INFORMATIK · EINFÜHRUNGSPHASE · E3

Grundlagen der Programmierung

Programmieren als Strukturierung von Verarbeitung und Interaktion

Das Themenfeld **E3 Grundlagen der Programmierung** führt systematisch von einer Problemstellung zu einer prüfbaren Lösung. Java ist dabei Arbeitsmedium, nicht Selbstzweck.

Eine Lernlinie verbindet Werte und Typen mit EVA, Ausdrücken, Bedingungen, Schleifen, Datenstrukturen und Methoden – bis zur ereignisgesteuerten GUI.

Die Kontexte wechseln, die fachliche Bewegung bleibt: Informationen werden als Daten und Zustände modelliert, Verarbeitung wird als Algorithmus strukturiert, in Java implementiert und anhand ihrer Wirkung geprüft.

> **OFFLINE-LERNFASSUNG · REDAKTIONELLER PILOT**
> Diese Fassung überträgt den vollständigen Live-Lernpfad in ein lineares Medium. Interaktive Werkzeugaufgaben sind als offline lesbare Arbeitsaufträge im Anhang materialisiert; automatische Ausführung, Diagnose und freischaltbare Musterlösungen bleiben der Online-Fassung vorbehalten.

Live-Stand: 12. Juli 2026
Quelle: https://informatik-dbook.de/einfuehrungsphase/E3.html
Herausgeber und Autor: Sebastian Grünberg, StR

Inhalt

Die Druckfassung folgt der Lernlinie der Live-Seite. Kerninhalte stehen im Hauptfluss; Referenzen, Fehlvorstellungen und Vertiefungen bleiben als sichtbar gekennzeichnete Schichten erhalten.

Hinweis: Links bleiben in Word und PDF anklickbar. Für eine vollständig netzunabhängige Bearbeitung sind die Aufgabenstellungen im Anhang enthalten.

Orientierung

RELATION **D-Book-Wiki-Verknüpfung**

Diese Seite ist der praktische Lern- und Diagnoseweg zu den Grundlagen der Programmierung. Die zugehörige D-Book-Wiki-Lineage ordnet dieselben Konzepte historisch und fachlich ein.

D-Book-Wiki: Von Hochsprachen zu objektorientierter Programmierung

E3 operationalisiert Werte, Ausdrücke, Kontrollstrukturen, Datenstrukturen, Methoden und GUI-Interaktion in Java; das Wiki erklärt die Entwicklung von formaler Programmbeschreibung über Compiler und Hochsprachen bis zu Java und objektorientierter Modellierung.

LERNWEG **Vom Wert zur ereignisgesteuerten Anwendung**

1. **Werte und Datentypen**
2. EVA und Variablenzustände
3. Ausdrücke und Operatoren
4. Bedingungen und Verzweigungen
5. Schleifen und Laufspuren
6. Arrays und Strings
7. Methoden und Schnittstellen
8. GUI und Ereignisse
9. **Algorithmus, Implementierung und Diagnose**

CURRICULUM **Kerncurriculum kompakt**

Einordnung und Inhalte des Themenfelds E.3

A) Allgemeine Einordnung

Programmierung gehört in der Einführungsphase zu den zentralen Primärerfahrungen der Informatik. Lernende entwickeln dabei vor allem Modellierungs- und Problemlösekompetenz, indem sie schrittweise von einer Idee zu einer überprüfbaren Umsetzung gelangen.

Java wird als objektorientierte Sprache genutzt. In E3 stehen jedoch imperative Grundlagen innerhalb einer objektorientierten Sprache im Vordergrund: Datentypen, Ausdrücke, Kontrollstrukturen, Datenstrukturen und Methoden.

B) Inhalte des Themenfelds E.3

- Grundaufbau von Programmen, Syntax und elementare Anweisungen → Kapitel 1
- Variablen, Zuweisungen, elementare Datentypen, Typkonvertierungen, Scanner-Eingaben und EVA → Kapitel 2
- Operatoren, Vergleiche und logische Ausdrücke → Kapitel 3
- Kontrollstrukturen: bedingte Anweisungen, Verzweigungen und Struktogramm-Notation → Kapitel 4
- Schleifen als kontrollierte Wiederholung mit Bedingung, Zustandsänderung und Laufspur → Kapitel 5
- Datenstrukturen mit Arrays und Strings als Anwendungs- und Vertiefungsfeld für Kontrollstrukturen → Kapitel 6
- Methoden, Schnittstellen und wiederverwendbare Verarbeitung → Kapitel 7
- Einstieg in GUI und ereignisgesteuerte Anwendungen als ereignisgesteuertes EVA-System → Kapitel 8
- Modellierung und Implementierung einfacher Algorithmen, Ausführung und Diagnose → Kapitel 9

C) Bedeutung für diese Kursseite

E3 modelliert und implementiert bereits einfache, endliche Lösungsverfahren. Q1.2 grenzt davon konkrete Such- und Sortierverfahren, ihren Vergleich sowie Laufzeit und Effizienz ab; Q1.3 vertieft rekursive Problemstrukturen.

Quelle

Hessisches Ministerium für Kultus, Bildung und Chancen (2024): Kerncurriculum gymnasiale Oberstufe Informatik, Themenfeld E.3, S. 30–31.

LERNABSCHNITT 1 VON 9 · E3

1 Einführung in Java, Syntax und Aufbau einfacher Programme

Wie Java-Code formal strukturiert wird und wo ausführbare Anweisungen stehen

1. Konzeptklärung: Programm, Quelltext und Syntax

Ein Algorithmus beschreibt einen eindeutigen, endlichen und ausführbaren Lösungsweg unabhängig von einer konkreten Programmiersprache. Ein Programm implementiert einen solchen Lösungsweg – oder einen Teil davon – in ausführbaren Anweisungen einer Programmiersprache. In Java wird diese Beschreibung als Quelltext in `.java`-Dateien notiert. Der Quelltext ist menschenlesbar, aber nicht direkt maschinenausführbar.

Begriff	Funktion im E3-Weg
Problem	fachliche Ausgangsfrage
Algorithmus	eindeutiges, endliches Lösungsverfahren
Modell / Darstellung	macht das Verfahren z. B. im Struktogramm sichtbar
Programm	Java-Implementierung eines Verfahrens oder Verarbeitungsschritts
Methode	benannte Programmeinheit für einen Teil- oder Gesamtverarbeitungsschritt
Ausführung	konkrete Wirkung bei bestimmten Eingaben und Zuständen

Für den Einstieg sind drei Syntaxideen besonders wichtig:

- Geschweifte Klammern `{ ... }` bilden zusammengehörige Blöcke.
- Die `main`-Methode ist der Startpunkt des Programms.
- Eine Anweisung wird in Java mit einem Semikolon `;` abgeschlossen.

2. Strukturelle Einordnung: Übersetzung und Laufzeit

Java-Programme durchlaufen zwei klar getrennte Schritte. Zunächst übersetzt der Compiler den Quelltext in Bytecode (`.class`), anschließend führt die Java Virtual Machine (JVM) diesen Bytecode aus. Die Trennung von Übersetzung und Ausführung macht nachvollziehbar, warum Übersetzungsfehler und Laufzeitfehler unterschiedliche Ursachen haben.

1. Quelltext .java-Datei, für Menschen lesbar
2. Compiler prüft Syntax und übersetzt
3. Bytecode .class-Datei, Zwischenform
4. JVM führt den Bytecode zur Laufzeit aus
5. Ausgabe entsteht erst beim Programmlauf

> RELATION **D-Book-Wiki-Vertiefung**
>
> Der E3-Ablauf Quelltext → Compiler → Bytecode → JVM wird im Wiki als Verbindung von Hochsprache, Laufzeitumgebung und plattformübergreifender Ausführung eingeordnet.
>
> Java und plattformübergreifende Programmausführung
>
> Von Hochsprachen zu objektorientierter Programmierung

3. Formale Umsetzung: Klasse, main-Methode und Anweisung

Eine ausführbare Java-Anwendung besitzt mindestens eine Klasse mit einer main-Methode als Startpunkt. Die Anweisungen innerhalb von `main` werden in der notierten Reihenfolge ausgeführt. Jede einzelne Anweisung muss syntaktisch abgeschlossen werden; bei einfachen Java-Anweisungen übernimmt das Semikolon diese Rolle.

> **BEGRIFF Leselogik: Startpunkt, Block und Abschluss**
>
> Beim Ausführen sucht die JVM die passende `main`-Methode. Erst dort beginnt der Programmablauf; die geschweiften Klammern markieren, welche Anweisungen zu diesem Startpunkt gehören. Das Semikolon zeigt dem Compiler, dass eine Anweisung beendet ist.

Im E3-Kontext dient die Klasse zunächst vor allem als notwendiger Java-Rahmen für ein lauffähiges Programm. Der systematische Objekt- und Klassenbegriff wird später in Q1.1 ausgearbeitet.

4. Beispiel: Minimales, vollständiges Java-Programm

```
public class HalloWelt {
    public static void main(String[] args) {
        System.out.println("Hallo Welt");
    }
}
```

Syntaxelement	Bedeutung im Beispiel
public class HalloWelt { ... }	Klassenrahmen des Programms
public static void main(String[] args) { ... }	Startpunkt des Programmlaufs
System.out.println("Hallo Welt");	auszuführende Anweisung
{ und }	Beginn und Ende eines Blocks
;	Abschluss einer Anweisung

Tabelle 2: Syntaxanalyse des HalloWelt-Programms

Die äußeren geschweiften Klammern umschließen die Klasse. Die inneren geschweiften Klammern umschließen die `main`-Methode. Anweisungen innerhalb von `main` werden in der notierten Reihenfolge ausgeführt. `System.out.println("Hallo Welt");` ist eine solche Anweisung; das Semikolon gehört zu ihr und markiert ihren Abschluss.

> **MATERIALISIERT Ausführungskette anzeigen**

Schritt	Wirkung	Fachliche Ebene
1	Der Quelltext wird als HalloWelt.java gespeichert.	Quelltext
2	Der Compiler prüft die Syntax.	Übersetzung
3	Bei korrektem Code entsteht Bytecode in HalloWelt.class.	Zwischenform
4	Die JVM sucht die main-Methode als Einstiegspunkt.	Ausführung
5	println erzeugt die sichtbare Ausgabe.	Konsole

Tabelle 3: Ausführungsschritte des HalloWelt-Programms

> **MATERIALISIERT Konsolenausgabe und Fehlerfall anzeigen**
>
> **Konsolenausgabe**

```
Hallo Welt
```

> **Fehlerfall**
>
> Fehlt das Semikolon nach `System.out.println("Hallo Welt")`, kann der Compiler die Anweisung nicht abschließen.

```
';' expected
```

Deutung: Das Programm steht nicht schon als fertige Ausführung im Quelltext. Erst nach erfolgreicher Übersetzung kann die JVM die `main`-Methode starten und die Ausgabe erzeugen.

Das Beispiel enthält genau eine Klasse, genau eine Startmethode und genau eine auszuführende Anweisung. Damit ist die grundlegende Programmstruktur vollständig sichtbar, ohne zusätzliche Sprachelemente vorwegzunehmen.

Dieses Minimalbeispiel zeigt nur die Ausgabe und den Java-Rahmen. Im nächsten Schritt werden zunächst feste Werte direkt zugewiesen; erst danach kommt `Scanner` als Eingabeschnittstelle hinzu.

5. Typische Fehlvorstellungen

Häufig wird angenommen, die Datei werde „Zeile für Zeile“ ohne Struktur ausgeführt. Tatsächlich startet die Ausführung an der `main`-Methode, nicht am Anfang der Datei.

Ebenfalls verbreitet ist die Gleichsetzung von Quelltext und lauffähigem Programm. Der Quelltext muss zuerst erfolgreich übersetzt werden; erst der erzeugte Bytecode ist ausführbar.

Für ein tragfähiges Grundverständnis ist die klare Unterscheidung der Ebenen entscheidend: Konzept (Programm als Anweisungsfolge), formale Syntax (Klasse, `main`-Methode, Klammern und Semikolon) und technischer Prozess (Compiler/JVM). Der Programmrahmen zeigt, wo ausführbare Anweisungen stehen. Im nächsten Schritt geht es darum, was solche Anweisungen mit Werten tun können: Werte speichern, ausgeben und später ersetzen.

ABSCHNITTSENDE · KAPITEL 1
Weiter: 2 Variablen als Speicherorte

LERNABSCHNITT 2 VON 9 · E3

2 Variablen als Speicherorte

Werte benennen, speichern, ausgeben und neu belegen

Nach der Programmgrundstruktur rückt der Inhalt einzelner Anweisungen in den Mittelpunkt: Werte können gespeichert, ausgegeben und später durch neue Werte ersetzt werden.

1. Konzeptklärung: Variable als Speicherort

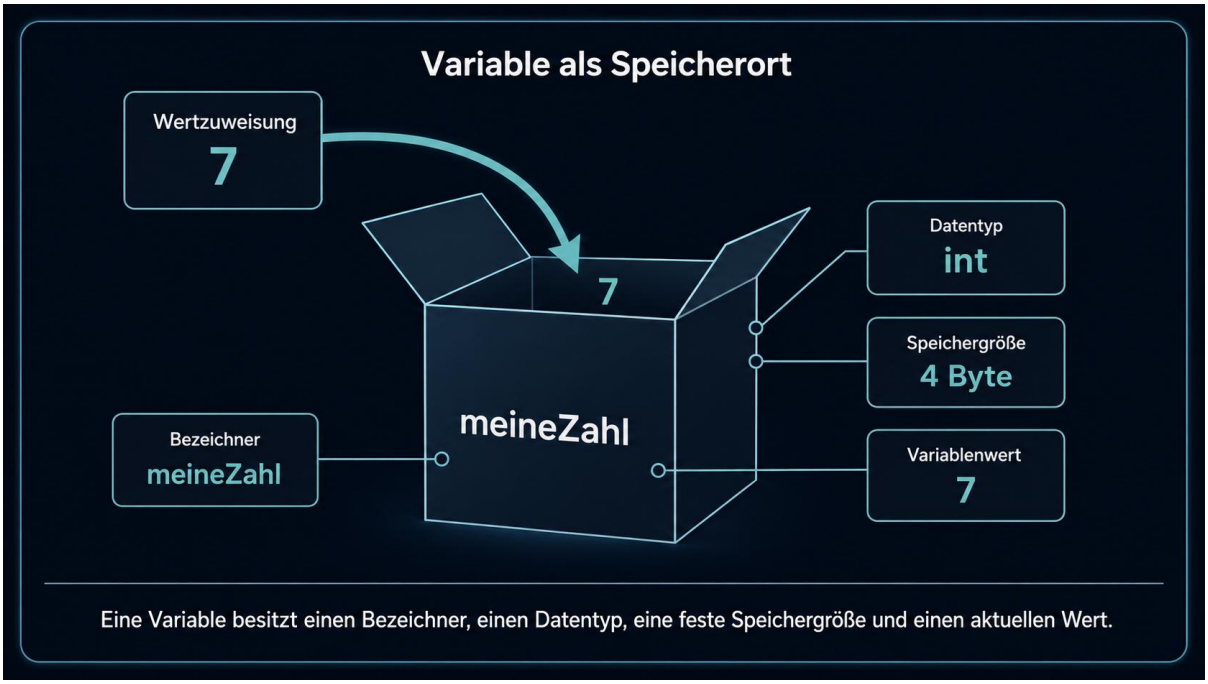

Abbildung 1: Didaktisch begrenztes Bildmodell: Im E3-Kern bestimmen Bezeichner, deklarierter Datentyp und aktueller Wert den Variablenzustand. Speichergröße und konkrete Speicherorganisation sind technische Vertiefungen.

Eine Variable verbindet einen Bezeichner mit einem deklarierten Datentyp und einem aktuellen Wert beziehungsweise Zustand. Über ihren Bezeichner wird der aktuelle Wert im Quelltext gelesen oder verändert; der Datentyp begrenzt Wertebereich und zulässige Operationen.

Variablen machen Programmzustände sichtbar und veränderbar. Während ein Programm läuft, können Werte gespeichert, ausgegeben und später durch neue Werte ersetzt werden. Der Bezeichner und der Datentyp bleiben dabei gleich; der gespeicherte Wert kann sich ändern.

2. Deklaration, Initialisierung und Neuzuweisung

Das folgende Beispiel bleibt bewusst bei `int`. Dadurch sind die vier Formen klar unterscheidbar:

- Die **Deklaration** legt Datentyp und Bezeichner fest.
- Die **Initialisierung** weist einer bereits deklarierten Variablen erstmals einen Wert zu.
- **Deklaration und Initialisierung** können auch in einer einzigen Anweisung zusammengefasst werden.
- Eine **Neuzuweisung** ersetzt den aktuell gespeicherten Wert durch einen neuen Wert.

Codezeile	Bedeutung	Zustand danach
int kontoStand;	Deklaration	Die Variable ist bekannt, aber noch nicht mit einem Wert belegt.
kontoStand = 100;	Initialisierung	kontoStand speichert erstmals 100.

Codezeile	Bedeutung	Zustand danach
int alter = 16;	Deklaration + Initialisierung	alter wird angelegt und speichert sofort 16.
kontoStand = 250;	Neuzuweisung	kontoStand speichert jetzt 250.

Tabelle 4: Deklaration, Initialisierung, kombinierte Deklaration und Neuzuweisung

```
public class VariablenDemo {
    public static void main(String[] args) {
        int kontoStand;          // Deklaration
        kontoStand = 100;        // Initialisierung

        int alter = 16;          // Deklaration + Initialisierung

        System.out.println("Konto:");
        System.out.println(kontoStand);

        System.out.println("Alter:");
        System.out.println(alter);

        kontoStand = 250;        // Neuzuweisung

        System.out.println("Neuer Kontostand:");
        System.out.println(kontoStand);
    }
}
```

Zunächst werden Werte direkt im Programm zugewiesen, damit Speicherort, Datentyp und Wertänderung sichtbar werden. Feste Werte sind hier ein bewusst reduziertes Lernmodell: Sie machen sichtbar, welcher Wert gerade in einer Variable gespeichert ist und wie eine Zuweisung diesen Zustand verändert.

BEGRIFF Sicherung: direkte Wertzuweisung deuten

Eine Variable wird als Speicherort mit Datentyp, Bezeichner und aktuellem Wert gelesen. Eine Zuweisung speichert einen festen Wert in diesem Speicherort; eine Neuzuweisung ersetzt den gespeicherten Zustand. `System.out.println(...)` macht den aktuellen Wert sichtbar.

BEGRIFF Anschluss an die Werkzeugarbeit

Die ersten beiden Aufgaben bleiben bewusst bei direkter Wertzuweisung. Sie sichern Java-Rahmen, Deklaration, Initialisierung, Neuzuweisung und Ausgabe, bevor Eingaben von außen hinzukommen.

- Java-Werkzeug öffnen: Werte direkt zuweisen und ausgeben
- Java-Werkzeug öffnen: Variablen initialisieren, neu zuweisen und Zustände ausgeben

EVA als frühes Grundmodell: Eingabe, Verarbeitung, Ausgabe

Nach dieser Sicherung wird `Scanner` als Eingabeschnittstelle eingeführt. Werte kommen nun nicht mehr nur als feste Startwerte aus dem Quelltext, sondern von außen ins Programm. Sie werden gespeichert, verarbeitet und anschließend wieder sichtbar gemacht.

Datentypen werden dadurch funktional wichtig: Die Art der Eingabe entscheidet mit, welcher Datentyp passt. Ab den folgenden Aufgaben werden Eingaben aktiv über `Scanner` modelliert.

- **Eingabe:** Werte gelangen über eine Schnittstelle ins Programm, im Konsolenprogramm zum Beispiel über `scanner.nextLine()`, `scanner.nextInt()` oder `scanner.nextDouble()`.
- **Verarbeitung:** Werte werden gespeichert, verändert, berechnet, verglichen oder später über Bedingungen zur Ablaufsteuerung genutzt.
- **Ausgabe:** Zustände, Ergebnisse oder Entscheidungen werden mit `System.out.println(...)` sichtbar gemacht.

Die Grundbewegung lautet deshalb: **Eingabe → Speicherung → Verarbeitung → Ausgabe**. Modellierung ist die Tätigkeit, mit der dieser Ablauf vor der Ausführung entworfen wird, kein zusätzlicher Laufzeitschritt in EVA.

Dieser Gedanke bereitet die spätere GUI-Programmierung vor: Dort liefern Komponenten Eingabezustände, ein Ereignis startet die Verarbeitung und Labels oder Felder zeigen die Ausgabe.

```
import java.util.Scanner;

public class BegruessungEVA {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Name: ");
        String name = scanner.nextLine();

        String begruessung = "Hallo, " + name;

        System.out.println(begruessung);
    }
}
```

- **Eingabe:** `scanner.nextLine()` liest den Namen.
- **Verarbeitung:** Aus dem Namen wird ein Begrüßungstext gebildet.
- **Ausgabe:** `System.out.println(begruessung)` zeigt das Ergebnis.

4. Datentypen als Modellierungsentscheidung

Nach Deklaration, Initialisierung und Neuzuweisung stellt sich die nächste Modellierungsfrage: Welche Art von Wert soll eine Variable speichern? Eine Anzahl ist etwas anderes als ein Messwert, ein Wahrheitswert, ein einzelnes Kennzeichen oder ein Text. Java verlangt deshalb, dass der Datentyp einer Variablen festgelegt wird.

Der Datentyp ist nicht nur eine Beschriftung. Er legt fest, welche Werte gespeichert werden dürfen, welche Operationen sinnvoll sind und welche Fehler der Compiler bereits vor der Ausführung erkennen kann. Für zählbare Werte eignet sich häufig `int`, für Messwerte mit Nachkommastellen `double`, für Ja/Nein-Informationen `boolean` und für einzelne Zeichen oder Kennzeichen `char`.

Texte werden in Java mit `String` gespeichert. `String` ist jedoch kein primitiver Datentyp, sondern ein Referenztyp. Für den Einstieg reicht zunächst: Texte können ausgegeben und gespeichert werden; die genauere Arbeit mit Strings folgt später im Datenstrukturkapitel.

Beispiel: Ein Konto mit Scanner-Eingaben und passenden Datentypen

```
import java.util.Scanner;

public class KontoDatentypenDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Kontoinhaber: ");
        String inhaber = scanner.nextLine();

        System.out.print("Kontostand in Cent: ");
        int kontoStandCent = scanner.nextInt();

        System.out.print("Zinssatz in Prozent: ");
        double zinssatzProzent = scanner.nextDouble();

        boolean positivesGuthaben = kontoStandCent > 0;

        System.out.println("Konto:");
        System.out.println(inhaber);

        System.out.println("Kontostand in Cent:");
        System.out.println(kontoStandCent);

        System.out.println("Zinssatz:");
        System.out.println(zinssatzProzent);

        System.out.println("Positives Guthaben:");
        System.out.println(positivesGuthaben);
```

```
    }
}
```

Information im Konto-Beispiel	Datentyp im Beispiel	Warum passend?
Kontoinhaber	String	Textinformation; wird mit scanner.nextLine() eingelesen
Kontostand in Cent	int	exakter Geldbetrag als ganzer Centwert; wird mit scanner.nextInt() eingelesen
Zinssatz in Prozent	double	Mess- und Rechenwert mit Nachkommastellen; wird mit scanner.nextDouble() eingelesen
positives Guthaben	boolean	Wahrheitswert aus der Verarbeitung kontoStandCent > 0

Tabelle 5: Datentypwahl im Konto-Beispiel

Das Beispiel zeigt: Datentypen entstehen nicht zufällig, sondern aus der Bedeutung der Daten. Wer ein Konto modelliert, muss entscheiden, welche Informationen eingegeben, in Variablen gespeichert, durch Operatoren oder Vergleiche verarbeitet und über `println` ausgegeben werden.

3. Typkonvertierungen: numerisch umwandeln und Text parsen

Typkonvertierung behandelt Werte in verschiedenen numerischen Typen; beim Parsen wird eine Zeichenkette als Zahl gelesen. Beides ist in E3 wichtig, weil Eingaben häufig zunächst als Text vorliegen.

```
int summe = 17;
int anzahl = 2;
double durchschnitt = (double) summe / anzahl; // numerische Umwandlung: 8.5

String textEingabe = "12.5";
double messwert = Double.parseDouble(textEingabe); // Parsen: Text → Zahl
```

Das Casting `(double)` sorgt hier dafür, dass die Division einen `double`-Wert liefert. `Double.parseDouble(...)` erwartet dagegen einen Text, der als Zahl geschrieben ist; eine ungültige Eingabe ist ein späterer Diagnose- und Validierungsfall.

BEGRIFF Anschluss an die Werkzeugarbeit

Im Java-Werkzeug wird dieser Gedanke anschließend auf einen neuen Kontext übertragen: Ein Versandauftrag soll mit passenden Datentypen umgesetzt und in der Konsole sichtbar gemacht werden. Danach werden die Modellierungsentscheidungen begründet.

- Java-Werkzeug öffnen: Versandauftrag mit Scanner-Eingaben und passenden Datentypen modellieren
- Java-Werkzeug öffnen: Datentypwahl, Scanner-Eingaben und EVA-Struktur begründen

BEGRIFF Sicherung: Modellierungsentscheidungen begründen

Am Ende der Werkzeugarbeit steht nicht nur die Frage, ob der Code läuft. Entscheidend ist auch, ob die gewählten Variablen den fachlichen Gegenstand sinnvoll repräsentieren. Dazu wird begründet, warum bestimmte Informationen als `int`, `double`, `boolean`, `char` oder `String` modelliert wurden.

VERTIEFUNG Referenz: vollständige Tabelle der primitiven Datentypen in Java

Die folgende Tabelle dient als Referenz. Im sichtbaren E3-Kern stehen `int`, `double`, `boolean`, `char`, `String` und Array/Feld im Mittelpunkt. Die weiteren primitiven Typen verdeutlichen Wertebereiche; konkrete Speicherorganisation ist kein notwendiger E3-Basisbegriff.

Datentyp	Größe	Wertebereich	Standardwert bei Feldern und Arrayelementen
boolean	JVM-abhängig (logisch 1 Bit)	true, false	false

Datentyp	Größe	Wertebereich	Standardwert bei Feldern und Arrayelementen
char	16 Bit	U+0000 bis U+FFFF	'\u0000'
byte	8 Bit	-128 bis 127	0
short	16 Bit	-32.768 bis 32.767	0
int	32 Bit	-2.147.483.648 bis 2.147.483.647	0
long	64 Bit	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807	0L
float	32 Bit	ca. ±1,4E-45 bis ±3,4E+38	0.0f
double	64 Bit	ca. ±4,9E-324 bis ±1,7E+308	0.0d

Tabelle 6: Primitive Datentypen in Java

Lokale Variablen werden nicht automatisch initialisiert. Sie müssen vor dem Lesen einen Wert erhalten.

4. Primitive Datentypen und Objekte sauber unterscheiden

Primitive Typen speichern unmittelbare Einzelwerte (z. B. Zahl, Zeichen, Wahrheitswert). Bei Referenztypen verweist eine Variable dagegen auf ein Objekt, das intern mehrere Daten und ggf. Methodenbezug zusammenfasst.

```
int punkte = 42;                  // primitiver Wert liegt direkt vor
int[] messwerte = {3, 5, 8};      // Variable speichert Referenz auf ein Array-Objekt
String name = "Lea";              // Referenz auf ein String-Objekt
```

Diese Unterscheidung ist für spätere Kapitel zentral: Arrays und andere Datenstrukturen arbeiten mit Referenzen auf Objekte, während numerische Berechnungen häufig auf primitiven Typen stattfinden.

VERTIEFUNG Vertiefung: Typumwandlung und Casting

Bei der Typumwandlung wird ein Wert in einen anderen Datentyp überführt. Java unterscheidet automatische (implizite) Umwandlung und explizites Casting.

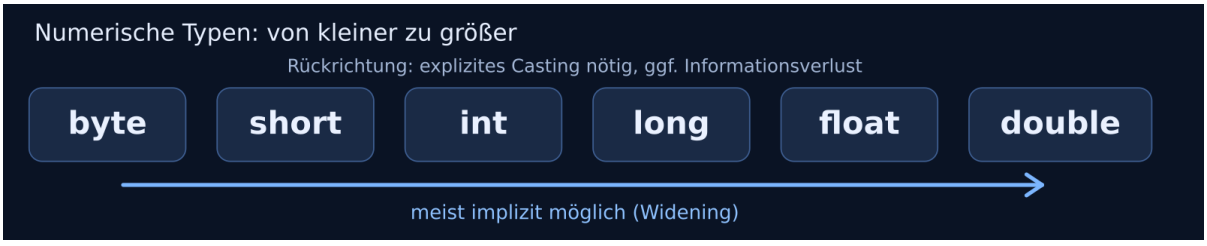

Abbildung 2: Richtung typischer numerischer Typumwandlungen

Von links nach rechts ist die Umwandlung typischerweise automatisch möglich; in Gegenrichtung nur mit explizitem Casting und möglichem Genauigkeits- oder Werteverlust.

Automatische (implizite) Umwandlung

```
int anzahl = 7;
double alsDouble = anzahl; // int -> double
float alsFloat = anzahl;   // int -> float
```

Arithmetische Beförderung bei kleinen Ganzzahltypen

```
byte a = 10;
short b = 20;
int summe = a + b; // byte/short werden in Ausdrücken zu int
```

Explizite Umwandlung (Casting)

```
double preis = 19.99;
int euro = (int) preis;   // 19, Nachkommastellen fallen weg

long gross = 4_000_000_000L;
int klein = (int) gross;  // Überlauf möglich

float note = 2.7f;
int ganzzahl = (int) note; // 2
```

5. Typische Fehlvorstellungen

Der Operator = wird oft als mathematische Gleichheit interpretiert. In Java bedeutet er Zuweisung: Der rechte Ausdruck wird ausgewertet und sein Ergebnis links gespeichert.

- Ein Datentyp ist keine Dekoration: Er legt zulässige Werte, sinnvolle Operationen und mögliche Compilerfehler mit fest.
- `int` ist für Zählwerte geeignet, aber nicht automatisch für Messwerte mit Nachkommastellen.
- Geldbeträge können bewusst als Centwerte mit `int` modelliert werden, um Nachkommastellenprobleme zu vermeiden.
- `String` ist Text, aber kein primitiver Datentyp.
- Typumwandlung löst nicht jedes Modellierungsproblem; besser ist oft, den passenden Datentyp von Anfang an zu wählen.

Damit wird der nächste Schritt vorbereitet: Operatoren verarbeiten typisierte Werte. Ob eine Operation sinnvoll ist und welches Ergebnis entsteht, hängt vom Datentyp der beteiligten Werte ab.

ABSCHNITTSENDE · KAPITEL 2
Weiter: 3 Operatoren und Vergleichsausdrücke

LERNABSCHNITT 3 VON 9 · E3

3 Operatoren und Vergleichsausdrücke

Rechnen, vergleichen und Ergebnistypen sichern

1. Konzeptklärung: Ausdruck als auswertbare Struktur

Variablen und Datentypen liefern die Wertebasis. Operatoren verarbeiten diese Werte: arithmetische Operatoren berechnen neue Zahlenwerte, Vergleichsoperatoren prüfen Beziehungen zwischen Werten, und logische Operatoren verbinden oder negieren Wahrheitswerte.

Im EVA-Modell gehören Operatoren zur Verarbeitungsschicht. Sie verarbeiten Werte, die zuvor eingelesen oder in Variablen gespeichert wurden. Feste Beispielwerte dienen in diesem Kapitel vor allem der nachvollziehbaren Auswertungsspur.

Für Java ist die genaue Schreibweise entscheidend. Einige Zeichen sehen ähnlich aus wie in der Mathematik, haben aber im Programm eine genau festgelegte Rolle: `=` weist einen Wert zu, `==` vergleicht zwei Werte.

Ein Ausdruck kombiniert Werte, Variablen, Konstanten und Operatoren zu einer auswertbaren Einheit. Das Ergebnis eines Ausdrucks hat einen Datentyp, zum Beispiel numerisch oder boolesch. Operatoren sind damit nicht nur Rechenzeichen, sondern zentrale Bausteine zur Formulierung von Berechnungen und Vergleichen.

> **BEGRIFF Operatorfamilien im E3-Kern**
>
> - **arithmetisch:** `+ - * / %` berechnen Werte,
> - **vergleichend:** `== != < <= > >=` liefern `boolean`,
> - **logisch:** `&&`, `||` und `!` verbinden oder negieren Wahrheitswerte.

2. Rechnen mit Zahlen und Rechnen mit Variablen

Beim Rechnen mit Zahlen stehen die Werte direkt im Ausdruck: `8 + 3` kann sofort zu `11` ausgewertet werden. Solche direkt notierten Werte nennt man Literalwerte.

Beim Rechnen mit Variablen stehen im Ausdruck dagegen Namen von Speicherorten: In `a + b` rechnet Java nicht mit den Buchstaben `a` und `b`, sondern liest zuerst die aktuell gespeicherten Werte dieser Variablen. Erst danach wird der Ausdruck ausgewertet.

Eine Zuweisung speichert immer das Ergebnis der aktuellen Auswertung. `int summe = a + b;` speichert also nicht dauerhaft die Formel `a + b`, sondern den Wert, der in diesem Moment aus den aktuellen Variablenwerten entsteht.

```
public class RechnenMitVariablen {
    public static void main(String[] args) {
        int a = 8;
        int b = 3;

        int summeMitZahlen = 8 + 3;
        int summeMitVariablen = a + b;

        a = 10;

        int neueSumme = a + b;

        System.out.println(summeMitZahlen);
        System.out.println(summeMitVariablen);
        System.out.println(neueSumme);
    }
}
```

Code	Was wird gelesen?	Ergebnis	Deutung

Code	Was wird gelesen?	Ergebnis	Deutung
int summeMitZahlen = 8 + 3;	8 und 3 stehen direkt im Ausdruck.	11	Rechnung mit Literalwerten.
int summeMitVariablen = a + b;	a speichert 8, b speichert 3.	11	Java liest die aktuellen Variablenwerte und rechnet damit.
a = 10;	rechter Wert 10	a speichert jetzt 10	Der Wert von a wird neu zugewiesen.
int neueSumme = a + b;	a speichert jetzt 10, b speichert 3.	13	Der Ausdruck wird mit dem neuen Wert von a ausgewertet.
summeMitVariablen	bereits gespeicherter Wert	11	Die alte Ergebnisvariable ändert sich nicht automatisch.

Tabelle 7: Auswertung von Rechnungen mit Zahlen und Variablen

BEGRIFF **Merksatz: Variablen in Ausdrücken**

In einem Ausdruck steht eine Variable für ihren aktuellen gespeicherten Wert. Wird das Ergebnis in einer neuen Variable gespeichert, bleibt dieses Ergebnis erhalten, auch wenn sich die ursprünglichen Variablen später ändern.

Erst wenn klar ist, welche Werte in einem Ausdruck gelesen werden, wird die genaue Operatoren-Notation wichtig.

VERTIEFUNG **Referenz: vollständige Operatorenübersicht und Kurzzuweisungen**

Die folgende Übersicht trennt die Bedeutung eines Operators von seiner Java-Schreibweise. Die Kurzschreibweise ist nur dort angegeben, wo eine Variable durch eine abgekürzte Zuweisung verändert werden kann.

Bedeutung / interne Leseschreibweise	Java-Notation	Kurzschreibweise	Beispiel	Wirkung / Ergebnistyp
Wert zuweisen / speichern	=	—	x = 8;	Variable speichert einen neuen Wert
addieren	+	+=	x = x + 3; / x += 3;	numerischer Wert; Kurzform verändert den gespeicherten Wert
subtrahieren	-	-=	x = x - 3; / x -= 3;	numerischer Wert; Kurzform verändert den gespeicherten Wert
multiplizieren	*	*=	x = x * 3; / x *= 3;	numerischer Wert; Kurzform verändert den gespeicherten Wert
dividieren	/	/=	x = x / 3; / x /= 3;	bei int / int ganzzahlige Division
Rest bestimmen	%	%=	x = x % 3; / x %= 3;	Rest der ganzzahligen Division
Gleichheit prüfen	==	—	a == b	boolean
Ungleichheit prüfen	!=	—	a != b	boolean
kleiner prüfen	<	—	a < b	boolean
kleiner oder gleich prüfen	<=	—	a <= b	boolean
größer prüfen	>	—	a > b	boolean
größer oder gleich prüfen	>=	—	a >= b	boolean

Tabelle 8: Java-Notation für arithmetische Operatoren, Vergleichsoperatoren und Kurzschreibweisen

Arithmetische Operatoren liefern Zahlenwerte. Vergleichsoperatoren liefern Wahrheitswerte. Klammern steuern die Auswertungsreihenfolge ausdrücklich und können dadurch ein anderes Ergebnis erzeugen.

Kurzschreibweisen wie `+=` oder `*=` verbinden eine Rechenoperation mit einer Zuweisung: Die Variable wird gelesen, der Ausdruck wird berechnet und das Ergebnis wird wieder in derselben Variable gespeichert.

2. Kanonisches Beispiel: Ausdrücke auswerten

```
public class OperatorenDemo {
    public static void main(String[] args) {
        int a = 8;
        int b = 3;

        int ohneKlammern = a + b * 2;
        int mitKlammern = (a + b) * 2;
        int ganzzahlQuotient = a / b;
        int rest = a % b;

        boolean groesserAlsZehn = ohneKlammern > 10;
        boolean gleich = a == b;
        boolean ungleich = a != b;
        boolean beidePositiv = a > 0 && b > 0;
        boolean mindestensEinWertGross = a > 10 || b > 10;
        boolean nichtGleich = !(a == b);

        System.out.println(ohneKlammern);
        System.out.println(mitKlammern);
        System.out.println(ganzzahlQuotient);
        System.out.println(rest);
        System.out.println(groesserAlsZehn);
        System.out.println(gleich);
        System.out.println(ungleich);
        System.out.println(beidePositiv);
        System.out.println(mindestensEinWertGross);
        System.out.println(nichtGleich);
    }
}
```

Die festen Werte dienen hier als Beispiel-Eingaben für die Laufspur. In einem interaktiven Konsolenprogramm würden sie mit `Scanner` eingelesen.

VERTIEFUNG **Vertiefung: Operatoren mit Scanner-Eingaben**

Die gleiche Verarbeitung kann auch mit Eingaben erfolgen: Werte werden eingelesen, gespeichert, durch Operatoren ausgewertet und ausgegeben.

```
import java.util.Scanner;

public class PunktebilanzMitScanner {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Basispunkte: ");
        int basisPunkte = scanner.nextInt();

        System.out.print("Bonuspunkte: ");
        int bonusPunkte = scanner.nextInt();

        System.out.print("Mindestpunkte: ");
        int mindestPunkte = scanner.nextInt();

        int gesamtPunkte = basisPunkte + bonusPunkte;
        int differenzZumZiel = mindestPunkte - gesamtPunkte;

        boolean zielErreicht = gesamtPunkte >= mindestPunkte;

        System.out.println("Gesamtpunkte:");
        System.out.println(gesamtPunkte);

        System.out.println("Differenz zum Ziel:");
        System.out.println(differenzZumZiel);
```

```
        System.out.println("Ziel erreicht:");
        System.out.println(zielErreicht);
    }
}
```

- **Eingabe:** Basispunkte, Bonuspunkte und Mindestpunkte werden eingelesen.
- **Verarbeitung:** `+` berechnet Gesamtpunkte, `-` berechnet die Differenz zum Zielwert.
- **Vergleich:** `>=` prüft, ob die Gesamtpunkte mindestens den Zielwert erreichen.
- **Ausgabe:** Zahlenwerte und Wahrheitswert werden sichtbar gemacht.

Bei einer Zuweisung wird zuerst der Ausdruck rechts vom Zuweisungsoperator `=` ausgewertet. Erst danach wird das Ergebnis in der linken Variable gespeichert. Deshalb prüft `=` in Java keine Gleichheit; der Gleichheitsvergleich wird mit `==` geschrieben.

Arithmetische Ausdrücke folgen Auswertungsregeln: Multiplikation und Division werden vor Addition und Subtraktion ausgewertet. Klammern können diese Reihenfolge verändern. Vergleichsausdrücke berechnen dagegen keine Zahl, sondern prüfen eine Aussage und liefern `true` oder `false`.

Wichtig ist die Trennung von Java-Notation und mathematischer Bedeutung: `=` weist einen Wert zu, `==` prüft Gleichheit, `!=` prüft Ungleichheit, `<=` bedeutet kleiner oder gleich und `>=` bedeutet größer oder gleich. Bei `int / int` entsteht eine ganzzahlige Division; `%` liefert den Rest dieser Division.

3. Auswertungsspur: Rechen-, Vergleichs- und Logikergebnisse

Ausdruck	Auswertungsschritte	Ergebnis	Ergebnistyp
a + b * 2	b * 2: 3 * 2 ergibt 6, danach a + 6: 8 + 6 ergibt 14	14	int
(a + b) * 2	a + b: 8 + 3 ergibt 11, danach 11 * 2 ergibt 22	22	int
a / b	8 / 3, ganzzahlige Division	2	int
a % b	Rest der ganzzahligen Division 8 / 3	2	int
ohneKlammern > 10	14 > 10 prüft eine Größer-als-Beziehung und ergibt true	true	boolean
a == b	8 == 3 prüft Gleichheit und ergibt false	false	boolean
a != b	8 != 3 prüft Ungleichheit und ergibt true	true	boolean
a <= b	8 <= 3 prüft kleiner oder gleich und ergibt false	false	boolean
a >= b	8 >= 3 prüft größer oder gleich und ergibt true	true	boolean
a > 0 && b > 0	beide Vergleiche sind true	true	boolean
a > 10 || b > 10	beide Vergleiche sind false	false	boolean

Tabelle 9: Auswertungsspur zu arithmetischen Ausdrücken und Vergleichsausdrücken

Die Auswertungsspur macht sichtbar, dass Ausdrücke nicht auf einmal magisch entstehen. Java wertet Teilausdrücke nach festen Regeln aus. Das Ergebnis eines arithmetischen Ausdrucks kann wieder gespeichert werden; das Ergebnis eines Vergleichsausdrucks ist ein Wahrheitswert.

MATERIALISIERT **Beispielausgabe des Programms anzeigen**

```
14
22
2
2
true
false
```

```
true
true
false
true
```

Deutung: Die ersten vier Ausgaben sind numerische Ergebnisse. Die letzten drei Ausgaben sind Wahrheitswerte aus Vergleichsausdrücken.

VERTIEFUNG **Vertiefung: Kurzzuweisungen als Zustandsänderung**

Kurzschreibweisen verbinden eine Rechenoperation mit einer Zuweisung. Die Variable wird dabei zuerst gelesen, anschließend wird der neue Wert berechnet und wieder in derselben Variable gespeichert.

```
public class KurzschreibweiseDemo {
    public static void main(String[] args) {
        int punktzahl = 10;

        punktzahl += 5;  // entspricht: punktzahl = punktzahl + 5;
        punktzahl *= 2;  // entspricht: punktzahl = punktzahl * 2;
        punktzahl -= 4;  // entspricht: punktzahl = punktzahl - 4;
        punktzahl /= 2;  // entspricht: punktzahl = punktzahl / 2;
        punktzahl %= 5;  // entspricht: punktzahl = punktzahl % 5;

        System.out.println(punktzahl);
    }
}
```

Anweisung	Langform	Wert danach
punktzahl += 5;	punktzahl = punktzahl + 5;	15
punktzahl *= 2;	punktzahl = punktzahl * 2;	30
punktzahl -= 4;	punktzahl = punktzahl - 4;	26
punktzahl /= 2;	punktzahl = punktzahl / 2;	13
punktzahl %= 5;	punktzahl = punktzahl % 5;	3

Tabelle 10: Laufspur zu Kurzschreibweisen als Zustandsänderungen

Die Kurzschreibweise ist besonders hilfreich, wenn ein gespeicherter Wert schrittweise verändert wird. Sie ersetzt keine neue Rechenregel, sondern verkürzt eine häufige Form der Neuzuweisung.

Numerische Ausdrücke können in Zahlenvariablen gespeichert werden; Vergleichs- und logische Ausdrücke liefern Wahrheitswerte und bilden so den direkten Übergang zu Bedingungen.

`a + b * 2`, `(a + b) * 2`, `a / b` und `a % b` liefern numerische Werte. Die Ergebnisse können in Variablen vom Typ `int` gespeichert oder direkt ausgegeben werden. `ohneKlammern > 10`, `a == b` und `a != b` liefern dagegen Wahrheitswerte. Die ergänzenden Vergleichsausdrücke `a <= b` und `a >= b` zeigen dieselbe Ergebnisklasse.

BEGRIFF **4. Logische Ausdrücke für zusammengesetzte Bedingungen**

Wenn mehrere Vergleichsergebnisse zu einer gemeinsamen Bedingung verbunden werden, kommen logische Operatoren hinzu. Sie arbeiten nicht auf Zahlen, sondern auf Wahrheitswerten.

Name	Java-Notation	Rolle
Logical and	&&	beide Wahrheitswerte müssen true sein
Logical or	||	mindestens ein Wahrheitswert muss true sein
Logical not	!	ein Wahrheitswert wird umgekehrt

Tabelle 11: Logische Operatoren in Java

A	B	A && B	A || B	!(A)

A	B	A && B	A || B	!(A)
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Tabelle 12: Wahrheitstabelle logischer Operatoren

`==` vergleicht hier zunächst primitive Werte. String-Inhalte werden mit `equals(...)` verglichen, nicht mit `==`. Im Zahlenkontext bedeutet `+` Addition; ist ein String beteiligt, kann `+` Zeichenketten verknüpfen.

Übergang zu Bedingungen: Ein Vergleich oder eine logische Verknüpfung liefert `true` oder `false`. Kontrollstrukturen verwenden diesen Wahrheitswert im nächsten Kapitel zur Ablaufsteuerung.

WERKZEUG **Anschluss an die Werkzeugarbeit**

Im Java-Werkzeug werden arithmetische Operatoren, Kurzschreibweisen und Vergleichsoperatoren im Taschenrechnerkontext verwendet. Die folgende Rechnerfolge überträgt diese Idee anschließend bewusst auf einen Taschenrechnerkontext: Zuerst werden arithmetische Ergebnisse berechnet, danach werden diese Ergebnisse mit Vergleichsoperatoren geprüft. Implementierungsaufgaben und Textaufgaben bleiben getrennt: Zuerst wird Code umgesetzt, danach werden eine Auswertungsspur und eine fehlerhafte Preisberechnung fachsprachlich geprüft und korrigiert.

- Java-Werkzeug öffnen: zwei Eingabewerte arithmetisch verarbeiten
- Java-Werkzeug öffnen: Rechenergebnisse vergleichen
- Java-Werkzeug öffnen: Auswertungsspur zu Beispiel-Eingaben erläutern
- Java-Werkzeug öffnen: Taschenrechner-Speicher mit Eingaben verändern
- Java-Werkzeug öffnen: Fehlerhafte Verarbeitung im EVA-Modell prüfen

Die verlinkten Aufgaben stehen zusätzlich als Offline-Arbeitsaufträge in Anhang A.

VERTIEFUNG **Vertiefung: Typische Fehlvorstellungen zu Operatoren**

Ein häufiger Fehler ist die Verwechslung von `=` und `==`. `=` weist einen Wert zu, `==` vergleicht zwei Werte auf Gleichheit.

Mathematische Vergleichszeichen werden in Java-Code nicht als Operatoren geschrieben. Java schreibt für Ungleichheit `!=`, für kleiner oder gleich `<=` und für größer oder gleich `>=`.

Kurzschreibweisen wie `+=` oder `*=` sind keine Vergleiche. Sie verändern den Wert einer bereits vorhandenen Variable.

Ebenso wird oft angenommen, jeder Ausdruck liefere eine Zahl. Vergleichsausdrücke liefern jedoch boolesche Werte und machen Aussagen über Beziehungen zwischen Werten prüfbar.

Ohne Klammerung wird die Priorität von Operatoren häufig falsch eingeschätzt. Explizite Klammern sichern die beabsichtigte Auswertungsreihenfolge.

Die klare Trennung zwischen Berechnung, Vergleich, Zuweisung und logischer Verknüpfung macht sichtbar, wie Ausdrücke Schritt für Schritt ausgewertet und fachsprachlich beschrieben werden können.

ABSCHNITTSENDE · KAPITEL 3

Weiter: 4 Kontrollstrukturen: Bedingte Anweisungen und Verzweigungen

LERNABSCHNITT 4 VON 9 · E3

4 Kontrollstrukturen: Bedingte Anweisungen und Verzweigungen

Bedingungen auswerten, Blöcke steuern, Alternativen modellieren und Verschachtelungen lesen

> **BEGRIFF Kontrollstrukturen: Zustand in Ablauf überführen**
>
> Ein Ausdruck liefert einen Wahrheitswert. Eine Kontrollstruktur nutzt ihn zur Ablaufsteuerung: `if` führt einen Block optional aus, `if-else` wählt genau einen von zwei Pfaden. Eine Bedingung kann eine `boolean`-Variable, ein Vergleichsausdruck oder eine logische Verknüpfung sein.

1. Bedingungen auswerten

Eine Bedingung ist ein Ausdruck, dessen Ergebnis `true` oder `false` ist. Solche Ausdrücke sind aus dem Operatorenkapitel bekannt: Sie lesen aktuelle Werte, vergleichen sie und liefern einen Wahrheitswert.

Der Wahrheitswert ist zunächst nur ein Ergebnis der Verarbeitung. Erst eine bedingte Anweisung nutzt dieses Ergebnis zur Ablaufsteuerung: Bei `true` kann ein bedingter Block ausgeführt werden, bei `false` wird er übersprungen.

Zustand	Bedingung	Auswertung	Ergebnis	Bedeutung
akkuProzent = 17	akkuProzent < 20	17 < 20	true	Akkuwarnung ist passend
temperaturCelsius = 28	temperaturCelsius > 30	28 > 30	false	Temperaturhinweis wird nicht ausgelöst
speicherFreiGb = 3	speicherFreiGb < 5	3 < 5	true	Speicherplatzwarnung ist passend
sensorAktiv = false	sensorAktiv	false	false	Gerätestatus bleibt unverändert

Tabelle 13: Bedingungen auswerten

2. Bedingte Anweisung mit if

Die einfachste Form ist eine einseitige bedingte Anweisung. Ein optional ausgeführter Block läuft nur dann, wenn die Bedingung `true` ergibt. Bei `false` wird dieser Block übersprungen, und die Fortsetzung läuft hinter dem `if`-Block weiter.

```
public class AkkuWarnungDemo {
    public static void main(String[] args) {
        int akkuProzent = 17;
        boolean akkuNiedrig = akkuProzent < 20;

        if (akkuNiedrig) {
            System.out.println("Akku bald laden.");
        }

        System.out.println("Statusprüfung beendet.");
    }
}
```

`akkuProzent` repräsentiert einen aktuellen Zustand. `akkuNiedrig` ist das Ergebnis der Verarbeitung: Der Vergleichsausdruck `akkuProzent < 20` liefert einen Wahrheitswert. Die bedingte Anweisung nutzt diesen Wahrheitswert zur Ablaufsteuerung.

- Der `if`-Block ist eine bedingte Ausgabe.
- Bei `true` wird der Warnhinweis ausgegeben.

- Bei `false` wird diese Ausgabe übersprungen.
- Die Abschlussausgabe läuft unabhängig von der Bedingung.

```
Eingabe/Zustand: akkuProzent = 17
Verarbeitung: akkuProzent < 20 wird zu true ausgewertet
Bedingte Ausgabe: Warnhinweis wird nur bei true ausgegeben
Fortsetzung: Statusprüfung beendet läuft immer
```

Die bedingte Anweisung verbindet einen aktuellen Zustand mit einer zustandsabhängigen Reaktion.

- Die Bedingung steht in runden Klammern.
- Der bedingte Block steht in geschweiften Klammern.
- Bei `false` wird der Block übersprungen.
- Danach läuft das Programm weiter.
- Das ist noch keine zweiseitige Verzweigung.

3. Verzweigung mit if-else

Eine Verzweigung ergänzt zur Bedingung eine Alternative. Dadurch wird nicht nur entschieden, ob ein Block ausgeführt wird, sondern welcher von zwei Blöcken ausgeführt wird.

```
public class VerzweigungDemo {
    public static void main(String[] args) {
        int punkte = 42;

        if (punkte >= 50) {
            System.out.println("Bestanden.");
        } else {
            System.out.println("Nicht bestanden.");
        }

        System.out.println("Pruefung beendet.");
    }
}
```

Eine Verzweigung mit `if-else` beschreibt zwei alternative Pfade. Ist die Bedingung `true`, wird der `if`-Block ausgeführt. Ist die Bedingung `false`, wird der `else`-Block ausgeführt. Bei `if-else` wird genau einer der beiden Blöcke ausgeführt.

Struktur	Bedingung true	Bedingung false	Fachliche Bedeutung
if	if-Block wird ausgeführt	if-Block wird übersprungen	ein zusätzlicher Block ist optional
if-else	if-Block wird ausgeführt	else-Block wird ausgeführt	genau eine Alternative wird gewählt

Tabelle 14: Vergleich von if und if-else

VERTIEFUNG Aufbauwissen: Verschachtelte bedingte Anweisungen

Bei einer Verschachtelung steht eine bedingte Anweisung innerhalb eines anderen bedingten Blocks. Die innere Bedingung wird nur geprüft, wenn der äußere Block ausgeführt wird. Dadurch lassen sich mehrstufige Entscheidungen modellieren: Zuerst wird eine Hauptfrage geklärt, danach werden nur im passenden Zweig weitere Folgefragen geprüft.

Eine verschachtelte bedingte Anweisung prüft eine zweite Bedingung also nur dann, wenn der äußere Zweig überhaupt erreicht wurde. Das unterscheidet sie von unabhängigen `if`-Anweisungen: Unabhängige `if`-Anweisungen prüfen mehrere Fragen nacheinander und unabhängig voneinander. Verschachtelte `if`-Anweisungen prüfen eine Folgefrage nur innerhalb eines erreichten Zweigs. Das ist sinnvoll, wenn die zweite Frage nur unter einer ersten Voraussetzung fachlich relevant ist.

```
public class GeraeteStatusDemo {
    public static void main(String[] args) {
        boolean geraetAktiv = true;
        int akkuProzent = 12;

        if (geraetAktiv) {
```

```
            System.out.println("Gerät ist aktiv.");

            if (akkuProzent < 20) {
                System.out.println("Akku bald laden.");
            } else {
                System.out.println("Akkustand ausreichend.");
            }
        } else {
            System.out.println("Gerät ist nicht aktiv.");
        }

        System.out.println("Statusprüfung beendet.");
    }
}
```

Zuerst wird geprüft, ob das Gerät aktiv ist. Nur wenn das Gerät aktiv ist, wird der Akkustand ausgewertet. Ist das Gerät nicht aktiv, wird die innere Akkuprüfung nicht erreicht. Die innere Verzweigung ist also abhängig vom äußeren Zweig. Die Abschlussausgabe steht außerhalb der Verschachtelung und läuft immer.

Schritt	Leselogik
1	Äußere Bedingung prüfen: geraetAktiv
2	Bei true: aktiver Zweig wird betreten
3	Innere Bedingung prüfen: akkuProzent < 20
4	Passende innere Ausgabe ausführen
5	Nach allen Blöcken: Statusprüfung beendet.

Tabelle 15: Leselogik einer verschachtelten bedingten Anweisung

Der Unterschied wird besonders deutlich, wenn zwei Prüfungen unabhängig nacheinander stehen:

```
if (geraetAktiv) {
    System.out.println("Gerät ist aktiv.");
}

if (akkuProzent < 20) {
    System.out.println("Akku bald laden.");
}
```

Diese beiden Prüfungen sind unabhängig. Die Akkuwarnung kann auch geprüft werden, wenn das Gerät nicht aktiv ist. Dagegen ist die Akkuprüfung im folgenden Beispiel eine Folgeprüfung innerhalb des aktiven Gerätezweigs:

```
if (geraetAktiv) {
    System.out.println("Gerät ist aktiv.");

    if (akkuProzent < 20) {
        System.out.println("Akku bald laden.");
    }
}
```

Im EVA-Modell bedeutet Verschachtelung: Die Verarbeitung wird nicht nur einmal abhängig vom Zustand gesteuert, sondern mehrstufig. Eine Eingabe oder ein Zustand führt zunächst zu einer Hauptentscheidung. Innerhalb des gewählten Zweigs können weitere Zustände geprüft werden. So entstehen zustandsabhängige Reaktionsketten.

In ereignisgesteuerten Programmen ist diese Denkweise besonders wichtig. Ein Ereignis startet einen Durchlauf. Der Event-Handler liest aktuelle Zustände und prüft dann schrittweise, welche Reaktion sinnvoll ist. Verschachtelte Bedingungen helfen, solche Folgeentscheidungen nachvollziehbar zu strukturieren.

Im Struktogramm entspricht eine Verschachtelung einer Auswahl innerhalb eines Zweigs. Dadurch wird sichtbar, dass die innere Entscheidung nur nach einer vorherigen Entscheidung erreicht wird.

4. Laufspur: Bedingung, Zweig und Fortsetzung

Die Laufspur zeigt, wie Java das `if-else`-Beispiel Schritt für Schritt auswertet. Im Beispiel ist `punkte` gleich `42`.

Schritt	Anweisung / Prüfung	Auswertung	Wirkung
1	int punkte = 42;	Wert speichern: punkte = 42	Ausgangszustand
2	punkte >= 50	42 >= 50 ergibt false	if-Block wird nicht gewählt
3	else	Bedingung war false	else-Block wird ausgeführt
4	System.out.println("Nicht bestanden.");	keine weitere Prüfung	Konsolenausgabe entsteht
5	System.out.println("Pruefung beendet.");	nach der Verzweigung	Programm läuft weiter

Tabelle 16: Laufspur einer if-else-Verzweigung

Die Laufspur zeigt: Eine Verzweigung unterbricht den Ablauf nicht dauerhaft. Sie wählt genau einen passenden Block aus. Danach läuft das Programm hinter der Verzweigung weiter.

5. Struktogramme: sprachunabhängige Ablaufstruktur

Ein Struktogramm stellt einen Algorithmus unabhängig von einer konkreten Programmiersprache durch Strukturblöcke für Sequenz, Auswahl, Wiederholung und Block- beziehungsweise Methodenaufruf dar.

Die Ablaufstruktur ist also sprachunabhängig. Java-nahe Variablendeklarationen in den E3-Struktogrammen dienen nur als Brücke zwischen Code und Modell; sie sind kein notwendiger Bestandteil der allgemeinen Struktogrammnotation.

Die sichtbaren Beispiele behalten einzelne Java-nahe Beschriftungen, damit sich die gewohnte Schreibweise gezielt auf die Modellstruktur beziehen lässt.

> **BEGRIFF Konvention: Struktur zuerst, Java als Brücke**
>
> - Der Java-Rahmen mit Klasse und `main`-Methode wird nicht dargestellt.
> - Anweisungen im Programmrumpf werden als Ablaufblöcke dargestellt.
> - Java-nahe Deklarationen wie `int punkte = 42;` sind eine didaktische Übergangsnotation.
> - Bedingungen stehen im Entscheidungskopf, z. B. `punkte >= 50?`.
> - Bei einer bedingten Anweisung kann der Nein-Zweig leer bleiben.
> - Bei einer Verzweigung enthalten Ja- und Nein-Zweig eigene Anweisungen.
> - Nach der Auswahl läuft das Programm unterhalb der Struktur weiter.

Struktogramm: Bedingte Anweisung

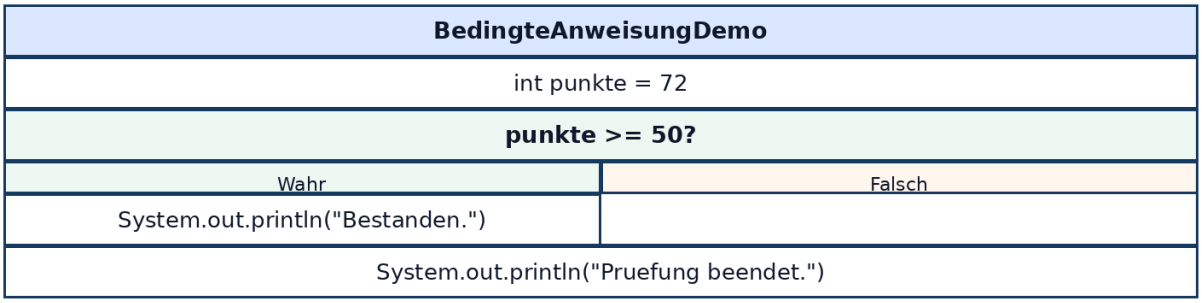

Abbildung 3: Struktogramm · notation

Struktogramm: Verzweigung

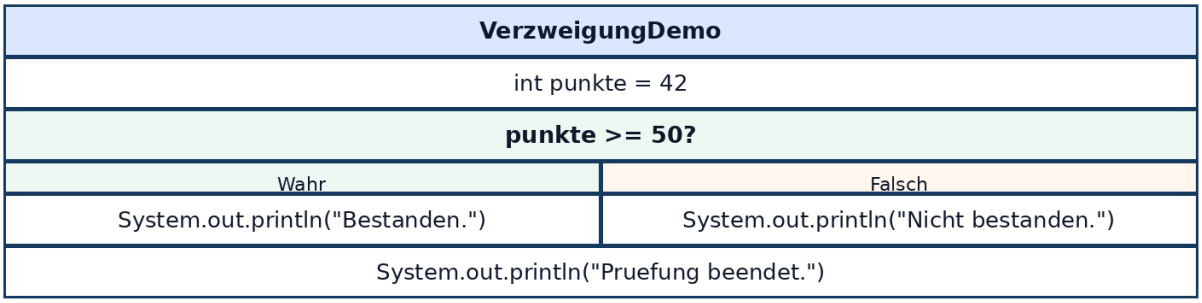

Abbildung 4: Struktogramm · if

Beide Struktogramme zeigen dieselbe Grundidee: Zuerst entsteht ein Programmzustand durch eine Variablenfestlegung. Danach wird eine Bedingung mit diesem aktuellen Wert ausgewertet. Erst aus dieser Auswertung ergibt sich, ob ein Block übersprungen wird oder welcher Zweig ausgeführt wird.

Damit wird sichtbar, dass die Bedingung nicht isoliert funktioniert. Sie bezieht sich auf zuvor gespeicherte Werte und steuert auf dieser Grundlage den weiteren Ablauf.

WERKZEUG **Anschluss an die Werkzeugarbeit**

Die Kino-Folge führt von einem optionalen Statushinweis mit `if` über eine Reservierungsprüfung als Struktogramm zur zweiseitigen Einlassentscheidung und endet mit einer offenen Problemaufgabe, in der mehrere Eingaben, Bedingungen und Alternativen selbst modelliert werden. Die Aufgaben arbeiten mit Scanner-Eingaben oder führen schrittweise dorthin: Eingabe über Scanner → Speicherung in Variablen → Verarbeitung durch Operatoren und Bedingungen → Ausgabe über `println`. Die abschließende offene Modellierungsaufgabe kann solche mehrstufigen Entscheidungen nutzen: Innerhalb eines gewählten Hauptzweigs können weitere Bedingungen geprüft und passende Hinweise ausgegeben werden.

- Java-Werkzeug öffnen: optionalen Hinweis mit if ausgeben
- Java-Werkzeug öffnen: Reservierungsprüfung als Struktogramm modellieren
- Java-Werkzeug öffnen: Entscheidung zwischen zwei Alternativen umsetzen
- Java-Werkzeug öffnen: eigenes Regelmodell entwerfen und implementieren

Die verlinkten Aufgaben stehen zusätzlich als Offline-Arbeitsaufträge in Anhang A.

VERTIEFUNG **Vertiefung: Typische Fehlvorstellungen zu Bedingungen**

- **`if` und `if-else` werden verwechselt:** Bei `if` kann ein Block ausgeführt oder übersprungen werden. Bei `if-else` wird genau einer von zwei Blöcken ausgeführt.
- **Bedingung und Anweisung werden verwechselt:** Die Bedingung liefert nur `true` oder `false`. Die Anweisungen im Block führen die eigentliche Aktion aus.
- **`=` und `==` werden verwechselt:** `=` weist zu. `==` vergleicht.
- **Klammern werden unterschätzt:** Runde Klammern enthalten die Bedingung. Geschweifte Klammern begrenzen den Anweisungsblock.
- **Innere Bedingungen werden zu früh erwartet:** Bei verschachtelten Bedingungen wird die innere Bedingung nur geprüft, wenn der äußere Block erreicht wurde.
- **`else`-Zugehörigkeit bleibt unklar:** Ein `else` gehört in Java immer zum nächstliegenden passenden `if`. Geschweifte Klammern machen die Zugehörigkeit eindeutig sichtbar.
- **Nach der Verzweigung läuft das Programm weiter:** Eine Verzweigung beendet nicht automatisch das Programm.

VERTIEFUNG **Ausblick: Mehrfachverzweigung mit switch**

`switch` ist keine Schleife, sondern eine Mehrfachverzweigung. Die Struktur modelliert mehrere mögliche Alternativpfade, wenn ein Wert auf feste Fälle geprüft wird.

```
switch (note) {
    case 1: text = "sehr gut"; break;
    case 2: text = "gut"; break;
    default: text = "weitere";
```

```
}
```

Struktogramm-Konvention: Mehrfachverzweigung

switchVerzweigung		
note		
1	2	Sonst
text = sehr gut	text = gut	text = weitere

Abbildung 5: Struktogramm · switch

Leselogik: Genau ein passender Fall wird gewählt, sonst der Default-Pfad. Das Struktogramm führt von der Auswahlbedingung zu genau einem der Alternativpfade.

Bedingte Anweisungen und Verzweigungen steuern, welcher Block ausgeführt wird. Schleifen nutzen Bedingungen anders: Sie entscheiden, ob ein Block erneut ausgeführt wird. Deshalb werden Schleifen im nächsten Abschnitt als eigene Form von Kontrollstruktur betrachtet; im GUI-Kapitel werden dieselben Bedingungen nach einem Ereignis zur zustandsabhängigen Reaktion genutzt.

ABSCHNITTSENDE · KAPITEL 4

Weiter: 5 Kontrollstrukturen: Schleifen

LERNABSCHNITT 5 VON 9 · E3

5 Kontrollstrukturen: Schleifen

Wiederholungen, Schleifenbedingungen, Zustandsänderungen und Laufspuren verstehen

Schleifen bauen auf Bedingungen auf. Der Unterschied zur Verzweigung liegt darin, dass der Schleifenkörper nach einer Ausführung erneut zur Bedingung zurückkehrt.

Zur Analyse einer terminierenden Schleife werden Startzustand, Fortsetzungsbedingung, Schleifenkörper und zustandsverändernder Schritt untersucht. Eine Schleife gehört zu einem Algorithmus, wenn Wiederholung und Abbruch eindeutig beschrieben sind; unbeabsichtigtes Nichtterminieren verletzt die erwartete Endlichkeit.

> BEGRIFF **Schleife als kontrollierte Wiederholung**
>
> Eine Schleife wiederholt nicht beliebig, sondern prüft wiederholt einen Zustand. Erst wenn die Schleifenbedingung nicht mehr erfüllt ist, läuft das Programm hinter der Schleife weiter.

While-Schleife

Die `while`-Schleife ist eine Bedingungsschleife. Sie arbeitet ähnlich wie eine bedingte Anweisung mit einer Bedingung, nutzt diese Bedingung aber anders: Bei `if` wird ein Block einmalig ausgeführt oder übersprungen. Bei `while` wird die Bedingung wiederholt geprüft.

Die Bedingung im Schleifenkopf ist deshalb eine **Fortsetzungsbedingung**. Solange diese Bedingung `true` ergibt, kann der Schleifenkörper ausgeführt werden. Nach jedem Durchlauf springt der Ablauf zurück zur Bedingung. Erst wenn die Bedingung `false` ergibt, wird die Schleife verlassen.

> BEGRIFF **Fortsetzungsbedingung und Abbruchfall**
>
> Die `while`-Bedingung beschreibt zunächst, wann die Schleife weiterlaufen darf. Der normale Abbruch entsteht dadurch, dass diese Bedingung irgendwann `false` wird. Man kann diesen Fall als impliziten Abbruchfall lesen: **Die Schleife endet, wenn die Fortsetzungsbedingung nicht mehr erfüllt ist.**
>
> Zusätzlich kann im Schleifenkörper eine explizite Abbruchbedingung geprüft werden. Wird sie erfüllt, kann die Schleife mit `break` sofort verlassen werden. Dadurch wird zwischen normaler Fortsetzung und gezieltem Sonderabbruch unterschieden.

Wichtig ist dabei die Erreichbarkeit des Abbruchs. Der Zustand, der in der Schleifenbedingung geprüft wird, muss sich im Ablauf verändern können. Wenn die Bedingung dauerhaft `true` bleibt und kein anderer Abbruch vorgesehen ist, entsteht eine Endlosschleife. Eine Endlosschleife kann bewusst gewünscht sein, etwa wenn ein späterer Abbruch über `break` geplant ist. Sie darf aber nicht unbemerkt entstehen.

Grundstruktur: Startzustand, Fortsetzungsbedingung, Körper und Schritt

Das erste Beispiel zeigt die elementare Struktur einer `while`-Schleife an einem einfachen Ladefortschritt. Der Kontext ist bewusst klein gehalten: Ein Startwert wird gesetzt, die Bedingung wird vor jedem Durchlauf geprüft, der Schleifenkörper gibt den aktuellen Zustand aus und verändert anschließend den geprüften Wert.

```
int ladeProzent = 0; // Startzustand: Die Variable steuert die Schleife.

// Schleifenkopf: while + Fortsetzungsbedingung
// Der Schleifenkörper wird ausgeführt, solange ladeProzent < 100 true ist.
while (ladeProzent < 100) {
    // Schleifenkörper:
    System.out.println("Ladestand: " + ladeProzent + "%");

    // Zustandsänderung:
    // Der geprüfte Wert verändert sich, damit die Schleife enden kann.
    ladeProzent = ladeProzent + 25;
}
```

```
// Anweisung nach der Schleife:
// Sie wird erst ausgeführt, wenn die Bedingung false ist.
System.out.println("Ladevorgang vollständig.");
```

Die Schleife läuft nicht, weil Java „weiß“, wie oft wiederholt werden soll. Sie läuft, weil die Bedingung vor jedem Durchlauf erneut ausgewertet wird. Erst die Zustandsänderung im Schleifenkörper sorgt dafür, dass die Bedingung später `false` werden kann.

Struktogramm: While-Grundstruktur

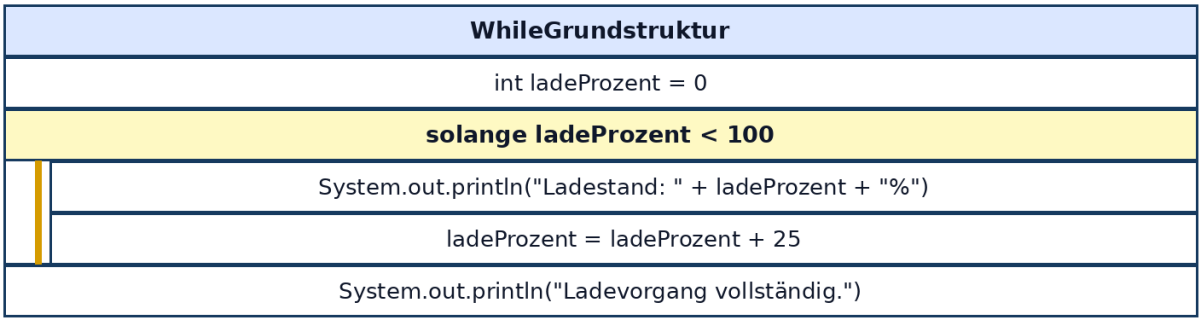

Abbildung 6: Struktogramm · while

Leselogik: Das Struktogramm zeigt zuerst den Startzustand `int ladeProzent = 0`. Danach wird die Bedingung `ladeProzent < 100` vor jedem möglichen Durchlauf geprüft. Ist sie wahr, werden die Ausgabe und die Zustandsänderung im Schleifenkörper ausgeführt. Danach führt der Ablauf zurück zur Bedingung. Ist sie falsch, wird die Schleife verlassen und die Anweisung nach der Schleife ausgeführt.

MATERIALISIERT Trace / Laufspur anzeigen

Durchlauf	ladeProzent vor Prüfung	Bedingung ladeProzent < 100	Ausgabe im Durchlauf	ladeProzent danach
1	0	true	Ladestand: 0%	25
2	25	true	Ladestand: 25%	50
3	50	true	Ladestand: 50%	75
4	75	true	Ladestand: 75%	100
5	100	false	keine Schleifenausgabe	100

Tabelle 17: Trace einer while-Schleife

MATERIALISIERT Konsolenausgabe anzeigen

```
Ladestand: 0%
Ladestand: 25%
Ladestand: 50%
Ladestand: 75%
Ladevorgang vollständig.
```

Deutung: Die Schleife wiederholt nicht blind. Vor jedem Durchlauf wird der aktuelle Zustand geprüft. Erst wenn die Bedingung `false` wird, endet der wiederholte Ablauf.

BEGRIFF EVA-Rückbezug: Eingaben können Schleifen steuern

Im EVA-Modell gehört die Schleife zur Verarbeitung: Ein Zustand wird geprüft, verarbeitet und verändert. Bei Konsolenprogrammen kann dieser Zustand auch durch Nutzereingaben entstehen. Dann lautet das typische Muster: **Eingabe lesen → Bedingung prüfen → Rückmeldung oder Verarbeitung ausführen → neue Eingabe lesen → erneut prüfen**.

Die Eingabe ist damit nicht nur ein Startwert. Sie kann im Schleifenkörper erneut gelesen werden und dadurch entscheiden, ob die Schleife weiterläuft oder endet. Gerade Eingabekontrollen und interaktive Programme beruhen auf dieser Verbindung von Eingabe, Verarbeitung und wiederholter Prüfung.

VERTIEFUNG **Vertiefung: while-Schleife mit bedingter Anweisung**

Eine `while`-Schleife steht selten allein. Sie kann mit Variablen, Zählern, Vergleichsausdrücken und bedingten Anweisungen kombiniert werden. Dabei bleibt die Rollenverteilung wichtig: Die `while`-Bedingung entscheidet, ob ein weiterer Durchlauf stattfindet. Eine bedingte Anweisung im Schleifenkörper entscheidet, was innerhalb eines bestimmten Durchlaufs geschieht.

```
int temperatur = 18; // Startwert
int anstieg = 4;     // Veränderung pro Durchlauf
int stunde = 1;      // Zähler für die Durchläufe

while (stunde <= 4) {
    System.out.println("Stunde: " + stunde);
    System.out.println("Temperatur: " + temperatur);

    // Bedingte Anweisung im Schleifenkörper:
    // Pro Durchlauf wird zusätzlich entschieden, welche Ausgabe passt.
    if (temperatur >= 26) {
        System.out.println("Kühlung prüfen.");
    } else {
        System.out.println("Temperatur im normalen Bereich.");
    }

    // Zustandsänderungen:
    temperatur = temperatur + anstieg;
    stunde = stunde + 1;
}

System.out.println("Messung beendet.");
```

Das Beispiel verbindet bekannte Elemente: Variablen speichern Zustände, Vergleichsausdrücke liefern Wahrheitswerte, `if-else` wählt innerhalb eines Durchlaufs eine Ausgabe aus, und die `while`-Schleife organisiert die Wiederholung. Der Zähler `stunde` begrenzt die Zahl der Durchläufe, während `temperatur` als veränderlicher Messwert im Schleifenkörper ausgewertet wird.

WERKZEUG **Werkzeugarbeit: Eingabeschleife und einfache Rateschleife**

Die ersten beiden Aufgaben übertragen die Grundstruktur der `while`-Schleife auf Eingaben: Eine Eingabe wird gelesen, geprüft und bei Bedarf erneut gelesen. B1 sichert eine einfache Eingabekontrolle. B2 überträgt die Fortsetzungsbedingung auf ein einfaches Ratespiel mit fester Geheimzahl. Zähler, Zufallszahlen und `break` bleiben in diesen beiden Aufgaben noch außen vor.

- Java-Werkzeug öffnen: gültige Zahl erzwingen
- Java-Werkzeug öffnen: einfache Rateschleife mit fester Geheimzahl

Die verlinkten Aufgaben stehen zusätzlich als Offline-Arbeitsaufträge in Anhang A.

VERTIEFUNG **Vertiefung: Explizite Abbruchbedingung und Schleifensteuerung mit break**

Normalerweise endet eine `while`-Schleife dadurch, dass ihre Fortsetzungsbedingung `false` wird. Manchmal gibt es innerhalb des Schleifenkörpers aber einen besonderen Zustand, bei dem die Schleife sofort verlassen werden soll. Dafür gibt es in Java die Anweisung `break`.

BEGRIFF **Explizite Abbruchbedingung im Schleifenkörper**

Die Fortsetzungsbedingung steht im Schleifenkopf. Sie beschreibt, wann die Schleife grundsätzlich weiterlaufen kann. Eine zusätzliche Abbruchbedingung kann im Schleifenkörper mit `if` geprüft werden. Wird dort `break` ausgeführt, wird die nächste umschließende Schleife sofort verlassen.

Dadurch kann auch eine Schleife kontrolliert enden, deren Fortsetzungsbedingung allein noch nicht `false` geworden ist. Das ist besonders wichtig, wenn eine Schleife für viele mögliche Eingaben offen bleibt und ein Sonderfall nicht zwangsläufig eintreten muss.

Das folgende Beispiel verarbeitet nacheinander Datenpakete. Die Fortsetzungsbedingung ist bewusst weit formuliert: Solange nicht `0` als normales Ende eingegeben wird, kann der Import weiterlaufen. Eine negative Paketgröße ist dagegen ein Sonderfall, weil eine solche Größe fachlich nicht sinnvoll ist. Dieser Fall muss in einer Eingabefolge nicht auftreten; falls er auftritt, wird die Schleife mit `break` sofort verlassen.

```
Scanner sc = new Scanner(System.in);
```

```
int paketGroesse = sc.nextInt();

while (paketGroesse != 0) {
    if (paketGroesse < 0) {
        System.out.println("Fehler: negative Paketgröße.");
        break;
    }

    System.out.println("Paket verarbeitet: " + paketGroesse);
    paketGroesse = sc.nextInt();
}

System.out.println("Import beendet.");
```

`break` ersetzt nicht die Schleifenbedingung. Es ergänzt sie um eine gezielte Steuerungsmöglichkeit im Schleifenkörper. So lässt sich fachlich sauber unterscheiden: Die Fortsetzungsbedingung `paketGroesse != 0` beschreibt die normale Wiederholung, die explizite Abbruchbedingung `paketGroesse < 0` beschreibt einen besonderen Zustand, bei dem der Ablauf sofort beendet wird.

Das Beispiel kann also auf zwei Wegen enden: normal durch die Eingabe `0` oder vorzeitig durch eine fehlerhafte negative Paketgröße. Wenn keiner dieser Fälle eintritt, wird ein weiteres Datenpaket eingelesen und erneut geprüft. Das ist die zentrale Idee von Schleifensteuerung: Fortsetzungsbedingung, Zustandsänderung und zusätzliche Abbruchfälle müssen zusammen gelesen werden.

VERTIEFUNG Zusatz: Zufallszahlen mit Random erzeugen

Nicht jeder Startwert muss aus einer festen Zuweisung oder aus einer Nutzereingabe stammen. Manche Programme erzeugen Werte während der Laufzeit selbst. In Java kann dafür `Random` aus dem Paket `java.util` genutzt werden.

```
import java.util.Random;

Random zufall = new Random();

int wuerfel = zufall.nextInt(6) + 1;     // 1 bis 6
int prozent = zufall.nextInt(101);       // 0 bis 100
int beliebigeZahl = zufall.nextInt();    // beliebiger int-Wert
double anteil = zufall.nextDouble();     // 0.0 bis kleiner als 1.0

System.out.println("Würfel: " + wuerfel);
System.out.println("Prozent: " + prozent);
System.out.println("Anteil: " + anteil);
```

- `import java.util.Random;` macht die Klasse `Random` verfügbar.
- `new Random()` erzeugt ein Zufallsobjekt.
- `nextInt(grenze)` liefert eine ganze Zahl von `0` bis `grenze - 1`.
- Mit `+ 1` kann der Wertebereich verschoben werden, zum Beispiel von `0..5` auf `1..6`.
- `nextInt()` liefert einen beliebigen `int`-Wert.
- `nextDouble()` liefert einen Dezimalwert von `0.0` bis kleiner als `1.0`.

Entscheidend ist, wo die Zufallszahl erzeugt wird: Vor einer Schleife entsteht ein Startwert, der während der Schleife gleich bleiben kann. Im Schleifenkörper entsteht dagegen pro Durchlauf ein neuer Wert.

```
import java.util.Random;

Random zufall = new Random();

int minute = 1;

while (minute <= 3) {
    int messwert = zufall.nextInt(5) + 20; // 20 bis 24
    System.out.println("Minute " + minute + ": " + messwert);

    minute = minute + 1;
}
```

Dieser Zusatzbereich bereitet Zufallswerte nur als allgemeines Sprachelement vor. Ob eine Zufallszahl ein Startwert, ein Messwert, ein Ereignis oder ein Sonderfall ist, entscheidet das jeweilige Programm durch seine Modellierung.

WERKZEUG **Werkzeugarbeit: Zufallswerte, Abbruchfälle und Struktogramm**

Die folgenden Aufgaben greifen die vorher erklärten Bausteine auf und führen sie in eine anspruchsvollere Gesamtstruktur. B3 ergänzt das Ratespiel um eine zufällig erzeugte Geheimzahl und einen Versuchszähler. B4 verbindet zufällige Werte mit mehreren Abbruchmöglichkeiten, darunter `break`. B5 überträgt das entstandene Gesamtprogramm in ein Struktogramm. Die Inhaltsseite liefert dafür die Strukturbegriffe; die konkrete Modellierung erfolgt in den Aufgaben.

- Java-Werkzeug öffnen: zufällige Geheimzahl und Versuche zählen
- Java-Werkzeug öffnen: Zufallswerte, Maximalversuche und Sonderabbruch mit break
- Java-Werkzeug öffnen: Gesamtprogramm als Struktogramm modellieren

Die verlinkten Aufgaben stehen zusätzlich als Offline-Arbeitsaufträge in Anhang A.

For-Schleife

Die `for`-Schleife ist die typische **Zählschleife**. Sie eignet sich besonders dann, wenn ein Programm einen Bereich systematisch durchlaufen soll: eine bekannte Anzahl von Etiketten, feste Kontrollzeitpunkte, Countdown-Werte, Teststufen oder Plätze in einem Saal. Im Unterschied zur `while`-Schleife werden Startwert, Fortsetzungsbedingung und Veränderung der Zählvariable direkt im Schleifenkopf gebündelt.

Der Schleifenkopf besitzt drei Teile. Die **Initialisierung** legt den Startzustand der Zählvariable fest. Die **Fortsetzungsbedingung** wird vor jedem möglichen Durchlauf geprüft. Das **Inkrement** oder **Dekrement** verändert die Zählvariable nach jedem Durchlauf. Der Schleifenkörper enthält die Anweisungen, die pro gezähltem Schritt ausgeführt werden.

BEGRIFF **Fachbegriffe: Zählvariable, Zählbereich, Inkrement und Dekrement**

- **Zählvariable:** Variable, die den aktuellen Schritt der Schleife beschreibt. In kurzen Zählschleifen ist `i` die übliche Standardvariable; sprechende Namen wie `platz` oder `reihe` sind sinnvoll, wenn sie die Modellierung klarer machen.
- **Zählbereich:** Wertebereich, den die Zählvariable systematisch durchläuft.
- **Inkrement:** Veränderung, bei der die Zählvariable größer wird. Für eine Erhöhung um `1` ist `i++` die Standardschreibweise.
- **Dekrement:** Veränderung, bei der die Zählvariable kleiner wird. Für eine Verringerung um `1` ist `i--` die Standardschreibweise.
- **Anpassbare Schrittweite:** Veränderung um einen anderen Wert, zum Beispiel `paketGroesse += schritt`.
- **Nichtlineare Veränderung:** Veränderung, bei der die Zählvariable nicht um einen festen Summanden verändert wird, zum Beispiel `datenrate *= 2`.

Teil im Schleifenkopf	Beispiel	Rolle im Ablauf
Initialisierung	int i = 1	wird genau einmal vor der ersten Prüfung ausgeführt
Fortsetzungsbedingung	i <= anzahlEtiketten	wird vor jedem Durchlauf geprüft
Inkrement	i++	wird nach jedem Schleifendurchlauf ausgeführt
Schleifenkörper	System.out.println(...)	wird nur ausgeführt, wenn die Fortsetzungsbedingung true ist

Tabelle 18: Syntax einer for-Schleife

Der Ablauf ist deshalb immer gleich: Initialisierung ausführen → Bedingung prüfen → Schleifenkörper ausführen → Zählvariable verändern → erneut prüfen. Wird die Bedingung `false`, läuft das Programm hinter der `for`-Schleife weiter.

Grundstruktur: Zählvariable und Inkrement

Das erste Beispiel modelliert einen kleinen Paketversand: Für eine bekannte Anzahl von Etiketten soll jeweils eine Druckausgabe erzeugt werden. Die Zählvariable `i` steht dabei für das gerade bearbeitete Etikett.

```
int anzahlEtiketten = 5;

for (int i = 1; i <= anzahlEtiketten; i++) {
    System.out.println("Etikett " + i + " drucken.");
}

System.out.println("Alle Etiketten wurden gedruckt.");
```

Die Schleife zählt von `1` bis `5`. In jedem Durchlauf wird die aktuelle Nummer ausgegeben. Nach dem Schleifenkörper erhöht `i++` die Zählvariable um `1`. Danach prüft Java wieder, ob `i <= anzahlEtiketten` noch gilt.

MATERIALISIERT Trace / Laufspur anzeigen

Durchlauf	i vor Prüfung	Bedingung i <= anzahlEtiketten	Ausgabe im Durchlauf	i danach
1	1	true	Etikett 1 drucken.	2
2	2	true	Etikett 2 drucken.	3
3	3	true	Etikett 3 drucken.	4
4	4	true	Etikett 4 drucken.	5
5	5	true	Etikett 5 drucken.	6
6	6	false	keine Schleifenausgabe	6

Tabelle 19: Trace einer einfachen for-Schleife

MATERIALISIERT Konsolenausgabe anzeigen

```
Etikett 1 drucken.
Etikett 2 drucken.
Etikett 3 drucken.
Etikett 4 drucken.
Etikett 5 drucken.
Alle Etiketten wurden gedruckt.
```

Deutung: Die Schleife endet nicht beim letzten ausgegebenen Wert, sondern erst bei der nächsten Prüfung. Nach `i++` wird `i` zu `6`. Erst dann ist die Bedingung `false`.

VERTIEFUNG Vertiefung: For-Varianten, Sonderfälle und verschachtelte Schleifen

Varianten: Inkrement, Dekrement und nichtlineare Veränderung

Eine `for`-Schleife ist nicht auf `i++` beschränkt. Die Veränderung der Zählvariable kann in festen Schritten wachsen, in festen Schritten kleiner werden oder nichtlinear verlaufen. Entscheidend ist, dass die Veränderung zur Fortsetzungsbedingung passt.

```
int startGroesse = 10;
int maxGroesse = 50;
int schritt = 10;

for (int paketGroesse = startGroesse; paketGroesse <= maxGroesse; paketGroesse += schritt) {
    System.out.println("Teste Paketgroesse: " + paketGroesse + " MB");
}

System.out.println("Lineare Paketgroessenpruefung beendet.");

int maxVersuche = 3;

for (int versuch = maxVersuche; versuch >= 1; versuch--) {
    System.out.println("Verbleibende Versuche: " + versuch);
}

System.out.println("Versuchszaehlung beendet.");
```

```
for (int datenrate = 1; datenrate <= 32; datenrate *= 2) {
    System.out.println("Teste Datenrate: " + datenrate + " Mbit/s");
}

System.out.println("Datenratentest beendet.");

for (int wartezeit = 32; wartezeit >= 1; wartezeit /= 2) {
    System.out.println("Wartezeit: " + wartezeit + " Sekunden");
}

System.out.println("Wartezeittest beendet.");
```

Im ersten Teil wächst `paketGroesse` linear um den Wert `schritt`. Im zweiten Teil wird `versuch` durch `versuch--` kleiner. Im dritten Teil verdoppelt sich `datenrate` pro Durchlauf; das ist ein nichtlineares Inkrement. Im vierten Teil wird `wartezeit` halbiert; das ist eine nichtlineare Verringerung.

Variante	Standardschreibweise	Ausgeschriebene Wirkung	Passende Fortsetzungsbedingung	Typischer Einsatz
Inkrement um 1	i++	i = i + 1	i <= grenze	klassische Zählschleife
lineares Inkrement mit Schrittweite	paketGroesse += schritt	paketGroesse = paketGroesse + schritt	paketGroesse <= maxGroesse	systematische Testwerte in festen Abständen
Dekrement um 1	versuch--	versuch = versuch - 1	versuch >= 1	Countdown oder verbleibende Versuche
nichtlineares Inkrement	datenrate *= 2	datenrate = datenrate * 2	datenrate <= 32	Verdopplung, Eskalationsstufen, Wachstumsmodelle
nichtlineare Verringerung	wartezeit /= 2	wartezeit = wartezeit / 2	wartezeit >= 1	Halbierung, Verfeinerung oder schrittweise Annäherung

Tabelle 20: Varianten der Veränderung einer Zählvariable

BEGRIFF Sonderfälle bei Zählbereichen

- Ist die Fortsetzungsbedingung schon vor dem ersten Durchlauf `false`, wird der Schleifenkörper gar nicht ausgeführt.
- Bei `<=` wird der Grenzwert eingeschlossen; bei `<` wird er ausgeschlossen.
- Inkrement und Fortsetzungsbedingung müssen zusammenpassen. Wer hochzählt, braucht normalerweise eine obere Grenze.
- Beim Rückwärtszählen muss die Bedingung zur fallenden Zählvariable passen, zum Beispiel `versuch >= 1`.
- Eine Schrittweite von `0` oder eine falsch gerichtete Veränderung kann eine Endlosschleife erzeugen.
- Nichtlineare Veränderungen können Werte überspringen. Dann ist nicht jeder Wert des Bereichs ein eigener Durchlauf.

Die ausgeschriebene Form bleibt als Deutungshilfe wichtig: `i++` bedeutet eine Erhöhung um `1`, `versuch--` eine Verringerung um `1`. Im Schleifenkopf wird jedoch die kompakte Schreibweise verwendet, weil sie in Java die übliche Standardschreibweise für einfache Zählvariablen ist.

Verschachtelte for-Schleifen

Eine `for`-Schleife kann im Schleifenkörper einer anderen `for`-Schleife stehen. Dann entsteht eine verschachtelte Wiederholung. Die äußere Schleife zählt einen übergeordneten Bereich; die innere Schleife wird für jeden äußeren Durchlauf neu gestartet.

```
int reihen = 3;
int plaetzeProReihe = 4;

for (int reihe = 1; reihe <= reihen; reihe++) {
    System.out.println("Reihe " + reihe + " pruefen.");

    for (int platz = 1; platz <= plaetzeProReihe; platz++) {
        System.out.println("  Platz " + platz + " pruefen.");
```

```
    }

    System.out.println("Reihe " + reihe + " abgeschlossen.");
}

System.out.println("Saalpruefung beendet.");
```

Im Beispiel zählt die äußere Schleife die Reihen. Für jede einzelne Reihe startet die innere Schleife wieder bei `platz = 1` und zählt alle Plätze dieser Reihe. Bei `3` Reihen und `4` Plätzen pro Reihe wird der innere Schleifenkörper insgesamt `12` Mal ausgeführt.

Die Einrückung ist hier fachlich wichtig: Sie zeigt, dass die Platzprüfung vollständig zum Schleifenkörper der Reihenschleife gehört. Erst wenn alle Plätze einer Reihe geprüft wurden, wird die Reihe abgeschlossen und die äußere Schleife geht zur nächsten Reihe weiter.

WERKZEUG **Werkzeugarbeit: For-Schleifen als Zählschleifen**

Die For-Aufgaben übertragen die Struktur der Zählschleife auf einen neuen Anwendungskontext: Eine Schulhof-Wetterstation plant Messpunkte, Kontrollzeitpunkte, Akkustände und adaptive Messpausen. Die Inhaltsseite erklärt dafür Syntax und Fachbegriffe; die Aufgaben verlangen den Transfer auf ein eigenständiges Modellierungsproblem.

- Java-Werkzeug öffnen: Messpunkte mit einer Zählschleife nummerieren
- Java-Werkzeug öffnen: Kontrollzeitpunkte mit festem Intervall planen
- Java-Werkzeug öffnen: Akkustand rückwärts zählen und bewerten
- Java-Werkzeug öffnen: adaptive Messpausen als Struktogramm modellieren

Die verlinkten Aufgaben stehen zusätzlich als Offline-Arbeitsaufträge in Anhang A.

VERTIEFUNG **Vertiefung: Do-while-Schleife**

Do-While-Schleife

Erklärung: Die `do-while`-Schleife ist fußgesteuert. Der Schleifenkörper wird mindestens einmal ausgeführt.

```
int versuche = 0;
do {
    versuche++;
} while (versuche < 1);
```

Struktogramm-Konvention: fußgesteuerte Schleife

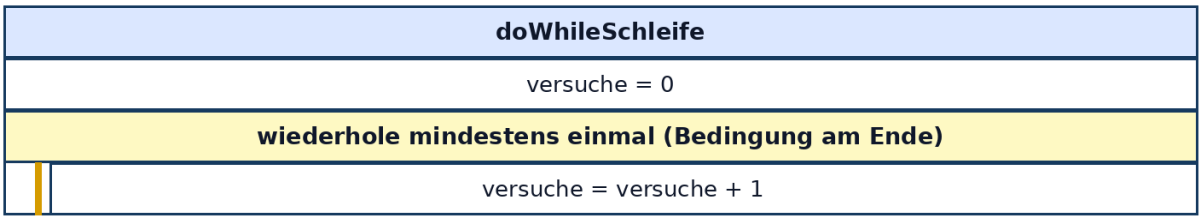

Abbildung 7: Struktogramm · do

Leselogik: Ein erster Durchlauf ist garantiert, erst danach wird geprüft. Im Struktogramm steht zuerst der Block, dann die Bedingung als Fortsetzungsentscheidung.

Schleifen auswählen

Ablaufproblem	Passende Struktur	Beispielkontext
Wiederholen, solange ein Zustand noch nicht passt	while	Zahlenratespiel: raten, bis der Tipp passt
Eingabe wiederholen, bis sie gültig ist	while oder do-while	gültige Zahl, gültiges Rechenzeichen
Feste Anzahl oder systematischer Durchlauf	for	Mathe-Challenge mit zehn Aufgaben oder Arraydurchlauf
Mindestens ein Durchlauf ist garantiert	do-while	Eingabe zuerst lesen, danach Fortsetzung prüfen

Tabelle 21: Schleifen nach Ablaufproblem auswählen

VERTIEFUNG **Vertiefung: Typische Fehlvorstellungen zu Schleifen**

- **Schleifen wiederholen nicht magisch:** Vor oder nach einem Durchlauf wird immer wieder eine Bedingung geprüft.
- **Der Zustand muss sich ändern können:** Wenn sich der geprüfte Zustand im Schleifenkörper nie ändert, droht eine Endlosschleife.
- `break` **beendet nicht nur den** `if`**-Block:** Wird `break` in einer bedingten Anweisung innerhalb einer Schleife ausgeführt, verlässt es die nächstliegende Schleife.
- **Zufallszahlen sind Zustände:** Wird eine Zufallszahl vor der Schleife erzeugt, bleibt sie gespeichert. Wird sie im Schleifenkörper erzeugt, entsteht pro Durchlauf ein neuer Wert.
- `while` **und** `do-while` **unterscheiden sich im Prüfzeitpunkt:** `while` prüft vor dem Durchlauf, `do-while` nach dem ersten Durchlauf.
- `for` **ist keine andere Magie:** Eine `for`-Schleife bündelt Startwert, Bedingung und Schrittweite an einer Stelle.
- **Arrays und Strings haben unterschiedliche Längen-Notation:** `array.length` ist ein Attribut, `string.length()` ist eine Methode.

BEGRIFF **Übergang: Schleifen als Bearbeitungswerkzeug**

Besonders wichtig werden For-Schleifen, wenn strukturierte Daten verarbeitet werden. Arrays und Strings besitzen geordnete Positionen; Schleifen können diese Positionen systematisch durchlaufen. Dadurch werden aus Kontrollstrukturen Bearbeitungswerkzeuge für Datenstrukturen: Werte ausgeben, prüfen, zählen, zusammenfassen und vergleichen.

ABSCHNITTSENDE · KAPITEL 5

Weiter: 6 Datenstrukturen: Arrays und Strings systematisch verarbeiten

LERNABSCHNITT 6 VON 9 · E3

6 Datenstrukturen: Arrays und Strings systematisch verarbeiten

Geordnete Daten mit Indizes, Schleifen, Bedingungen, Zählern und Akkumulatoren bearbeiten

0. Einstieg: Von Wiederholung zu strukturierter Datenarbeit

Bisher wurden einzelne Werte gespeichert, berechnet, verglichen und mit Kontrollstrukturen verarbeitet. Einzelne Variablen reichen aus, wenn ein Programm nur einen einzelnen Wert benötigt. Viele Aufgaben arbeiten aber mit mehreren zusammengehörigen Werten: Messwerte einer Wetterstation, Akkustände, Kontrollzeiten oder die Zeichen eines Textes. Mit Datenstrukturen werden aus Einzelwerten geordnete Sammlungen.

Der Index verbindet Datenstruktur und Schleife. Eine Schleife besucht systematisch alle gültigen Positionen. Bedingungen prüfen die besuchten Werte, Zähler erfassen passende Fälle und Akkumulatoren fassen Werte über mehrere Durchläufe zusammen. Dadurch werden Schleifen zum Bearbeitungswerkzeug für strukturierte Daten.

BEGRIFF **Merksatz: Datenstruktur und Schleife zusammendenken**

- **Datenstruktur:** geordnete Sammlung zusammengehöriger Werte.
- **Index:** Position innerhalb der Sammlung.
- **Schleife:** systematischer Durchlauf durch gültige Positionen.
- **Bedingung, Zähler, Akkumulator:** Auswertung der besuchten Werte.

A. Arrays

A1. Arrays: Syntax, Regeln und Fachbegriffe

Ein Array speichert mehrere Werte gleichen Typs unter einem Namen. Die Array-Variable verweist auf diese Sammlung. Ein Array-Element ist ein einzelner Wert an einer bestimmten Position. Die Länge eines Arrays wird bei der Erzeugung festgelegt und bleibt danach unverändert.

Indizes beginnen bei `0`. Der letzte gültige Index ist deshalb `array.length - 1`. `array.length` ist ein Attribut, keine Methode. Ein neues `int[]`, das ohne Startwerte erzeugt wird, enthält zunächst Standardwerte: bei `int` also `0`.

```
int[] festeMesswerte = {18, 21, 19, 24}; // Startwerte

int anzahlMessungen = 4;
int[] messwerte = new int[anzahlMessungen];  // Länge festlegen

int ersterMesswert = festeMesswerte[0];      // Element lesen
festeMesswerte[0] = 20;                      // Element verändern
```

VERTIEFUNG **Referenz: Array-Syntax lesen**

Ausdruck	Bedeutung
int[] messwerte	Array-Variable für eine Sammlung von int-Werten
{18, 21, 19}	Array-Erzeugung mit Startwerten
new int[anzahlMessungen]	Array mit zur Laufzeit festgelegter Länge
messwerte[i]	Element an Position i lesen

Ausdruck	Bedeutung
messwerte[i] = wert	Element an Position i überschreiben
messwerte.length	Anzahl der Elemente

Tabelle 22: Array-Syntax lesen

Index	0	1	2	3
Wert	18	21	19	24

Tabelle 23: Index und Wert in einer Messwertreihe

A2. Arrays mit Schleifen bearbeiten

Arrays machen die For-Schleife besonders nützlich: Die Schleife zählt gültige Indizes, und der Arrayzugriff liest oder verändert den Wert an der jeweiligen Position.

```
int[] messwerte = {18, 21, 19, 24};

for (int i = 0; i < messwerte.length; i++) {
    System.out.println("Messwert: " + messwerte[i]);
}
```

`i` ist der Index, nicht der Messwert. `messwerte[i]` liest den Wert an Position `i`. `i = 0` startet beim ersten gültigen Index, `i < messwerte.length` endet vor dem ersten ungültigen Index, und `i++` geht zur nächsten Position. So besucht die Schleife jeden gültigen Index genau einmal.

Durchlauf	Index i	Bedingung i < messwerte.length	Zugriff	Wert	Ausgabe
1	0	true	messwerte[0]	18	Messwert: 18
2	1	true	messwerte[1]	21	Messwert: 21
3	2	true	messwerte[2]	19	Messwert: 19
4	3	true	messwerte[3]	24	Messwert: 24
5	4	false	kein Zugriff	-	keine Ausgabe

Tabelle 24: Trace eines Array-Durchlaufs

VERTIEFUNG **Vertiefung: Bearbeitungsmuster, Sonderfälle und Fehlersuche bei Arrays**

Arrayprogramme bestehen selten nur aus einem Durchlauf. Typisch sind wiederkehrende Muster, bei denen Schleifen, Bedingungen, Zähler und Akkumulatoren zusammenspielen.

BEGRIFF **Typische Array-Bearbeitungsmuster**

Ausgeben: Jedes Element wird der Reihe nach sichtbar gemacht.

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

Einlesen: Jeder gültige Index bekommt genau einen Wert.

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

Aggregieren: Ein Akkumulator sammelt Messwerte über mehrere Durchläufe.

```
int summe = 0;
for (int i = 0; i < messwerte.length; i++) {
    summe += messwerte[i];
}
double durchschnitt = summe / (double) messwerte.length;
```

Extremwerte bestimmen: Der bisher niedrigste Wert wird laufend mit dem nächsten Element verglichen.
Vorbedingung: Das Array enthält mindestens ein Element.

```
int niedrigsterWert = messwerte[0];
for (int i = 1; i < messwerte.length; i++) {
    if (messwerte[i] < niedrigsterWert) {
        niedrigsterWert = messwerte[i];
    }
}
```

Klassifizieren und zählen: Eine Bedingung prüft jedes Element; ein Zähler erfasst Werte, die eine Bedingung erfüllen.

```
int kritischeWerte = 0;
for (int i = 0; i < messwerte.length; i++) {
    if (messwerte[i] >= 30) {
        kritischeWerte++;
    }
}
```

A4. Sonderfälle und typische Fehler bei Arrays

BEGRIFF Stolperstellen beim Arraydurchlauf

- Der erste Index ist `0`, nicht `1`.
- Die Länge ist nicht der letzte Index; der letzte gültige Index ist `messwerte.length - 1`.
- `i <= messwerte.length` führt beim letzten Durchlauf zu einem ungültigen Zugriff.
- `new int[anzahl]` enthält zunächst Standardwerte; bei `int` sind das `0`-Werte.
- Ein noch nicht vollständig gefülltes Array kann ungewollte `0`-Werte enthalten.
- Min/Max sollte nicht blind mit `0` starten; besser ist `messwerte[0]`, wenn mindestens ein Wert vorhanden ist.
- Bei einem Durchschnitt muss die Anzahl größer als `0` sein.
- Bei Eingabevalidierung darf der Index nur erhöht werden, wenn wirklich ein gültiger Wert gespeichert wurde.

```
int i = 0;
while (i < messwerte.length) {
    int messwert = sc.nextInt();

    if (messwert < -50 || messwert > 60) {
        System.out.println("Ungültiger Messwert");
    } else {
        messwerte[i] = messwert;
        i++;
    }
}
```

Der Index `i` wird nur erhöht, wenn ein gültiger Messwert gespeichert wurde. Ungültige Eingaben werden nicht ins Array übernommen.

WERKZEUG Werkzeugarbeit: Arrays

Die Array-Aufgaben übertragen die zuvor neutral gezeigten Durchlauf- und Auswertungsmuster auf den Notenkontext. Sie führen vom sicheren Grundmuster zum fachsprachlichen Verständnis und anschließend zur Problemaufgabe. Zuerst wird eine feste Notenskala mit einem Array durchlaufen. Danach wird begründet, warum der Indexdurchlauf korrekt funktioniert. Abschließend wird eine Klassenarbeit mit Array, Eingabekontrolle, Durchschnitt, Extremwerten und Zählern ausgewertet.

- Java-Werkzeug öffnen: Notenskala als Array ausgeben
- Java-Werkzeug öffnen: Arraydurchlauf mit Index begründen
- Java-Werkzeug öffnen: Klassenarbeit mit Array auswerten

Die verlinkten Aufgaben stehen zusätzlich als Offline-Arbeitsaufträge in Anhang A.

B. Strings

B1. Strings: Syntax, Regeln und Fachbegriffe

`String` ist als Textwert aus früheren Kapiteln bekannt. Neu ist die Strukturperspektive: Ein String ist eine geordnete Zeichenfolge. Jedes Zeichen hat eine Position, und auch hier beginnen die Indizes bei `0`. `text.length()` liefert die Anzahl der Zeichen; der letzte gültige Index ist `text.length() - 1`. Einzelne Zeichen werden mit `text.charAt(i)` gelesen.

Strings sind Objekte und Referenztypen. Außerdem sind sie unveränderlich: Methoden liefern neue Strings, sie verändern den ursprünglichen String nicht automatisch.

```
String kurs = "Informatik";

int laenge = kurs.length();
char erstesZeichen = kurs.charAt(0);
char fuenftesZeichen = kurs.charAt(4);
```

Index	0	1	2	3	4	5	6	7	8	9
Zeichen	I	n	f	o	r	m	a	t	i	k

Tabelle 25: Indexzugriff in einem String

VERTIEFUNG **Vertiefung: String-Methoden, Anwendungsfälle und Sonderfälle**

String-Methoden werden im E3-Pfad als Werkzeuge gelesen: Sie prüfen, finden, schneiden aus, lesen Zeichen oder formen Text um. Die folgenden Methoden bilden den Hauptpfad für die Werkzeugarbeit.

Funktion	Methode	Rückgabe	Beispiel	Deutung
Prüfen	length()	int	kursCode.length() >= 5	Länge als Bedingung nutzen
Prüfen	equals(...)	boolean	eingabe.equals("ende")	Textinhalte vergleichen
Prüfen	contains(...)	boolean	dateiname.contains(".")	Teiltext vorhanden?
Finden	indexOf(...)	int	int pos = dateiname.indexOf(".")	erste Fundposition bestimmen
Ausschneiden	substring(start, ende)	String	dateiname.substring(0, pos)	Teil vor einer Grenze bilden
Ausschneiden	substring(start)	String	dateiname.substring(pos + 1)	Teil ab einer Position bilden
Zeichen lesen	charAt(index)	char	kursCode.charAt(0)	ein einzelnes Zeichen lesen
Umformen	replace(alt, neu)	String	meldung.replace(" ", "-")	Textstellen ersetzen
Umformen	toLowerCase()	String	kommando.toLowerCase()	Schreibweise vereinheitlichen
Umformen	toUpperCase()	String	kennzeichen.toUpperCase()	Großschreibung erzeugen

Tabelle 26: String-Funktionen nach Anwendungsfällen

B3. Anwendungsfall: Dateiname zerlegen

Der Dateiname `messwerte.csv` zeigt die typische Arbeit mit Positionen: Zuerst wird die Trennposition gefunden, dann werden die Teile vor und hinter dem Punkt ausgeschnitten.

```
String dateiname = "messwerte.csv";

int punkt = dateiname.indexOf(".");
```

```
String name = dateiname.substring(0, punkt);
String endung = dateiname.substring(punkt + 1);

String endungGross = endung.toUpperCase();

System.out.println("Name: " + name);
System.out.println("Endung: " + endungGross);
```

`indexOf(".")` findet die Trennposition. `substring(0, punkt)` liefert den Teil vor dem Punkt; `substring(punkt + 1)` startet hinter dem Punkt. Der Punkt selbst wird nicht übernommen. Bei `substring(start, ende)` ist der Startindex enthalten, der Endindex aber ausgeschlossen. `toUpperCase()` liefert eine neue Schreibweise der Endung; das Ergebnis muss gespeichert oder direkt weiterverarbeitet werden.

B4. Anwendungsfall: Kommando normieren und prüfen

Bei einem Kommando greifen Umformung und Prüfung ineinander: Eine Eingabe wird zuerst in eine einheitliche Schreibweise überführt und danach mit Bedingungen geprüft.

```
String eingabe = " Start Messung ";
                                                                    String kommando =
eingabe.trim().toLowerCase().replace(" ", "_");

System.out.println("Kommando: " + kommando);

if (kommando.length() >= 5) {
    System.out.println("Länge ok");
} else {
    System.out.println("Länge zu kurz");
}

if (kommando.contains(".")) {
    System.out.println("Punkt gefunden");
} else {
    System.out.println("Kein Punkt enthalten");
}
```

`trim()` entfernt führende und nachfolgende Leerzeichen. Danach erzeugt die Umformung Kleinbuchstaben und ersetzt innere Leerzeichen durch Unterstriche. Die Prüfung kontrolliert Länge und ein bestimmtes Zeichen. Damit verbinden sich String-Methoden direkt mit `if`/`else`.

B5. Sonderfälle und typische Fehler bei Strings

BEGRIFF Stolperstellen bei String-Zugriffen

- `indexOf(...)` liefert `-1`, wenn der gesuchte Text nicht gefunden wird.
- Vor `substring(...)` muss geprüft werden, ob die Trennposition existiert.
- `substring(start, ende)` enthält `start`, aber nicht `ende`.
- `charAt(0)` ist bei einem leeren String ungültig.
- `substring(1)` ist bei sehr kurzen Strings ein Sonderfall.
- String-Vergleich erfolgt mit `equals(...)`, nicht mit `==`.
- String-Methoden verändern den ursprünglichen String nicht automatisch.
- Groß-/Kleinschreibung ist relevant.
- Ein Trennzeichen kann fehlen, einmal vorkommen oder mehrfach vorkommen; einfache E3-Beispiele erwarten zunächst genau eine Trennposition.

```
String kommando = "START";
kommando.toLowerCase();              // Ergebnis wird verworfen
kommando = kommando.toLowerCase();   // Ergebnis wird gespeichert

int punkt = dateiname.indexOf(".");

if (punkt == -1) {
    System.out.println("Punkt fehlt");
} else {
    System.out.println("Punkt gefunden");
}
```

WERKZEUG Werkzeugarbeit: Strings

Die String-Aufgaben übertragen die zuvor neutral gezeigten Prüf-, Such-, Ausschneide- und Umformungsmuster auf Benutzernamen und Profilkennungen. Zuerst werden Länge und Unterstrich

geprüft. Danach wird begründet, wie ein Benutzername mit `indexOf`, `substring` und `charAt` zerlegt und formatiert wird. Abschließend wird aus einem Anzeigenamen eine Profilkennung erzeugt und geprüft.

- Java-Werkzeug öffnen: Benutzername prüfen
- Java-Werkzeug öffnen: Benutzernamen mit Index und substring erklären
- Java-Werkzeug öffnen: Profilkennung erzeugen und prüfen

Die verlinkten Aufgaben stehen zusätzlich als Offline-Arbeitsaufträge in Anhang A.

C. Arrays und Strings vergleichen

Aspekt	Array	String
Grundidee	geordnete Elemente gleichen Typs	geordnete Zeichenfolge
Länge	array.length	string.length()
Zugriff	array[i]	string.charAt(i)
Startindex	0	0
letzter gültiger Index	array.length - 1	string.length() - 1
Veränderbarkeit	Elemente veränderbar	unveränderlich
typischer Durchlauf	for (int i = 0; i < array.length; i++)	for (int i = 0; i < string.length(); i++)
typische Verarbeitung	Messwerte summieren, Mittelwert bilden, Min/Max bestimmen, Schwellenwerte zählen	Dateiname prüfen, Trennposition finden, Teilstrings ausschneiden, Kommando normieren

Tabelle 27: Arrays und Strings vergleichen

Arrays und Strings sind keine identischen Strukturen, aber sie teilen die Idee geordneter Positionen. Der Indexzugriff ist jeweils die Brücke zur Verarbeitung mit Schleifen.

D. Übergang zu Methoden und Q1.2

Daten werden eingegeben oder bereitgestellt, in Arrays oder Strings strukturiert und anschließend mit Schleifen, Bedingungen, Zählern und Akkumulatoren verarbeitet. Das EVA-Modell ordnet diesen Ablauf als Eingabe, Verarbeitung und Ausgabe. Methoden kapseln wiederkehrende Verarbeitung, damit Programme lesbarer und gezielter prüfbar werden. Die Aufgaben übertragen diese Muster anschließend auf konkrete Anwendungskontexte.

Für den Blick nach vorne bleiben die Qualifikationsphasen ein Ausblick: Objektstrukturen in Q1.1, algorithmische Verfahren wie Suchen und Sortieren in Q1.2, vertiefte Datenstrukturen in Q1.4 und Datenmodelle in Q2.1. Der direkte E3-Anschluss liegt jetzt bei Methoden: Wiederkehrende Verarbeitung von Einzelwerten, Arrays und Strings wird benannt, gekapselt und wiederverwendbar gemacht.

ABSCHNITTSENDE · KAPITEL 6
Weiter: 7 Methoden, Strukturierung und erste Modellierung

LERNABSCHNITT 7 VON 9 · E3

7 Methoden, Strukturierung und erste Modellierung

Verarbeitung benennen, kapseln, aufrufen und wiederverwenden

Bisher wurden Werte gespeichert, Ausdrücke ausgewertet, Bedingungen geprüft, Schleifen formuliert und Arrays oder Strings durchlaufen. Dadurch entstehen Programme, die fachlich schon viel leisten, aber schnell unübersichtlich werden. Methoden lösen dieses Strukturproblem: Eine wiederkehrende oder fachlich abgeschlossene Verarbeitung erhält einen Namen, eine Schnittstelle und einen eigenen Rumpf.

> **BEGRIFF Leitidee**
>
> Methoden kapseln Verarbeitung: Sie bekommen Werte über Parameter, führen eine klar abgegrenzte Teilaufgabe aus und liefern entweder ein Ergebnis zurück oder führen eine benannte Aktion aus.

Warum jetzt Methoden?

Nach Datenstrukturen wird besonders sichtbar, dass dieselben Grundmuster immer wiederkehren: Werte werden berechnet, geprüft, gezählt, gesucht, umgeformt oder ausgegeben. Methoden stehen deshalb nicht neben den bisherigen Konzepten, sondern bündeln sie zu benannten Verarbeitungsroutinen. Die Übersicht wird im ausklappbaren Abschnitt zu Methodenarten an Beispielen konkretisiert.

Bisheriger Baustein	Neue Kapselung durch Methoden
Operatoren	Berechnungsmethode
Bedingungen	Prüfmethode
Schleifen	Auswertungsmethode
Arrays und Strings	Verarbeitungsmethode
GUI später	Event-Handler ruft Fachmethode auf

Tabelle 28: Bisherige Inhalte werden in Methoden gebündelt

1. Methode als benannte Teilaufgabe

Eine Methode ist eine benannte, abgegrenzte Verarbeitungsroutine innerhalb einer Klasse. Sie wird nicht automatisch ausgeführt, sondern erst durch einen Methodenaufruf gestartet. Der volle objektorientierte Methodenbegriff als Verhalten von Objekten wird in Q1.1 systematisch entwickelt.

Eine Methode ist nicht automatisch ein Algorithmus. Sie kann einen Algorithmus, einen Teilschritt, eine Prüfung, eine Transformation oder eine reine Aktion implementieren.

- Der Methodenkopf beschreibt die Schnittstelle nach außen.
- Der Methodenrumpf enthält die eigentliche Verarbeitung.
- `main` ist auch eine Methode, aber eine besondere Startmethode.
- Eigene Methoden stehen innerhalb der Klasse, aber außerhalb von `main`.

> **BEGRIFF Definition**
>
> Eine Methode ist eine benannte Teilaufgabe. Von außen sind Name, Parameter und Rückgabetyp sichtbar. Im Methodenrumpf steht die Verarbeitung. Ausgeführt wird die Methode erst durch einen Methodenaufruf.

2. Allgemeine Syntax: Methodenkopf, Parameterliste und Methodenrumpf

Eine Methodendeklaration besteht aus Methodenkopf, Parameterliste und Methodenrumpf. Das folgende Beispiel bleibt bewusst neutral:

```
public static int berechneDifferenz(int startwert, int endwert) {
    int differenz = endwert - startwert;
    return differenz;
}
```

Teil	Beispiel	Bedeutung
Sichtbarkeit	public	im E3-Kontext Arbeitsstandard; später genauer in Q1.1
static	static	Aufruf aus main ohne Objekt
Rückgabetyp	int	Typ des Ergebnisses
Methodenname	berechneDifferenz	fachlicher Name der Teilaufgabe
Parameterliste	int startwert, int endwert	Eingabewerte der Methode
Methodenrumpf	{ ... }	Verarbeitung
lokale Variable	int differenz	Hilfswert nur innerhalb der Methode
Rückgabe	return differenz;	Ergebnis zurück an die Aufrufstelle

Tabelle 29: Bestandteile einer Methodendeklaration

`public static` ist hier eine E3-Arbeitskonvention: Die Methode gehört zur Klasse und kann aus `main` direkt aufgerufen werden. Eine ausführliche Objektorientierungsdeutung folgt später in Q1.1.

3. Methodenaufruf, Argumente und Parameterbindung

Die Methodendeklaration legt fest, welche Eingabewerte erwartet werden. Der Methodenaufruf liefert konkrete Argumentwerte; sie werden beim Start der Methode den Parametern zugeordnet. Parameter sind lokale Variablen der Methode.

```
int wert = berechneDifferenz(12, 20);
```

Schritt	Bedeutung
Aufruf	berechneDifferenz(12, 20) startet die Methode.
Argumente	12 und 20 stehen an der Aufrufstelle.
Parameterbindung	startwert = 12, endwert = 20
Verarbeitung	differenz = endwert - startwert
Rückgabe	return differenz liefert 8 zurück.
Nutzung	wert speichert den Rückgabewert.

Tabelle 30: Laufspur eines Methodenaufrufs

Deutung: Argumente stehen beim Aufruf, Parameter stehen in der Methodendeklaration. Parameter sind lokale Variablen der Methode. Ein Rückgabewert muss an der Aufrufstelle genutzt oder gespeichert werden, wenn man ihn weiterverwenden möchte.

4. Rückgabetyp, Rückgabe und Rückgabewert unterscheiden

Der Rückgabetyp steht im Methodenkopf und legt fest, welche Art von Ergebnis die Methode liefern muss. Die Rückgabeanweisung `return ...;` beendet die Methode und sendet einen Wert an die Aufrufstelle zurück. Der Rückgabewert ist der konkrete Wert, der bei einem bestimmten Methodenaufruf entsteht.

```
public static int berechneAenderung(int startwert, int endwert) {
    return endwert - startwert;
}
```

Begriff	Wo sichtbar?	Bedeutung
Rückgabetyp	public static int ...	legt den Ergebnistyp fest
Rückgabeanweisung	return endwert - startwert;	beendet die Methode und liefert einen Wert zurück
Rückgabewert	z. B. 6	konkretes Ergebnis eines Methodenaufrufs
Aufrufstelle	int aenderung = berechneAenderung(18, 24);	nutzt oder speichert den Rückgabewert

Tabelle 31: Rückgabetyp Rückgabe und Rückgabewert unterscheiden

Eine Methode mit Rückgabetyp muss auf jedem relevanten Ausführungspfad einen passenden Wert zurückgeben. Eine `void`-Methode hat keinen Rückgabewert.

VERTIEFUNG 5. Methodenarten und bisherige Inhalte bündeln

Methoden lassen sich danach ordnen, welche fachliche Rolle sie übernehmen. Zugleich zeigen sie, wie die bisherigen Java-Bausteine aus E3 gebündelt werden: Operatoren werden zu Berechnungsmethoden, Bedingungen zu Prüfmethoden, Schleifen und Arrays zu Auswertungsmethoden, String-Operationen zu Transformationsmethoden und Ausgaben zu Aktionsmethoden.

Bisheriger Baustein	Methodenart	Typischer Rückgabetyp	Beispielrolle
Ausgabe / Hinweis	Aktion	void	Hinweis ausgeben
Operatoren	Berechnung	int / double	Wert berechnen
Bedingungen	Prüfung	boolean	Zustand prüfen
Strings	Transformation	String	Text umformen
Schleifen / Arrays	Auswertung	int / double / boolean	Werte zählen, Extremwert bestimmen, zusammenfassen

Tabelle 32: Bisherige Bausteine und Methodenarten

Aktion / Ausgabe

```
public static void zeigeStartHinweis() {
    System.out.println("Messsystem wird gestartet.");
}
```

- `void`-Methode
- führt eine Aktion aus
- liefert keinen Rückgabewert

Berechnung / Operatoren

```
public static int berechneRestAkku(int startAkku, int verbrauch) {
    return startAkku - verbrauch;
}
```

- Operatoren werden in einer benannten Berechnung gekapselt.
- Die Methode liefert einen `int`-Rückgabewert.

Prüfung / Bedingungen

```
public static boolean istAkkuKritisch(int akkuProzent) {
    return akkuProzent <= 20;
}
```

- Eine Bedingung wird zu einer benannten Prüfmethode.
- Der Rückgabewert kann später in `if` oder `while` genutzt werden.

Transformation / Strings

```
public static String normalisiereSensorcode(String code) {
    return code.toUpperCase().replace(" ", "-");
}
```

- String-Verarbeitung wird in eine benannte Umformung ausgelagert.
- Die Methode liefert einen neuen String zurück.
- Der ursprüngliche String wird nicht automatisch verändert.

Auswertung / Schleifen und Arrays

```
public static int bestimmeHoechstenWert(int[] werte) {
    int maximum = werte[0];

    for (int i = 1; i < werte.length; i++) {
        if (werte[i] > maximum) {
            maximum = werte[i];
        }
    }

    return maximum;
}
```

- Schleife und Bedingung werden zu einer wiederverwendbaren Auswertungsroutine.
- Das Array ist Eingabe der Methode.
- Die Methode liefert ein Ergebnis zurück.

5. `main` als Steuerung, Methoden als Fachlogik

BEGRIFF Strukturidee

Im bevorzugten E3-Strukturmodell koordiniert `main` den Ablauf: Eingaben bereitstellen, Methoden aufrufen, Ergebnisse ausgeben. Die eigentliche Fachlogik liegt möglichst in Methoden; dies ist eine hilfreiche Strukturentscheidung, keine universelle Sprachregel.

```
public class AkkuAuswertung {
    public static void main(String[] args) {
        int[] akkustaende = {80, 64, 25, 18};

        int niedrigsterAkku = bestimmeNiedrigstenAkku(akkustaende);
        int warnungen = zaehleAkkuWarnungen(akkustaende);

        System.out.println("Niedrigster Akku: " + niedrigsterAkku);
        System.out.println("Akkuwarnungen: " + warnungen);
    }

    public static int bestimmeNiedrigstenAkku(int[] werte) {
        int minimum = werte[0];

        for (int i = 1; i < werte.length; i++) {
            if (werte[i] < minimum) {
                minimum = werte[i];
            }
        }

        return minimum;
    }

    public static int zaehleAkkuWarnungen(int[] werte) {
        int anzahl = 0;

        for (int i = 0; i < werte.length; i++) {
            if (werte[i] <= 20) {
```

```
                anzahl++;
            }
        }

        return anzahl;
    }
}
```

- `main` zeigt den groben Ablauf.
- Methoden enthalten die Detailverarbeitung.
- Die Fachlogik wird dadurch benennbar und gezielt prüfbar.
- Das bereitet GUI vor: Event-Handler sollen später möglichst Methoden aufrufen statt Fachlogik unübersichtlich direkt im Handler zu enthalten.

VERTIEFUNG **Vertiefung: Gültigkeitsbereich, Referenzwirkungen, Überladung und typische Fehler**

Gültigkeitsbereich, lokale Variablen und Referenzen

Methoden strukturieren Programme auch dadurch, dass Namen nur in einem begrenzten Bereich gelten. Werte müssen über Parameter in eine Methode hinein und über `return` wieder heraus.

- Parameter gelten nur innerhalb der Methode.
- Lokale Variablen gelten nur innerhalb des Methodenrumpfs.
- Mehrere Methoden können gleich benannte lokale Variablen besitzen.
- `return` beendet die Methode sofort.
- Bei Arrays erhält der Parameter eine Kopie des Referenzwertes; beide Referenzwerte bezeichnen anschließend dasselbe Arrayobjekt. Änderungen an dessen Elementen können deshalb außerhalb sichtbar werden.
- Strings sind unveränderlich; String-Methoden liefern neue Strings.

```
public static int verdopple(int wert) {
    int ergebnis = wert * 2;
    return ergebnis;
}
```

`wert` und `ergebnis` sind außerhalb der Methode nicht sichtbar.

```
public static void setzeErstenWert(int[] daten, int wert) {
    daten[0] = wert;
}
```

Die Methode verändert ein Element des übergebenen Arrays. Das ist ein anderer Fall als eine Methode, die nur einen berechneten Wert zurückgibt.

```
public static String klein(String text) {
    return text.toLowerCase();
}
```

Der ursprüngliche String wird nicht verändert; die Methode liefert einen neuen String zurück.

Sonderfälle und typische Fehler

Bei Methoden entstehen viele Fehler nicht durch neue Syntax allein, sondern durch eine falsche Vorstellung von Aufruf, Schnittstelle, Rückgabe und Gültigkeitsbereich.

Fehlerbild	Klärung
Methode wird definiert, aber nie aufgerufen.	Eine Deklaration startet keine Ausführung.
Methode steht versehentlich innerhalb von main.	Eigene Methoden stehen in der Klasse, aber außerhalb von main.
Rückgabetyp und return passen nicht zusammen.	Der zurückgegebene Wert muss zum Rückgabetyp passen.
return fehlt bei einer Methode mit Rückgabetyp.	Jeder mögliche Ablauf muss ein Ergebnis liefern.
Rückgabewert wird berechnet, aber nicht gespeichert.	Wer das Ergebnis später braucht, muss es an der Aufrufstelle nutzen.

Fehlerbild	Klärung
void-Methode wird wie ein Wert verwendet.	void bedeutet: Aktion ohne Rückgabewert.
Parameter und Argumente werden verwechselt.	Parameter stehen in der Deklaration, Argumente am Aufruf.
Anzahl oder Typen der Argumente passen nicht.	Der Aufruf muss zur Parameterliste passen.
Lokale Variable wird außerhalb der Methode verwendet.	Lokale Namen sind außerhalb ihres Blocks nicht sichtbar.
return steht zu früh.	return beendet die Methode; Code danach wird nicht mehr erreicht.
Scanner- und Ausgabe-Logik wird in Fachmethoden vermischt.	Fachmethoden sollten möglichst Werte verarbeiten und Ergebnisse liefern.
Array-Parameter erzeugen Seiteneffekte.	Änderungen an Array-Elementen können außerhalb der Methode sichtbar bleiben.
String-Methoden werden ohne Speicherung aufgerufen.	String-Methoden liefern neue Strings und verändern den ursprünglichen String nicht automatisch.

Tabelle 33: Typische Methodenfehler

6. Werkzeugarbeit: Methoden

WERKZEUG **Werkzeugarbeit: Methoden in der Messkampagne**

Die Methodenaufgaben übertragen die Strukturidee auf eine Schulhof-Wetterstation. Zuerst wird eine Messwertänderung als einfache Methode implementiert. Danach wird eine Bereichsprüfung als boolean-Methode angewendet. Anschließend wird fachsprachlich begründet, wie Argumente, Parameterbindung und Rückgabewerte zusammenhängen. In der Problemaufgabe wird eine Messkampagne mit Strings, Arrays, Schleifen und Methoden modularisiert. Abschließend wird die modulare Lösung als Struktogramm dargestellt.

- Java-Werkzeug öffnen: Messwertänderung als Methode berechnen
- Java-Werkzeug öffnen: Messwertbereich mit Methode prüfen
- Java-Werkzeug öffnen: Parameterbindung und Rückgabe erklären
- Java-Werkzeug öffnen: Messkampagne modularisieren
- Java-Werkzeug öffnen: Modulare Messkampagne als Struktogramm darstellen

Die verlinkten Aufgaben stehen zusätzlich als Offline-Arbeitsaufträge in Anhang A.

7. Übergang zur GUI

In der GUI löst ein Button-Klick einen Event-Handler aus. Dieser liest Eingaben, ruft die Fachlogik als Methode auf und schreibt das Ergebnis zurück in eine Komponente.

ABSCHNITTSENDE · KAPITEL 7

Weiter: 8 Grafische Benutzeroberflächen und ereignisgesteuerte Programmierung

LERNABSCHNITT 8 VON 9 · E3

8 Grafische Benutzeroberflächen und ereignisgesteuerte Programmierung

Interaktion über Fenster, Eingaben und Aktionen

Eine grafische Benutzeroberfläche verbindet sichtbare Eingabe, Aktion und Ausgabe. Im Anhalteweg-Beispiel bilden `JFrame`, `JLabel`, `JTextField` und Button den sichtbaren Arbeitsweg.

Die GUI ist Präsentations- und Interaktionsschicht: Komponenten zeigen Zustände und Ergebnisse. Der Event-Handler liest Eingaben und koordiniert die Reaktion; die Fachmethode berechnet oder prüft.

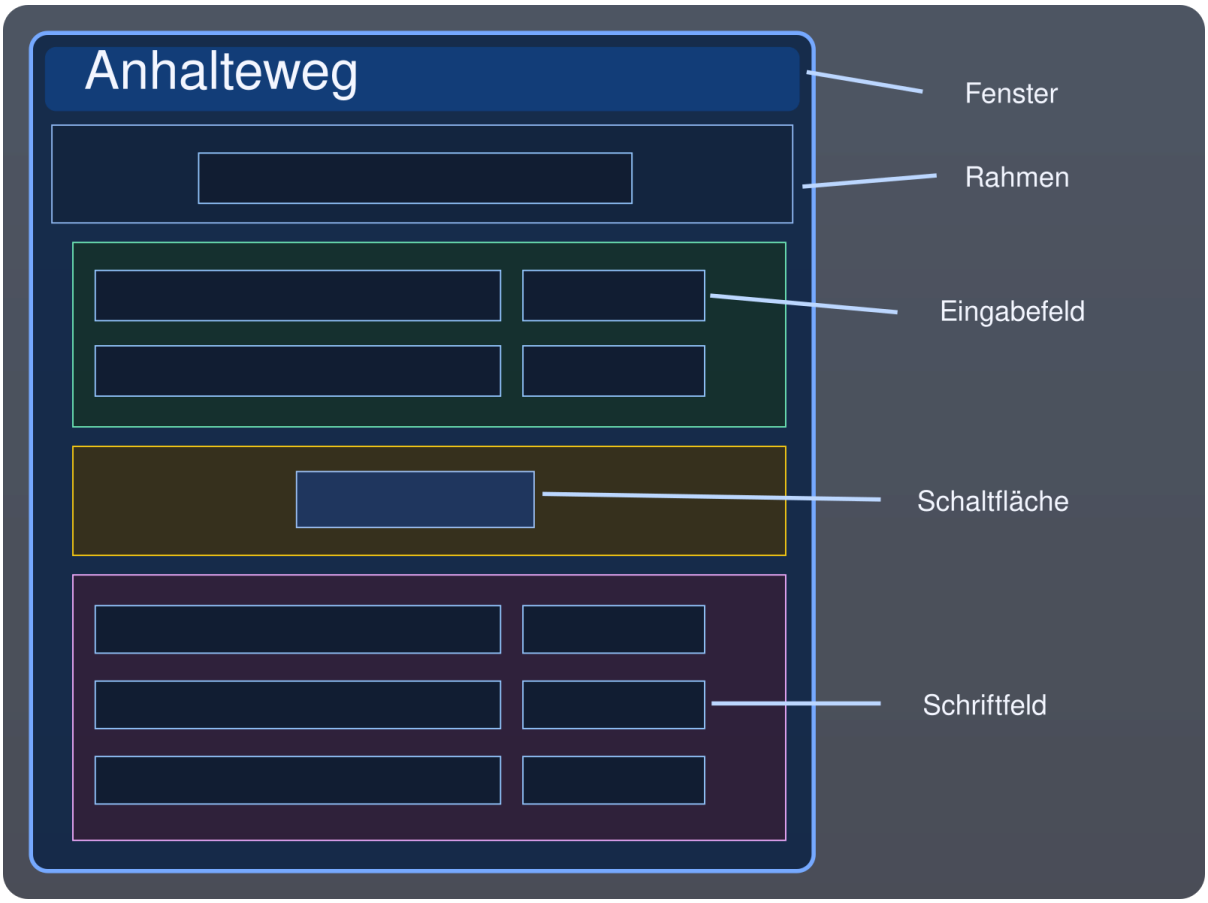

Abbildung 8: Bausteine der GUI „Anhalteweg“: Sichtbare Komponenten sind in einem Fenster und in Bereichen gruppiert.

RELATION **D-Book-Wiki-Vertiefung**

Historische Linie: Von Rechenmaschinen zu Computern

E3 untersucht die GUI als sichtbare Interaktionsschicht und als ereignisgesteuertes EVA-System. Die History-Ebene zeigt am Xerox Alto, wie grafische Oberfläche, Maus, Vernetzung und persönliche Arbeitsstation zu einer neuen Form der Computernutzung verbunden wurden.

Xerox Alto und grafische Interaktion

Von Rechenmaschinen zu Computern

Die Inhaltsseite erklärt die Funktionslogik einer GUI-Anwendung; das Wiki ordnet den Wandel von der Rechenmaschine zur interaktiven Arbeitsumgebung historisch ein.

1. Oberfläche, Ereignissteuerung und Fachlogik

Eine einfache GUI-Anwendung lässt sich in drei Ebenen denken: Die **Oberfläche** nimmt Eingaben entgegen und zeigt Ergebnisse, die **Ereignissteuerung** reagiert auf Benutzeraktionen, und die **Fachlogik** führt Berechnungen und Prüfungen aus.

BEGRIFF Drei Ebenen einer einfachen GUI-Anwendung

Ebene	Aufgabe
Oberfläche	Anzeigen, Eingaben aufnehmen und Zustände sichtbar machen
Ereignissteuerung	Reaktion registrieren und beim Ereignis ausführen
Fachlogik	Berechnungen, Prüfungen und Regeln unabhängig von der Oberfläche formulieren

Tabelle 34: Drei Ebenen einer einfachen GUI-Anwendung

GUI-Komponenten sind nicht die Fachlogik. Sie liefern Zustände oder zeigen Ergebnisse; die Verarbeitung bleibt als Fachlogik getrennt.

Der Event-Handler bildet die Vermittlungsschicht zwischen Oberfläche und Fachlogik: Er liest Zustände aus Komponenten, ruft eine passende Verarbeitungsmethode auf und schreibt das Ergebnis in die Oberfläche zurück.

Abbildung 9: Diagramm

GUI, Event-Handler und Fachlogik haben unterschiedliche Aufgaben und greifen im Ereignisablauf gezielt ineinander.

2. Linearer Ablauf versus ereignisgesteuerter Ablauf

Das klassische EVA-Modell bleibt erhalten, aber der Programmablauf wird im GUI-Kontext durch Ereignisse gesteuert.

Konsolenprogramm (linear)

1. Start
2. Eingabe
3. Verarbeitung

4. Ausgabe
5. Ende

Der Ablauf folgt einer festen Reihenfolge von oben nach unten.

GUI-Programm (ereignisgesteuert)

6. Start
7. Oberfläche bleibt aktiv

wartet auf Ereignisse

reagiert wiederholt im passenden Event-Handler

Lineare Konsolenprogramme folgen einer festen Kette. GUI-Programme bleiben aktiv und reagieren immer wieder auf Ereignisse.

BEGRIFF EVA im GUI-Kontext

EVA-Schritt	GUI-Umsetzung
Eingabe	Zustände aus Komponenten lesen, z. B. getText() oder isSelected()
Verarbeitung	Fachlogik ausführen, Bedingungen prüfen, Methoden aufrufen
Ausgabe	Komponenten aktualisieren, z. B. setText(...)

Tabelle 35: EVA im GUI-Kontext

Wichtig: Ein Button ist keine direkte Dateneingabe. Er ist ein **Auslöser**, der ein Ereignis erzeugt und damit die Verarbeitung startet.

VERTIEFUNG Vertiefung: EVA-Darstellung im GUI-Kontext

Abbildung 10: EVA im GUI-Kontext: Nutzendeingaben und Klickereignisse werden verarbeitet und in einer Anzeige ausgegeben.

3. Sichtbare Basisbausteine

BEGRIFF KCGO-Kern einer einfachen GUI

`JFrame` bildet das Fenster, `JLabel` zeigt Text, `JTextField` nimmt eine Eingabe auf und `JButton` ist die Ereignisquelle. Der Handler liest die Eingabe, ruft eine Fachmethode auf und aktualisiert die Ausgabe.

VERTIEFUNG Aufbauschicht: Hauptfenster, Container und Komponenten

Ein `JFrame` ist das Hauptfenster einer Swing-Anwendung. Komponenten werden darin nicht frei schwebend platziert, sondern in eine Container-Struktur eingefügt. Ein Container kann weitere Komponenten enthalten und strukturieren. Das Fenster ist der oberste Container; ein `JPanel` kann als innerer Container Komponenten gruppieren. Sichtbare Komponenten sind z. B. `JLabel`, `JTextField`, `JButton` oder `JCheckBox`. Typisch ist die Reihenfolge: Objekt erzeugen → konfigurieren → anordnen/positionieren → hinzufügen → später auswerten oder verändern.

Informatikfachlich bedeutet das: Fenster, Panels, Buttons und Textfelder sind Objekte mit Eigenschaften, Methoden und internem Zustand. Container strukturieren diese Objekte und erzeugen gemeinsam eine hierarchische Objektstruktur der Oberfläche.

```
import javax.swing.*;

public class ContainerStrukturDemo {
    public static void main(String[] args) {
        JFrame fenster = new JFrame("Container-Struktur");
        fenster.setSize(360, 220);
        fenster.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fenster.setLayout(null);

        JPanel panel = new JPanel();
        panel.setLayout(null);
        panel.setBounds(20, 20, 300, 120);

        JLabel beschriftung = new JLabel("Panel als innerer Container");
        beschriftung.setBounds(20, 20, 220, 25);
        panel.add(beschriftung);

        JButton button = new JButton("Aktion");
        button.setBounds(20, 60, 100, 25);
        panel.add(button);

        fenster.add(panel);
        fenster.setVisible(true);
    }
}
```

- `JFrame` ist das Hauptfenster.
- `JPanel` ist ein innerer Container.
- `JLabel` und `JButton` werden dem Panel hinzugefügt.
- Das Panel wird anschließend dem Fenster hinzugefügt.
- Dadurch entsteht eine hierarchische Struktur aus Containern und Komponenten.

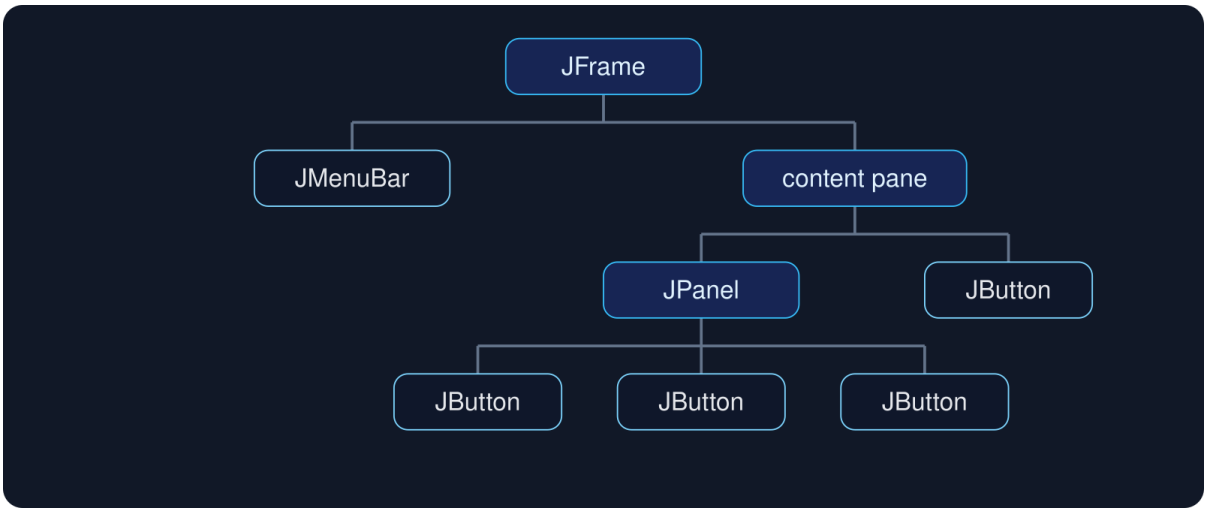

Abbildung 11: Eine Swing-Oberfläche ist hierarchisch aufgebaut: Ein JFrame enthält eine Menüleiste und eine content pane. In der content pane liegen Komponenten oder weitere Container wie JPanel.

Bausteine einer einfachen Swing-GUI

Swing-Komponenten folgen einem gemeinsamen Muster: Der Konstruktor erzeugt ein Objekt, `setBounds(...)` beziehungsweise ein Layout-Manager bestimmt die Darstellung, und mit `add(...)` wird die Komponente in einen Container eingefügt.

VERTIEFUNG Benötigte Importpakete für die Swing-Beispiele

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

JFrame – Fenster erzeugen

```
JFrame fenster = new JFrame("Login");
```

JLabel – Text ausgeben

```
JLabel lblInfo = new JLabel("Bitte anmelden");
```

JTextField – Eingabe aufnehmen

```
JTextField txtUser = new JTextField();
```

JButton – Ereignis auslösen

```
JButton btnLogin = new JButton("Anmelden");
```

setLayout(null) – absolute Positionierung aktivieren

```
fenster.setLayout(null);
```

setBounds(...) – Position und Größe festlegen

```
btnLogin.setBounds(80, 90, 120, 25);
```

add(...) – Komponente einfügen

```
fenster.add(btnLogin);
```

setVisible(true) – Fenster anzeigen

```
fenster.setVisible(true);
```

Die Codeboxen sind bewusst isoliert dargestellt. In einem vollständigen Programm müssen diese Bausteine in sinnvoller Reihenfolge zusammengesetzt werden. Für die Einführungsphase wird zunächst mit `setLayout(null)` und `setBounds(...)` gearbeitet, weil dadurch der Zusammenhang zwischen Objekt, Position und Anzeige besonders sichtbar bleibt.

VERTIEFUNG **Warum Komponenten oft Attribute sind**

Eine lokale Variable ist außerhalb ihres Blocks beziehungsweise ihrer Methode nicht zugreifbar. Wichtige GUI-Komponenten werden deshalb oft als Attribute, also als Felder der Klasse, gespeichert: Der Konstruktor erzeugt und ordnet sie, der Listener liest oder verändert später dieselben Komponenten.

BEGRIFF **Benötigte Importpakete**

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

8. 1. Attribute deklarieren private JTextField txtName; private JLabel lblAusgabe;
9. 2. Im Konstruktor erzeugen public MeineGUI() { txtName = new JTextField(20); lblAusgabe = new JLabel("Noch keine Eingabe."); }
10. 3. Im Listener verwenden public void actionPerformed(ActionEvent e) { String name = txtName.getText(); lblAusgabe.setText("Hallo, " + name + "!"); }

Wenn Komponenten dauerhaft verfügbar sind, kann der Event-Handler mit ihnen arbeiten: Er liest Zustände aus, verarbeitet Werte und schreibt Ergebnisse zurück.

4. Zustände von GUI-Komponenten lesen und verändern

- GUI-Komponenten sind Objekte mit internem Zustand.
- `get`- und `is`-Methoden lesen Zustände.
- `set`-Methoden verändern Zustände.
- Die Komponente bleibt bestehen; nur ihr Zustand ändert sich.
- `setText("")` löscht nicht das Objekt, sondern setzt den Textzustand auf die leere Zeichenkette.

VERTIEFUNG **Beispiele: Zustände lesen und verändern**

Zustände lesen

Komponente	Code
JTextField	String text = feld.getText();

Komponente	Code
JLabel	String info = label.getText();
JCheckBox	boolean aktiv = check.isSelected();
JRadioButton	boolean gewaehlt = radio.isSelected();
JComboBoxAusblick	int index = combo.getSelectedIndex();
JSpinnerAusblick	int wert = (int) spinner.getValue();

Tabelle 36: Beispiele zum Lesen von Zuständen aus GUI-Komponenten

Zustände verändern

Komponente	Code
JLabel	label.setText("Fertig");
JTextField	feld.setText("");
JCheckBox	check.setSelected(true);
JRadioButton	radio.setSelected(true);
JComboBoxAusblick	combo.setSelectedIndex(2);
JSpinnerAusblick	spinner.setValue(5);

Tabelle 37: Beispiele zum Verändern von Zuständen in GUI-Komponenten

VERTIEFUNG **Vertiefung: Spezielle Layout-Formen**

Bisher wurde in E3 bewusst mit `setLayout(null)` und `setBounds(...)` gearbeitet, damit Positionen und Größen explizit sichtbar werden. In größeren Swing-Programmen werden stattdessen Layout-Manager eingesetzt. Sie ordnen Komponenten automatisch an und reagieren auf Fenstergrößen sowie die verfügbare Containerfläche.

In dieser Vertiefung werden drei zentrale Layouts aus der Swing-Grundausbildung betrachtet: `FlowLayout`, `BorderLayout` und `GridLayout`. Als kurzer Ausblick gelten außerdem `BoxLayout`, `CardLayout` und `GridBagLayout`, die hier nicht weiter vertieft werden.

VERTIEFUNG **FlowLayout – zeilenweise Anordnung**

Komponenten werden zeilenweise von links nach rechts angeordnet. Wenn kein Platz mehr vorhanden ist, beginnt automatisch eine neue Zeile. Die sichtbare Anordnung hängt daher direkt von der Fensterbreite ab.

Hinweis: Das Beispiel nutzt hier `extends JFrame`, weil viele Swing-Quellen so kompakt erklären. Die bisherigen E3-Beispiele arbeiten weiterhin ohne Vererbung über ein separates `JFrame`-Objekt.

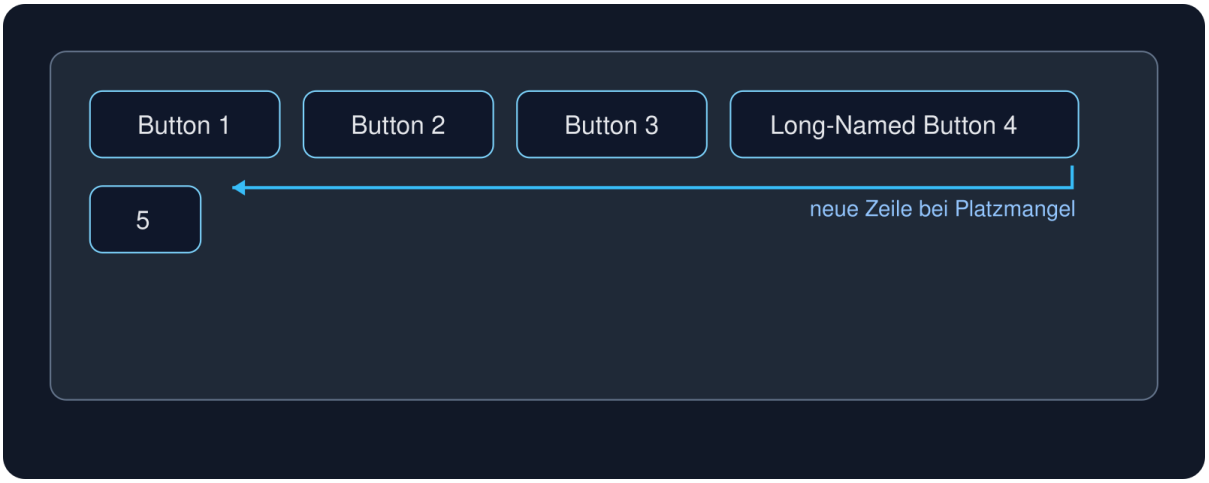

Abbildung 12: FlowLayout ordnet Komponenten zeilenweise an; bei zu geringer Breite beginnt eine neue Zeile.

```
import javax.swing.*;
import java.awt.FlowLayout;

public class FlowLayoutDemo extends JFrame {
    public FlowLayoutDemo() {
        setTitle("FlowLayoutDemo");
        setSize(500, 160);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLayout(new FlowLayout(FlowLayout.LEFT));

        add(new JButton("Button 1"));
        add(new JButton("Button 2"));
        add(new JButton("Button 3"));
        add(new JButton("Long-Named Button 4"));
        add(new JButton("5"));

        setVisible(true);
    }

    public static void main(String[] args) {
        new FlowLayoutDemo();
    }
}
```

VERTIEFUNG **BorderLayout – fünf Bereiche**

Die Containerfläche wird in fünf Bereiche eingeteilt: `NORTH`, `WEST`, `SOUTH`, `EAST` und `CENTER`. Bei Größenänderungen wächst besonders der `CENTER`-Bereich; die Randbereiche passen sich entsprechend an. Direkt pro Position kann höchstens eine Komponente eingefügt werden (für mehrere Elemente nutzt man z. B. ein zusätzliches `JPanel`).

Abbildung 13: BorderLayout teilt die Containerfläche in fünf Bereiche.

```
import javax.swing.*;
import java.awt.BorderLayout;

public class BorderLayoutDemo extends JFrame {
    public BorderLayoutDemo() {
        setTitle("BorderLayoutDemo");
        setSize(420, 240);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLayout(new BorderLayout());

        add(new JButton("North"), BorderLayout.NORTH);
        add(new JButton("West"), BorderLayout.WEST);
        add(new JButton("South"), BorderLayout.SOUTH);
        add(new JButton("East"), BorderLayout.EAST);
        add(new JButton("Center"), BorderLayout.CENTER);

        setVisible(true);
    }

    public static void main(String[] args) {
        new BorderLayoutDemo();
    }
}
```

VERTIEFUNG **GridLayout – gitterförmige Anordnung**

Komponenten werden gitterförmig angeordnet. Jede Komponente ist gleich groß und nutzt den verfügbaren Platz vollständig aus. `new GridLayout(0, 2)` bedeutet dabei: beliebig viele Zeilen und genau zwei Spalten.

Abbildung 14: GridLayout ordnet Komponenten in gleich große Rasterzellen ein.

```
import javax.swing.*;
import java.awt.GridLayout;

public class GridLayoutDemo extends JFrame {
    public GridLayoutDemo() {
        setTitle("GridLayoutDemo");
        setSize(420, 220);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setLayout(new GridLayout(0, 2));

        add(new JButton("Button 1"));
        add(new JButton("Button 2"));
        add(new JButton("Button 3"));
        add(new JButton("Long-Named Button 4"));
        add(new JButton("5"));

        setVisible(true);
    }

    public static void main(String[] args) {
        new GridLayoutDemo();
    }
}
```

VERTIEFUNG **Vertiefung: Weitere Swing-Komponenten**

Swing stellt viele weitere GUI-Komponenten bereit. In E3 werden davon nur ausgewählte Komponenten betrachtet. Die folgenden Beispiele sind optional und zeigen typische Eingabe-, Auswahl- und Strukturierungselemente. Alle Komponenten folgen demselben Grundprinzip: Objekt erzeugen, konfigurieren, positionieren, hinzufügen und bei Bedarf im Event-Handler auswerten.

BEGRIFF **Benötigtes Importpaket**

```
import javax.swing.*;
```

VERTIEFUNG **JCheckBox – unabhängige Auswahl**

Eine JCheckBox eignet sich für Optionen, die unabhängig voneinander ein- oder ausgeschaltet werden können. Die Eingabe wird als Wahrheitswert modelliert und mit `isSelected()` ausgewertet. Im EVA-Sinn liefert sie Eingabezustände, die in der Verarbeitung geprüft werden.

```
JCheckBox newsletter = new JCheckBox("Newsletter abonnieren");
newsletter.setBounds(20, 20, 220, 25);
fenster.add(newsletter);

boolean ausgewaehlt = newsletter.isSelected();
```

VERTIEFUNG **JRadioButton + ButtonGroup – genau eine Auswahl**

JRadioButton wird verwendet, wenn aus mehreren Alternativen genau eine gewählt werden soll. Eine `ButtonGroup` organisiert die gegenseitige Ausschließlichkeit der Auswahl. Für EVA bedeutet das: klare Eingabeoptionen, deren Zustand im Event-Handler abgefragt werden kann.

```
JRadioButton klein = new JRadioButton("klein");
klein.setBounds(20, 20, 100, 25);

JRadioButton gross = new JRadioButton("groß");
gross.setBounds(20, 50, 100, 25);

ButtonGroup gruppe = new ButtonGroup();
gruppe.add(klein);
gruppe.add(gross);

fenster.add(klein);
fenster.add(gross);

if (gross.isSelected()) {
    ausgabe.setText("Groß wurde gewählt.");
}
```

VERTIEFUNG **JComboBox – Auswahl aus einer Liste**

Eine JComboBox bietet eine kompakte Auswahl aus mehreren vorgegebenen Werten. Typisch ist die Auswertung mit `getSelectedItem()`. Dadurch werden freie Texteingaben reduziert und mögliche Eingabefehler im Eingabeschritt minimiert.

```
String[] farben = {"Rot", "Grün", "Blau"};
JComboBox<String> auswahl = new JComboBox<>(farben);
auswahl.setBounds(20, 20, 120, 25);
fenster.add(auswahl);

String farbe = (String) auswahl.getSelectedItem();
```

VERTIEFUNG **JTextArea – mehrzeilige Texteingabe**

Eine JTextArea erlaubt längere oder mehrzeilige Eingaben. Mit `getText()` wird der gesamte Textinhalt ausgelesen. In EVA kann so umfangreichere Eingabe gesammelt und danach verarbeitet werden.

```
JTextArea textbereich = new JTextArea();
textbereich.setBounds(20, 20, 220, 80);
fenster.add(textbereich);

String text = textbereich.getText();
```

VERTIEFUNG **JPasswordField – verdeckte Texteingabe**

Ein JPasswordField ist für verdeckte Eingaben gedacht. Die Zeichen werden nicht offen angezeigt; zur Auswertung dient typischerweise `getPassword()`, das ein `char[]` liefert. Hier wird dies didaktisch vereinfacht in einen String umgewandelt.

```
JPasswordField passwortFeld = new JPasswordField();
passwortFeld.setBounds(20, 20, 160, 25);
fenster.add(passwortFeld);

String passwort = new String(passwortFeld.getPassword());
```

VERTIEFUNG **JPanel – Komponenten gruppieren**

Ein JPanel kann Komponenten gruppieren und später auch eigene Layouts enthalten. Es ist damit ein Strukturierungselement innerhalb des Hauptfensters und verdeutlicht das Container-Prinzip. In EVA unterstützt es vor allem den geordneten Aufbau der Eingabe- und Ausgabeelemente.

```
JPanel panel = new JPanel();
panel.setLayout(null);
panel.setBounds(10, 10, 250, 100);

JLabel label = new JLabel("Gruppe");
label.setBounds(20, 20, 100, 25);
panel.add(label);

fenster.add(panel);
```

Weitere Komponenten wie JList, JMenuBar, JOptionPane oder JSlider sind möglich, werden hier aber nur als Ausblick betrachtet. Entscheidend ist das Grundprinzip: passende Komponente auswählen, als Objekt erzeugen, konfigurieren, hinzufügen und über Methoden bzw. Ereignisse nutzen.

Zustände allein machen eine GUI noch nicht interaktiv. Erst durch Ereignisse wird festgelegt, wann diese Zustände gelesen oder verändert werden.

5. Listener und Ereignisreaktion

Ereignisverarbeitung trennt den Aufbau der Oberfläche vom Verhalten des Programms. Der GUI-Aufbau beschreibt, welche Objekte sichtbar sind; der Listener beschreibt, wie auf ein Ereignis reagiert wird. Ein Ereignis ist z. B. ein Button-Klick oder eine Auswahländerung. Über `addActionListener(...)` wird eine Reaktion registriert. Die Methode `actionPerformed(...)` wird erst ausgeführt, wenn das Ereignis eintritt.

Ein Button ist damit Auslöser, nicht Datenspeicher und nicht die Verarbeitung selbst. Welche Verarbeitungsroutine beim Klick ausgeführt wird, steht im registrierten Listener.

BEGRIFF **Ereignismodell**

Ereignisquelle → Ereignisobjekt → Listener → `actionPerformed(...)` → Reaktion

```
Button-Klick
    ↓
ActionEvent entsteht
    ↓
registrierter Listener reagiert
    ↓
actionPerformed(...) wird ausgeführt
```

BEGRIFF **Benötigte Importpakete**

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

Im sichtbaren Kern wird nur das anonyme `ActionListener`-Muster verwendet. Es ist im folgenden vollständigen Anhalteweg-Beispiel integriert.

VERTIEFUNG **Vertiefung: Listener-Bausteine und weitere Ereignisinformationen**

```
btnLogin.addActionListener(new ActionListener() {
    @Override
```

```
    public void actionPerformed(ActionEvent e) {
        // Dieser Block wird erst beim Klick auf den Button ausgeführt.
    }
});
```

`ActionEvent e` ist ein Ereignisobjekt mit Informationen über das ausgelöste Ereignis, zum Beispiel welche Komponente die Aktion ausgelöst hat. Für den Einstieg ist vor allem wichtig: Der Code in `actionPerformed(...)` läuft nicht beim Programmstart, sondern erst dann, wenn der Button angeklickt wird.

```
Object quelle = e.getSource();
```

Diese Information muss in einfachen Programmen nicht immer ausgewertet werden. Sie erklärt aber, warum `actionPerformed(...)` ein Ereignisobjekt als Parameter erhält.

Auch die in der Vertiefung gezeigten Komponenten können im Event-Handler ausgewertet werden, zum Beispiel mit `isSelected()`, `getSelectedItem()` oder `getText()`.

Im Event-Handler werden Eingabezustände gelesen und Reaktionen koordiniert. Dort lassen sich bekannte Kontrollstrukturen wie `if`, `else`, Schleifen oder Methodenaufrufe verwenden. Die Fachlogik sollte aber nicht vollständig im Listener verschwinden: Ein Button-Klick kann eine Methode auslösen, die Berechnung oder Prüfung selbst bleibt als Verarbeitungsroutine benannt. Damit werden die Inhalte aus den vorherigen Kapiteln zu Methoden und Kontrollstrukturen direkt in den GUI-Kontext übertragen.

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String name = eingabe.getText();

        if (name.equals("")) {
            ausgabe.setText("Bitte Namen eingeben.");
        } else {
            ausgabe.setText("Hallo, " + name + "!");
        }
    }
});
```

- `getText()` liest die Eingabe aus dem Textfeld.
- `if` prüft eine Bedingung und unterscheidet zwischen leerer und nicht leerer Eingabe.
- `setText(...)` setzt abhängig vom Ergebnis unterschiedliche Ausgaben.
- Die Kontrollstruktur steht innerhalb von `actionPerformed(...)`, weil sie erst beim Button-Klick ausgeführt werden soll.

6. Vollständiger GUI-EVA-Ablauf am Anhalteweg-Beispiel

GUI-Programme bestehen aus Aufbau und Verhalten. Zum Aufbau gehören Fenster, Komponenten, Positionen und Container. Zum Verhalten gehören die Reaktion auf Ereignisse, das Lesen von Eingaben, die Verarbeitung und die Ausgabe. Der Event-Handler verbindet beides: Er arbeitet mit den sichtbaren Objekten, sobald ein Benutzerereignis eintritt.

Das folgende Minimalbeispiel kombiniert die vorherigen Bausteine vollständig und zeigt den Weg von Eingabe über Ereignis und Verarbeitung zur Ausgabe.

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class GuiEvaMinimal {
    public static void main(String[] args) {
        JFrame fenster = new JFrame("Anhalteweg");
        fenster.setSize(320, 160);
        fenster.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fenster.setLayout(null);

        JTextField eingabe = new JTextField();
        eingabe.setBounds(20, 20, 160, 25);
        fenster.add(eingabe);

        JButton button = new JButton("Berechnen");
        button.setBounds(190, 20, 100, 25);
```

```
        fenster.add(button);

        JLabel ausgabe = new JLabel("Noch keine Berechnung.");
        ausgabe.setBounds(20, 70, 260, 25);
        fenster.add(ausgabe);

        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {

                        double geschwindigkeitKmh = Double.parseDouble(eingabe.getText());

                        double anhalteweg = berechneAnhalteweg(geschwindigkeitKmh);

                        ausgabe.setText("Anhalteweg: " + anhalteweg + " m");
            }
        });

        fenster.setVisible(true);
    }


              public static double berechneAnhalteweg(double geschwindigkeitKmh) {

                        double reaktionsweg = (geschwindigkeitKmh / 10.0) * 3.0;

                        double bremsweg = (geschwindigkeitKmh / 10.0) * (geschwindigkeitKmh / 10.0);

                        return reaktionsweg + bremsweg;
    }
}
```

- **Aufbau:** `JFrame`, `JTextField`, `JButton` und `JLabel` werden erzeugt und hinzugefügt.
- **Ereignis:** Ein Button-Klick startet `actionPerformed(...)`.
- **Eingabe:** `getText()` liest Text aus dem Textfeld, `Double.parseDouble(...)` macht daraus eine Zahl.
- **Verarbeitung:** `berechneAnhalteweg(...)` berechnet Reaktionsweg und Bremsweg und addiert beide Werte.
- **Ausgabe:** `setText(...)` aktualisiert das Label.
- **Trennung:** Dieselbe Methode könnte auch aus einem Konsolenprogramm oder Test heraus aufgerufen werden.

Modellgrenze: Das Beispiel verwendet eine vereinfachte Faustformel. Reale Anhaltewege hängen unter anderem von Reaktionszeit, Fahrbahn, Reifen und Bremsvorgang ab. Das Minimalbeispiel setzt zudem eine gültige Zahleneingabe voraus; Eingabevalidierung ist ein anschließender Diagnose- und Transferfall.

MATERIALISIERT **Startzustand und Ereignis anzeigen**

Startzustand

- Textfeld enthält `"50"`.
- Label zeigt `"Noch keine Berechnung."`.
- Button wartet auf einen Klick.

Ereignis

Der Benutzer klickt `Berechnen`. Dadurch startet der registrierte Event-Handler.

MATERIALISIERT **Event-Trace anzeigen**

Schritt	Anweisung / Aktion	Wirkung
1	actionPerformed(...)	Der Event-Handler startet.
2	eingabe.getText()	Liefert "50".
3	Double.parseDouble("50")	Liefert 50.0.
4	berechneAnhalteweg(50.0)	Liefert 40.0: Reaktionsweg 15.0 m plus Bremsweg 25.0 m.
5	ausgabe.setText(...)	Setzt den Labeltext auf "Anhalteweg: 40.0 m".

Tabelle 38: Event-Trace des Anhalteweg-Beispiels

MATERIALISIERT **Endzustand anzeigen**

Textfeld

Bleibt `"50"`.

Label

Zeigt `"Anhalteweg: 40.0 m"`.

Deutung: Der Button berechnet nicht selbst. Er löst ein Ereignis aus. Der Event-Handler liest den Zustand des Textfelds, ruft die Fachlogik auf und schreibt das Ergebnis zurück in das Label.

BEGRIFF **Schnittstellen-Merksatz**

Benutzeraktion → GUI-Komponente → Event-Handler → Verarbeitung → GUI-Ausgabe

Die GUI ist die Schnittstelle zwischen Benutzer und Programm. Der Event-Handler ist die Schnittstelle zwischen Oberfläche und Fachlogik: Er liest Zustände aus Komponenten, ruft Verarbeitung auf und schreibt Ergebnisse zurück.

Für größere Programme gilt: Die Oberfläche steuert Ein- und Ausgabe. Berechnungen und Prüfungen sollten möglichst in getrennten Methoden liegen, damit der Event-Handler als Vermittlungsschicht übersichtlich bleibt. Das betrifft zum Beispiel den Anhalteweg oder Prüfregeln.

VERTIEFUNG **Typische Fehlvorstellungen beim Event-Handling**

BEGRIFF **Benötigte Importpakete**

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

BEGRIFF **Ein Button speichert Fachinformation.**

Ein Button ist Auslöser, nicht Datenspeicher. Die eigentlichen Daten stehen z. B. in Textfeldern, Labels oder Variablen.

```
// Der Button löst nur aus.
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String name = eingabe.getText();
        ausgabe.setText("Hallo, " + name + "!");
    }
});
```

BEGRIFF **actionPerformed(...) läuft beim Programmstart.**

Der Code in `actionPerformed(...)` wird erst ausgeführt, wenn das registrierte Ereignis eintritt.

```
System.out.println("Programmstart");

button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button wurde geklickt");
    }
});
```

BEGRIFF **setText(...) verändert automatisch die Fachlogik.**

`setText(...)` verändert nur den Textzustand der Komponente.

```
double summe = a + b;                // Fachlogik
ausgabe.setText("Summe: " + summe);  // GUI-Ausgabe
```

BEGRIFF **Ein Klick startet das ganze Programm neu.**

Ein Klick startet nicht `main(...)` neu.

```
public static void main(String[] args) {
    // Wird beim Programmstart ausgeführt.
}
```

```
public void actionPerformed(ActionEvent e) {
    // Wird beim Klick ausgeführt.
}
```

BEGRIFF GUI-Code und Verarbeitung müssen immer zusammen im Listener stehen.

Bei größeren Programmen sollte Fachlogik ausgelagert werden.

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {

                                                                double geschwindigkeitKmh =


      Double.parseDouble(eingabe.getText());


                                                        double anhalteweg =

                                                                berechneAnhalteweg(geschwindigkeitKmh);

        ausgabe.setText(
            "Anhalteweg: " + anhalteweg
        );
    }
});


                                                public static double berechneAnhalteweg(double
geschwindigkeitKmh) {

                                                        double reaktionsweg = (geschwindigkeitKmh / 10.0) *
3.0;

                                                        double bremsweg = (geschwindigkeitKmh / 10.0) *
(geschwindigkeitKmh / 10.0);

                                                        return reaktionsweg + bremsweg;
}
```

VERTIEFUNG Weitere Beispiele

Die folgenden größeren Anwendungsbeispiele vertiefen dieselbe Leitidee: sichtbare Eingabe und Ausgabe, Auslösung durch Benutzerereignisse und Verarbeitung in passenden Methoden.

VERTIEFUNG Variantenbeispiel: Begrüßung mit Texteingabe

BEGRIFF Benötigte Importpakete

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String name = eingabe.getText().trim();

        if (name.equals("")) {
            name = "Gast";
        }

        ausgabe.setText("Hallo, " + name + "!");
    }
});
```

VERTIEFUNG Weiteres Beispiel: Addition zweier Zahlen

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class GuiAddition {
    public static void main(String[] args) {
        JFrame fenster = new JFrame("Addition");
        fenster.setSize(360, 180);
```

```
        fenster.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        fenster.setLayout(null);

        JTextField eingabeA = new JTextField();
        eingabeA.setBounds(20, 20, 80, 25);
        fenster.add(eingabeA);

        JTextField eingabeB = new JTextField();
        eingabeB.setBounds(110, 20, 80, 25);
        fenster.add(eingabeB);

        JButton button = new JButton("Addieren");
        button.setBounds(200, 20, 110, 25);
        fenster.add(button);

        JLabel ausgabe = new JLabel("Noch kein Ergebnis.");
        ausgabe.setBounds(20, 70, 280, 25);
        fenster.add(ausgabe);

        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                double a = Double.parseDouble(eingabeA.getText());
                double b = Double.parseDouble(eingabeB.getText());
                double summe = a + b;
                ausgabe.setText("Summe: " + summe);
            }
        });

        fenster.setVisible(true);
    }
}
```

VERTIEFUNG **Weiterführende Dokumentation**

Die hier behandelten Inhalte sind eine didaktische Auswahl. Insbesondere Hauptfenster/Container, Swing-Komponenten, Layout-Manager und Ereignisverarbeitung wurden fachlich reduziert übernommen. Swing umfasst deutlich mehr Themen wie Dialogfenster, Zeichnen mit `paintComponent`, GUI-Builder, Threads oder MVC; diese werden hier nur als Ausblick genannt. Professionelle Swing-Oberflächen werden auf dem Event Dispatch Thread erzeugt und aktualisiert; die E3-Beispiele reduzieren diesen Aspekt bewusst. Die fachliche Struktur orientiert sich unter anderem an der Vorlesungsunterlage Grafische Benutzeroberflächen mit Swing von O. Bittel, HTWG Konstanz. Weitere Details finden sich in der offiziellen Java-Dokumentation:

- Swing API
- AWT API

Diese Bausteine führen zum Abschlussgedanken: Aus strukturierten Abläufen und Methoden entstehen benannte Verarbeitungsroutinen, an die Q1.2 und Q1.3 fachlich anschließen.

ABSCHNITTSENDE · KAPITEL 8

Weiter: 9 Algorithmus, Implementierung und Diagnose

LERNABSCHNITT 9 VON 9 · E3

9 Algorithmus, Implementierung und Diagnose

Einfache Lösungsverfahren strukturieren, in Java umsetzen und ihre Wirkung prüfen

In E3 wird die gesamte fachliche Bewegung sichtbar: Ein Problem wird fachlich verstanden und durch Daten sowie Zustände modelliert. Ein einfacher Algorithmus ordnet die Verarbeitung; ein Struktogramm kann ihn sprachunabhängig darstellen. Java implementiert ihn als Programm, gegebenenfalls in mehreren Methoden. Ein Trace zeigt die konkrete Ausführung für bestimmte Eingaben, und Diagnose unterscheidet Fehler sowie Modellgrenzen.

Bereich	Leitfrage
E3	Wie modelliere und implementiere ich ein einfaches, endliches Lösungsverfahren?
Q1.2	Wie funktionieren konkrete Such- und Sortierverfahren, und wie unterscheiden sie sich hinsichtlich ihrer Laufzeit?
Q1.3	Wie lassen sich Probleme rekursiv in kleinere Probleme derselben Art zerlegen?

Tabelle 39: Einordnung von E3, Q1.2 und Q1.3

VERTIEFUNG **Transfer: E3-Bausteine in E5**

WERKZEUG **E5-Ausblick im Java-Werkzeug**

Diese Aufgaben nutzen E3-Bausteine wie Schleifen, Strings, Arrays, char-Rechnung und Modulo in kryptologischen Verfahren. Die fachliche Kryptologie wird in E5 vertieft.

- Aufgabe starten: E5-Ausblick: Zeichen mit Caesar verschieben
- Im Java-Werkzeug üben: E5-Ausblick: OTP mit festem Schlüsselarray

Die verlinkten Aufgaben stehen zusätzlich als Offline-Arbeitsaufträge in Anhang A.

Diagnoseperspektive: Programmieren als prüfende Praxis

Programmieren bedeutet in E3 nicht nur, Code zu schreiben. Entscheidend ist, die Wirkung einer Anweisung, eines Ausdrucks oder einer Methode prüfen zu können. Die Diagnose trennt Fehlerarten nach ihrer Ursache.

Fehlerart	Beispiel	Diagnosefrage
Übersetzungsfehler	Syntax- oder Typfehler, z. B. ein fehlendes Semikolon	Erkennt der Compiler den Fehler?
Laufzeitfehler	ungültiger Arrayindex	Welche Werte liegen beim Zugriff vor?
Logik-/Modellfehler	Programm läuft, liefert aber wegen einer falschen Bedingung oder eines unpassenden Modells ein fachlich falsches Ergebnis.	Beschreibt die Verarbeitung das fachliche Problem richtig?
Eingabe-/Validierungsfehler	Text statt einer erwarteten Zahl	Verletzt die Eingabe eine angenommene Vorbedingung?
Ereignis-/Verdrahtungsfehler	Listener fehlt oder die falsche Komponente wird ausgewertet.	Ist die Ereignisquelle korrekt mit einem Listener verbunden?

Tabelle 40: Diagnosematrix zu typischen Programmierfehlern

ABSCHNITTSENDE · E3-HAUPTPFAD

Weiter: Begriffe und Beziehungen

REFERENZTEIL · E3

Begriffe und Beziehungen

Die Live-Seite erschließt diese Begriffe über Inline-Erklärungen und das interaktive Wissensnetz. Im linearen Medium werden sie als lokales Register und als Lernbewegung sichtbar gemacht.

Datentypen	Variable
Operatoren	Referenztypen
Kontrollstrukturen	Objekt
Schleifen	Typumwandlung
Arrays	Casting
Strings	Ausdruck
Methoden	Struktogramm
GUI	Index
ereignisgesteuerte Anwendungen	Methodenaufruf
Algorithmus	Methodenkopf
Programm	Methodenrumpf
Anweisungen	Button
Quelltext	EVA-Modell
Compiler	Container
Bytecode	Konstruktor
Klasse	Layout-Manager
main-Methode	

RELATION **Lernbewegung im E3-Begriffsraum**

- Daten werden in Variablen mit Datentypen gespeichert.
- Operatoren bilden Ausdrücke; Vergleichsausdrücke liefern Wahrheitswerte.
- Kontrollstrukturen nutzen Wahrheitswerte, um Abläufe zu verzweigen oder zu wiederholen.
- Schleifen bearbeiten geordnete Daten in Arrays und Strings.
- Methoden kapseln Verarbeitung und schaffen wiederverwendbare Schnittstellen.
- GUI-Komponenten erzeugen Zustände und Ereignisse; Event-Handler verbinden Oberfläche und Fachlogik.
- Ein Algorithmus beschreibt die eindeutige, endliche und ausführbare Lösungsstruktur, die das Programm implementiert.

Interaktive Fassung: https://informatik-dbook.de/einfuehrungsphase/E3.html

ABSCHNITTSENDE · BEGRIFFSREGISTER
Weiter: Anhang A · Offline-Arbeitsaufträge

ARBEITSTEIL · E3

Anhang A · Offline-Arbeitsaufträge

Dieser Anhang materialisiert genau die 35 Java-Aufgaben, die auf der E3-Liveseite verlinkt sind. Er enthält Arbeitsauftrag, Kriterien und – soweit vorhanden – Start- oder Bezugscode. Hinweise, automatische Diagnose und Musterlösungen bleiben im interaktiven Werkzeug, damit die dort vorgesehene Freischaltlogik erhalten bleibt.

E3 · Programmstart und Variablen

J-E3-P1 · Werte direkt zuweisen und ausgeben

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Werte direkt zuweisen und ausgeben**

Implementiere ein vollständiges Java-Konsolenprogramm, das mindestens eine Variable mit passendem Datentyp anlegt, initialisiert und mit System.out.println(...) sichtbar ausgibt. Der Schwerpunkt liegt auf Java-Rahmen, Variable, Datentyp, Zuweisung und Konsolenausgabe. Feste Startwerte im Quelltext reichen dafür vollständig aus.

Startcode

```
public class WerteDirektZuweisen {
    public static void main(String[] args) {
        // Lege mindestens eine Variable mit einem festen Wert an.
        // Gib den gespeicherten Wert mit System.out.println(...) aus.
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Tragfähig ist die Aufgabe, wenn der Code ausführbar ist, Variablen mit passenden Datentypen anlegt, Werte speichert und diese Werte sichtbar ausgibt. Die Musterlösung nutzt feste Startwerte; weiterentwickelte Lösungen sind akzeptabel, wenn sie dieselbe Kernstruktur nachvollziehbar erfüllen.

J-E3-P2 · Versandauftrag: Werte zuweisen, korrigieren und Zustände ausgeben

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Entwickle ein Java-Konsolenprogramm, das Informationen eines Versandauftrags als Variablen speichert, einen Anfangszustand ausgibt, einen gespeicherten Wert korrigiert und den korrigierten Zustand erneut ausgibt.**

Problem: Ein Versandauftrag soll als Programmzustand mit ganzzahligen Werten sichtbar werden.

Ziel: Entwickle ein Java-Konsolenprogramm, das Informationen eines Versandauftrags als Variablen speichert, einen Anfangszustand ausgibt, einen gespeicherten Wert korrigiert und den korrigierten Zustand erneut ausgibt.

Modellierungsfragen

- Welche Informationen beschreiben den Zustand eines Versandauftrags?
- Welche dieser Informationen lassen sich als ganze Zahlen repräsentieren?
- Welcher gespeicherte Wert kann sich im Verlauf durch eine Korrektur ändern?
- Welche Ausgaben machen Anfangszustand und korrigierten Zustand unterscheidbar?

Bedingungen

- Mindestens zwei Informationen werden als Variablen repräsentiert.
- Mindestens zwei Informationen werden als ganzzahlige Werte modelliert.
- Eine getrennte Deklaration mit späterer Initialisierung wird verwendet.
- Eine kombinierte Deklaration mit Initialisierung wird verwendet.
- Ein Anfangszustand und ein korrigierter Zustand werden sichtbar.
- Mindestens ein gespeicherter Wert wird später verändert.

Erfolgskriterien

- **Variablenmodell:** Mehrere Informationen des Versandauftrags werden als Variablen repräsentiert.
- **Ganzzahlige Werte:** Mindestens zwei Informationen werden als int-Werte modelliert.
- **Deklaration und Initialisierung:** Getrennte und kombinierte Deklaration/Initialisierung sind erkennbar.
- **Zustandsänderung:** Ein gespeicherter Wert wird später neu zugewiesen.
- **Ausgabe:** Anfangszustand und korrigierter Zustand sind in der Ausgabe unterscheidbar.

Startcode

```
public class VersandauftragZustand {
    public static void main(String[] args) {
        // Überlege, welche ganzzahligen Informationen den Versandauftrag beschreiben.
        // Deklariere mindestens eine Variable zuerst ohne Wert.
        // Initialisiere sie anschließend mit einem festen Wert.
        // Lege mindestens eine weitere Variable direkt mit einem festen Wert an.
        // Gib den Anfangszustand des Versandauftrags aus.
        // Korrigiere mindestens einen gespeicherten Wert durch Neuzuweisung.
        // Gib den korrigierten Zustand erneut aus.
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Tragfähig sind ausführbarer Code, ein klarer Versandauftrag-Kontext, sinnvolle ganzzahlige Variablen, getrennte Deklaration und Initialisierung, kombinierte Deklaration mit Initialisierung, spätere Neuzuweisung als Korrektur sowie sichtbare Anfangs- und Endzustände. Unterschiedliche Versandmodelle sind akzeptabel, wenn sie diese Kernanforderungen nachvollziehbar erfüllen.

J-E3-P3 · Versandauftrag: Scanner-Eingaben passend typisieren

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Entwickle ein Java-Konsolenprogramm, das mehrere Informationen eines Versandauftrags einliest, passend typisiert, eine Wahrheitsinformation aus einer Bedingung ableitet und die Verarbeitung nachvollziehbar ausgibt.**

Problem: Ein Versandauftrag wird nicht mehr nur mit festen Startwerten beschrieben, sondern über Eingaben von außen aufgebaut.

Ziel: Entwickle ein Java-Konsolenprogramm, das mehrere Informationen eines Versandauftrags einliest, passend typisiert, eine Wahrheitsinformation aus einer Bedingung ableitet und die Verarbeitung nachvollziehbar ausgibt.

Modellierungsfragen

- Welche Informationen des Versandauftrags sollen von außen eingegeben werden?
- Welche Eingaben sind Textinformationen, welche sind Zahlenwerte?
- Für welche Information ist ein Dezimalwert fachlich sinnvoll?
- Welche Ja/Nein-Information kann aus den Eingaben abgeleitet werden?
- Welche Ausgaben machen Eingabe und Verarbeitung nachvollziehbar?

Bedingungen

- Mehrere Werte werden über Scanner gelesen.
- Mindestens eine Textinformation wird gelesen.
- Mindestens ein ganzzahliger Wert und ein Dezimalwert werden verarbeitet.

- Mindestens eine Wahrheitsinformation wird aus einer Bedingung abgeleitet.
- Eingaben und Verarbeitungsergebnisse werden verständlich beschriftet ausgegeben.

Erfolgskriterien

- **Scanner-Eingaben:** Mehrere Versandinformationen werden über Scanner gelesen.
- **Datentypwahl:** Text, Ganzzahl, Dezimalwert und Wahrheitswert sind passend eingesetzt.
- **Abgeleiteter Wahrheitswert:** Eine boolean-Information entsteht aus einem Vergleich.
- **Ausgabe:** Die Ausgabe macht Eingaben und Verarbeitungsergebnisse nachvollziehbar.

Startcode

```
import java.util.Scanner;

public class VersandauftragDatentypen {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Lies zuerst eine Textbeschreibung mit nextLine().
        // Lies danach passende Zahlenwerte des Versandauftrags.
        // Berechne eine boolean-Variable aus einem Vergleich.
        // Gib die Eingaben und Verarbeitungsergebnisse beschriftet aus.
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Die Aufgabe erzwingt keine exakten Variablennamen oder Versandattribute. Tragfähig sind ausführbarer Code, Scanner-Eingaben, String, int, double, eine berechnete boolean-Variable und eine lesbare Ausgabe des Versandauftrags.

J-E3-P4 · Datentypwahl, Scanner-Eingaben und EVA-Struktur begründen

Niveau: Begründen · Bearbeitungsmodus: schriftliche Erklärung · Online im Java-Werkzeug öffnen

AUFGABE **Datentypwahl, Scanner-Eingaben und EVA-Struktur begründen**

Begründe die Modellierungsentscheidungen deines Versandauftrag-Programms. Erkläre für jede relevante Information, ob sie eingegeben, verarbeitet oder ausgegeben wird, welchen Datentyp du gewählt hast, welche Scanner-Methode dazu passt und warum dieser Datentyp fachlich passt. Benenne außerdem, welche Verarbeitung stattfindet und welche Ausgabe entsteht. Du kannst dich auf deinen Code aus der vorherigen Aufgabe beziehen. Eine gute Begründung unterscheidet Eingabe, Verarbeitung und Ausgabe.

Schriftliche Sicherung: Begründe, welche Werte eingegeben, verarbeitet und ausgegeben werden, welche Scanner-Methoden und Datentypen passen und warum diese Entscheidungen fachlich sinnvoll sind.

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Die Textdiagnose ist formativ und keine Vollbewertung. Die Aufgabe verlangt keine neue Implementierung.

E3 · Rechnerfolge

J-E3-A1 · Zwei Eingabewerte arithmetisch verarbeiten

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Zwei Eingabewerte arithmetisch verarbeiten**

Implementiere ein Java-Konsolenprogramm, das zwei ganze Zahlen über Scanner einliest und arithmetisch verarbeitet. Berechne und gib beschriftet aus: Summe, Differenz, Produkt, ganzzahligen Quotienten und Rest. Der fachliche Schwerpunkt liegt auf Eingabe, arithmetischer Verarbeitung, Zuweisung und beschrifteter Ausgabe. Eine einfache sequenzielle Lösung reicht vollständig aus.

Startcode

```
import java.util.Scanner;

public class Main {
```

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Lies zwei ganze Zahlen ein.
        // Berechne Summe, Differenz, Produkt, Quotient und Rest.
        // Gib jedes Ergebnis beschriftet aus.
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Tragfähig ist die Aufgabe, wenn der Code ausführbar ist, zwei Werte über Scanner einliest, mehrere arithmetische Operatoren inklusive % nutzt und die Ergebnisse beschriftet ausgibt. Die Musterlösung zeigt eine einfache sequenzielle Verarbeitung; weiterentwickelte Lösungen sind akzeptabel, wenn die Kernanforderungen nachvollziehbar erfüllt sind.

J-E3-A2 · Taschenrechner: Rechenergebnisse vergleichen

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Erweitere dein A1-Programm so, dass arithmetische Ergebnisse weitergeführt, mindestens drei Vergleichsausdrücke gebildet und die entstehenden Wahrheitswerte verständlich ausgegeben werden.**

Problem: Ein Taschenrechnerprogramm erzeugt aus zwei Eingaben mehrere Rechenergebnisse. Diese Ergebnisse können anschließend verglichen werden.

Ziel: Erweitere dein A1-Programm so, dass arithmetische Ergebnisse weitergeführt, mindestens drei Vergleichsausdrücke gebildet und die entstehenden Wahrheitswerte verständlich ausgegeben werden.

Modellierungsfragen

- Welche Rechenergebnisse oder Eingaben lassen sich sinnvoll vergleichen?
- Welche Vergleiche liefern für den Taschenrechner-Kontext aussagekräftige Wahrheitswerte?
- Welche Vergleichsergebnisse sollen als boolean-Zustand gespeichert oder sichtbar ausgegeben werden?
- Wie werden Rechenergebnisse und Vergleichsergebnisse in der Ausgabe unterscheidbar?

Bedingungen

- Arithmetische Ergebnisse aus A1 werden weitergeführt.
- Mindestens drei Vergleichsausdrücke werden gebildet.
- Vergleichsergebnisse werden als Wahrheitswerte sichtbar.
- Rechenergebnisse und Vergleichsergebnisse werden verständlich beschriftet ausgegeben.

Erfolgskriterien

- **Arithmetik weitergeführt:** Die arithmetischen Ergebnisse aus A1 bleiben im Programm sichtbar.
- **Vergleichsausdrücke:** Mindestens drei Vergleichsausdrücke werden gebildet.
- **Wahrheitswerte:** Vergleichsergebnisse werden als boolean-Werte gespeichert oder sichtbar ausgegeben.
- **Ausgabe:** Rechen- und Vergleichsergebnisse sind unterscheidbar beschriftet.

Startcode

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Lies zwei ganze Zahlen ein.
        // Berechne die arithmetischen Ergebnisse aus A1.
        // Ergänze mindestens drei Vergleichsausdrücke.
        // Speichere die Vergleichsergebnisse in boolean-Variablen.
        // Gib Rechen- und Vergleichsergebnisse beschriftet aus.
    }
}
```

> PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**
>
> Tragfähig ist die Aufgabe, wenn das Taschenrechnerprogramm aus A1 um Vergleichsausdrücke erweitert wird, die Vergleichsergebnisse als boolean-Werte gespeichert oder sichtbar ausgegeben werden und die Konsolenausgaben verständlich beschriftet sind. Unterschiedliche Vergleichsentscheidungen sind akzeptabel, wenn arithmetische Ergebnisse und Vergleichsergebnisse fachlich nachvollziehbar bleiben.

J-E3-A3 · Auswertungsspur zu Beispiel-Eingaben erläutern

Niveau: erlaeutere · Bearbeitungsmodus: schriftliche Erklärung · Online im Java-Werkzeug öffnen

> AUFGABE **Auswertungsspur zu Beispiel-Eingaben erläutern**
>
> Erläutere die Auswertung eines arithmetischen Ausdrucks und eines Vergleichsausdrucks zu konkreten Beispiel-Eingaben, zum Beispiel 8 und 3. Es muss kein neuer Code geschrieben werden. Gehe darauf ein, welche Eingabewerte verwendet werden, in welcher Reihenfolge Teilausdrücke ausgewertet werden und welchen Ergebnistyp die Ausdrücke haben.

Schriftliche Sicherung: Beschreibe, wie Java den Ausdruck schrittweise auswertet.

> PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**
>
> Die Textdiagnose ist formativ und keine Vollbewertung. Die Aufgabe verlangt keine neue Implementierung.

J-E3-A4 · Taschenrechner-Speicher mit Eingaben verändern

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

> AUFGABE **Entwickle ein Java-Konsolenprogramm, das mehrere numerische Eingaben liest, Rechenergebnisse bildet, einen Speicherzustand mit Kurzschreibweisen verändert, Vergleichsergebnisse erzeugt und den Ablauf verständlich ausgibt.**
>
> **Problem:** Ein Taschenrechner-Speicher hält einen Startzustand fest und wird durch Rechenoperationen verändert.
>
> **Ziel:** Entwickle ein Java-Konsolenprogramm, das mehrere numerische Eingaben liest, Rechenergebnisse bildet, einen Speicherzustand mit Kurzschreibweisen verändert, Vergleichsergebnisse erzeugt und den Ablauf verständlich ausgibt.
>
> **Modellierungsfragen**
>
> - Welche Eingaben bilden den Startzustand des Rechners?
> - Welche Rechenoperationen verändern oder erklären den Speicherzustand?
> - Welche Kurzschreibweisen passen zu einer Veränderung desselben Speicherwerts?
> - Welche Vergleichsfragen machen den Endzustand auswertbar?
> - Welche Ausgaben machen Rechenweg, Speicherzustand und Wahrheitswerte unterscheidbar?
>
> **Bedingungen**
>
> - Mehrere numerische Eingaben werden gelesen.
> - Ein Speicherzustand wird verändert.
> - Mehrere arithmetische Operationen, Kurzschreibweisen und Vergleiche werden sichtbar.
> - Ergebnisse werden verständlich ausgegeben.
>
> **Erfolgskriterien**
>
> - **Eingaben:** Mehrere numerische Eingaben werden über Scanner gelesen.
> - **Speicherzustand:** Ein Speicherwert wird angelegt und später verändert.
> - **Operatoren:** Arithmetische Operatoren, Kurzschreibweisen und Vergleichsausdrücke sind sichtbar.
> - **Wahrheitswerte:** Vergleichsergebnisse werden als Wahrheitswerte sichtbar.
> - **Ausgabe:** Rechenweg, Speicherzustand und Wahrheitswerte sind unterscheidbar beschriftet.

Startcode

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

```
public class TaschenrechnerSpeicher {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Lies zwei Ausgangswerte und einen Speicherstart ein.
        // Berechne mehrere Ergebnisse.
        // Verändere den Speicherwert mit Kurzschreibweisen.
        // Gib Vergleichsergebnisse als true/false aus.
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Tragfähig ist die Aufgabe, wenn Scanner-Eingaben, Rechenoperatoren, Kurzschreibweisen, Speicherwertveränderungen und Vergleichsausdrücke sichtbar werden. Unterschiedliche sequenzielle Verarbeitungen sind akzeptabel, wenn Speicherwert, Rechenweg und boolean-Ergebnisse fachlich nachvollziehbar bleiben.

J-E3-A5 · Fehlerhafte Verarbeitung im EVA-Modell prüfen

Niveau: Prüfen · Bearbeitungsmodus: schriftliche Erklärung · Online im Java-Werkzeug öffnen

AUFGABE **Fehlerhafte Verarbeitung im EVA-Modell prüfen**

Prüfe die fehlerhafte Java-Implementierung einer Versandkostenrechnung als EVA-Problem. Benenne Eingabewerte beziehungsweise Startwerte, Verarbeitung und Ausgabe. Korrigiere die Modellierungsfehler, Datentypfehler, Rechenfehler, Zuweisungen, Vergleiche, Kurzschreibweisen oder Ausgabefehler. Erläutere kurz, warum deine Korrekturen fachlich richtig sind. Achte besonders darauf, ob die gewählten Datentypen zu den gespeicherten Werten passen, ob der fachliche Prozess durch die Verarbeitung richtig repräsentiert wird und ob die Java-Operatoren korrekt verwendet werden.

Sollbeschreibung

```
Ein Taschenrechner-Speicher soll eine einfache Versandkostenrechnung abbilden.

Es gibt 3 Pakete.
Jedes Paket kostet 250 Cent.
Zusätzlich fällt eine Grundgebühr von 400 Cent an.
Ein Rabatt von 150 Cent wird abgezogen.
Das Gewicht pro Paket beträgt 2.5 kg.
Am Ende soll geprüft werden, ob das Gewicht pro Paket mindestens 2.0 kg beträgt.

Fachliche Zielsetzung:
- Eingabewerte oder Startwerte bilden die Grundlage der Verarbeitung.
- Die Paketkosten ergeben sich aus Anzahl der Pakete mal Preis pro Paket.
- Der Speicherwert beginnt bei 0 Cent.
- Die Grundgebühr wird addiert.
- Die Paketkosten werden addiert.
- Der Rabatt wird abgezogen.
- Das Gewicht pro Paket ist ein Dezimalwert.
- Der Gewichtsvergleich liefert einen Wahrheitswert.
- Die Ausgaben machen Gesamtpreis und Gewichtsentscheidung sichtbar.
```

Fehlerhafte Java-Implementierung

```
public class FehlerhaftePreisberechnung {
    public static void main(String[] args) {
        int anzahlPakete = 3;
        int preisProPaketCent = 250;
        int grundgebuehrCent = 400;
        int rabattCent = 150;
        int gewichtProPaketKg = 2.5;

        int gesamtpreisCent = anzahlPakete + preisProPaketCent;

        int speicherCent = 0;
        speicherCent += gesamtpreisCent;
        speicherCent *= grundgebuehrCent;
        speicherCent += rabattCent;

        boolean schweresPaket = gewichtProPaketKg = 2.0;

        System.out.println("Gesamtpreis in Cent:");
```

```
        System.out.println(speicherCent);

        System.out.println("Schweres Paket:");
        System.out.println(schweresPaket);
    }
}
```

Schriftliche Sicherung: Korrigiere die Modellierungsfehler, Datentypfehler und Rechenfehler im fehlerhaften Code.

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Die Textdiagnose ist formativ und keine Vollbewertung. Die Aufgabe verlangt keine neue Implementierung.

E3 · Bedingte Anweisungen und Verzweigungen

J-E3-IF1 · Kino-Status: optionalen Hinweis mit if ausgeben

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Kino-Status: optionalen Hinweis mit if ausgeben**

Implementiere ein vollständiges Java-Konsolenprogramm für einen kleinen Kino-Statushinweis. Lies eine Anzahl freier Plätze über Scanner ein. Berechne mit einem Vergleich, ob nur noch wenige Plätze frei sind. Wenn nur noch wenige Plätze frei sind, soll ein Hinweis ausgegeben werden. Danach soll das Programm immer ausgeben, dass die Statusprüfung beendet ist. Der Schwerpunkt liegt auf der einseitigen bedingten Anweisung: Ein zusätzlicher Block wird nur ausgeführt, wenn die Bedingung true ist. Für die Kernlösung reicht ein if-Block ohne else vollständig aus.

Startcode

```
import java.util.Scanner;

public class KinoStatusHinweis {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Lies die Anzahl freier Plätze ein.
        // Berechne, ob nur noch wenige Plätze frei sind.
        // Nutze if für den optionalen Hinweis.
        // Gib nach dem if immer aus, dass die Statusprüfung beendet ist.
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Tragfähig ist die Aufgabe, wenn der Hinweis korrekt an eine Bedingung gebunden ist und die Abschlussausgabe unabhängig vom if-Block erfolgt. Es wird nicht auf exakte Ausgabetexte geprüft.

J-E3-IF2 · Kino-Reservierung: bedingte Anweisung als Struktogramm modellieren

Niveau: modellieren · Bearbeitungsmodus: formative Bearbeitung · Online im Java-Werkzeug öffnen

AUFGABE **Kino-Reservierung: bedingte Anweisung als Struktogramm modellieren**

Modelliere das folgende Szenario als Struktogramm: Ein Kino prüft vor dem Einlass, ob eine Reservierung vorliegt. Die Eingabe ist ein Zahlenwert: 1 bedeutet Reservierung vorhanden, 0 bedeutet keine Reservierung. Wenn eine Reservierung vorhanden ist, soll der Hinweis "Reservierungscode bereithalten" ausgegeben werden. Danach läuft die Prüfung immer weiter und gibt "Einlassprüfung läuft weiter" aus. Der Java-Rahmen wird nicht dargestellt. Sichtbar werden sollen Eingabe, Vergleich, optionaler Hinweis und Fortsetzung. Der Nein-Zweig kann leer bleiben, weil bei false nur der optionale Hinweis übersprungen wird.

Szenario

```
Ein Kino prüft vor dem Einlass, ob eine Reservierung vorliegt.
Eingabe: 1 bedeutet Reservierung vorhanden, 0 bedeutet keine Reservierung.
Wenn eine Reservierung vorhanden ist, wird ausgegeben: Reservierungscode bereithalten.
Danach wird immer ausgegeben: Einlassprüfung läuft weiter.
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Keine automatische Struktogramm-Prüfung: Entscheidend ist, dass das Struktogramm eine bedingte Anweisung zeigt, der Ja-Zweig eine passende Aktion enthält, der Nein-Zweig leer bleibt oder als Platzhalter sichtbar ist und die Fortsetzung unterhalb der Bedingung steht.

J-E3-IF3 · Kinoeinlass: Entscheidung zwischen zwei Alternativen umsetzen

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Kinoeinlass: Entscheidung zwischen zwei Alternativen umsetzen**

Implementiere ein vollständiges Java-Konsolenprogramm für eine einfache Kinoeinlassentscheidung. Lies das Alter einer Person und die Altersfreigabe des Films über Scanner ein. Prüfe, ob die Person alt genug ist. Wenn die Person alt genug ist, soll der Einlass erlaubt werden. Sonst soll der Einlass verweigert werden. Am Ende soll immer ausgegeben werden, dass die Prüfung beendet ist. Der Kern ist die Entscheidung zwischen zwei Alternativen: Genau einer der beiden Zweige wird ausgeführt.

Startcode

```
import java.util.Scanner;

public class KinoEinlassEntscheidung {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Tragfähig ist die Aufgabe, wenn Alter und Altersfreigabe gelesen werden, ein Vergleich die Entscheidung vorbereitet, if-else als Strukturkern vorhanden ist, beide Alternativen sinnvolle Meldungen ausgeben und die Abschlussausgabe nach der Verzweigung steht.

J-E3-IF4 · Kinoeinlass: eigenes Regelmodell entwerfen und implementieren

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Entwickle ein eigenes Regelmodell für eine Kinoeinlassprüfung und setze es als Java-Konsolenprogramm um.**

Problem: Ein Kino möchte vor dem Einlass Regeln prüfen und Hinweise ausgeben.

Ziel: Entwickle ein eigenes Regelmodell für eine Kinoeinlassprüfung und setze es als Java-Konsolenprogramm um.

Modellierungsfragen

- Welche Eingaben braucht dein Einlassmodell?
- Welche Regel bildet die Hauptentscheidung?
- Welche Regel erzeugt nur einen optionalen Hinweis?
- Welche Bedingungen müssen dafür als Wahrheitswerte formuliert werden?
- Welche Ausgaben machen Entscheidung und Hinweis unterscheidbar?

Bedingungen

- Mehrere passende Eingaben werden über Scanner gelesen.
- Mindestens zwei Bedingungen beschreiben Regeln des gewählten Einlassmodells.
- Ein optionaler Hinweis wird bedingt ausgegeben.
- Eine Hauptentscheidung unterscheidet zwischen Alternativen.
- Am Ende wird die Prüfung verständlich abgeschlossen.

Erfolgskriterien

- **Eingaben:** Die Eingaben passen zum gewählten Regelmodell.
- **Bedingungen:** Die Bedingungen sind fachlich nachvollziehbar.

- **Hinweis und Hauptentscheidung:** Optionaler Hinweis und Hauptentscheidung sind unterscheidbar.
- **Verzweigung:** Die Verzweigung trifft eine erkennbare Einlassentscheidung oder gleichwertige Hauptentscheidung.
- **Ausgaben:** Die Ausgaben machen die Entscheidung verständlich.
- **Modellierung:** Die Modellierungsentscheidung ist im Kommentar oder Begründungstext nachvollziehbar.

Startcode

```
import java.util.Scanner;

public class KinoEinlassProblem {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Regelmodell:
        // Eingaben:
        // Regeln:

        // Lies mehrere passende Werte ein.
        // Formuliere mindestens zwei Bedingungen.
        // Unterscheide optionalen Hinweis und Hauptentscheidung.
        // Gib am Ende immer eine Abschlussmeldung aus.
    }
}
```

Schriftliche Sicherung: Beschreibe kurz, welche Eingaben du gewählt hast und welche Regeln dein Programm umsetzt.

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Tragfähig sind unterschiedliche Modellierungen, wenn sie innerhalb des Kino-Kontexts sinnvolle Eingaben, Bedingungen, Entscheidungen und Ausgaben enthalten. Bewertet wird nicht die Übereinstimmung mit einer festen Regelkette, sondern die fachliche Nachvollziehbarkeit des entworfenen Regelmodells.

E3 · Eingabekontrolle und Spielschleife

J-E3-B1 · gültige Mindestzahl erzwingen

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **gültige Mindestzahl erzwingen**

Implementiere ein Programm, das so lange Zahlen über Scanner einliest, bis eine Zahl eingegeben wurde, die mindestens 100 ist. Gib für jede zu kleine Eingabe den Text "Ungültig" aus. Gib am Ende die gültige Zahl mit dem Text "Gültig: " aus. Es soll kein Versuchszähler ausgegeben werden.

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Lies Zahlen ein, bis eine Zahl mindestens 100 erreicht.
    }
}
```

J-E3-B2 · Zahlenratespiel mit fester Geheimzahl

Niveau: passe-an · Bearbeitungsmodus: Werkzeugaufgabe · Online im Java-Werkzeug öffnen

AUFGABE **Zahlenratespiel mit fester Geheimzahl**

Passe die Eingabeschleife zu einem Zahlenratespiel an. Der Nutzer gibt wiederholt Tipps über Scanner ein. Die Geheimzahl ist fest 42. Lies eine Eingabe, prüfe sie und lies so lange neue Eingaben, bis der Tipp richtig ist. Für jeden falschen Tipp soll genau der Text "Zu klein" oder "Zu groß" ausgegeben werden. Wenn der Tipp richtig ist, soll "Gewonnen" ausgegeben werden. Überlege, welcher Vergleichsoperator für deine Fortlaufbedingung geeignet ist.

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Implementiere das Zahlenratespiel.
        // Gib Texte wie "Zu klein", "Zu groß" und "Gewonnen" in Anführungszeichen aus.
    }
}
```

J-E3-B3 · Zufallszahl raten und Versuche zählen

Niveau: Erweitern · Bearbeitungsmodus: Werkzeugaufgabe · Online im Java-Werkzeug öffnen

AUFGABE Zufallszahl raten und Versuche zählen

Erweitere das Zahlenratespiel so, dass die Geheimzahl nicht fest im Quelltext steht, sondern mit Random erzeugt wird. Nutze im Java-Tool Random zufall = new Random(1); und erzeuge damit eine Geheimzahl im Bereich von 1 bis 100. Lies Tipps ein, bis die Geheimzahl getroffen wurde. Gib nach falschen Tipps "Zu klein" oder "Zu groß" aus. Zähle außerdem alle gelesenen Tipps und gib am Ende "Gewonnen" sowie "Versuche: " mit der gezählten Anzahl aus.

Startcode

```
import java.util.Scanner;
import java.util.Random;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Random zufall = new Random(1);

        // Erzeuge hier die Geheimzahl im Bereich 1 bis 100.
        int eingabe = sc.nextInt();

        while (eingabe != geheim) {
            if (eingabe < geheim) {
                System.out.println("Zu klein");
            } else {
                System.out.println("Zu groß");
            }

            eingabe = sc.nextInt();
        }

        System.out.println("Gewonnen");
    }
}
```

J-E3-B4 · Zufallszahl mit verbotener Zahl und Maximalversuchen

Niveau: Erweitern · Bearbeitungsmodus: Werkzeugaufgabe · Online im Java-Werkzeug öffnen

AUFGABE Zufallszahl mit verbotener Zahl und Maximalversuchen

Erweitere das Spiel so, dass die Geheimzahl weiterhin mit Random im Bereich von 1 bis 100 erzeugt wird. Nutze im Java-Tool wieder Random zufall = new Random(1);. Zusätzlich soll in jedem Durchlauf der while-Schleife mit demselben Random-Objekt eine neue verbotene Zahl im Bereich von 1 bis 100 erzeugt werden. Wenn der aktuelle Tipp diese verbotene Zahl trifft, gib "Verbotene Zahl getroffen" aus und verlasse die Schleife mit break. Unabhängig davon soll nach maximal drei Versuchen abgebrochen werden. Für den dritten falschen Tipp soll noch "Zu klein" oder "Zu groß" ausgegeben werden; danach soll die Schleife mit break verlassen werden. Nach der Schleife soll das Programm "Gewonnen" ausgeben, wenn der Tipp der Geheimzahl entspricht, sonst "Verloren".

Startcode

```
import java.util.Scanner;
import java.util.Random;
```

```
class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Random zufall = new Random(1);

        int geheim = zufall.nextInt(100) + 1;
        int eingabe = sc.nextInt();
        int versuche = 1;

        while (eingabe != geheim) {

        }
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Die Aufgabe führt zwei Abbruchmechanismen zusammen: die normale while-Fortsetzung über den Tipp, den Maximalversuch und einen zusätzlichen break-Abbruch bei einer pro Durchlauf neu erzeugten verbotenen Zufallszahl. Für reproduzierbare Tool-Tests wird Random mit Seed 1 verwendet.

J-E3-B5 · Zufalls-Ratespiel als Struktogramm darstellen

Niveau: modellieren · Bearbeitungsmodus: formative Bearbeitung · Online im Java-Werkzeug öffnen

AUFGABE **Zufalls-Ratespiel als Struktogramm darstellen**

Überführe das fertige Zufalls-Ratespiel aus B4 in ein Struktogramm. Stelle nicht den Java-Rahmen mit import, class und main dar, sondern die Ablaufstruktur im Programmrumpf. Sichtbar werden sollen: das Anlegen von Scanner und Random, die Erzeugung der Geheimzahl als Zufallszahl im Bereich 1 bis 100, die erste Eingabe, der Versuchszähler, die while-Schleife, die pro Durchlauf neu erzeugte verbotene Zufallszahl, der break-Abbruch bei verbotener Zahl, der break-Abbruch nach maximal drei Versuchen, die Rückmeldungen, die neue Eingabe, die Zählerveränderung und die abschließende Entscheidung zwischen Gewonnen und Verloren.

Zu modellierendes Gesamtprogramm

```
Das Programm erzeugt die zu erratende Geheimzahl zufällig im Bereich 1 bis 100.
Danach liest es Tipps ein.
Nach jedem falschen Tipp wird ausgegeben, ob der Tipp zu klein oder zu groß war.
Die Variable versuche zählt die gelesenen Tipps.
In jedem Durchlauf der while-Schleife wird zusätzlich eine verbotene Zufallszahl im Bereich 1 bis 100
erzeugt.
Trifft der aktuelle Tipp diese verbotene Zahl, wird die Schleife mit break verlassen.
Nach dem dritten falschen Tipp wird ebenfalls noch die passende Rückmeldung ausgegeben und die Schleife mit
break verlassen.
Nach der Schleife entscheidet eine Verzweigung, ob Gewonnen oder Verloren ausgegeben wird.
Im Struktogramm wird der Java-Rahmen nicht dargestellt. Sichtbar werden die Anweisungen, die Zufallszahl-
Erzeugung, die while-Schleife, die if-Strukturen, break und die Fortsetzung nach der Schleife.
```

Startcode

```
import java.util.Scanner;
import java.util.Random;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Random zufall = new Random(1);

        int geheim = zufall.nextInt(100) + 1;
        int eingabe = sc.nextInt();
        int versuche = 1;

        while (eingabe != geheim) {
            int verboten = zufall.nextInt(100) + 1;

            if (eingabe == verboten) {
                System.out.println("Verbotene Zahl getroffen");
                break;
            }
```

```
            if (versuche == 3) {
                if (eingabe < geheim) {
                    System.out.println("Zu klein");
                } else {
                    System.out.println("Zu groß");
                }
                break;
            }

            if (eingabe < geheim) {
                System.out.println("Zu klein");
            } else {
                System.out.println("Zu groß");
            }

            eingabe = sc.nextInt();
            versuche = versuche + 1;
        }

        if (eingabe == geheim) {
            System.out.println("Gewonnen");
        } else {
            System.out.println("Verloren");
        }
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Keine automatische Struktogramm-Prüfung: Entscheidend ist, dass das Struktogramm die Ablaufstruktur des Gesamtprogramms zeigt. Erwartet werden Zufallszahl-Erzeugung für Geheimzahl und verbotene Zahl, while-Schleife, verbotene-Zahl-Prüfung mit break, dritter-Versuch-Prüfung mit break, Rückmeldungsverzweigung, Eingabe- und Zähleränderung sowie die abschließende Gewonnen/Verloren-Verzweigung.

E3 · For-Schleifen: Mess- und Kontrollpläne

J-E3-FOR1 · Messpunkte einer Wetterstation nummerieren

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Messpunkte einer Wetterstation nummerieren**

Implementiere ein Java-Konsolenprogramm für eine Schulhof-Wetterstation. Das Programm liest die Anzahl geplanter Messungen über Scanner ein und erzeugt anschließend für jede Messung eine nummerierte Ausgabe in der Form "Messung 1", "Messung 2" usw. Die Lösung soll eine for-Schleife mit einer Zählschleife verwenden. Sichtbar werden Startwert, Fortsetzungsbedingung und Inkrement der Zählschleife im Kopf.

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int anzahl = sc.nextInt();

        // Gib die geplanten Messungen als nummerierte Messpunkte aus.
    }
}
```

J-E3-FOR2 · Kontrollzeitpunkte mit festem Intervall planen

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Kontrollzeitpunkte mit festem Intervall planen**

Implementiere ein Java-Konsolenprogramm für den Kontrollplan einer Schulhof-Wetterstation. Das Programm liest Startminute, Endminute und Kontrollintervall über Scanner ein und erzeugt alle Kontrollzeitpunkte in der Form "Kontrolle bei Minute ...". Für diese Aufgabe gilt: Das Intervall ist positiv und die Startminute ist kleiner oder gleich der Endminute. Die for-Schleife soll das eingelesene Intervall als variables lineares Inkrement der Zählschleife mit minute += intervall nutzen.

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int startMinute = sc.nextInt();
        int endMinute = sc.nextInt();
        int intervall = sc.nextInt();

        // Gib alle Kontrollzeitpunkte von startMinute bis endMinute aus.
    }
}
```

J-E3-FOR3 · Messkampagne mit verschachtelten for-Schleifen erläutern

Niveau: erlaeutere · Bearbeitungsmodus: schriftliche Erklärung · Online im Java-Werkzeug öffnen

AUFGABE **Messkampagne mit verschachtelten for-Schleifen erläutern**

Erläutere, welche Problemstellung durch das Programm bearbeitet wird. Beschreibe den Realitätsausschnitt, die Eingaben, die Bedeutung der äußeren und inneren for-Schleife, die Zählvariablen, das lineare Wachstum, das nichtlineare Wachstum und die Funktion des Zählers messungen.

Szenario und Beispiel

```
Untersuche das folgende Programm und die Beispielausgabe. Rekonstruiere, welche Messplanung das Programm
beschreibt und wie die beiden for-Schleifen zusammenwirken.

Beispieleingabe:
3
8

Beispielausgabe:
Tag 1
  Messpause: 1 min
  Messpause: 2 min
  Messpause: 4 min
  Messpause: 8 min
Tag 2
  Messpause: 1 min
  Messpause: 2 min
  Messpause: 4 min
  Messpause: 8 min
Tag 3
  Messpause: 1 min
  Messpause: 2 min
  Messpause: 4 min
  Messpause: 8 min
Messungen insgesamt: 12
```

Bezugscode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int tage = sc.nextInt();
        int maxPause = sc.nextInt();

        int messungen = 0;
```

```
        for (int tag = 1; tag <= tage; tag++) {
            System.out.println("Tag " + tag);

            for (int pause = 1; pause <= maxPause; pause *= 2) {
                System.out.println("  Messpause: " + pause + " min");
                messungen++;
            }
        }

        System.out.println("Messungen insgesamt: " + messungen);
    }
}
```

Schriftliche Sicherung: Schreibe keinen neuen Code. Erläutere, welche Messplanung der Bezugscode beschreibt und wie die beiden for-Schleifen zusammenwirken.

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Textaufgabe: Bewertet wird die fachliche Erläuterung von Realitätsausschnitt, Eingaben, verschachtelten Schleifen, linearem und nichtlinearem Wachstum sowie Gesamtzähler.

J-E3-FOR4 · Messstrategie für eine Schulhof-Wetterstation entwickeln

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Entwickle ein eigenes, fachlich nachvollziehbares Messmodell und setze es als Java-Konsolenprogramm um.**

Problem: Eine Schulhof-Wetterstation braucht eine Messstrategie.

Ziel: Entwickle ein eigenes, fachlich nachvollziehbares Messmodell und setze es als Java-Konsolenprogramm um.

Modellierungsfragen

- Welche Messgröße oder welcher Zustand steuert den Ablauf deiner Messstrategie?
- Welche Eingaben braucht dein Modell, damit die Messstrategie berechnet werden kann?
- Was zählt oder verändert die for-Schleife in deinem Modell?
- Welche Bedingung im Schleifenrumpf hat fachliche Bedeutung?
- Welche Zusammenfassung zeigt am Ende, was die Messstrategie geleistet hat?

Bedingungen

- Mindestens drei Werte werden eingelesen.
- Eine for-Schleife steuert eine wiederholte Messhandlung.
- Im Schleifenrumpf wird mindestens eine fachlich begründete Bedingung geprüft.
- Am Ende wird das gewählte Messmodell zusammenfassend sichtbar.

Erfolgskriterien

- **Messmodell:** Das Messmodell ist erkennbar.
- **Eingaben:** Die Eingaben passen zum Modell.
- **for-Schleife:** Eine for-Schleife steuert eine wiederholte Messhandlung.
- **Bedingung:** Die Bedingung im Schleifenrumpf hat fachliche Funktion.
- **Ausgaben:** Ausgaben sind verständlich beschriftet.
- **Zusammenfassung:** Die Zusammenfassung passt zum gewählten Modell.

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Lies mindestens drei passende Werte ein.
        // Plane eine Messstrategie mit einer for-Schleife.
        // Nutze im Schleifenrumpf eine fachlich begründete Bedingung.
        // Gib am Ende eine Zusammenfassung aus.
    }
```

```
}
```

> **PRÜFFOKUS** **Woran eine tragfähige Bearbeitung erkennbar ist**
>
> Bewertet wird nicht die Übereinstimmung mit einem festen Lösungsweg, sondern ob eine fachlich nachvollziehbare Messstrategie mit for-Schleife, Eingaben, bedingter Anweisung und beschrifteter Ausgabe umgesetzt wird.

E3 · Datenstrukturen: Arrays

J-E3-D1 · Notenskala als Array ausgeben

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

> **AUFGABE** **Notenskala als Array ausgeben**
>
> Implementiere ein Java-Konsolenprogramm, das eine Notenskala als int-Array speichert und vollständig ausgibt. Lege ein Array notenSkala mit den Werten 1 bis 6 an. Durchlaufe das Array mit einer for-Schleife von Index 0 bis vor notenSkala.length. Gib jede Note in der Form "Note: 1" aus.

Startcode

```
class Main {
    public static void main(String[] args) {
        // Lege die Notenskala als int-Array an.
        // Durchlaufe alle gültigen Indizes mit einer for-Schleife.
        // Gib jede Note beschriftet aus.
    }
}
```

J-E3-D2 · Arraydurchlauf mit Index begründen

Niveau: Begründen · Bearbeitungsmodus: schriftliche Erklärung · Online im Java-Werkzeug öffnen

> **AUFGABE** **Arraydurchlauf mit Index begründen**
>
> Begründe, warum der Arraydurchlauf korrekt ist. Erkläre, warum die Zählvariable i bei 0 startet, warum die Bedingung i < noten.length lautet, warum noten[i] den Wert an der aktuellen Position liest und warum i <= noten.length hier falsch wäre. Erläutere außerdem, wie summe als Akkumulator funktioniert und warum für den Durchschnitt ein double-Anteil nötig ist.

Bezugscode

```
class Main {
    public static void main(String[] args) {
        int[] noten = {2, 3, 1, 4};

        int summe = 0;

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

        double durchschnitt = summe / (double) noten.length;
        System.out.println("Durchschnitt: " + durchschnitt);
    }
}
```

Schriftliche Sicherung: Schreibe keinen neuen Code. Erkläre die Indexlogik und den Akkumulator im vorgegebenen Programm.

> **PRÜFFOKUS** **Woran eine tragfähige Bearbeitung erkennbar ist**
>
> Die Textdiagnose prüft die fachliche Begründung der Indexlogik, des Akkumulators und der double-Division. Es wird kein neuer Code bewertet.

J-E3-D3 · Klassenarbeit mit Array auswerten

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Entwickle ein Java-Konsolenprogramm, das eine frei wählbare Anzahl gültiger Noten übernimmt und anschließend zentrale Kennwerte der Klassenarbeit ausgibt.**

Problem: Eine Lehrkraft möchte die Ergebnisse einer Klassenarbeit auswerten. Die Anzahl der gültigen Noten wird eingegeben, danach werden Noten erfasst und ausgewertet.

Ziel: Entwickle ein Java-Konsolenprogramm, das eine frei wählbare Anzahl gültiger Noten übernimmt und anschließend zentrale Kennwerte der Klassenarbeit ausgibt.

Modellierungsfragen

- Welche Daten müssen gespeichert werden, bevor die Auswertung möglich ist?
- Welche Eingaben sind gültig und welche dürfen nicht in die Sammlung übernommen werden?
- Welche Kennwerte beschreiben das Ergebnis der Klassenarbeit?
- Welche Zähler oder Zustände verändern sich während der Auswertung?
- Welche Ausgaben machen Durchschnitt, Extremwerte und Klassifikation nachvollziehbar?

Bedingungen

- Die Anzahl der gültig zu übernehmenden Noten wird eingelesen.
- Noten im Bereich 1 bis 6 gelten als gültige Daten.
- Die Auswertung bezieht sich ausschließlich auf gültig übernommene Noten.
- Durchschnitt, Extremwerte und Bestehensklassifikation werden aus den übernommenen Noten abgeleitet.

Erfolgskriterien

- **Eingaben:** Anzahl und Noten werden über Scanner gelesen.
- **Speicherung:** Gültige Noten werden in einer passenden int-Sammlung gespeichert.
- **Validierung:** Die Auswertung bleibt auf gültige Noten bezogen.
- **Auswertung:** Alle gespeicherten gültigen Noten werden berücksichtigt.
- **Durchschnitt:** Der Durchschnitt wird als Dezimalwert berechnet.
- **Extremwerte:** Beste und schlechteste Note werden bestimmt.
- **Klassifikation:** Bestanden/nicht bestanden wird nachvollziehbar gezählt oder abgeleitet.
- **Ausgabe:** Ergebnisse werden verständlich beschriftet ausgegeben.

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int anzahl = sc.nextInt();
        int[] noten = new int[anzahl];

        // Erfasse gültige Noten für die Auswertung.
        // Berechne anschließend die geforderten Kennwerte.
        // Gib die Auswertung verständlich beschriftet aus.
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Kriteriencheck: Bewertet wird, ob eine frei wählbare Anzahl gültiger Noten übernommen, die Auswertung auf gültige Noten bezogen und Durchschnitt, beste/schlechteste Note sowie bestanden/nicht bestanden nachvollziehbar ausgegeben werden. Die automatische Prüfung erkennt nur zentrale funktionale und strukturelle Kriterien und bewertet nicht alle Modellierungsentscheidungen.

E3 · Datenstrukturen: Strings

J-E3-E1 · Benutzername prüfen

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Benutzername prüfen**

Implementiere ein Java-Konsolenprogramm, das einen Benutzernamen prüft. Lies den Benutzernamen als String mit Scanner ein. Prüfe, ob er mindestens 8 Zeichen lang ist. Prüfe außerdem, ob er einen Unterstrich "_" enthält. Gib passende Rückmeldungen aus: "Länge ok" oder "Länge zu kurz" und "Unterstrich vorhanden" oder "Unterstrich fehlt".

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        String benutzername = sc.nextLine();

        // Prüfe Länge und Unterstrich.
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Geschlossene String-Prüfung: Bewertet wird hier ein eng geführtes Muster aus String-Eingabe, length(), contains("_"), if/else und den vorgegebenen Rückmeldungen. Die Vollbewertung ist hier tragfähig, weil die Aufgabe die Rückmeldungen und Prüfungen ausdrücklich vorgibt.

J-E3-E2 · Benutzernamen mit Index und substring erklären

Niveau: Begründen · Bearbeitungsmodus: schriftliche Erklärung · Online im Java-Werkzeug öffnen

AUFGABE **Benutzernamen mit Index und substring erklären**

Begründe, wie der vorgegebene Code den Benutzernamen zerlegt und formatiert. Erkläre, welche Bedeutung der Unterstrich-Index hat, warum substring(0, position) den Vornamen liefert, warum substring(position + 1) den Nachnamen liefert, warum der Endindex bei substring ausgeschlossen ist und wie charAt, toUpperCase und toLowerCase zur Formatierung genutzt werden.

Beispiel

```
Eingabe:
lEA_mUELLER

Ausgabe:
Vorname: Lea
Nachname: Mueller
```

Bezugscode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        String benutzername = sc.nextLine();

        int position = benutzername.indexOf("_");
        String vorname = benutzername.substring(0, position);
        String nachname = benutzername.substring(position + 1);

        String vornameFormatiert = ("" + vorname.charAt(0)).toUpperCase()
                + vorname.substring(1).toLowerCase();

        String nachnameFormatiert = ("" + nachname.charAt(0)).toUpperCase()
                + nachname.substring(1).toLowerCase();

        System.out.println("Vorname: " + vornameFormatiert);
        System.out.println("Nachname: " + nachnameFormatiert);
    }
}
```

Schriftliche Sicherung: Schreibe keinen neuen Code. Erkläre, wie Index, substring und Formatierung zusammenarbeiten.

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Die Textdiagnose prüft die fachliche Erklärung von Index, substring-Grenzen und String-Formatierung. Es wird kein neuer Code bewertet.

J-E3-E3 · Profilkennung erzeugen und prüfen

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Entwickle ein Java-Konsolenprogramm, das einen Anzeigenamen einliest, daraus eine normalisierte Profilkennung bildet und diese anschließend anhand einfacher Regeln prüft.**

Problem: Aus einem Anzeigenamen soll eine technische Profilkennung erzeugt werden, die für ein Informatiksystem einfacher weiterzuverarbeiten ist.

Ziel: Entwickle ein Java-Konsolenprogramm, das einen Anzeigenamen einliest, daraus eine normalisierte Profilkennung bildet und diese anschließend anhand einfacher Regeln prüft.

Modellierungsfragen

- Welche Eingabe beschreibt den Ausgangstext?
- Welche Verarbeitungsschritte erzeugen eine technische Kennung?
- Welche Zwischenwerte müssen gespeichert oder direkt weiterverarbeitet werden?
- Welche Prüfregeln gelten für die erzeugte Kennung?
- Welche Ausgaben trennen erzeugte Kennung und Prüfergebnisse?

Bedingungen

- Ein Anzeigename wird als String eingelesen.
- Aus dem Anzeigenamen wird eine technische Profilkennung abgeleitet.
- Die Profilkennung wird normalisiert und verständlich ausgegeben.
- Die Mindestlänge 8 und das Punkt-Verbot werden geprüft.

Erfolgskriterien

- **String-Eingabe:** Die String-Eingabe wird gelesen.
- **Normalisierung:** Die Profilkennung wird normalisiert.
- **Leerzeichen:** Leerzeichen werden ersetzt.
- **Ausgabe:** Die Profilkennung wird verständlich ausgegeben.
- **Mindestlänge:** Die Mindestlänge wird geprüft.
- **Punkt-Verbot:** Das Punkt-Verbot wird geprüft.
- **Rückmeldungen:** Rückmeldungen sind verständlich.

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        String anzeigename = sc.nextLine();

        // Erzeuge eine Profilkennung aus der Eingabe.
        // Prüfe die geforderten Regeln.
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Kriteriencheck: Bewertet wird, ob aus einer Texteingabe eine fachlich nachvollziehbare Profilkennung erzeugt wird, indem der Anzeigename normalisiert, Leerzeichen ersetzt, die Profilkennung ausgegeben und anschließend Länge sowie Punkt-Verbot geprüft werden. Die automatische Prüfung erkennt zentrale String-Operationen und Rückmeldestrukturen, bewertet aber nicht alle möglichen Modellierungsentscheidungen oder Variablennamen.

E3 · Methoden: Messkampagne modularisieren

J-E3-C1 · Messwertänderung als Methode berechnen

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Messwertänderung als Methode berechnen**

Implementiere ein Java-Konsolenprogramm für eine Schulhof-Wetterstation. Das Programm liest zwei Messwerte ein: einen Startwert und einen Endwert. Implementiere die Methode berechneAenderung(int startwert, int endwert). Die Methode soll endwert - startwert zurückgeben. Rufe die Methode in main auf, speichere den Rückgabewert und gib ihn in der Form "Änderung: ..." aus.

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int startwert = sc.nextInt();
        int endwert = sc.nextInt();

        // Rufe die Methode auf und gib die Änderung aus.
    }

    public static int berechneAenderung(int startwert, int endwert) {
        // Gib die Änderung zurück.
        return 0;
    }
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Geschlossene Methodensyntax: Bewertet werden Einlesen, Methodenkopf, Parameter, return-Ausdruck, Methodenaufruf, gespeicherter Rückgabewert und beschriftete Ausgabe.

J-E3-C2 · Messwertbereich mit Methode prüfen

Niveau: wende-an · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Messwertbereich mit Methode prüfen**

Eine Schulhof-Wetterstation soll prüfen, ob ein Messwert in einem erlaubten Bereich liegt. Lies einen Messwert, einen Minimalwert und einen Maximalwert ein. Implementiere die Methode istImBereich(int wert, int minimum, int maximum). Sie soll true zurückgeben, wenn wert mindestens minimum und höchstens maximum ist. Rufe die Methode in main auf. Gib "Messwert gültig" aus, wenn die Methode true liefert, sonst "Messwert ungültig".

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Lies Messwert, Minimum und Maximum ein.
        // Rufe eine passende Prüfmethode auf.
        // Gib aus, ob der Messwert gültig oder ungültig ist.
    }

    // Ergänze hier deine Methode istImBereich(...).
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Anwendungsaufgabe: Das Methodenmuster aus C1 wird auf eine boolean-Prüfmethode übertragen, die main für eine if/else-Rückmeldung nutzt.

J-E3-C3 · Parameterbindung und Rückgabe erklären

Niveau: Begründen · Bearbeitungsmodus: schriftliche Erklärung · Online im Java-Werkzeug öffnen

AUFGABE **Parameterbindung und Rückgabe erklären**

Schreibe keinen neuen Code. Erkläre, was beim Aufruf der beiden Methoden passiert. Gehe auf Argumente, Parameter, Parameterbindung, lokale Variablen, Rückgabewerte und die Rolle von main ein. Erkläre besonders, welche Werte beim Aufruf berechneAenderung(start, ende) an startwert und endwert gebunden werden und wie der Rückgabewert weiterverwendet wird.

Bezugscode

```
class Main {
    public static void main(String[] args) {
        int start = 18;
        int ende = 24;

        int aenderung = berechneAenderung(start, ende);
        boolean kritisch = istKritisch(ende, 30);

        System.out.println("Änderung: " + aenderung);
        System.out.println("Kritisch: " + kritisch);
    }

    public static int berechneAenderung(int startwert, int endwert) {
        return endwert - startwert;
    }

    public static boolean istKritisch(int messwert, int grenze) {
        return messwert >= grenze;
    }
}
```

Schriftliche Sicherung: Schreibe keinen neuen Code. Erkläre Argumente, Parameterbindung, Rückgabe und die Rolle von main.

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Begründungsaufgabe: Entscheidend ist die fachsprachliche Erklärung von Argumenten, Parametern, Parameterbindung, lokalen Variablen, Rückgabewerten und der Rolle von main.

J-E3-C4 · Messkampagne modularisieren

Niveau: Implementieren · Bearbeitungsmodus: Implementierung · Online im Java-Werkzeug öffnen

AUFGABE **Entwickle ein Java-Konsolenprogramm, das die Messkampagne modular verarbeitet: Eingaben lesen, gültige Messwerte speichern, Kommando normalisieren, Mittelwert berechnen, kritische Werte zählen und Ergebnisse ausgeben.**

Problem: Eine Schulhof-Wetterstation verarbeitet ein Kommando, eine Messreihe und einen Grenzwert für kritische Werte.

Ziel: Entwickle ein Java-Konsolenprogramm, das die Messkampagne modular verarbeitet: Eingaben lesen, gültige Messwerte speichern, Kommando normalisieren, Mittelwert berechnen, kritische Werte zählen und Ergebnisse ausgeben.

Modellierungsfragen

- Welche Daten beschreiben die Messkampagne und welche müssen gespeichert werden?
- Welche Werte müssen geprüft werden, bevor sie in die Messreihe eingehen?
- Welche Verarbeitungsschritte steuert main und welche werden in Methoden ausgelagert?
- Welche Rückgabewerte brauchen die Auswertungsmethoden?
- Welche Ausgaben machen Normalisierung und Auswertung nachvollziehbar?

Bedingungen

- Ein Kommando, die Anzahl der Messwerte, ein Grenzwert und Messwerte werden eingelesen.
- Messwerte im Bereich -50 bis 60 gelten als gültige Daten.
- Die Auswertung bezieht sich auf gültig übernommene Messwerte.

- Die Methoden normalisiereKommando, istGueltigerMesswert, berechneMittelwert und zaehleKritischeWerte bilden die geforderte Schnittstelle.
- main koordiniert Eingabe, Methodenaufrufe und Ausgabe.

Erfolgskriterien

- **Eingaben:** Kommando, Anzahl, Grenzwert und Messwerte werden eingelesen.
- **Speicherung:** Gültige Messwerte werden gespeichert.
- **Validierung:** Die Messreihe und Auswertung bleiben auf gültige Messwerte bezogen.
- **Normalisierung:** Das Kommando wird nachvollziehbar normalisiert.
- **Mittelwert:** Der Mittelwert wird über die gespeicherten Werte berechnet.
- **Kritische Werte:** Kritische Werte werden anhand des Grenzwerts gezählt.
- **Methoden:** Die Methoden werden sinnvoll genutzt.
- **Ausgabe:** Ergebnisse werden beschriftet ausgegeben.

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Messkampagne:
        // Lies die Daten der Messkampagne ein.
        // Fülle die Messwertsammlung nur mit gültigen Messwerten.
        // Rufe die Methoden für Normalisierung und Auswertung auf.
        // Gib die Ergebnisse beschriftet aus.
    }

    // Ergänze unterhalb von main die vier geforderten Methoden.
}
```

PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**

Problemaufgabe: Output-Tests prüfen zentrale Fälle; die Rubrik ergänzt strukturelle Hinweise auf validiertes Array-Füllen, String-Normalisierung und ausgelagerte Auswertungsmethoden.

J-E3-C5 · Modulare Messkampagne als Struktogramm darstellen

Niveau: modellieren · Bearbeitungsmodus: formative Bearbeitung · Online im Java-Werkzeug öffnen

AUFGABE **Modulare Messkampagne als Struktogramm darstellen**

Überführe deine modulare Messkampagne aus C4 in ein Struktogramm. Nutze deine eigene Lösung oder rekonstruiere die Struktur aus dem C4-Auftrag. Stelle nicht den Java-Rahmen dar, sondern den Programmrumpf von main. Methoden erscheinen als benannte Aufrufblöcke.

Struktogrammauftrag

```
Stelle die Struktur deiner C4-Lösung dar. Der Java-Rahmen mit import, class und main wird nicht gezeichnet.
Sichtbar werden sollen die fachlichen Ablaufblöcke: Eingaben für Messkampagne, Erzeugung der Messreihe,
validiertes Füllen der Messreihe, Methodenaufrufe zur Normalisierung und Auswertung sowie beschriftete
Ausgaben. Methoden dürfen als Call-Blöcke dargestellt werden; ihre Rümpfe müssen nicht aufgeklappt werden.
```

Startcode

```
// Nutze deine Lösung aus C4 als Vorlage.
// Stelle im Struktogramm nur den Programmrumpf von main dar.
//
// Sichtbar werden sollen:
// - Eingaben
// - Array-Erzeugung
// - validierte Eingabeschleife
// - Methodenaufrufe als benannte Call-Blöcke
// - Ausgaben
//
// Methodenrümpfe werden nicht vollständig aufgeklappt.
```

> PRÜFFOKUS **Woran eine tragfähige Bearbeitung erkennbar ist**
>
> Keine automatische Struktogramm-Prüfung: Entscheidend ist, dass das Struktogramm die Struktur der modularen Messkampagne zeigt. Erwartet werden Eingaben, Array-Erzeugung, validierte Eingabeschleife, Methodenaufrufe und Ausgaben. Methoden dürfen als benannte Aufrufblöcke dargestellt werden.

E5 · Caesar-Chiffre als E3-Transfer

J-E5-F1 · einzelnes Zeichen verschieben

Niveau: Implementieren · Bearbeitungsmodus: Werkzeugaufgabe · Online im Java-Werkzeug öffnen

> AUFGABE **einzelnes Zeichen verschieben**
>
> Lies einen Großbuchstaben und einen Schlüssel k ein. Verschiebe den Buchstaben nach Caesar um k Stellen. Es werden nur Großbuchstaben A bis Z betrachtet.

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Lies einen Buchstaben und einen Schlüssel ein.
    }
}
```

E5 · One-Time-Pad als Array-Transfer

J-E5-G1 · OTP mit Schlüsselarray

Niveau: Implementieren · Bearbeitungsmodus: Werkzeugaufgabe · Online im Java-Werkzeug öffnen

> AUFGABE **OTP mit Schlüsselarray**
>
> Lies einen Text aus Großbuchstaben ein. Lies danach genau so viele Schlüsselwerte ein, wie der Text Zeichen hat. Verschlüssele jedes Zeichen um den Schlüsselwert an derselben Position.

Startcode

```
import java.util.Scanner;

class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        String text = sc.nextLine();
        int[] key = new int[text.length()];

        // Lies die Schlüsselwerte ein und verschlüssele text mit key[i].
    }
}
```