Element entfernen
    besucher.remove(0);

    System.out.println();

    // Ausgabe der veränderten Liste
    System.out.println("Veränderte Besucherliste:");
    for (String b : besucher) {
      System.out.println(b);
    }
  }

}
```

Ausgabe

```
Ursprüngliche Besucherliste:
Data
Luke
Ripley

Besucher an Position 1: Luke

Veränderte Besucherliste:
Mr. Spock
Ripley
```

Erläuterung

Nachdem eine Liste mit veränderlicher Länge erzeugt wurde, sollen nun einzelne Elemente gezielt gelesen, verändert und entfernt werden.

- Ausgangsliste
  - Die ArrayList wird hier direkt im Code mit Beispielwerten gefüllt.
  - Ziel ist es, Operationen auf bestehenden Listenelementen zu demonstrieren.

- besucher.get(1)
  - Liefert das Element an der angegebenen Position.
  - Die Indizierung beginnt – wie bei Arrays – bei 0.
  - get(1) greift daher auf das zweite Element zu (hier: Luke).

- besucher.set(1, "Mr. Spock")
  - Ersetzt das Element an Position 1 (hier: Luke) durch einen neuen Wert (hier: Mr. Spock).
  - Die Länge der Liste bleibt dabei unverändert (hier: 3).

- besucher.remove(0)
  - Entfernt das Element an Position 0 (hier: Data).
  - Alle nachfolgenden Elemente rücken automatisch nach.
  - Dadurch ändern sich die Indizes der verbleibenden Elemente; dies unterscheidet ArrayList deutlich von Arrays.
  - Die Länge der Liste verringert sich um 1 (hier: auf 2).

- Ausgabe der veränderten Liste
  - Die Liste wird erneut mit einer for-each-Schleife ausgegeben.
  - Die neue Reihenfolge und die veränderten Inhalte werden sichtbar.

Zusammenfassung

- ArrayList ermöglicht Listen mit veränderlicher Länge.
- Man kann Elemente
  - hinzufügen (add)
  - lesen (get)
  - verändern (set)
  - entfernen (remove)
- ArrayList ersetzt Arrays nicht, sondern ergänzt sie für andere Einsatzfälle.

49. Zufall I: Zufallszahl zwischen 0 und 1

Code

```
public class ZufallNullEins {

  public static void main(String[] args) {

    double zufall1 = Math.random();
    System.out.println("Zufallszahl zw. 0.0 und 1.0: " + zufall1);

    int zufall2 = (int)(Math.random() * 10);
    System.out.println("Zufallszahl als ganze Zahl zw. 0 und 9: " + zufall2);
  }

}
```

Ausgabe (Beispiel)

```
Zufallszahl zw. 0.0 und 1.0: 0.732845291
Zufallszahl als ganze Zahl zw. 0 und 9: 7
```

Erläuterung

- double zufall1 = Math.random();
  - Math.random() liefert eine Pseudo-Zufallszahl vom Typ double im Intervall [0.0, 1.0), d. h. ≥ 0 und < 1.
  - „Pseudo" bedeutet, dass der Wert kein „echter" Zufall ist, sondern auf `deterministischen Berechnungen` beruht und ggf. reproduzierbar ist (nicht für kryptografische Zwecke geeignet).
  - Jede Ausführung erzeugt typischerweise einen anderen Wert.

- int zufall2 = (int)(Math.random() * 10);
  - Um ganze Zufallszahlen zu erhalten (hier: Werte von 0 bis 9), multipliziert man den erzeugten Wert mit der gewünschten Obergrenze (hier: 10) und konvertiert ihn in int.
  - Beachte: Die Obergrenze ist exklusiv – 10 wird hier nicht erzeugt.

50. Zufall II: Würfelwurf

Code

```
import java.util.Random;

public class ZufallWuerfelwurf {

  public static void main(String[] args) {

    // 1. Zufallszahl ohne Seed
    Random rand1 = new Random();
    int wuerfel1 = rand1.nextInt(6) + 1;

    System.out.println("Würfelwurf ohne Seed: " + wuerfel1);

    // 2. Zufallszahl mit Seed
    Random rand2 = new Random(12345);  // Seed 12345 für reproduzierbare Werte
    int wuerfel2 = rand2.nextInt(6) + 1;

    System.out.println("Würfelwurf mit Seed:  " + wuerfel2);
  }

}
```

Ausgabe (Beispiel)

```
Würfelwurf ohne Seed: 3
Würfelwurf mit Seed:  4
```

Erläuterung

Würfelwurf ohne Seed

- Random rand1 = new Random();
  - Erstellt ein neues Objekt vom Typ Random, das Pseudo-Zufallszahlen *ohne* Startwert (Seed, s. u.) erzeugt.
  - Für Random wird java.util.Random; benötigt.

- rand1.nextInt(6)
  - Liefert eine ganzzahlige Zufallszahl im Bereich 0 bis 5 (inklusive 0, exklusive 6).
  - Beachte: Die Obergrenze (6) ist exklusiv, d. h. 6 wird hier nicht erzeugt.

- \+ 1
  - Verschiebt den Wertebereich von 0–5 auf 1–6, sodass das Ergebnis einem klassischen Würfelwurf entspricht.
  - Ohne +1 würde der Würfel „0" enthalten, was in der Realität nicht vorkommt.

Würfelwurf mit Seed

- Random rand2 = new Random(12345);
  - Erstellt ein neues Objekt vom Typ Random, das Pseudo-Zufallszahlen *mit* Seed erzeugt.
  - Seed: fester Startwert für die Zufallszahlenberechnung, so dass mit jeder Ausführung mit demselben Seed die gleiche Folge von Zufallszahlen erzeugt wird.
  - Der Seed kann beliebig gewählt werden (z. B. 1, 42, 987654).

- Vorteile von Random gegenüber Math.random()
  - Flexibilität: kann verschiedene Typen liefern (int, double, boolean).
  - Vorhersagbarkeit: Seed ermöglicht reproduzierbare Folgen von Zufallszahlen, nützlich für Tests oder Simulationen.

Exkurs: Zufallszahlen – Pseudozufall und Sicherheit

- Math.random() und Random erzeugen Pseudo-Zufallszahlen.
  - Zufall entsteht algorithmisch (also mittel Berechnungen), nicht physikalisch.
  - Interne Berechnung folgt festen Regeln.
  - Bei bekanntem Seed ist die Zahlenfolge reproduzierbar.
  - Ausreichend für Spiele, Simulationen, Tests.

- SecureRandom erzeugt kryptografisch sichere Pseudo-Zufallszahlen.
  - Ebenfalls algorithmisch.
  - Nutzt aber zusätzliche Entropiequellen des Betriebssystems.
  - Folgen sind praktisch nicht vorhersagbar.
  - Geeignet für sicherheitskritische Anwendungen (z. B. Passwörter, Schlüssel).

- „Echter" Zufall
  - Erfordert nicht reproduzierbare physikalische Zufallsquellen, z. B.
    - ~ Rauschen elektronischer Bauteile,
    - ~ Sensorrauschen,
    - ~ radioaktiver Zerfall.
  - In Java
    - ~ Kein direkter Zugriff auf physikalische Zufallsquellen, sondern nur indirekt über Betriebssystem-Entropie (z. B. Mausbewegungen, Systemrauschen).
    - ~ Ergebnis ist dennoch Teil eines algorithmischen Prozesses.

- Zusammenfassung
  - Zufall ist nicht gleich Zufall.
  - Entscheidend ist der Einsatzzweck.

Generator	algorithmisch (Pseudo-Zufall)	vorher-sagbar	kryptografisch sicher
Math.random()	ja	ja	nein
Random	ja	ja	nein
SecureRandom	ja	praktisch nein	ja

51. Zufall III: Zufallsauswahl aus einem Array

Code

```
import java.util.Random;

public class ZufallAuswahlAusArray {

  public static void main(String[] args) {

    String[] farben = {"Rot", "Grün", "Blau", "Gelb"};

    // Pseudo-Zufallsgenerator erstellen
    Random rand = new Random();

    // Zufälliger Index zwischen 0 und farben.length-1
    int index = rand.nextInt(farben.length);

    // Zugriff auf Array mit zufälligem Index
    String zufallsFarbe = farben[index];

    System.out.println("Zufällige Farbe: " + zufallsFarbe);
  }

}
```

Ausgabe (Beispiel)

```
Zufällige Farbe: Blau
```

Erläuterung

- String[] farben = {"Rot", "Grün", "Blau", "Gelb"};
  - Array farben, initialisiert mit den vier Element-Werten Rot, Grün, Blau und Gelb.
  - Arrays sind Objekte; Zugriff erfolgt über einen Index, der bei 0 beginnt (farben[0] → "Rot").

- Random rand = new Random()
  - Erstellt einen Pseudo-Zufallsgenerator.
  - Ohne Seed: jeder Programmstart erzeugt andere Folgen von Zufallszahlen.
  - Optional kann ein Seed angegeben werden, um reproduzierbare Folgen zu erhalten (z. B. new Random(12345)).

- rand.nextInt(farben.length)
  - farben.length liefert die Länge des Arrays, hier 4.
  - rand.nextInt(...) erzeugt eine ganzzahlige Zufallszahl zwischen 0 (inklusive) und farben.length (exklusive), hier also 0–3.
  - Die Obergrenze ist exklusiv, d. h. farben.length selbst wird nie erzeugt.

- String zufallsFarbe = farben[index];
  - Mit dem zufälligen Index index wird ein Element aus dem Array ausgewählt.
  - Ergebnis: eine zufällige Farbe aus dem Array.

- Anwendung
  - Nützliche Technik für zufällige Auswahl aus Listen oder Arrays, z. B. Namen, Spielaktionen, Menüpunkte oder Test-/Simulationsdaten.

52. Methoden I: Ohne Rückgabewert (void)

Code

```
public class MethodenVoid {

  static void hallo() {  // Methode "hallo" (ohne Parameter und Rückgabewert)

    System.out.println("Hallo");
  }

  public static void main(String[] args) {

    hallo();             // Aufruf der Methode "hallo" (ohne Parameter)
  }

}
```

Ausgabe

```
Hallo
```

Erläuterung

Anm.: Grundlegende Aspekte wie Klasse, main-Methode, static und geschweifte Klammern wurden bereits in Beispiel 1 oben erläutert und werden hier nur knapp aufgegriffen.

- static void hallo()
  - Definition einer eigenständigen Methode innerhalb der Klasse.
  - void bedeutet: Die Methode liefert keinen Rückgabewert, sondern führt nur Anweisungen aus.
  - Der Methodenname ist hallo.

- Leere runde Klammern ()
  - Methoden können Parameter erwarten. Parameter sind Variablen, über die Werte beim Aufruf einer Methode an diese übergeben werden können.
  - hallo definiert keine Parameter, d. h. die Methode erwartet keine übergebenen Werte. Entsprechend stehen nach hallo leere runde Klammern (diese dürfen auch nicht weggelassen werden).

- Methodenrumpf { ... }
  - Enthält den Code, der beim Aufruf der Methode ausgeführt wird.
  - Hier erfolgt mit System.out.println lediglich eine Konsolenausgabe.

- Methodenaufruf hallo();
  - Startet die Ausführung der Methode hallo.
  - Da kein Rückgabewert existiert, steht der Aufruf allein und wird keinem Ausdruck zugewiesen.
  - Da keine Parameter an hallo übergeben werden, stehen nach hallo leere runde Klammern (diese dürfen auch nicht weggelassen werden).

- Sichtbarkeit (public)
  - Die Methode hallo ist hier nicht public, was ausreichend ist, da sie nur innerhalb derselben Klasse verwendet wird.
  - public wird erst relevant, wenn Methoden von anderen Klassen aus aufgerufen werden sollen.

53. Methoden II: Mit Rückgabewert (return)

Code

```
public class MethodenReturn {

  static int quad(int x) {   // Methode "quad" (mit Parameter und Rückgabewert)

    return x * x;
  }

  public static void main(String[] args) {

    int ergebnis = quad(4);  // Aufruf der Methode "quad" (mit Parameter (4))
    System.out.println(ergebnis);
  }

}
```

Ausgabe

```
16
```

Erläuterung

- static int quad(int x)
  - Methode mit Rückgabewert (entsprechend kein void angegeben).
  - Das erste int gibt den Datentyp des Rückgabewerts an.
  - x ist ein Parameter, über den beim Aufruf ein Wert an die Methode übergeben wird. Das zweite int zeigt, dass die Methode für x eine Ganzzahl erwartet.
  - Grundsätzlich kann man nicht nur einen, sondern auch mehrere Parameter übergeben. Dann ist sowohl beim Aufruf der Methode, als auch bei der Methode selbst auf die Reihenfolge zu achten, in der die Parameter notiert werden.

- Parameterübergabe
  - Beim Aufruf quad(4) wird der Wert 4 an den Parameter x übergeben.
  - Innerhalb der Methode wird ausschließlich mit x gearbeitet.

- return x * x;
  - Berechnet das Quadrat von x.
  - Gibt das Ergebnis an den Aufrufer zurück.
  - Sobald return ausgeführt wird, endet die Methode quad sofort (eventuell nachfolgender Code in der Methode würde nicht ausgeführt).

- Verwendung des Rückgabewerts
  - Der zurückgegebene Wert kann
    - ~ einer Variablen zugewiesen,
    - ~ ausgegeben oder
    - ~ weiterverarbeitet werden.
  - Hier wird das Ergebnis in ergebnis gespeichert und anschließend ausgegeben. Der Datentyp von ergebnis muss passend definiert sein.

Kurzvergleich: void-Methoden vs. Methoden mit Rückgabewert

Merkmal	void-Methode	Methode mit Rückgabewert
Zweck	Aktion ausführen	Ergebnis berechnen
Rückgabewert	keiner	genau ein Wert
return notwendig	nein	ja
Ergebnis weiter nutzbar	nein	ja

- void-Methoden werden eingesetzt, wenn eine Methode etwas ausführen, aber kein Ergebnis zurückgeben soll (z. B. Ausgaben, Aktionen, Statusmeldungen).

- Methoden mit Rückgabewert kapseln Berechnungen. Sie liefern ein Ergebnis, das außerhalb der Methode weiterverwendet werden kann.

Methoden im Allgemeinen

- Methoden werden nicht nur aus formalen Gründen verwendet, sondern um Programme übersichtlich, wartbar und erweiterbar zu halten. Anstatt den gesamten Code in der main-Methode zu sammeln, werden logisch zusammengehörige Aufgaben in eigene Methoden ausgelagert. Das hat mehrere Vorteile:
  - Strukturierung: Jede Methode erfüllt eine klar abgegrenzte Aufgabe. Der Programmablauf wird leichter verständlich.
  - Wiederverwendbarkeit: Eine Methode kann mehrfach aufgerufen werden, ohne Code zu duplizieren.
  - Lesbarkeit: Der Code in main wird kürzer und beschreibt eher *was* passiert, während die Methoden definieren *wie* es passiert.
  - Fehlerreduktion: Änderungen müssen nur an einer Stelle vorgenommen werden; das senkt die Gefahr von Inkonsistenzen.
  - Erweiterbarkeit: Programme lassen sich einfacher anpassen oder ausbauen, wenn Funktionen sauber gekapselt sind.

- In Java ist die Reihenfolge von Methoden innerhalb einer Klasse beliebig, da der Compiler die gesamte Klasse kennt. Im Beispiel oben kann die Methode hallo vor oder nach main stehen.

54. Debugging I: Compilerfehler

Code

```
public class DebuggingCompilerfehler {

  public static void main(String[] args) {

    int a = 5;                      // Compilerfehler 1: fehlendes Semikolon

    if (a == 5) {                   // Compilerfehler 2: Zuweisung statt Vergleich
      System.out.println("a ist 5");
    }
  }

}
```

Ausgabe

```
<keine Ausgabe (*)>
```

Erläuterung

- Programm kann nicht in ausführbaren Bytecode übersetzt werden → zwei Compilerfehler müssen zuerst behoben werden.

- Fehler 1: Fehlendes Semikolon am Ende der Anweisung int a = 5.

- Fehler 2: Zuweisung statt Vergleich; if erwartet einen boolean-Ausdruck als Ergebnis eines Vergleichs (==). Eine Zuweisung (hier: a = 5) liefert jedoch einen int → Compiler meldet „Type mismatch".

- Der Java-Compiler erkennt Compilerfehler bereits vor dem Programmstart und (*) gibt die Zeilennummer der fehlerhaften Stelle aus.

55. Debugging II: Laufzeitfehler

Code

```
public class DebuggingLaufzeitfehler {

  public static void main(String[] args) {

    int a = 42;
    int b = 0;
    int ergebnis = a / b;

    System.out.println(ergebnis);
  }

}
```

Ausgabe

```
<keine Ausgabe (*)>
```

Erläuterung

- Programm startet, bricht jedoch während der Ausführung ab.

- Fehler: Division durch Null → Laufzeitfehler.

- (*) Java zeigt eine Exception an (z. B. ArithmeticException: / by zero), wodurch typische Laufzeit-Fehlermeldungen sichtbar werden.

56. Debugging III: Logikfehler, Debug-Ausgabe

Code

```
public class DebuggingLogikfehler {

  public static void main(String[] args) {

    int[] zahlen = {1, 2, 3};
    int summe = 0;

    for (int i = 0; i < zahlen.length; i++) {
      summe = zahlen[i];
    }

    System.out.println("Summe: " + summe);
  }

}
```

Ausgabe

```
3
```

Erläuterung

- Programm läuft ohne Absturz, berechnet in der Codezeile summe = zahlen[i]; aber die Summe falsch → Logikfehler.

- Korrektur: summe += zahlen[i];

- Tipp: Debug-Ausgaben können zum Auffinden des Fehlers helfen:

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

  Hier werden Zwischenergebnisse strukturiert ausgegeben, wodurch die Codezeile mit der fehlerhaften Berechnung (summe = zahlen[i];) lokalisiert und anschließend korrigiert werden kann.

57. Debugging IV: Logikfehler – Endlosschleife

Code

```
public class DebuggingLogikfehlerEndlosschleife {

  public static void main(String[] args) {

    int i = 1;

    while (i <= 5) {
```

```
      System.out.println("i = " + i);
      // i++;  // fehlt (da hier auskommentiert)!
    }
  }

}
```

Ausgabe

```
i = 1
i = 1
i = 1
<...>
<Endlose Wiederholung, Programm muss manuell abgebrochen werden>
```

Erläuterung

- Das Programm startet korrekt und verursacht weder einen Compiler- noch einen Laufzeitfehler.
- Die while-Schleife besitzt jedoch keine Veränderung der Abbruchbedingung im Schleifenkörper.
- Die Variable i wird initialisiert, aber nie erhöht (i++ ist auskommentiert).
- Die Bedingung i <= 5 bleibt daher immer wahr → Endlosschleife.
- Korrektur:

  ```
  while (i <= 5) {
    System.out.println("i = " + i);
    i++;
  }
  ```

- Bemerkung
  - Endlosschleifen sind eine besondere Form von Logikfehlern.
  - Sie sind besonders schwer zu erkennen, da das Programm syntaktisch korrekt ist und keine Fehlermeldung ausgibt – jedoch nicht terminiert (= sich nicht selbstständig beendet).
  - Häufige Ursachen sind:
    ~ fehlende Änderung der Zählvariable (z. B. i++)
    ~ falsche Abbruchbedingung
    ~ Änderung der falschen Variable
  - Mögliche Folgen sind eine hohe CPU-Auslastung, blockierte Ressourcen, Speicherüberläufe, Programmabstürze und Datenverlust. In kritischen Systemen kann dies Sicherheitsrisiken verursachen.

58. Exceptions I: Laufzeitfehler abfangen (try/catch)

Code

```java
public class ExceptionsTryCatch {

  public static void main(String[] args) {

    int[] zahlen = {10, 20, 30};
    int index = 5;
    int wert;

    try {
      wert = zahlen[index];  // Fehler: Zugriff auf einen ungültigen Array-Index
    }
    catch (ArrayIndexOutOfBoundsException fehler) {
      System.out.println("Fehler: ungültiger Index - Ersatzwert wird verwendet.");
      wert = -1;  // Ersatzwert festlegen
    }

    if (wert != -1) {
      System.out.println("Weiterverarbeiteter Wert: " + wert);
    }
  }

}
```

Ausgabe (Beispiel)

```
Fehler: ungültiger Index - Ersatzwert wird verwendet.
```

Erläuterung

- Grundidee
  - In den Debugging-Programmbeispielen oben führte der Zugriff auf einen ungültigen Array-Index zu einem Programmabbruch.
  - Hier wird nun gezeigt, wie ein solcher Laufzeitfehler abgefangen werden kann. Das Programm kann anschließend kontrolliert weiterarbeiten.

- try
  - try enthält Code, in dem ein Fehler auftreten könnte.
  - Tritt kein Fehler auf, wird der Block normal ausgeführt.
  - Im Beispiel wird versucht, mit zahlen[index] auf ein Arrayelement zuzugreifen.
  - Da index = 5 außerhalb des gültigen Bereichs liegt, entsteht eine Exception.

- catch (ArrayIndexOutOfBoundsException fehler)
  - catch fängt eine in try aufgetretene Exception ab.
  - Hier wird catch ausgeführt, wenn der Zugriff auf einen ungültigen Array-Index erfolgt.
  - Statt eines Programmabsturzes erscheint mit „Fehler: ungültiger Index - Ersatzwert wird verwendet.“ eine verständliche Fehlermeldung.
  - fehler ist die Variable, in der Java Informationen über den aufgetretenen Fehler speichert (wird im Programmbeispiel hier nicht weiterverarbeitet).
  - wert = -1;
    - ~ Zusätzlich wird hier ein Ersatzwert festgelegt.
    - ~ Die Variable wert besitzt damit einen definierten Zustand.
    - ~ Häufig verwendet man einen speziellen Wert, der für die Variable im normalen Programmablauf nicht erwartet wird (hier: -1).
  - Nach dem catch-Block läuft das Programm normal weiter.

- if (wert != -1) { …
  - Nur wenn kein Fehler aufgetreten ist, wird hier der echte Arraywert ausgegeben.
  - Im Fehlerfall wurde wert auf -1 gesetzt → die Ausgabe wird übersprungen.

- Wichtige Erkenntnisse
  - try/catch verhindert den Programmabsturz.
  - Im catch-Block kann festgelegt werden, wie das Programm auf einen Fehler reagiert.
  - Ersatzwerte sind eine häufige Strategie, um Programme auch nach einem Fehler weiterarbeiten zu lassen.

- Die weitere Programmlogik kann anschließend prüfen, ob ein regulärer Wert oder ein Fehlerfall vorliegt.

Exkurs: Welche Exceptions gibt es?

- Wenn ein Programm ohne try/catch abstürzt, zeigt Java eine Fehlermeldung an.
- Darin steht die Bezeichnung der Exception, die zum Abfangen im catch-Block angegeben wird.
- So weiß Java, auf welche Exception reagiert werden soll.
- Eine vollständige Übersicht zu Exceptions findet man in der Java-Dokumentation. Gängige sind:

Exception	typische Ursache
ArithmeticException	Bei einer Rechnung, die mathematisch nicht erlaubt ist (z. B. Division durch 0).
ArrayIndexOutOfBoundsException	Beim Zugriff auf ein Array mit einem Index außerhalb des gültigen Bereichs (z. B. Index 5 bei einem Array der Länge 3).
IllegalArgumentException	Wenn einer Methode ein Wert übergeben wird, der zwar den richtigen Datentyp hat, aber keinen sinnvollen oder erlaubten Wert darstellt (z. B. negative Länge oder ungültiger Bereich).
InputMismatchException	Bei Eingaben mit Scanner, wenn der eingegebene Wert nicht zum erwarteten Datentyp passt (z. B. Text statt einer ganzen Zahl bei nextInt()).
IOException	Wenn bei Ein- oder Ausgabeoperationen ein Problem auftritt, z. B. wenn eine Datei nicht gefunden oder nicht gelesen werden kann.
NullPointerException	Wenn auf ein Objekt zugegriffen wird, das nicht erzeugt oder nicht zugewiesen wurde (Variable enthält null).
NumberFormatException	Wenn ein Text in eine Zahl umgewandelt werden soll, der keine gültige Zahl darstellt (z. B. "abc" statt "123").

59. Exceptions II: Finally – garantiert ausgeführt

Code

```
public class ExceptionsFinally {

  static int positiveGanzzahldivision(int a, int b) {

    boolean istErfolgreich = false;  // Annahme: Berechnung funktioniert nicht

    try {
      int ergebnis = a / b;
      istErfolgreich = true;
      return ergebnis;
    }
    catch (ArithmeticException fehler) {
      System.out.println("Fehler: Division durch 0.");
      System.out.println();
      return -1;  // Ersatzwert
    }
    finally {
      System.out.println("Berechnung abgeschlossen.");
      System.out.println("Erfolgreich? " + istErfolgreich);
    }
  }

  public static void main(String[] args) {

    int wert = positiveGanzzahldivision(10, 0);  // Vorgabe: nur pos. Ganzzahlen
    System.out.println("Rückgabewert: " + wert);
  }

}
```

Ausgabe (Beispiel)

```
Fehler: Division durch 0.

Berechnung abgeschlossen.
Erfolgreich? false
Rückgabewert: -1
```

Erläuterung

- Grundidee
  - In der vorigen Aufgabe wurde gezeigt, dass ein Fehler abgefangen werden kann und das Programm anschließend weiterläuft.
  - Zusätzlich zum Abfangen eines Fehlers gibt es manchmal Anweisungen, die auf jeden Fall ausgeführt werden sollen – unabhängig davon, ob ein Fehler aufgetreten ist oder ob die Methode vorzeitig mit return beendet werden soll.
  - Dafür gibt es finally.

- try
  - Enthält die Berechnung.
  - Wenn kein Fehler auftritt, wird das Ergebnis berechnet und der Status istErfolgreich wird auf true gesetzt.
  - Anschließend soll mit return die Methode beendet und ein Wert zurückgegeben werden. Bevor die Methode jedoch tatsächlich verlassen wird, führt Java den finally-Block aus.

- catch (ArithmeticException fehler)
  - Wird ausgeführt, wenn eine Division durch 0 auftritt.
  - Eine Fehlermeldung wird ausgegeben.
  - Für return gilt hier dasselbe wie im try-Block, jedoch wird -1 als Ersatzwert zurückgegeben.

- finally
  - Wird nach try oder catch immer ausgeführt – und zwar unabhängig davon,
    ~ ob im try eine Exception aufgetreten ist oder nicht
    ~ oder ob die Methode durch return beendet werden soll.
  - Deshalb werden die Zeilen
    ~ „Berechnung abgeschlossen.“
    ~ „Erfolgreich? …“
    in jedem Fall ausgegeben.

- Reihenfolge der Ausführung
  - Die typische Struktur lautet: try – catch – finally.
  - Zu einem try gehört mindestens ein catch oder ein finally (oder beides).
  - Auch wenn return (im try oder im catch) bereits erreicht wurde:
    1. finally wird ausgeführt.
    2. Danach kehrt die Methode zu main zurück.
    3. Der Rückgabewert wird ausgegeben.

- Wichtige Erkenntnisse
  - finally ergänzt try/catch um einen Block, der immer ausgeführt wird.
  - Dadurch bleiben Programmabläufe zuverlässig, selbst bei Fehlern oder vorzeitigem Abbruch mit return.
  - Relevant ist finally v. a. bei komplexen Anwendungen, um typische Aufräumarbeiten durchzuführen, z. B. Ressourcen (Dateien o. a.) schließen oder Speicher freigeben.

60. Anwendung: Array-Eingabe, Summenbildung (Variante ohne E-V-A)

Das Programm liest mehrere Ganzzahlen vom Benutzer ein, speichert sie in einem Array und berechnet anschließend die Summe aller eingegebenen Werte.

Code

```
import java.util.Scanner;

public class ArrayEingabeSummeOhneEVA {

  public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);
    int[] zahlen = new int[3];
    int summe = 0;

    for (int i = 0; i < zahlen.length; i++) {
      zahlen[i] = sc.nextInt();
      summe += zahlen[i];
    }

    System.out.println("Summe: " + summe);
    sc.close();
  }

}
```

Ausgabe (Beispiel)

```
Summe: 42
```

Erläuterung

- Scanner sc = new Scanner(System.in);
  - Ermöglicht das Einlesen von Benutzereingaben über die Tastatur.

- int[] zahlen = new int[3];
  - Deklariert ein Array für drei Ganzzahlen.
  - Speichert die vom Benutzer eingegebenen Werte.

- int summe = 0;
  - Zwischenspeicherung der Gesamtsumme, initialisiert mit 0.

- for (int i = 0; i < zahlen.length; i++)
  - Schleife durchläuft alle Indizes des Arrays.
  - zahlen.length sorgt für Anpassung an die Array-Länge.

- zahlen[i] = sc.nextInt();
  - Liest eine Zahl vom Benutzer ein und speichert sie an Position i im Array.

- summe += zahlen[i];
  - Addiert den zuletzt eingegebenen Wert zur Gesamtsumme.

- System.out.println("Summe: " + summe);
  - Gibt die berechnete Summe aus.

Alle Aufgaben – Eingabe, Verarbeitung (Summenbildung) und Ausgabe – werden hier direkt in main erledigt. Der Code ist kompakt und übersichtlich, aber weniger modular und schwerer erweiterbar.

61. Anwendung: Array-Eingabe, Summenbildung (Variante mit E-V-A)

Dieses Beispiel erweitert das vorherige Programm, indem es Eingabe, Verarbeitung und Ausgabe in eigene Methoden kapselt.

Code

```
import java.util.Scanner;

public class ArrayEingabeSummeMitEVA {

  static int[] eingabeZahlen(int anzahl) {

    Scanner sc = new Scanner(System.in);
    int[] zahlen = new int[anzahl];

    for (int i = 0; i < zahlen.length; i++) {
      zahlen[i] = sc.nextInt();
    }

    return zahlen;
  }

  static int berechneSumme(int[] zahlen) {

    int summe = 0;

    for (int z : zahlen) {
      summe += z;
    }

    return summe;
  }

  static void ausgabeSumme(int summe) {

    System.out.println("Summe: " + summe);
  }

  public static void main(String[] args) {

    int[] zahlen = eingabeZahlen(3);
    int summe = berechneSumme(zahlen);

    ausgabeSumme(summe);
  }

}
```

Ausgabe (Beispiel)

```
Summe: 42
```

Erläuterung

- Grundidee der Struktur
  - Das Programm ist in drei logisch getrennte Aufgaben gegliedert:
    - ~ Eingabe: Daten erfassen,
    - ~ Verarbeitung: Daten berechnen / auswerten,
    - ~ Ausgabe: Ergebnisse anzeigen.
  - Damit wird das E-V-A-Prinzip umgesetzt, das für saubere, gut wartbare Programme grundlegend ist.
  - Jede Aufgabe wird von einer eigenen Methode übernommen.
  - main dient nur noch als Ablaufsteuerung: Sie beschreibt *was* passiert, nicht *wie*.

- Eingabe: static int[] eingabeZahlen(int anzahl)
  - Liest eine festgelegte Anzahl von Ganzzahlen ein.
  - Erstellt dafür ein Array, füllt es und gibt es zurück.
  - Kapselt die gesamte Eingabe.

- Verarbeitung: static int berechneSumme(int[] zahlen)
  - Berechnet die Summe aller Elemente im übergebenen Array.
  - Nutzt eine for-each-Schleife, da der Index nicht benötigt wird.
  - Gibt die Summe zurück.
  - Kapselt die Berechnungslogik sauber vom Rest des Programms; enthält somit den eigentlichen Kernalgorithmus.

- Ausgabe: static void ausgabeSumme(int summe)
  - Gibt die Summe formatiert aus.
  - Hat keinen Rückgabewert (void), da nur eine Ausgabe erfolgt.
  - Trennt die Darstellung des Ergebnisses von der Berechnung und erhöht so die Lesbarkeit.

- main-Methode als Steuerzentrale
  - Ruft die drei Methoden in logischer Reihenfolge auf:
    1. Eingabe der Daten,
    2. Verarbeitung in Form der Berechnung der Summe,
    3. Ausgabe des Ergebnisses.
  - main beschreibt damit *was* passiert, nicht *wie* es im Detail umgesetzt wird.

Vergleich von beiden Aufgabenlösungen

- Vorheriges Beispiel: Alles kompakt in main. Kurz, wenig Variablen und Methoden.

- Hiesiges Beispiel: Modular strukturiert nach E-V-A-Prinzip. Gut wiederverwendbar und erweiterbar.

- Beide Varianten sind korrekt – welche sinnvoller ist, hängt von verschiedenen Faktoren ab. Für kleine, überschaubare Programme reicht eine kompakte Lösung. Für größere oder erweiterbare Programme ist die strukturierte Variante mit Methoden vorzuziehen.