DAS HANDWERK DES CODES

Imperative Programmierung in Java meistern (Band 1)

Dietrich Boles

Impressum

Autor: Dietrich Boles

Kontakt: dietrich@boles.de

Originalquelle (Ursprungsort): https://www.boles.de/programmierkurs-java/

Vorwort

Liebe Leserinnen und Leser,

willkommen zum ersten und wichtigsten Schritt Ihrer Reise in die Welt der Softwareentwicklung!

Wenn Sie dieses Buch aufschlagen, stehen Sie vermutlich noch ganz am Anfang. Sie wissen vielleicht noch nicht, wie man Klassen entwirft, Objekte instanziiert oder Algorithmen formuliert. Bisher kannten Sie die digitale Welt vor allem als reiner Anwender: Sie haben auf Knöpfe gedrückt, Menüs bedient, und die Maschine hat scheinbar wie von Zauberhand reagiert. Die Welt der Software bestand für Sie aus fertigen Produkten, deren innere Mechanismen verborgen blieben.

Mit dem Aufschlagen dieses Buches lade ich Sie ein, die Seiten zu wechseln und diese verborgene Welt selbst zu gestalten.

Die Realität ist: Ein Computer ist im Grunde eine sehr dumme, aber unglaublich schnelle Maschine. Er besitzt keine Intuition und keinen eigenen Willen. Er braucht jemanden, der ihm haargenau diktiert, was er tun soll – Schritt für Schritt, Anweisung für Anweisung. Genau das ist die Kunst der imperativen Programmierung.

In diesem Buch erlernen Sie das grundlegende Handwerk. Sie lernen eine tröstliche, berechenbare und verlässliche Welt kennen: die sequenzielle Programmierung. Sie werden dem Computer Befehle erteilen, und er wird eine Anweisung nach der anderen ausführen. Wenn Sie Ihr Programm zweimal mit denselben Eingabedaten starten, erhalten Sie exakt dasselbe Ergebnis – die Welt ist hier noch deterministisch. Sie lernen, wie Sie Daten im Speicher ablegen (Variablen), wie Sie den Fluss Ihres Programms durch logische Entscheidungen lenken (if-Abfragen) und wie Sie mühsame Arbeitsschritte automatisiert wiederholen lassen (Schleifen).

Dieses Buch gießt jedoch nur das Fundament. Es ist der Auftakt einer vierteiligen Reise, die Sie vom Anfänger zum modernen, professionellen Software-Architekten ausbilden wird:

- **Band 1 (Dieses Buch): Das Handwerk des Codes.** Hier erlernen Sie das imperative Fundament, die Syntax und die grundlegende Logik der Sprache Java.
- **Band 2: Die Architektur der Objekte.** Sobald Sie das imperative Handwerk beherrschen, werden Sie merken, dass reine Befehlsfolgen bei großen Programmen schnell unübersichtlich und fehleranfällig werden. Im zweiten Band lernen Sie die objektorientierte Programmierung kennen, bei der wir wehrlose Daten und Funktionen zu intelligenten, sich selbst verteidigenden Systemen (Objekten) verschmelzen.
- **Band 3: Der Fluss der Daten.** Je komplexer moderne Anwendungen werden, desto schwieriger wird es, riesige Mengen an veränderlichen Objekten zu bändigen. Im dritten Band tauchen wir in die funktionale Programmierung ein. Sie erleben einen Paradigmenwechsel, bei dem Sie dem Computer nicht mehr das *Wie*, sondern das *Was*

diktieren und hochgradig fehlerresistente Datenfabriken bauen.

- **Band 4: Die Choreografie der Threads.** Den krönenden Abschluss bildet die parallele und nebenläufige Programmierung. Da moderne Prozessoren immer mehr Rechenkerne erhalten, müssen zeitgemäße Programme unzählige Aufgaben gleichzeitig erledigen. Sie werden lernen, wie Sie das Chaos geteilter Ressourcen beherrschen und hochskalierbare, parallele Architekturen entwerfen.

Doch bevor man ein Orchester dirigieren kann, muss man lernen, Noten zu lesen. Die Umstellung vom reinen Anwender zum Programmierer erfordert anfangs Geduld und Disziplin. Die strikte Grammatik einer Programmiersprache wie Java verzeiht keine Tippfehler oder fehlenden Semikolons. Ihr Code wird anfangs Fehler werfen. Doch sobald Sie den Schalter im Kopf umgelegt haben und Ihr erstes eigenes Programm exakt das tut, was Sie ihm befohlen haben, werden Sie ein unvergleichliches Gefühl der Ermächtigung spüren.

Nutzen Sie Ihre neuen Fähigkeiten mit Neugierde. Experimentieren Sie mit den Code-Beispielen, testen Sie Ihre Grenzen aus und vor allem: Bauen Sie Ihre ersten Programme, auf die Sie stolz sein können!

Viel Erfolg beim Erlernen Ihres neuen Handwerks.

Dietrich Boles, Oldenburg, März 2026

Inhaltsverzeichnis

Teil I: Die Werkstatt einrichten (Grundlagen)

1 Grundlagen der Programmierung

Willkommen in Ihrer neuen Werkstatt! Bevor Sie zum ersten Mal den Hammer schwingen oder die Säge ansetzen – also bevor wir unsere erste Zeile Java-Code schreiben –, müssen wir uns mit den grundlegenden Konzepten unseres Handwerks vertraut machen.

In diesem ersten Kapitel legen wir das theoretische Fundament. Wir klären, was Programmieren eigentlich bedeutet, wie Computer im Innersten funktionieren und welche Werkzeuge wir benötigen, um einer Maschine unseren Willen aufzuzwingen. Wenn Sie diese Grundlagen verinnerlicht haben, wird Ihnen das Erlernen der eigentlichen Programmiersprache im nächsten Kapitel deutlich leichter fallen.

1.1 Terminologie und Algorithmus

Wenn wir von Softwareentwicklung sprechen, werfen wir oft mit Begriffen wie „Programm“, „Code“ oder „Algorithmus“ um uns. Lassen Sie uns diese Begriffe entzaubern.

Ein **Computerprogramm** (oder kurz Programm) ist im Grunde nichts anderes als eine detaillierte Arbeitsanweisung für einen Computer. Ähnlich wie ein Rezept für einen Koch oder eine Bauanleitung für einen Tischler, sagt ein Programm dem Computer exakt, welche Schritte er nacheinander ausführen muss, um ein bestimmtes Ziel zu erreichen.

Das Herzstück eines jeden Programms ist der **Algorithmus**. Ein Algorithmus ist eine präzise formulierte, endliche Folge von Anweisungen zur Lösung eines Problems.

Jeder gute Algorithmus muss bestimmte Eigenschaften erfüllen:

- **Eindeutigkeit (Determinismus):** Jeder Schritt muss glasklar definiert sein. Es darf keinen Interpretationsspielraum geben. Ein Rezept, das sagt „eine Prise Salz hinzufügen“, ist für einen Menschen verständlich, für einen Computer jedoch unbrauchbar. Wie viele Körner sind eine Prise?
- **Ausführbarkeit:** Der Computer muss in der Lage sein, die Schritte mit den ihm zur Verfügung stehenden Mitteln auszuführen.
- **Endlichkeit (Finitheit):** Die Beschreibung des Algorithmus muss eine endliche Länge haben.
- **Terminierung:** Der Algorithmus muss nach einer endlichen Anzahl von Schritten zu einem Ende kommen und ein Ergebnis liefern. Er darf sich nicht in einer Endlosschleife verfangen.

Ein alltägliches Beispiel für einen Algorithmus:

Stellen Sie sich vor, Sie möchten einem Roboter beibringen, eine Tür zu öffnen. Der Algorithmus könnte wie folgt aussehen:

1. Gehe auf die Tür zu, bis der Abstand weniger als 30 Zentimeter beträgt.
2. Hebe den rechten Arm auf Höhe der Türklinke.
3. Schließe die Hand um die Türklinke.
4. Drücke die Türklinke um 45 Grad nach unten.
5. Ziehe den Arm in Richtung des Körpers.
6. Lasse die Türklinke los.

Beim Programmieren machen wir exakt das Gleiche: Wir zerlegen ein großes Problem in winzige, unmissverständliche Teilschritte.

1.2 Programmiersprachen

Warum können wir dem Computer unsere Algorithmen nicht einfach auf Deutsch oder Englisch mitteilen?

Menschliche Sprachen (sogenannte natürliche Sprachen) sind wundervoll ausdrucksstark, aber sie haben einen massiven Nachteil: Sie sind extrem mehrdeutig und stark vom Kontext abhängig. Der Satz „Umfahren Sie das Schild“ kann bedeuten, dass man einen Bogen um das Schild machen soll – oder dass man es mit dem Auto umkippen soll. Ein Computer besitzt keine Lebenserfahrung, um solche Mehrdeutigkeiten aufzulösen. Er würde verzweifeln (oder das Schild rammen).

Zudem versteht ein Computer auf seiner untersten Ebene ohnehin keine Wörter, sondern nur Strompegel: Strom an (1) und Strom aus (0). Diese unterste Ebene nennt man **Maschinensprache**. Früher mussten Programmierer tatsächlich endlose Kolonnen von Nullen und Einsen eingeben, um Programme zu schreiben. Das war mühsam, extrem fehleranfällig und für Menschen kaum lesbar.

Um dieses Problem zu lösen, wurden **Programmiersprachen** erfunden (auch Hochsprachen genannt). Eine Programmiersprache wie Java ist eine Art Brückensprache zwischen Mensch und Maschine.

Sie bietet uns ein Vokabular aus englischen Begriffen (wie if, while, class) und strikten grammatikalischen Regeln (der sogenannten **Syntax**). Wenn wir uns streng an die Syntax halten, ist unser Code für Menschen gut lesbar und gleichzeitig absolut eindeutig, sodass er fehlerfrei in die Nullen und Einsen der Maschine übersetzt werden kann.

1.3 Entwicklungsphasen

Programmieren bedeutet nicht, sich sofort an die Tastatur zu setzen und wild Code einzutippen. Wahre Handwerkskunst erfordert Planung. Die Entwicklung eines Programms durchläuft in der Regel verschiedene Phasen, die den Bauplan eines Hauses widerspiegeln:

1. **Problemanalyse (Was ist zu tun?):** Zunächst müssen wir exakt verstehen, welches Problem gelöst werden soll. Welche Eingabedaten haben wir? Welches Ergebnis wird erwartet?
2. **Entwurf (Wie wollen wir es lösen?):** In dieser Phase formulieren wir den Algorithmus. Wir planen die Struktur unseres Programms, oft noch ganz ohne eine bestimmte Programmiersprache auf Papier oder Whiteboards.
3. **Implementierung (Das eigentliche Programmieren):** Jetzt übersetzen wir unseren Entwurf in eine konkrete Programmiersprache (z. B. Java). Wir schreiben den sogenannten **Quellcode** (Source Code).
4. **Testen (Funktioniert es?):** Wir führen unser Programm mit verschiedenen Testdaten aus und prüfen, ob das gewünschte Ergebnis geliefert wird. Treten Fehler (Bugs) auf, müssen wir diese suchen und beheben (Debugging).
5. **Wartung:** Auch nach der Fertigstellung wird Software weiter gepflegt. Sie wird an neue Betriebssysteme angepasst oder um neue Funktionen erweitert.

In diesem Buch werden wir uns vor allem auf die Phasen des Entwurfs, der Implementierung und des Testens konzentrieren.

1.4 Computer und Speicher

Um effizient programmieren zu können, müssen wir grob verstehen, wie unser Werkstück – der Computer – aufgebaut ist. Die meisten modernen Computer basieren auf der sogenannten Von-Neumann-Architektur. Für uns als Programmierer sind vor allem zwei Komponenten von entscheidender Bedeutung:

Der **Prozessor (CPU - Central Processing Unit):** Er ist der eigentliche Arbeiter in unserer Werkstatt. Er führt die Befehle unseres Programms aus, führt mathematische Berechnungen durch und fällt logische Entscheidungen. Er ist unfassbar schnell, hat aber ein extrem schlechtes Gedächtnis.

Der **Hauptspeicher (RAM - Random Access Memory):** Er ist die Werkbank des Prozessors. Hier werden unser laufendes Programm und alle Daten, die das Programm gerade verarbeitet (unsere Variablen), abgelegt. Der Hauptspeicher ist sehr schnell,

aber flüchtig: Sobald der Strom abgeschaltet wird, ist alles, was auf der Werkbank lag, unwiederbringlich verloren.

Hinweis: Daten, die dauerhaft gespeichert werden sollen, müssen auf der Festplatte (dem Lagerraum) abgelegt werden.

Wenn wir in unseren Programmen später "Variablen" anlegen, reservieren wir uns faktisch kleine Bereiche auf dieser Werkbank (im RAM), in die wir Zahlen, Buchstaben oder Wahrheitswerte ablegen und jederzeit wieder abrufen können.

1.5 Laufzeitsystem und Compiler

Wie wir bereits in Abschnitt 1.2 gelernt haben, schreiben wir unseren Code in einer Hochsprache wie Java, aber der Prozessor versteht nur Maschinencode (Nullen und Einsen). Wie kommt unser Text nun zum Prozessor?

Hier kommen zwei unverzichtbare Helfer ins Spiel:

Der **Compiler** (Übersetzer) ist ein spezielles Programm, das unseren geschriebenen Quellcode einliest. Er prüft zunächst, ob wir uns penibel an alle Grammatikregeln (Syntax) der Programmiersprache gehalten haben. Fehlt auch nur ein einziges Semikolon, bricht der Compiler mit einer Fehlermeldung ab. Ist der Code fehlerfrei, übersetzt der Compiler unseren Text in ein Format, das näher an der Maschinensprache liegt.

Das **Laufzeitsystem** (Runtime Environment) ist die Umgebung, in der unser übersetztes Programm dann tatsächlich ausgeführt wird. Das Laufzeitsystem nimmt das vom Compiler übersetzte Programm, reicht die Instruktionen Schritt für Schritt an den Prozessor weiter und verwaltet den Speicher (unsere Werkbank) für uns.

Wie genau dieser Prozess in Java abläuft – und warum Java hier einen ganz besonderen, genialen Weg geht – werden wir im nächsten Kapitel detailliert betrachten.

1.6 Entwicklungswerkzeuge

Ein Handwerker ist nur so gut wie sein Werkzeug. Um Programme schreiben, übersetzen und ausführen zu können, benötigen wir eine grundlegende Ausstattung:

- **Editor:** Ein reines Textverarbeitungsprogramm, in dem wir unseren Quellcode schreiben. Im Gegensatz zu Microsoft Word formatiert ein Code-Editor den Text nicht fett oder kursiv, sondern hilft uns durch farbliche Hervorhebungen von

Schlüsselwörtern (Syntax Highlighting).
- **Compiler:** Das bereits erwähnte Übersetzungswerkzeug.
- **Debugger:** Eine Art digitale Lupe. Mit einem Debugger können wir unser Programm im Zeitlupe ausführen – Schritt für Schritt. Wir können jederzeit anhalten und in den Speicher (RAM) schauen, um herauszufinden, warum unser Programm sich nicht so verhält, wie wir es erwarten.

Früher mussten Programmierer all diese Werkzeuge einzeln aufrufen. Heute nutzen wir sogenannte **Integrierte Entwicklungsumgebungen** (IDE – Integrated Development Environment). Eine IDE ist das ultimative Schweizer Taschenmesser des Programmierers. Sie vereint Editor, Compiler und Debugger in einer einzigen, komfortablen Benutzeroberfläche. Sie weist uns schon beim Tippen auf Fehler hin, ergänzt automatisch fehlende Klammern und nimmt uns viel lästige Routinearbeit ab.

Im kommenden Kapitel werden wir unsere Werkzeuge installieren, unsere Entwicklungsumgebung einrichten und endlich unser allererstes Java-Programm schreiben!

2 Java

Nachdem wir im ersten Kapitel unsere Werkstatt gedanklich eingerichtet und die theoretischen Grundlagen des Programmierens kennengelernt haben, wird es nun Zeit, unser Hauptwerkzeug auszuwählen und kennenzulernen. In diesem Buch ist unser Werkzeug der Wahl die Programmiersprache Java.

Warum gerade Java? Warum nicht C++, Python oder C#? In diesem Kapitel werfen wir einen kurzen Blick auf die Geschichte und die besonderen Eigenschaften von Java. Wir klären die wichtigsten Fachbegriffe, richten unsere Arbeitsumgebung ein und schreiben – als krönenden Abschluss – unser allererstes funktionierendes Java-Programm.

2.1 Was ist Java? (Historie und Eigenschaften)

Java ist eine der weltweit am häufigsten verwendeten Programmiersprachen. Sie wurde Anfang der 1990er Jahre von einem Team um James Gosling bei der Firma Sun Microsystems (die heute zu Oracle gehört) entwickelt und 1995 offiziell veröffentlicht. Ursprünglich war Java kurioserweise dazu gedacht, Haushaltsgeräte wie interaktive Fernseher oder Kühlschränke zu programmieren. Der große Durchbruch kam jedoch mit dem aufkommenden World Wide Web, da Java es ermöglichte, kleine Programme (sogenannte Applets) sicher im Webbrowser auszuführen.

Heute ist Java aus der professionellen Softwareentwicklung nicht mehr wegzudenken. Von Android-Smartphone-Apps über Kassensysteme bis hin zu den gewaltigen Server-Anwendungen großer Banken und Online-Shops – überall verrichtet Java unauffällig, aber extrem zuverlässig seinen Dienst.

Das liegt vor allem an den herausragenden Eigenschaften der Sprache:

- **Plattformunabhängigkeit:** Das berühmte Motto von Java lautet *"Write Once, Run Anywhere"* (Schreibe es einmal, führe es überall aus). Ein Java-Programm, das Sie auf einem Windows-PC schreiben, läuft ohne Änderungen auch auf einem Apple Mac oder einem Linux-Server.
- **Objektorientierung:** Obwohl wir uns in diesem Buch fast ausschließlich mit dem imperativen (schrittweisen) Fundament beschäftigen, ist Java in seinem Herzen eine objektorientierte Sprache. Das bedeutet, dass sie darauf ausgelegt ist, große Probleme in gut handhabbare, logische Bausteine (Objekte) zu zerlegen.
- **Sicherheit und Robustheit:** Java nimmt dem Programmierer viele fehleranfällige Aufgaben ab. Die Sprache kümmert sich beispielsweise automatisch um die Speicherverwaltung (Garbage Collection). Wir müssen nicht selbst aufräumen,

wenn wir Daten im Arbeitsspeicher nicht mehr benötigen – Java erledigt das für uns.

2.2 Zentrale Begriffe und Arbeitsweise

Erinnern Sie sich an Kapitel 1, als wir über Compiler und Laufzeitsysteme gesprochen haben? Hier wählt Java einen genialen Mittelweg, um seine Plattformunabhängigkeit zu erreichen. Um das zu verstehen, müssen wir drei zentrale Begriffe klären:

1. JDK (Java Development Kit)

Das JDK ist der Werkzeugkasten für uns Programmierer. Wenn wir Java-Programme *schreiben* wollen, müssen wir das JDK installieren. Es enthält den Compiler und viele weitere nützliche Hilfsprogramme.

2. JRE (Java Runtime Environment)

Das JRE ist das Laufzeitsystem. Wenn jemand ein fertiges Java-Programm nur *ausführen* (also nutzen) möchte, benötigt er lediglich das JRE. Es enthält die eigentliche "Maschine", die das Programm zum Leben erweckt.

3. JVM (Java Virtual Machine)

Die JVM ist das Herzstück des JRE. Sie ist eine Art "simulierter Computer im Computer".

Die Arbeitsweise von Java (Der Zweistufen-Prozess):

Wenn wir ein Programm in C++ schreiben, übersetzt der Compiler den Quellcode direkt in die Maschinensprache (Nullen und Einsen) des jeweiligen Betriebssystems (z.B. Windows). Versuchen wir, dieses Programm auf einem Mac auszuführen, versteht der Mac die Windows-Maschinensprache nicht.

Java löst dieses Problem in zwei Stufen:

1. **Das Kompilieren:** Wir schreiben unseren Quellcode in einer Textdatei (mit der Endung .java). Der Java-Compiler übersetzt diesen Text nun *nicht* in Maschinensprache, sondern in einen Zwischencode, den sogenannten **Bytecode**. Dieser Bytecode wird in einer neuen Datei (mit der Endung .class) gespeichert.
2. **Das Ausführen:** Die JVM (Java Virtual Machine) liest diesen Bytecode ein. Es gibt für jedes Betriebssystem (Windows, Mac, Linux) eine eigene, speziell angepasste JVM. Die JVM verhält sich wie ein Dolmetscher: Sie liest den universellen Bytecode und übersetzt ihn blitzschnell, in genau dem Moment, in dem das Programm

ausgeführt wird, in die Maschinensprache des jeweiligen Computers.

Deshalb müssen wir unser Programm nur einmal schreiben und kompilieren. Der Bytecode läuft überall dort, wo eine JVM installiert ist!

2.3 Installation der Werkzeuge

Bevor wir loslegen können, müssen wir unsere Werkstatt einrichten. Da es im Internet zahllose ausführliche und stets aktualisierte Anleitungen hierzu gibt, beschränken wir uns hier auf das Wesentliche.

Sie benötigen das aktuelle **JDK (Java Development Kit)**. Da Java Open Source ist, gibt es verschiedene Anbieter, die kostenlose JDKs zur Verfügung stellen (z.B. Adoptium/Eclipse Temurin, Amazon Corretto oder das Oracle JDK).

Laden Sie sich ein aktuelles JDK für Ihr Betriebssystem herunter und folgen Sie den Installationsanweisungen.

2.4 Entwicklungsumgebungen (IDE)

Wir könnten unsere Java-Programme theoretisch mit dem einfachen Windows-Editor schreiben und über eine unübersichtliche Kommandozeile (die schwarze Konsole) kompilieren. Das wäre so, als würden wir einen Baumstamm mit einem Taschenmesser fällen wollen. Es geht, aber es ist mühsam und schmerzhaft.

Profis nutzen **Integrierte Entwicklungsumgebungen (IDE - Integrated Development Environment)**. Eine IDE ist eine hochmoderne, voll ausgestattete Werkbank. Sie bietet uns:

- Einen intelligenten Texteditor, der Java-Befehle farbig markiert.
- Eine automatische Fehlererkennung schon während des Tippens (vergleichbar mit der Rechtschreibprüfung in Word).
- Einen eingebauten Compiler, der per Knopfdruck unser Programm übersetzt und startet.

Die drei beliebtesten, kostenlosen IDEs für Java sind derzeit:

- **Eclipse**
- **IntelliJ IDEA (Community Edition)**
- **Visual Studio Code (mit Java-Erweiterungen)**

Es spielt für den Verlauf dieses Buches keine Rolle, für welche IDE Sie sich entscheiden.

Installieren Sie eine dieser Umgebungen, öffnen Sie sie und erstellen Sie ein "Neues Java Projekt".

2.5 Programmerstellung und Java-Beispielprogramm

Nun ist es so weit! Wir schreiben unser allererstes Java-Programm. Es ist eine jahrzehntealte Tradition unter Programmierern, dass das erste Programm in einer neuen Sprache "Hello World" (Hallo Welt) heißt. Seine einzige Aufgabe ist es, diesen kurzen Gruß auf dem Bildschirm auszugeben.

Tippen Sie den folgenden Code exakt so in Ihre IDE ab. Achten Sie penibel auf Groß- und Kleinschreibung (Java unterscheidet zwischen System und system!). Vergessen Sie keine der Klammern oder das Semikolon am Ende. Die jeweilige Anzahl an Leerzeichen an den entsprechenden Stellen ist übrigens beliebig.

```
public class HelloWorld {

    public static void main(String[] args) {

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

    }

}
```

Wenn Sie nun in Ihrer IDE auf den Ausführen-Knopf (oft ein grünes "Play"-Symbol) drücken, übernimmt die IDE die Arbeit. Sie ruft heimlich den Compiler auf, erzeugt den Bytecode, startet die JVM und präsentiert Ihnen im Ausgabefenster (der Konsole) das magische Ergebnis:

```
Hallo Welt!
```

Gratulation! Sie haben soeben Ihr erstes funktionierendes Computerprogramm geschrieben. Aber was genau haben wir da eigentlich getippt? Auch wenn wir viele der Konzepte erst in späteren Kapiteln im Detail verstehen werden, wollen wir den Code kurz sezieren:

- public class HelloWorld { ... }: In Java muss *alles*, was wir schreiben, innerhalb einer sogenannten "Klasse" (class) stehen. Die Klasse ist quasi der äußere Rahmen unseres Bauplans. Wir haben unseren Bauplan HelloWorld genannt. Die geschweiften Klammern { und } markieren den Anfang und das Ende dieses Bauplans.

- public static void main(String[] args) { ... }: Das ist der wichtigste Satz für die Ausführung. Diese Zeile definiert die sogenannte **Hauptmethode** (Main-Methode). Wenn die Java Virtual Machine gestartet wird, sucht sie immer exakt nach diesem Satz. Die Main-Methode ist das Eingangstor, der Startpunkt unseres Programms. Hier beginnt Java mit der schrittweisen (imperativen) Abarbeitung unserer Befehle.
- System.out.println("Hallo Welt!");: Das ist unsere erste echte Arbeitsanweisung! Wir befehlen dem System (System), genauer gesagt dem Standard-Ausgabekanal (out), eine Zeile Text zu drucken (println, kurz für *print line*). Der Text, der gedruckt werden soll, steht in Anführungszeichen innerhalb der runden Klammern.
- **Das Semikolon (;):** Beachten Sie das winzige Zeichen am Ende der Anweisung. In menschlichen Sprachen beenden wir einen Satz mit einem Punkt. In Java beenden wir *jede* imperative Anweisung mit einem Semikolon. Es sagt dem Compiler: "Hier ist der Befehl zu Ende, jetzt kommt der nächste."

Wir haben erfolgreich ein Fundament gegossen und unsere Werkzeuge getestet. Ab dem nächsten Kapitel (Einführung) werden wir dieses Gerüst mit Leben füllen, mit echten Daten arbeiten und dem Computer beibringen, eigenständig zu rechnen und zu entscheiden!

Teil II: Erste Schritte und Materialien (Daten und Ausdrücke)

3 Einführung in die Programmierung mit Java

Nachdem wir in Kapitel 2 unsere Werkstatt eingerichtet und unser erstes kleines „Hallo Welt“-Schild aufgehängt haben, wird es nun Zeit, richtig loszulegen. In diesem Kapitel unternehmen wir einen ersten Streifzug durch die wichtigsten Konzepte von Java.

Stellen Sie sich dieses Kapitel wie einen Rundgang durch Ihre neue Werkstatt vor: Wir schauen uns an, wie wir Materialien lagern (Variablen), wie wir sie bearbeiten (Ausdrücke), wie wir Entscheidungen treffen (Bedingungen) und wie wir monotone Arbeiten automatisieren (Schleifen). Damit wir uns voll auf die Konzepte konzentrieren können, arbeiten wir in diesem Kapitel ausschließlich mit zwei grundlegenden Daten-Materialien: ganzen Zahlen (wie 5, -10 oder 42) und Wahrheitswerten (wahr oder falsch).

3.1 Grundlagen, Ausdrücke und Variablen

Wenn unser Programm rechnen soll, muss es sich Zahlen merken können. Wie wir in Kapitel 1 gelernt haben, nutzt der Computer dafür seinen Arbeitsspeicher (RAM). In Java können wir uns kleine Bereiche in diesem Speicher reservieren und ihnen Namen geben. Diese benannten Speicherplätze nennen wir **Variablen**.

Stellen Sie sich eine Variable wie eine beschriftete Kiste vor. Auf der Kiste steht der Name, und in der Kiste liegt der Wert. Da Java eine sehr ordentliche Sprache ist, müssen wir auf jede Kiste auch schreiben, *welche Art* von Dingen hineindarf (der sogenannte Datentyp). Für ganze Zahlen nutzen wir den Datentyp int (kurz für *integer*, englisch für ganze Zahl).

Ein **Ausdruck** ist wiederum eine Rechnung, die ein Ergebnis liefert (z. B. 5 + 3).

Schauen wir uns unser erstes Code-Beispiel an:

```
public class VariablenBeispiel {
    public static void main(String[] args) {
        // 1. Deklaration (Wir bauen eine Kiste für ganze Zahlen)
        int alter;

        // 2. Zuweisung (Wir legen den Wert 25 in die Kiste)
        alter = 25;

        // 3. Initialisierung (Deklaration und Zuweisung in einem Schritt)
        int verdoppelt = alter * 2;

        // 4. Ausgabe des Ergebnisses
        System.out.println(verdoppelt);
```

```
    }
}
```

Was passiert hier Schritt für Schritt?

- Zunächst sehen wir Zeilen, die mit // beginnen. Das sind **Kommentare**. Java ignoriert diese Zeilen komplett. Sie dienen nur uns Menschen als Notizzettel im Code.
- int alter;: Wir sagen Java: "Reserviere einen Speicherplatz für eine ganze Zahl und nenne ihn alter." (Das nennt man Deklaration).
- alter = 25;: Das Gleichheitszeichen ist in Java der **Zuweisungsoperator**. Es bedeutet nicht "ist gleich" wie in der Mathematik, sondern "nimm den Wert auf der rechten Seite und lege ihn in die Variable auf der linken Seite".
- int verdoppelt = alter * 2;: Hier kombinieren wir alles. Wir bauen eine neue Kiste namens verdoppelt. Auf der rechten Seite werten wir einen Ausdruck aus: Wir nehmen den aktuellen Wert aus alter (25), multiplizieren ihn mit 2 (das * ist das Mal-Zeichen) und legen das Ergebnis (50) in die neue Kiste.
- System.out.println(verdoppelt);: Gibt die Zahl 50 auf dem Bildschirm aus.

3.2 Boolesche Ausdrücke

Neben Zahlen müssen Programme oft Wahrheitswerte verarbeiten. Ein Türschloss ist entweder *offen* oder *geschlossen*. Ein Nutzer ist *eingeloggt* oder *nicht eingeloggt*. Für solche Ja/Nein-Zustände gibt es den Datentyp boolean (benannt nach dem Mathematiker George Boole).

Eine boolean-Variable kann exakt zwei Zustände annehmen: true (wahr) oder false (falsch).

Wir können solche Wahrheitswerte direkt zuweisen oder durch Vergleiche (sogenannte **boolesche Ausdrücke**) ermitteln lassen. Dafür nutzen wir Vergleichsoperatoren wie < (kleiner), > (größer) oder == (prüft auf Gleichheit – Achtung: zwei Gleichheitszeichen!).

```
public class BooleanBeispiel {
    public static void main(String[] args) {
        int meinAlter = 20;
        int mindestAlter = 18;

        // Ein direkter Wahrheitswert
        boolean hatTicket = true;

        // Ein boolescher Ausdruck: Ist meinAlter größer oder
        // gleich mindestAlter?
        boolean darfEintreten = meinAlter >= mindestAlter;
```

```
        System.out.println("Hat ein Ticket: ");
        System.out.println(hatTicket);

        System.out.println("Darf eintreten: ");
        System.out.println(darfEintreten);
    }
}
```

Hier rechnet Java in der Zeile meinAlter >= mindestAlter nicht im mathematischen Sinne, sondern beantwortet eine logische Frage: "Ist 20 größer oder gleich 18?". Die Antwort ist "Ja", also wird der Wert true in der Variablen darfEintreten gespeichert.

3.3 Ein- und Ausgabe (Scanner und System.out)

Bisher waren unsere Programme stumm und unflexibel. Die Werte waren fest im Code verankert ("hartcodiert"). Ein richtiges Programm muss aber mit dem Benutzer interagieren können.

Die Ausgabe kennen Sie bereits: System.out.println() druckt Text oder Variableninhalte auf die Konsole und macht danach einen Zeilenumbruch.

Für die Eingabe nutzen wir in Java ein spezielles Werkzeug namens Scanner. Es "scannt" das, was der Benutzer auf der Tastatur eintippt.

```
// Wir müssen den Scanner aus der Java-Bibliothek importieren
import java.util.Scanner;

public class EingabeBeispiel {
    public static void main(String[] args) {
        // Wir bereiten den Scanner vor, um von der
        // Standardeingabe (Tastatur) zu lesen
        Scanner tastatur = new Scanner(System.in);

        System.out.println("Bitte geben Sie Ihr Geburtsjahr ein:");

        // Das Programm wartet hier, bis der Nutzer eine Zahl eintippt und
        // Enter drückt!
        int geburtsjahr = tastatur.nextInt();

        int aktuellesJahr = 2024;
        int alter = aktuellesJahr - geburtsjahr;

        System.out.println("Sie sind (oder werden) dieses Jahr so alt:");
        System.out.println(alter);
    }
```

```
}
```

Mit tastatur.nextInt() stoppt das Programm vorübergehend. Es wartet geduldig, bis Sie in der unteren schwarzen Konsole Ihrer IDE eine ganze Zahl (z.B. 1990) eingeben und mit der Enter-Taste bestätigen. Erst dann fließt das Programm weiter.

3.4 Bedingte Anweisung und Alternativanweisung

Ein Programm, das immer nur stur von oben nach unten abläuft, ist nicht sehr intelligent. Wir wollen, dass unser Code auf unterschiedliche Situationen unterschiedlich reagiert. Dafür nutzen wir die if-Anweisung (Wenn-Dann-Bedingung).

Wir prüfen in den runden Klammern eine Bedingung (boolescher Ausdruck). Ist diese true, wird der nachfolgende Codeblock ausgeführt. Optional können wir mit else (Ansonsten) eine Alternative definieren, falls die Bedingung false ist.

```
import java.util.Scanner;

public class BedingungBeispiel {
    public static void main(String[] args) {
        Scanner tastatur = new Scanner(System.in);
        System.out.println("Wie alt bist du?");
        int alter = tastatur.nextInt();

        // Die if-Anweisung prüft die Bedingung
        if (alter >= 18) {
            // Wird nur ausgeführt, wenn alter >= 18 (true) ist
            System.out.println("Du bist volljährig.");
            System.out.println("Du darfst diesen Vertrag unterschreiben.");
        } else {
            // Wird nur ausgeführt, wenn alter >= 18 (false) ist
            System.out.println("Du bist leider noch minderjährig.");
            int fehlendeJahre = 18 - alter;
            System.out.println("Komm wieder in so vielen Jahren: ");
            System.out.println(fehlendeJahre);
        }
    }
}
```

Beachten Sie die geschweiften Klammern { und }. Sie klammern mehrere Anweisungen zu einem **Block** zusammen. Java weiß so genau, welche Befehle zum "Wenn"-Teil und welche zum "Ansonsten"-Teil gehören.

3.5 Wiederholungsanweisung und Blockanweisung

Oft stehen wir vor dem Problem, dass wir eine bestimmte Aktion mehrfach ausführen wollen. Anstatt denselben Code zehnmal untereinander zu kopieren, nutzen wir Schleifen. Die einfachste Schleife in Java ist die while-Schleife (Solange-Schleife).

Sie funktioniert wie ein if, aber mit einem entscheidenden Unterschied: Wenn der Block abgearbeitet ist, springt Java wieder nach oben und prüft die Bedingung erneut! Das geht so lange, bis die Bedingung irgendwann false wird.

Lassen Sie uns einen simplen Countdown programmieren:

```
public class SchleifenBeispiel {
    public static void main(String[] args) {
        int counter = 5;

        // Solange der counter größer als 0 ist, wiederhole den Block:
        while (counter > 0) {
            System.out.println("Countdown: ");
            System.out.println(counter);

            // Ganz wichtig: Wir müssen den Wert verändern,
            // sonst läuft die Schleife für immer!
            counter = counter - 1;
        }

        System.out.println("Start!");
    }
}
```

Wenn Sie das Programm starten, zählt es 5, 4, 3, 2, 1 und gibt dann "Start!" aus. Der Block { ... } hält auch hier die Anweisungen zusammen, die bei jedem Schleifendurchlauf wiederholt werden sollen.

3.6 Fehler finden und Testen

Programmieren ist Handwerk, und wo gehobelt wird, da fallen Späne. Sie werden Fehler machen. Das ist völlig normal und gehört zum Alltag jedes noch so erfahrenen Entwicklers. Wir unterscheiden grundsätzlich drei Arten von Fehlern:

1. Syntaxfehler (Grammatikfehler):

Wenn Sie sich nicht an die strikten Regeln von Java halten, verweigert der Compiler die Arbeit. Ein vergessenes Semikolon (;), eine fehlende Klammer (}) oder ein Tippfehler

(Sustem.out statt System.out) führen zu einem roten Fehler in Ihrer IDE. Das Programm lässt sich erst gar nicht starten. Diese Fehler sind leicht zu beheben, da die IDE Ihnen meist genau zeigt, in welcher Zeile das Problem liegt.

2. Semantische Fehler (Logikfehler):

Diese sind tückischer. Das Programm ist syntaktisch völlig korrekt, es lässt sich kompilieren und ausführen. Aber es tut nicht das, was Sie sich vorgestellt haben.

Betrachten wir folgendes (fehlerhaftes) Programm, das eigentlich von 1 bis 3 zählen sollte:

```
public class FehlerBeispiel {
    public static void main(String[] args) {
        int zahl = 1;

        // ACHTUNG: Logikfehler in der nächsten Zeile!
        while (zahl > 3) {
            System.out.println(zahl);
            zahl = zahl + 1;
        }

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

Wenn Sie dieses Programm ausführen, wird es sofort "Fertig." auf dem Bildschirm ausgeben. Die Schleife wird kein einziges Mal ausgeführt. Warum?

Weil die Bedingung zahl > 3 beim ersten Prüfen (ist 1 größer als 3?) direkt false ergibt! Der korrekte Ausdruck hätte zahl <= 3 (kleiner oder gleich) lauten müssen.

3. Laufzeitfehler (Runtime Errors / Exceptions):

Diese Fehlerkategorie bildet die dramatischste Form des Scheiterns. Ihr Programm ist grammatikalisch korrekt (kein Syntaxfehler) und die generelle Logik mag für den Normalfall auch stimmen. Das Programm lässt sich problemlos starten. Doch plötzlich, mitten in der Ausführung (zur Laufzeit), stößt der Computer auf eine Situation, die er unmöglich verarbeiten kann.

Stellen Sie sich vor, Sie haben eine Kreissäge nach Anleitung perfekt zusammengebaut und sie soll Holz schneiden. Doch plötzlich halten Sie versehentlich einen massiven Stahlträger in das Sägeblatt. Die Maschine blockiert sofort. In Java führt eine solche

Situation dazu, dass das Programm in Panik gerät und mit einer roten Fehlermeldung (einer sogenannten *Exception* oder Ausnahme) abstürzt.

Ein klassisches Beispiel ist die mathematisch unmögliche Division durch Null:

```
public class LaufzeitfehlerBeispiel {
    public static void main(String[] args) {
        int anzahlKekse = 12;
        int anzahlKinder = 0; // Hier liegt das Problem!

        System.out.println("Wir bereiten die Verteilung vor...");

        // ACHTUNG: Laufzeitfehler!
        // Der Computer versucht durch 0 zu teilen und stürzt sofort ab!
        int kekseProKind = anzahlKekse / anzahlKinder;

        // Diese Zeile wird NIEMALS erreicht:
        System.out.println("Jedes Kind bekommt: " + kekseProKind);
    }
}
```

Wenn Sie dieses Programm ausführen, druckt es noch "Wir bereiten die Verteilung vor...". In der nächsten Zeile reißt der Faden jedoch abrupt ab. Die Konsole spuckt eine Fehlermeldung wie java.lang.ArithmeticException: / by zero aus.

Wie wir solche Abstürze elegant abfangen, indem wir ein Sicherheitsnetz spannen, lernen wir im zweiten Buch (Objektorientierung). Für den Moment gilt: Wenn Ihr Programm plötzlich rot aufleuchtet und abbricht, lesen Sie die Fehlermeldung! Sie sagt Ihnen meist exakt, in welcher Zeile der Absturz passierte.

Der Schreibtischtest

Egal welche Art von Fehler Sie suchen: Solche Fehler findet man oft nur durch sorgfältiges Testen. Spielen Sie beim Testen gedanklich selbst den Computer. Gehen Sie den Code Zeile für Zeile mit dem Finger durch, notieren Sie sich die aktuellen Werte der Variablen auf einem Blatt Papier und prüfen Sie, ob die Bedingungen wahr oder falsch sind. Diesen extrem hilfreichen Prozess nennt man auch "Schreibtischtest".

4 Datentypen

Im vorherigen Kapitel haben wir einen ersten, schnellen Rundgang durch unsere Werkstatt gemacht. Dabei haben wir uns auf zwei grundlegende Materialien beschränkt: ganze Zahlen (int) und Wahrheitswerte (boolean).

Wenn Sie jedoch ein Möbelstück bauen, reicht Ihnen nicht nur eine Sorte Holz. Sie brauchen dicke Balken für das Grundgerüst, feines Furnier für die Oberfläche und Metall für die Scharniere. Genauso verhält es sich in der Programmierung: Unterschiedliche Arten von Informationen erfordern unterschiedliche Materialien. In diesem Kapitel schauen wir uns das komplette Materiallager von Java an. Wir lernen, wie der Computer Daten speichert, welche Datentypen es gibt und wie wir sie mit Operatoren bearbeiten können.

4.1 Der Computer-Speicher

Erinnern Sie sich an unsere Metapher aus dem ersten Kapitel: Der Arbeitsspeicher (RAM) ist die Werkbank des Computers. Wenn wir eine Variable anlegen, reservieren wir Platz auf dieser Werkbank.

Der Speicher eines Computers ist intern in winzige Zellen unterteilt, die sogenannten **Bytes**. Ein Byte besteht wiederum aus 8 Bits (die nur die Werte 0 oder 1 annehmen können). Wenn wir in Java eine Variable anlegen, fragt uns der Computer im Grunde: *"Wie groß soll die Kiste sein, die ich für dich auf die Werkbank stellen soll?"*

Wenn wir nur die Zahl 5 speichern wollen, brauchen wir eine kleine Kiste. Wollen wir die Einwohnerzahl der Erde speichern, brauchen wir eine sehr große Kiste. Wenn wir versuchen, eine riesige Zahl in eine zu kleine Kiste zu quetschen, läuft sie über – das Programm liefert falsche Ergebnisse oder stürzt ab. Java zwingt uns daher, für jede Variable sofort bei der Erschaffung (Deklaration) den passenden Datentyp anzugeben. Dieser Typ bestimmt unverrückbar, wie groß die Kiste ist und was hineingelegt werden darf.

4.2 Einfache Datentypen

Java besitzt exakt acht sogenannte **primitive (einfache) Datentypen**. Sie sind die absoluten Grundbausteine der Sprache. Den Typ boolean (für true oder false) kennen Sie bereits. Die restlichen sieben Typen sind allesamt für Zahlen gedacht.

Ganzzahlige Datentypen:

- byte: Die kleinste Kiste (1 Byte groß). Speichert Zahlen von -128 bis 127.
- short: Eine kleine Kiste (2 Bytes groß). Speichert Zahlen von -32.768 bis 32.767.
- int: Der Standard-Typ für ganze Zahlen (4 Bytes groß). Reicht bis ca. 2 Milliarden.
- long: Die Riesenkiste für ganze Zahlen (8 Bytes groß). Reicht für fast unvorstellbar große Zahlen.

Gleitkommazahlen (Kommazahlen):

- float: Für Kommazahlen mit normaler Präzision (4 Bytes).
- double: Für Kommazahlen mit doppelter Präzision (8 Bytes). In der Praxis nutzt man fast immer double, wenn man mit Brüchen oder Kommawerten rechnet.

Schauen wir uns an, wie wir diese Typen in Code gießen:

```
public class ZahlenBeispiel {
    public static void main(String[] args) {
        // Deklaration und Zuweisung verschiedener Ganzzahlen
        byte kleineZahl = 100;
        int standardZahl = 50000;

        // Bei sehr großen Zahlen für 'long' hängen wir ein 'L' an
        long weltBevoelkerung = 8000000000L;

        // Kommazahlen (Achtung: Im Englischen nutzt man den Punkt, kein
        // Komma!)
        double kontostand = 1542.75;

        // Bei 'float' müssen wir ein 'f' anhängen, um dem Compiler
        // zu sagen, dass es kein 'double' ist
        float temperatur = 36.6f;

        System.out.println("Weltbevölkerung: ");
        System.out.println(weltBevoelkerung);
        System.out.println("Temperatur: ");
        System.out.println(temperatur);
    }
}
```

Erklärung des Codes:

Sie sehen hier, dass Java bei der Schreibweise penibel ist. Wenn wir ein Programm schreiben, nutzt Java für Kommazahlen automatisch immer den Typ double. Wollen wir

explizit den kleineren Typ float verwenden, müssen wir die Zahl mit einem kleinen f markieren (36.6f). Genauso verhält es sich mit großen Zahlen beim Typ long, hier signalisieren wir das mit einem großen L.

4.3 Variablen und Literale

In der Programmierung unterscheiden wir strikt zwischen dem Behälter und dem Inhalt.

- Die **Variable** ist der benannte Behälter (z. B. alter).
- Das **Literal** ist der konkrete, fest in den Code geschriebene Wert (z. B. 25 oder true oder 3.14).

Literale sind die rohen Rohstoffe, die wir in unsere Variablen-Kisten füllen.

```
public class LiteralBeispiel {
    public static void main(String[] args) {
        // 'anzahl' ist die Variable, '12' ist das ganzzahlige Literal
        int anzahl = 12;

        // Wir können Variablen auch den Wert anderer Variablen zuweisen
        int kopie = anzahl;

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

Wenn Java die Zeile int kopie = anzahl; ausführt, nimmt es den Wert aus der Kiste anzahl heraus (die 12), fertigt eine Kopie davon an und legt diese Kopie in die Kiste kopie. Die ursprüngliche Variable anzahl wird dabei nicht geleert oder verändert!

4.4 Der Datentyp Character

Was ist, wenn wir keine Zahlen, sondern Buchstaben speichern wollen? Dafür gibt es den achten und letzten primitiven Datentyp: char (von engl. character = Schriftzeichen).

Ein char speichert exakt ein einziges Zeichen. Das kann ein Buchstabe, eine Ziffer oder ein Sonderzeichen sein. Um Java zu signalisieren, dass wir ein Zeichen-Literal meinen und nicht etwa den Namen einer Variablen, müssen wir das Zeichen in **einfache Anführungszeichen** (Apostrophe) setzen.

```
public class CharacterBeispiel {
    public static void main(String[] args) {
        // Ein einzelner Buchstabe
        char anfangsBuchstabe = 'A';
```

```
        // Auch eine Ziffer kann ein Zeichen sein
        char zifferAlsZeichen = '7';

        // Ein Sonderzeichen
        char paragraph = '§';

        System.out.println(anfangsBuchstabe);
        System.out.println(paragraph);
    }
}
```

Ein interessantes Detail am Rande: Unter der Haube ist der Computer nicht in der Lage, Buchstaben zu speichern. Er kennt nur Zahlen. Deshalb ist jedem Zeichen weltweit eine eindeutige Zahl zugeordnet (der sogenannte Unicode). Das große 'A' entspricht beispielsweise der Zahl 65. Der Datentyp char ist also insgeheim einfach nur eine Kiste für Zahlen, die Java bei der Ausgabe für uns automatisch wieder in Buchstaben zurückübersetzt.

4.5 Zeichenketten (String)

Ein einzelner Buchstabe reicht oft nicht aus. Wir wollen ganze Namen, Sätze oder Textblöcke speichern. Dafür bietet Java den Typ String (englisch für Zeichenkette).

Wichtige Anmerkung: String ist **kein** einfacher (primitiver) Datentyp wie int oder boolean. Er ist ein sogenannter *Referenzdatentyp*. Die tieferen Geheimnisse von Referenztypen lüften wir erst in Kapitel 10. Für den Moment behandeln wir String einfach so, als wäre er ein normaler Datentyp. Sie erkennen die Sonderrolle von String aber schon daran, dass er großgeschrieben wird, während die primitiven Typen (int, char, double) kleingeschrieben werden.

Text-Literale (String-Literale) werden in **doppelte Anführungszeichen** gesetzt.

```
public class StringBeispiel {
    public static void main(String[] args) {
        String vorname = "Anna";
        String nachname = "Müller";

        // Mit dem Plus-Operator (+) können wir Strings aneinanderketten
        // (Konkatenation)
        // Wir fügen hier ein Leerzeichen (" ") zwischen den Namen ein.
        String ganzerName = vorname + " " + nachname;

        System.out.println("Der Name lautet:");
        System.out.println(ganzerName);
    }
```

```
}
```

Wenn Sie den Plus-Operator + mit Strings verwenden, rechnet Java nicht mathematisch, sondern "klebt" die Texte einfach aneinander.

4.6 Ausdrücke und Operatoren

Materialien in Kisten zu lagern ist nützlich, aber erst wenn wir mit ihnen arbeiten, entsteht ein echtes Programm. Wir kombinieren Variablen und Literale mithilfe von **Operatoren** zu **Ausdrücken**.

Die wichtigsten arithmetischen (mathematischen) Operatoren sind:

- + (Addition)
- - (Subtraktion)
- * (Multiplikation)
- / (Division)
- % (Modulo - der Rest bei einer Division)

Hier gibt es zwei riesige Stolpersteine für Anfänger, die wir uns im Code genau ansehen müssen:

```
public class OperatorBeispiel {
    public static void main(String[] args) {
        int a = 10;
        int b = 3;

        // 1. Die klassische Division
        int ergebnisDivision = a / b;

        // 2. Der Modulo-Operator (Restwert)
        int ergebnisModulo = a % b;

        System.out.println("10 geteilt durch 3 ist:");
        System.out.println(ergebnisDivision); // Gibt 3 aus!

        System.out.println("Der Rest der Division ist:");
        System.out.println(ergebnisModulo);   // Gibt 1 aus!

        // 3. Fließkomma-Division
        double exakteDivision = 10.0 / 3.0;
        System.out.println("Exakte Division:");
        System.out.println(exakteDivision);   // Gibt 3.3333333333333335 aus
    }
}
```

Die Integer-Division (Die Falle):

Wenn Sie in Java zwei ganze Zahlen (int) durcheinander teilen, ist das Ergebnis *immer* wieder eine ganze Zahl. Java rundet dabei nicht auf oder ab, sondern schneidet die Nachkommastellen einfach gnadenlos ab! Deshalb ist 10 / 3 in Java exakt 3. Wenn Sie ein exaktes Ergebnis mit Kommastellen wollen, muss mindestens eine der beiden Zahlen ein double sein (wie im dritten Beispiel 10.0 / 3.0).

Der Modulo-Operator (%):

Dieser Operator ist in der Programmierung unglaublich nützlich. Er beantwortet die Frage aus der Grundschule: "Was bleibt als Rest übrig, wenn ich schriftlich teile?". 10 geteilt durch 3 ist 3, Rest 1. Der Ausdruck 10 % 3 liefert also genau diese 1. Wofür braucht man das? Sie können damit zum Beispiel kinderleicht prüfen, ob eine Zahl gerade ist: Wenn zahl % 2 exakt 0 ergibt, ist die Zahl gerade.

Damit haben wir unser Materiallager komplett ausgestattet! Sie kennen nun die primitiven Datentypen, können mit Texten (String) arbeiten und wissen, wie man Ausdrücke mathematisch korrekt formuliert und welche Fallen bei der Division lauern.

Teil III: Den Fluss steuern (Anweisungen)

5 Einfache Anweisungen und Sequenzen

In den bisherigen Kapiteln haben wir unsere Werkstatt eingerichtet und das Materiallager gefüllt. Sie wissen nun, wie Sie Variablen als Behälter für Zahlen (int, double), Zeichen (char) oder Wahrheitswerte (boolean) anlegen. Sie wissen auch, wie man diese Werte durch Ausdrücke (wie 5 + 3) miteinander verknüpft.

Doch ein Ausdruck allein baut noch keinen Schrank und führt auch noch kein Programm aus. Wenn Sie in Java einfach nur 5 + 3 in den Code schreiben, wird der Compiler sich beschweren. Warum? Weil es keine Anweisung ist. Es ist, als würden Sie in der Werkstatt stehen und laut "Holzbrett und Säge!" rufen. Niemand weiß, was Sie damit meinen. Sie müssen einen Befehl erteilen: "Säge das Holzbrett durch!"

In diesem Kapitel lernen wir das eigentliche Herzstück der imperativen Programmierung kennen: Die Anweisungen (englisch: *statements*) und ihre streng chronologische Abfolge (die Sequenz).

5.1 Grundlagen und einfache Anweisungen

Eine Anweisung ist ein vollständiger Arbeitsauftrag an den Computer. In der imperativen Programmierung arbeitet der Prozessor diese Aufträge exakt in der Reihenfolge ab, in der wir sie aufschreiben.

Das wichtigste Erkennungsmerkmal einer einfachen Anweisung in Java ist das **Semikolon (;)** am Ende. Das Semikolon ist für den Java-Compiler das, was der Punkt für das Ende eines deutschen Satzes ist. Es signalisiert: "Dieser Befehl ist jetzt vollständig, führe ihn aus."

Ein Ausdruck (x + 1) wird zu einer **Ausdrucksanweisung** (englisch: *expression statement*), wenn er etwas bewirkt und mit einem Semikolon abgeschlossen wird. Zu den erlaubten Ausdrucksanweisungen gehören Zuweisungen, das Erhöhen/Verringern von Werten (Inkrement/Dekrement) und Methodenaufrufe (wie die Ausgabe auf der Konsole).

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

        // Dies ist eine Ausdrucksanweisung (Methodenaufruf)
        System.out.println("Start");

        // Dies ist eine Ausdrucksanweisung (Zuweisung)
```

```
        x = x + 5;

        // FALSCH: Das Folgende ist nur ein Ausdruck, keine Anweisung!
        // Der Compiler würde hier einen Fehler werfen, wenn Sie das
        // Kommentarzeichen (//) entfernen:
        // x + 10;
    }
}
```

5.2 Deklarationsanweisung und Zuweisung

Zwei der wichtigsten einfachen Anweisungen haben wir bereits in den vorherigen Kapiteln intuitiv genutzt. Nun wollen wir sie formal korrekt einordnen.

Die **Deklarationsanweisung** teilt dem Compiler mit, dass wir einen neuen Speicherplatz (eine Variable) reservieren möchten. Sie besteht immer aus dem Datentyp, gefolgt vom gewünschten Namen der Variablen und dem obligatorischen Semikolon.

Die **Zuweisung** (englisch: *assignment*) ist die Anweisung, um einen Wert in diesen Speicherplatz zu legen. Sie nutzt den Zuweisungsoperator (=). Auf der linken Seite des Gleichheitszeichens muss immer zwingend der Name der Variablen stehen, auf der rechten Seite der Wert (oder ein Ausdruck, der einen Wert berechnet).

```
public class ZuweisungBeispiel {
    public static void main(String[] args) {
        // 1. Reine Deklarationsanweisungen
        int laenge;
        int breite;
        int flaeche;

        // 2. Reine Zuweisungsanweisungen
        laenge = 5;
        breite = 3;

        // Zuweisung mit einem Ausdruck auf der rechten Seite
        flaeche = laenge * breite;

        // 3. Kombinierte Deklarations- und Zuweisungsanweisung
        // (Initialisierung)
        int umfang = (laenge + breite) * 2;

        System.out.println("Die Fläche beträgt: ");
        System.out.println(flaeche);

        System.out.println("Der Umfang beträgt: ");
        System.out.println(umfang);
```

```
    }
}
```

Erklärung:

Bei der Zeile flaeche = laenge * breite; passiert im Computer Folgendes: Zuerst wird die *rechte* Seite betrachtet. Der Computer schaut in der Kiste laenge nach (findet eine 5), schaut in der Kiste breite nach (findet eine 3), multipliziert beide (ergibt 15) und packt erst ganz am Ende das fertige Ergebnis in die Kiste auf der *linken* Seite (flaeche).

5.3 Leeranweisung und Blockanweisung

Manchmal gibt es Situationen, in denen die Grammatik von Java eine Anweisung verlangt, wir aber eigentlich gar nichts tun wollen. Dafür gibt es die **Leeranweisung** (englisch: *empty statement*). Sie besteht aus nichts anderem als einem einsamen Semikolon.

In der Praxis ist eine Leeranweisung jedoch sehr oft das Resultat eines gefährlichen Tippfehlers, den wir uns später bei den Schleifen (Kapitel 7) noch genau ansehen werden.

Das genaue Gegenteil der Leeranweisung ist die **Blockanweisung**. Wir haben sie ebenfalls schon flüchtig kennengelernt. Wenn wir mehrere einzelne Anweisungen in geschweifte Klammern { und } einfassen, verschmelzen sie für den Compiler zu einer einzigen, großen Anweisung – einem Block.

Blöcke sind extrem wichtig für die Strukturierung, aber sie haben eine weitreichende Konsequenz: den **Gültigkeitsbereich (Scope)** von Variablen.

```
public class BlockBeispiel {
    public static void main(String[] args) {
        // Leeranweisungen (bewirken absolut gar nichts)
        ;
        ;

        int aussen = 100;

        // Wir öffnen eine Blockanweisung
        {
            int innen = 50;
            System.out.println("Wir sind im Block!");
            System.out.println(aussen); // Zugriff auf 'aussen' ist erlaubt
            System.out.println(innen);  // Zugriff auf 'innen' ist erlaubt
```

```
            // Die Variable 'aussen' kann hier sogar verändert werden
            aussen = aussen + 1;
        } // Hier endet der Block

        System.out.println("Wir haben den Block verlassen.");
        System.out.println(aussen); // Wird 101 ausgeben

        // FALSCH: Der folgende Code würde einen Compiler-Fehler werfen!
        // Die Variable 'innen' hat am Ende des Blocks aufgehört zu
        // existieren.
        // System.out.println(innen);
    }
}
```

Die goldene Regel des Gültigkeitsbereichs:

Eine Variable, die innerhalb eines Blocks deklariert wird (wie innen), "lebt" nur so lange, bis die schließende Klammer } des Blocks erreicht wird. Danach löscht Java die Variable sofort aus dem Speicher. Von außen kann man also niemals in einen Block hineingreifen. Ein Block kann jedoch sehr wohl auf Variablen zugreifen, die *vor* ihm in einem übergeordneten Bereich (wie aussen) definiert wurden.

5.4 Sequenzen von Anweisungen

Wir kommen nun zum Herzschlag der imperativen Programmierung: der Sequenz. Eine Sequenz ist schlichtweg die Hintereinanderausführung von Anweisungen.

Der Computer beginnt bei der ersten Anweisung in unserer Main-Methode, führt sie aus, geht zur nächsten, führt diese aus, und so weiter. Er überspringt nichts und er tauscht keine Zeilen. Dieser Determinismus (die absolute Vorhersagbarkeit) ist die wichtigste Eigenschaft unseres Handwerks.

Um zu demonstrieren, wie essenziell die chronologische Reihenfolge der Anweisungen ist, schauen wir uns ein klassisches Problem der Programmierung an: Das **Vertauschen von zwei Werten** (Swap).

Stellen Sie sich vor, Sie haben ein Glas mit roter Farbe (Variable A) und ein Glas mit blauer Farbe (Variable B). Sie möchten die Inhalte tauschen. Wenn Sie einfach die rote Farbe in das blaue Glas kippen, vermischt sich alles, und der ursprüngliche blaue Wert ist für immer verloren. Was brauchen Sie in der echten Welt? Ein drittes, leeres Hilfsglas! Exakt so lösen wir das Problem auch in Code:

```java
public class SequenzBeispiel {
    public static void main(String[] args) {
        int glasA = 99; // Die "rote" Farbe
        int glasB = 11; // Die "blaue" Farbe

        System.out.println("Vor dem Tausch:");
        System.out.println("Glas A: " + glasA); // Gibt 99 aus
        System.out.println("Glas B: " + glasB); // Gibt 11 aus

        // Sequenz zum Vertauschen der Werte:
        // Schritt 1: Wir kippen den Inhalt von A in ein leeres Hilfsglas
        int hilfsGlas = glasA;

        // Schritt 2: Jetzt können wir den Inhalt von B sicher in A umfüllen
        glasA = glasB;

        // Schritt 3: Zum Schluss füllen wir das Hilfsglas (altes A) in B
        glasB = hilfsGlas;

        System.out.println("Nach dem Tausch:");
        System.out.println("Glas A: " + glasA); // Gibt jetzt 11 aus
        System.out.println("Glas B: " + glasB); // Gibt jetzt 99 aus
    }
}
```

Warum die Reihenfolge alles ist:

Versuchen Sie gedanklich, in der Tausch-Sequenz die Schritte 2 und 3 umzudrehen oder wegzulassen. Der gesamte Algorithmus würde sofort in sich zusammenbrechen und falsche Ergebnisse liefern. In der imperativen Programmierung ist nicht nur wichtig, *was* Sie dem Computer befehlen, sondern exakt *wann* Sie es tun.

Wir beherrschen nun die Fähigkeit, Variablen anzulegen, Ausdrücke zu berechnen und strikt aufeinanderfolgende Arbeitsaufträge (Sequenzen) zu erteilen. Unser Programm fließt bisher wie ein gerader Kanal stur von oben nach unten.

In der Realität muss eine Werkstatt aber auf unterschiedliche Materialien reagieren können. Wenn das Holz hart ist, nutze eine andere Säge als bei weichem Holz.

Sind Sie bereit für **Kapitel 6: Verzweigungen (Bedingte Anweisungen)**, in dem wir unserem Programm beibringen, eigenständig Entscheidungen mit if, else und switch zu treffen?

6 Verzweigungen (Bedingte Anweisungen)

Bisher glichen unsere Programme einer Einbahnstraße ohne Abzweigungen: Der Computer hat jede einzelne Anweisung strikt von oben nach unten abgearbeitet. In einer echten Werkstatt funktioniert das jedoch nicht immer. Wenn Sie ein Stück Holz bearbeiten, müssen Sie Entscheidungen treffen: *Wenn* das Holz sehr hart ist, *dann* müssen Sie eine andere Säge verwenden, *ansonsten* reicht die normale Handsäge.

Damit ein Programm intelligent wirkt, muss es in der Lage sein, den Ablaufplan (den Fluss der Anweisungen) an bestimmte Bedingungen anzupassen. Wir nennen dies **Verzweigungen** oder auch **Kontrollstrukturen**. In diesem Kapitel lernen wir die drei wichtigsten Werkzeuge kennen, um den Ausführungsfluss in Java zu steuern: die if-Anweisung, die if-else-Anweisung und die switch-Anweisung.

6.1 Die if-Anweisung

Die if-Anweisung (zu Deutsch: Wenn-Anweisung) ist die einfachste Form der Verzweigung. Sie erlaubt es uns, einen bestimmten Block von Code *nur dann* auszuführen, wenn eine vorher definierte Bedingung erfüllt ist. Ist die Bedingung nicht erfüllt, ignoriert der Computer den Code-Block komplett und macht einfach danach weiter.

Die Grammatik einer if-Anweisung ist streng vorgegeben:

1. Das Schlüsselwort if.
2. Ein boolescher Ausdruck (eine Ja/Nein-Frage) in runden Klammern ().
3. Eine Anweisung oder (viel häufiger) ein Anweisungsblock in geschweiften Klammern {}.

Schauen wir uns das an einem Beispiel an. Wir bauen ein Warnsystem für eine Maschine, das nur Alarm schlägt, wenn die Temperatur zu hoch wird:

```
import java.util.Scanner;

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

        System.out.println("Bitte aktuelle Temperatur eingeben:");
        int temperatur = tastatur.nextInt();

        // Die if-Anweisung
        if (temperatur > 100) {
```

```
            // Dieser Block wird NUR betreten, wenn temperatur > 100 'true'
            //  ergibt
            System.out.println("ALARM!");
            System.out.println("Die Maschine ist überhitzt!");
            System.out.println("Kühlsystem wird aktiviert...");
        }

        // Diese Zeile wird IMMER ausgeführt, egal ob die if-Bedingung
        // zutraf oder nicht
        System.out.println("Programm wird fortgesetzt.");
    }
}
```

Wie der Computer vorgeht:

Wenn Sie das Programm starten und die Zahl 80 eingeben, prüft Java: Ist 80 > 100? Das Ergebnis ist false. Java überspringt den gesamten Block zwischen { und } und gibt direkt "Programm wird fortgesetzt." aus.

Geben Sie jedoch 110 ein, ergibt die Prüfung true. Java betritt den Block, druckt die drei Alarm-Zeilen auf die Konsole und macht erst danach mit dem restlichen Programm weiter.

6.2 Die if-else-Anweisung

Oft reicht ein einfaches "Wenn-Dann" nicht aus. Meistens haben wir eine Entweder-Oder-Situation. *Wenn* der Benutzer das richtige Passwort eingibt, *dann* gewähre ihm Zugang, *ansonsten* sperre ihn aus.

Für dieses "Ansonsten" bietet Java das Schlüsselwort else. Der else-Block fängt alle Fälle ab, bei denen die if-Bedingung false ergeben hat.

Darüber hinaus können wir mehrere Bedingungen hintereinanderschalten, indem wir else if verwenden. Damit bauen wir eine sogenannte Kaskade. In einer solchen Kaskade wird der Computer von oben nach unten jede Bedingung prüfen. Sobald die erste Bedingung true ist, wird der entsprechende Block ausgeführt und die restliche Kaskade sofort abgebrochen.

Hier ist ein Beispiel für eine Alterskontrolle im Kino:

```
import java.util.Scanner;

public class IfElseBeispiel {
```

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

        System.out.println("Wie alt bist du?");
        int alter = tastatur.nextInt();

        if (alter < 12) {
            // Wird ausgeführt für alle von 0 bis 11 Jahren
            System.out.println("Du darfst nur Kinderfilme sehen.");
        }
        else if (alter < 18) {
            // Wird ausgeführt, wenn alter NICHT < 12 ist (also ab 12),
            // aber < 18 ist (also 12 bis 17 Jahre)
            System.out.println("Du darfst Jugendfilme sehen.");
        }
        else {
            // Der Fallback: Wird ausgeführt, wenn KEINE der obigen
            // Bedingungen zutraf (also 18 Jahre und älter)
            System.out.println("Du bist volljährig. " +
                               "Viel Spaß im Actionfilm!");
        }

        System.out.println("Bitte gehen Sie in Saal 1.");
    }
}
```

Die Macht der Reihenfolge:

Bei einer else if-Kaskade ist die Reihenfolge der Fragen entscheidend. Hätten wir ganz oben gefragt if (alter < 18), dann hätte ein 10-Jähriger die Ausgabe "Du darfst Jugendfilme sehen" erhalten, weil die Bedingung 10 < 18 wahr ist. Die Kaskade wäre dort beendet gewesen, und die Überprüfung auf < 12 wäre niemals erreicht worden. Planen Sie Ihre Verzweigungen also immer sorgfältig!

6.3 Die switch-Anweisung

Wenn wir eine Variable haben und diese auf viele verschiedene, konkrete Werte prüfen wollen (z.B. bei der Auswahl in einem Menü), werden if-else-Kaskaden schnell sehr unübersichtlich. Dafür hat Java ein spezielles Werkzeug erfunden: die switch-Anweisung (zu Deutsch: Schalter-Anweisung).

Anstatt logische Ausdrücke (<, >, !=) zu prüfen, schauen wir uns bei switch den exakten Wert einer einzigen Variable an. Wir definieren verschiedene Fälle (englisch: case). Wenn der Wert der Variable mit einem case übereinstimmt, springt das Programm genau

dorthin.

Lassen Sie uns ein simples Werkzeug-Menü für unsere Werkstatt programmieren:

```
import java.util.Scanner;

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

        System.out.println("Wähle ein Werkzeug:");
        System.out.println("1 - Hammer");
        System.out.println("2 - Säge");
        System.out.println("3 - Bohrmaschine");

        int auswahl = tastatur.nextInt();

        // Wir prüfen den konkreten Wert der Variablen 'auswahl'
        switch (auswahl) {
            case 1:
                System.out.println("Du nimmst den Hammer.");
                System.out.println("Vorsicht auf die Finger!");
                // GANZ WICHTIG: Das break beendet den Fall!
                break;
            case 2:
                System.out.println("Du nimmst die Säge.");
                break;
            case 3:
                System.out.println("Du nimmst die Bohrmaschine.");
                break;
            default:
                // Der default-Block ist wie das 'else'. Er fängt alles ab,
                // was oben nicht definiert wurde (z.B. Eingabe von 4
                // oder 99).
                System.out.println("Ungültige Eingabe. " +
                                   "Dieses Werkzeug haben wir nicht.");
                break;
        }

        System.out.println("An die Arbeit!");
    }
}
```

Der tückische Fall-Through (Warum break so wichtig ist):

In der switch-Anweisung lauert eine der berühmtesten Fehlerquellen für Programmieranfänger. Was passiert, wenn wir das Schlüsselwort break; (zu Deutsch:

abbrechen/ausbrechen) weglassen?

Gibt der Nutzer im obigen Beispiel eine 1 ein, springt Java zu case 1: und führt den Code aus. Ohne das break am Ende würde Java danach einfach blind in den case 2: hineinlaufen und dessen Code ebenfalls ausführen, dann in case 3: und so weiter! Dieses "Durchfallen" nennt man im Fachjargon *Fall-Through*.

Das eiserne Gesetz der switch-Anweisung lautet daher: Beenden Sie jeden Fall zwingend mit einem break; (es sei denn, Sie wollen das Durchfallen ausnahmsweise als speziellen Trick nutzen, was wir Anfängern jedoch vorerst strengstens abraten).

Mit if, else und switch haben Sie nun die volle Kontrolle über den logischen Fluss Ihres Programms. Sie können Entscheidungen fällen, Fehler abfangen und Menüs bauen. Doch was tun wir, wenn wir den Code in einem bestimmten Block nicht nur einmal ausführen wollen, sondern zehnmal, hundertmal oder so lange, bis eine Aufgabe endgültig erledigt ist?

Hier ist eine deutlich kompaktere Version des Abschnitts, die direkt auf den Punkt kommt, aber alle wichtigen Konzepte (Pfeilsyntax, Vollständigkeit und yield) für den Java-Einsteiger verständlich erklärt:

6.4 Der moderne Switch-Ausdruck (Switch Expression)

Die klassische switch-Anweisung hat für uns Handwerker zwei große Nachteile: Wenn Sie auch nur ein einziges break vergessen, rutscht das Programm fälschlicherweise in den nächsten Block ab (der gefürchtete *Fall-through*). Zudem müssen wir oft mühsam in jedem case dieselbe Variable neu zuweisen.

Die moderne Lösung dafür ist der **Switch-Ausdruck**. Ein Ausdruck (Expression) liefert immer direkt ein Ergebnis zurück. Wir können das gesamte switch-Konstrukt also einfach auf die rechte Seite eines Gleichheitszeichens stellen!

```
int tagNummer = 3;

// Das gesamte Konstrukt berechnet einen Wert und speichert ihn in 'tagName'
String tagName = switch (tagNummer) {
    case 1 -> "Montag";
    case 2 -> "Dienstag";
    case 3 -> "Mittwoch";
    case 4, 5 -> "Donnerstag oder Freitag";
      // Mehrere Fälle kommagetrennt zusammenfassen
    default -> "Wochenende oder ungültig";
}; // WICHTIG: Hier am Ende muss nun ein Semikolon stehen!
```

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

Die wichtigsten architektonischen Neuerungen:

1. **Der Pfeil (->):** Er ersetzt den Doppelpunkt und macht das fehleranfällige break komplett überflüssig. Sobald ein Treffer gefunden und der Wert rechts vom Pfeil geliefert wurde, beendet sich das switch absolut sicher von selbst.
2. **Zwang zur Vollständigkeit:** Da die Variable tagName zwingend einen Wert erhalten muss, zwingt der Compiler uns dazu, alle möglichen Fälle abzudecken (meist durch einen default-Zweig). Vergessen wir ihn, weigert sich Java, das Programm zu übersetzen.

Komplexe Code-Blöcke mit yield

Wenn ein bestimmter Fall mehr Logik erfordert, als in eine einzige Zeile passt, können Sie nach dem Pfeil geschweifte Klammern { } öffnen. Um am Ende dieses Blocks den finalen Wert an die Variable auszuliefern, nutzen Sie das spezielle Schlüsselwort **yield** (nicht return!):

```
int tagNummer = 3;

String tagName = switch (tagNummer) {
    case 1 -> "Montag";
    case 3 -> {
        System.out.println("LOG: Die Mitte der Woche ist erreicht.");
        int berechnung = 10 * 5;

        // 'yield' feuert das Ergebnis aus dem Block heraus in die Variable!
        yield "Mittwoch (Wert: " + berechnung + ")";
    }
    default -> "Ein anderer Tag";
};
```

Faustregel für die Praxis: Wenn Sie mit einem switch einen Wert berechnen oder zuweisen wollen, nutzen Sie immer den modernen Switch-Ausdruck mit dem Pfeil (->). Geht es nur darum, Befehle auszuführen (z.B. Spielfigur bewegen oder Menü öffnen), reicht die klassische Variante mit Doppelpunkt und break.

7 Schleifen und Sprünge (Wiederholungsanweisungen)

In den vorangegangenen Kapiteln haben wir gelernt, wie wir Anweisungen nacheinander ausführen (Sequenz) und wie wir das Programm Entscheidungen treffen lassen (Verzweigung). Doch was ist, wenn wir eine bestimmte Aufgabe mehrfach erledigen müssen?

Stellen Sie sich vor, Sie stehen in der Werkstatt und müssen 100 Nägel in ein großes Brett schlagen. Sie schreiben dafür in Ihrem Bauplan nicht hundertmal untereinander die Anweisung schlageNagel(). Sie fassen den Ablauf zusammen: *"Solange noch Nägel da sind, schlage einen Nagel ein"*.

Genau für diese Automatisierung von Wiederholungen bietet Java die sogenannten **Schleifen** (Wiederholungsanweisungen). Sie sind das absolute Rückgrat der Programmierung, denn sie erlauben es Computern, Millionen von Rechenschritten in Sekundenbruchteilen auszuführen, für die wir nur wenige Zeilen Code schreiben müssen.

7.1 Die while-Anweisung

Die while-Schleife (Solange-Schleife) haben wir in der Einführung (Kapitel 3) bereits kurz angetestet. Sie ist die fundamentalste aller Schleifen.

Ihre Struktur ist simpel: Sie besteht aus dem Schlüsselwort while, einer Bedingung in runden Klammern und einem Code-Block. Vor jedem Durchlauf prüft Java die Bedingung. Ist sie true, wird der Block ausgeführt. Danach springt das Programm wieder nach oben und prüft erneut. Das geht so lange, bis die Bedingung irgendwann false ergibt.

```
public class WhileBeispiel {
    public static void main(String[] args) {
        int zaehler = 1;

        System.out.println("Die Schleife startet:");

        // Solange der Zähler kleiner oder gleich 5 ist...
        while (zaehler <= 5) {
            System.out.println("Aktuelle Zahl: " + zaehler);

            // WICHTIG: Wir müssen den Zähler verändern,
            // sonst entsteht eine Endlosschleife!
```

```
            zaehler = zaehler + 1;
        }

        System.out.println("Die Schleife ist beendet.");
        System.out.println("Der Zähler steht am Ende auf: " + zaehler);
    }
}
```

Die Gefahr der Endlosschleife:

Die while-Schleife hat eine eiserne Regel: *Innerhalb* des Blocks muss zwingend etwas passieren, das die Bedingung irgendwann zu false macht. Vergessen Sie die Zeile zaehler = zaehler + 1;, bleibt der Wert für immer auf 1. Das Programm wird unendlich oft "Aktuelle Zahl: 1" auf den Bildschirm drucken, bis Sie es gewaltsam abbrechen.

7.2 Die do-Anweisung

Die while-Schleife ist eine sogenannte *kopfgesteuerte* Schleife. Sie fragt erst den Türsteher (die Bedingung), bevor sie den Raum (den Block) betritt. Wenn die Bedingung schon ganz am Anfang false ist, wird der Block kein einziges Mal ausgeführt.

Manchmal wollen wir aber, dass ein Code-Block **mindestens ein einziges Mal** ausgeführt wird, bevor überhaupt die erste Prüfung stattfindet. Dafür gibt es die do-Schleife (auch do-while-Schleife genannt). Sie ist *fußgesteuert*.

Ein klassisches Beispiel ist das Erzwingen einer korrekten Benutzereingabe: Wir müssen den Nutzer ja mindestens einmal nach einer Zahl fragen, bevor wir prüfen können, ob sie gültig ist.

```
import java.util.Scanner;

public class DoWhileBeispiel {
    public static void main(String[] args) {
        Scanner tastatur = new Scanner(System.in);
        int passwort;

        // Führe diesen Block aus (mindestens einmal!)
        do {
            System.out.println("Bitte geben Sie die PIN (1234) ein:");
            passwort = tastatur.nextInt();

            if (passwort != 1234) {
                System.out.println("Falsche PIN! Versuchen Sie es erneut.");
            }
```

```
        // ... und wiederhole das Ganze SOLANGE die PIN falsch ist.
        // Achtung: Bei der do-Schleife MUSS ein Semikolon am Ende stehen!
        } while (passwort != 1234);

        System.out.println("Tresor geöffnet!");
    }
}
```

7.3 Die for-Anweisung

Wenn Sie genau wissen, wie oft eine Schleife laufen soll (z. B. "zähle von 1 bis 10"), ist die while-Schleife oft etwas unhandlich, da Zählvariable, Bedingung und Hochzählen über mehrere Zeilen verstreut sind.

Die for-Schleife (Zählschleife) fasst all diese Elemente in einer einzigen, extrem kompakten Zeile zusammen. In den runden Klammern der for-Schleife stehen, durch Semikolons getrennt, exakt drei Bereiche:

1. **Initialisierung:** Wird nur ein einziges Mal ganz zu Beginn ausgeführt (hier deklarieren wir meist unsere Zählvariable).
2. **Bedingung:** Wird vor *jedem* Durchlauf geprüft.
3. **Aktualisierung:** Wird am Ende *jedes* Durchlaufs ausgeführt (hier zählen wir hoch oder runter).

```
public class ForBeispiel {
    public static void main(String[] args) {

        System.out.println("Wir zählen gerade Zahlen:");

        // 1. int i = 2; (Starte bei 2)
        // 2. i <= 10;   (Mache weiter, solange i <= 10 ist)
        // 3. i = i + 2  (Erhöhe i nach jedem Durchlauf um 2)
        for (int i = 2; i <= 10; i = i + 2) {
            System.out.println(i);
        }

        // FALSCH: Die Variable 'i' existiert hier nicht mehr!
        // Der Gültigkeitsbereich (Scope) von 'i' ist auf die for-Schleife
        //  beschränkt.
        // System.out.println("Letzter Wert: " + i);
    }
}
```

Hinweis für Profis: Da das Hochzählen um 1 (i = i + 1) in der Programmierung so unfassbar oft vorkommt, gibt es dafür eine Abkürzung: den Inkrement-Operator ++. Sie werden in

fast jedem Java-Code der Welt Schleifen sehen, die so aussehen: for (int i = 0; i < 10; i++).

7.4 Die break- und continue-Anweisung

Normalerweise bestimmt allein die Bedingung im Kopf (oder Fuß) der Schleife, wann diese endet. Es gibt jedoch Situationen, in denen wir mittendrin ausbrechen oder einen Schritt überspringen müssen. Hierfür nutzen wir Sprunganweisungen.

Die **break-Anweisung** kennen Sie bereits aus der switch-Anweisung. Wenn Java innerhalb einer Schleife auf break; trifft, wird die gesamte Schleife sofort, bedingungslos und endgültig abgebrochen. Das Programm geht in der Zeile nach der Schleife weiter.

Die **continue-Anweisung** ist sanfter. Sie bricht nicht die ganze Schleife ab, sondern nur den *aktuellen Durchlauf*. Java überspringt den restlichen Code im Block und springt sofort wieder nach oben zur nächsten Runden-Prüfung.

```
public class SprungBeispiel {
    public static void main(String[] args) {
        System.out.println("Start der for-Schleife:");

        for (int i = 1; i <= 5; i++) {

            // Wenn i genau 3 ist, überspringen wir diesen Durchlauf
            if (i == 3) {
                System.out.println("Wir überspringen die 3!");
                continue;
            }

            // Wenn i genau 5 ist, brechen wir die komplette Schleife ab
            if (i == 5) {
                System.out.println("Bei 5 brechen wir komplett ab!");
                break;
            }

            // Diese Ausgabe wird für 3 und 5 niemals erreicht
            System.out.println("Zahl: " + i);
        }

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

Ausgabe dieses Programms:

```
Zahl: 1
Zahl: 2
Wir überspringen die 3!
Zahl: 4
Bei 5 brechen wir komplett ab!
Schleife beendet.
```

7.5 Die Label-Anweisung und return-Anweisung

Zum Abschluss der Kontrollstrukturen schauen wir uns zwei mächtige, aber mit Vorsicht zu genießende Sprungbefehle an.

Die Label-Anweisung:

Wenn Sie eine Schleife *in* einer Schleife haben (verschachtelte Schleifen) und ein break aufrufen, bricht Java immer nur die innere Schleife ab. Wenn Sie aus beiden Schleifen gleichzeitig ausbrechen wollen, können Sie der äußeren Schleife einen Namen (ein Label) geben und das break gezielt dorthin schicken. Ein Label ist einfach ein Name gefolgt von einem Doppelpunkt (:).

```
public class LabelBeispiel {
    public static void main(String[] args) {

        // Wir benennen die äußere Schleife "AussenSchleife"
        AussenSchleife:
        for (int woche = 1; woche <= 3; woche++) {
            System.out.println("Woche " + woche + " beginnt.");

            for (int tag = 1; tag <= 7; tag++) {
                if (woche == 2 && tag == 3) {
                    System.out.println("Mittwoch in Woche 2: Notfall! " +
                                       "Wir brechen alles ab!");
                    // Das normale 'break' würde nur die innere Schleife
                    // beenden.
                    // Mit dem Label beenden wir beide Schleifen sofort!
                    break AussenSchleife;
                }
                System.out.println("  Arbeitstag " + tag);
            }
        }

        System.out.println("Arbeit niedergelegt.");
    }
}
```

Warnung: Nutzen Sie Labels nur im absoluten Notfall! Zu viele Sprünge quer durch den

Code machen ein Programm schnell unlesbar – Programmierer sprechen dann abfällig von "Spaghetti-Code".

Die return-Anweisung:

Noch radikaler als break ist return (zu Deutsch: zurückkehren). Während break nur die Schleife beendet, beendet return; sofort die *gesamte Funktion* (in unserem bisherigen Fall also die gesamte main-Methode). Sobald Java ein return; liest, ist das Programm schlagartig zu Ende.

```
public class ReturnBeispiel {
    public static void main(String[] args) {
        int fehlerCode = 404;

        if (fehlerCode != 0) {
            System.out.println("Ein kritischer Fehler ist aufgetreten!");
            // Beendet die main-Methode sofort.
            return;
        }

        // Wenn return ausgeführt wurde, wird dieser Code niemals erreicht
        System.out.println("Dieses Programm lief fehlerfrei.");
    }
}
```

Die wahre Macht von return werden wir erst erkennen, wenn wir unsere eigenen Funktionen schreiben und Werte an den Aufrufer "zurückgeben" wollen.

Damit haben wir das Arsenal unserer Grundwerkzeuge komplett! Wir können Variablen speichern, rechnen, Entscheidungen fällen und Aufgaben wiederholen.

Bisher schreiben wir all unseren Code in einen einzigen großen Block (die main-Methode). In einer echten, großen Werkstatt würde dabei schnell das Chaos ausbrechen. Wir müssen anfangen, Aufgaben auszulagern und Werkzeuge modular zu bauen.

Teil IV: Werkzeuge auslagern (Funktionen)

8 Funktionen

Bisher haben wir all unseren Code in einen einzigen, großen Block geschrieben: die main-Methode. Stellen Sie sich das so vor, als würden Sie in Ihrer Werkstatt alle Arbeiten – vom Sägen über das Leimen bis hin zum Lackieren – auf exakt demselben Tisch ausführen. Bei einem kleinen Vogelhaus mag das noch gutgehen. Wenn Sie aber einen dreitürigen Kleiderschrank bauen, bricht auf Ihrem Tisch schnell das absolute Chaos aus. Sie verlieren den Überblick, wo welche Werkzeuge und Materialien liegen.

In der professionellen Programmierung lösen wir dieses Problem durch **prozedurale Zerlegung**. Wir lagern bestimmte, in sich geschlossene Aufgaben in eigene, kleine Unterprogramme aus. In Java nennen wir diese Unterprogramme **Methoden** (oder allgemeingültiger: Funktionen). In diesem Kapitel lernen wir, wie wir unsere eigenen, maßgeschneiderten Werkzeuge bauen, denen wir Materialien (Parameter) übergeben und die uns fertige Ergebnisse (Rückgabewerte) zurückliefern.

Wichtiger Vorab-Hinweis: Da wir uns in diesem Buch auf die imperative Programmierung konzentrieren und noch keine eigenen "Objekte" erschaffen, werden wir in diesem Kapitel allen unseren Funktionen das Schlüsselwort static voranstellen. Was genau static im Detail bedeutet, klären wir im darauffolgenden Buch zur Objektorientierung. Für uns ist es im Moment einfach eine feste Grammatikregel, um unsere Werkzeuge nutzen zu können.

8.1 Prozedurale Zerlegung und ihre Vorteile

Warum sollten wir unseren Code in Funktionen aufteilen? Dafür gibt es drei unschlagbare Gründe:

1. **Wiederverwendbarkeit (Don't Repeat Yourself - DRY):** Wenn Sie in Ihrem Programm an fünf verschiedenen Stellen den Rabatt für einen Kunden berechnen müssen, schreiben Sie die komplizierte Formel nicht fünfmal auf. Sie schreiben eine Funktion berechneRabatt und rufen diese einfach fünfmal auf. Müssen Sie die Formel später ändern, tun Sie dies nur an einer einzigen Stelle.
2. **Lesbarkeit:** Eine main-Methode, die nur aus Aufrufen wie leseDatenEin(), verarbeiteDaten() und druckeErgebnis() besteht, liest sich so flüssig wie ein Inhaltsverzeichnis. Sie verbirgt die komplexen Details und zeigt sofort, was das Programm auf oberster Ebene tut.
3. **Einfacheres Testen (Teile und Herrsche):** Wenn Ihr Programm aus 1000 Zeilen am Stück besteht und einen Fehler wirft, müssen Sie überall suchen. Wenn Ihr

Programm aus 20 kleinen Funktionen à 50 Zeilen besteht, können Sie jede Funktion einzeln isoliert testen und den Fehler extrem schnell eingrenzen.

8.2 Prozeduren (static void)

Die einfachste Form einer Funktion ist die **Prozedur**. Eine Prozedur führt einfach nur eine Reihe von Anweisungen aus, liefert aber kein konkretes Ergebnis (wie z.B. eine Zahl) an den Aufrufer zurück. In Java signalisieren wir dieses "Liefert nichts zurück" durch das Schlüsselwort void (englisch für: leer / ungültig).

Lassen Sie uns ein Programm schreiben, das eine Begrüßung auf der Konsole ausgibt. Wir lagern die Begrüßung in eine eigene Prozedur aus.

```
public class ProzedurBeispiel {

    // Unsere erste eigene Funktion (Prozedur)
    // Sie steht auf derselben Ebene wie die main-Methode!
    public static void druckeBegruessung() {
        System.out.println("************************************");
        System.out.println("* Willkommen in unserer Werkstatt *");
        System.out.println("************************************");
    }

    // Hier beginnt das eigentliche Programm
    public static void main(String[] args) {
        System.out.println("Programm startet...");

        // Wir rufen unsere Funktion auf (inklusive der runden Klammern!)
        druckeBegruessung();

        System.out.println("Wir bereiten die Werkzeuge vor...");

        // Wir können die Funktion beliebig oft aufrufen
        druckeBegruessung();

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

Wie der Aufruf funktioniert:

Wenn der Computer in der main-Methode auf die Zeile druckeBegruessung(); trifft, pausiert er die main-Methode. Er springt nach oben in unsere Prozedur, führt dort alle Anweisungen (die drei println-Zeilen) aus und springt danach exakt an die Stelle in der main-Methode zurück, an der er sie verlassen hat, um dort weiterzumachen.

8.3 Funktionen mit Rückgabewert

Oft wollen wir, dass eine Funktion etwas berechnet und uns das Ergebnis dieser Berechnung als konkreten Wert zurückliefert. In diesem Fall ersetzen wir das Wörtchen void durch den Datentyp, den die Funktion zurückgeben soll (z.B. int oder double).

Zusätzlich müssen wir in der Funktion zwingend das Schlüsselwort return verwenden, um den berechneten Wert tatsächlich an den Aufrufer zurückzusenden.

```
public class RueckgabeBeispiel {

    // Diese Funktion verspricht, am Ende eine ganze Zahl (int) zurückzugeben
    public static int berechneGeheimeZahl() {
        int zahl = 40 + 2;
        // Das return sendet den Wert (42) an die main-Methode zurück
        // und beendet die Funktion sofort!
        return zahl;
    }

    public static void main(String[] args) {
        System.out.println("Ich frage das Orakel...");

        // Wir rufen die Funktion auf. Der Aufruf wird quasi durch
        // das Ergebnis (42) ersetzt und in der Variable 'ergebnis'
        // gespeichert.
        int ergebnis = berechneGeheimeZahl();

        System.out.println("Die geheime Zahl lautet: " + ergebnis);
    }
}
```

8.4 Parameter und Lokale Variablen

Unsere Funktionen sind bisher sehr statisch. Die berechneGeheimeZahl rechnet immer 40 + 2. Richtig mächtig werden Funktionen erst, wenn wir ihnen von außen Rohmaterialien übergeben können, mit denen sie arbeiten sollen. Diese Einwurfschlitze für Daten nennen wir **Parameter**.

Parameter werden in den runden Klammern der Funktionsdefinition deklariert, ganz ähnlich wie normale Variablen.

```
public class ParameterBeispiel {

    // Diese Funktion erwartet zwei ganze Zahlen als Eingabe:
    // Sie nennt diese intern 'laenge' und 'breite'
```

```
    public static int berechneFlaeche(int laenge, int breite) {
        int flaeche = laenge * breite;
        return flaeche;
    }

    public static void main(String[] args) {
        // Beim Aufruf MÜSSEN wir nun exakt zwei int-Werte übergeben.
        // Diese nennt man 'Argumente'.
        int meinZimmer = berechneFlaeche(4, 5);
        int meinGarten = berechneFlaeche(10, 20);

        System.out.println("Zimmerfläche: " + meinZimmer);
        System.out.println("Gartenfläche: " + meinGarten);
    }
}
```

Wenn wir berechneFlaeche(4, 5) aufrufen, nimmt Java die 4 und kopiert sie in die Parametervariable laenge. Die 5 wird in breite kopiert. Dann beginnt die Funktion zu rechnen.

8.5 Gültigkeitsbereich und Eigenschaften

Wir kommen nun zu einem der wichtigsten Konzepte der Programmierung, das bei Anfängern oft für Verwirrung sorgt: dem **Gültigkeitsbereich (Scope)** von Variablen.

- **Lokale Variablen:** Alle Variablen, die Sie *innerhalb* einer Funktion deklarieren (inklusive der Parameter in den runden Klammern), sind strikt lokal! Sie existieren nur für die Dauer des Funktionsaufrufs. Sobald das return erreicht ist oder die Funktion endet, löscht Java diese Variablen unwiderruflich aus dem Arbeitsspeicher.
- **Abgeschottete Welten:** Die main-Methode kann nicht in die Variablen der Funktion berechneFlaeche hineinsehen. Und umgekehrt kann berechneFlaeche nicht auf die Variablen der main-Methode zugreifen.

Schauen Sie sich dieses Beispiel genau an:

```
public class ScopeBeispiel {

    public static void veraendereZahl(int zahl) {
        // Diese 'zahl' ist eine Kopie!
        System.out.println("In der Funktion (vorher): " + zahl);
        zahl = 999;
        System.out.println("In der Funktion (nachher): " + zahl);
    }

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

```
        System.out.println("In main (vorher): " + zahl);

        // Wir übergeben den WERT 10 an die Funktion, NICHT unsere Kiste
        // 'zahl'!
        veraendereZahl(zahl);

        // Unsere Kiste 'zahl' in der main-Methode ist völlig unangetastet
        // geblieben!
        System.out.println("In main (nachher): " + zahl);
        // Gibt immer noch 10 aus!
    }
}
```

Obwohl die Variable in der main-Methode und der Parameter in der Funktion zufällig beide den Namen zahl tragen, sind es für den Computer zwei völlig unterschiedliche Kisten, die in unterschiedlichen Räumen stehen. Java verwendet standardmäßig *Call by Value* (Wertübergabe): Es wird immer nur eine Kopie des Inhalts übergeben.

8.6 Nutzung von Funktionsbibliotheken

Sie müssen das Rad nicht immer neu erfinden. Die Entwickler von Java haben bereits Tausende von nützlichen Werkzeugen für uns programmiert und in sogenannten Bibliotheken zusammengefasst.

Eine der nützlichsten Bibliotheken ist die Klasse Math. Sie enthält dutzende mathematische Funktionen, die wir sofort nutzen können, ohne sie selbst programmieren zu müssen. Wir rufen sie einfach mit Math. gefolgt vom Funktionsnamen auf.

```
public class BibliothekBeispiel {
    public static void main(String[] args) {
        // Math.max gibt die größere der beiden Zahlen zurück
        int groesser = Math.max(15, 42);
        System.out.println("Die größere Zahl ist: " + groesser);

        // Math.pow (power) berechnet eine Potenz (hier: 2 hoch 3)
        // Achtung: Math.pow liefert immer ein 'double' zurück!
        double potenz = Math.pow(2.0, 3.0);
        System.out.println("2 hoch 3 ist: " + potenz);

        // Math.sqrt (square root) berechnet die Quadratwurzel
        double wurzel = Math.sqrt(81.0);
        System.out.println("Die Wurzel aus 81 ist: " + wurzel);
    }
}
```

Durch das Auslagern in solche Bibliotheken bleibt unser eigener Code sauber und kurz.

8.7 Rekursion

Zum Abschluss dieses Kapitels betreten wir einen der faszinierendsten, aber auch kniffligsten Bereiche der Programmierung: die Rekursion.

Eine Funktion darf in ihrem eigenen Rumpf (ihrem Block) nicht nur andere Funktionen aufrufen – sie darf sich auch **selbst aufrufen**! Das nennt man Rekursion. Stellen Sie sich vor, Sie stehen zwischen zwei Spiegeln und sehen, wie sich Ihr Spiegelbild unendlich oft in sich selbst wiederholt.

Damit eine Rekursion nicht zu einer endlosen Schleife wird und den Speicher des Computers sprengt, braucht jeder rekursive Algorithmus zwingend eine **Abbruchbedingung** (den sogenannten Basisfall).

Das klassische Beispiel für Rekursion ist die Berechnung der Fakultät einer Zahl aus der Mathematik. Die Fakultät von 5 (geschrieben 5!) ist 5 * 4 * 3 * 2 * 1.

Man kann das auch so formulieren: Die Fakultät von 5 ist 5 * Fakultät von 4.

Und die Fakultät von 4 ist 4 * Fakultät von 3.

Das geht so weiter, bis wir bei der 1 ankommen. Die Fakultät von 1 ist einfach 1 (unser Abbruch!).

In Code übersetzt sieht dieses magische Prinzip so aus:

```
public class RekursionBeispiel {

    // Eine rekursive Funktion
    public static int berechneFakultaet(int n) {
        // 1. Die Abbruchbedingung (Basisfall)
        // Ohne dies würde sich die Funktion endlos selbst aufrufen!
        if (n == 1) {
            return 1;
        }
        // 2. Der rekursive Aufruf
        else {
            return n * berechneFakultaet(n - 1);
        }
    }

    public static void main(String[] args) {
        int zahl = 5;
```

```
        int ergebnis = berechneFakultaet(zahl);

        System.out.println("Die Fakultät von " + zahl + " ist: " + ergebnis);
    }
}
```

Wie rechnet der Computer hier?

Wenn Sie berechneFakultaet(5) aufrufen, passiert Folgendes:

1. n = 5. Ist nicht 1. Also rechne: 5 * berechneFakultaet(4) ... und warte auf das Ergebnis.
2. berechneFakultaet(4) startet. Ist nicht 1. Rechne: 4 * berechneFakultaet(3) ... warte.
3. berechneFakultaet(3) startet... 3 * berechneFakultaet(2) ... warte.
4. berechneFakultaet(2) startet... 2 * berechneFakultaet(1) ... warte.
5. berechneFakultaet(1) startet. n ist 1! Abbruchbedingung erfüllt. Gibt 1 zurück!
6. Jetzt rattert die Kette rückwärts wieder hoch: 2 * 1 ist 2. Das wird zurückgegeben. 3 * 2 ist 6. 4 * 6 ist 24. 5 * 24 ist 120. Die main-Methode erhält die fertige 120.

Rekursion erfordert etwas Übung im logischen Denken, ist aber ein extrem mächtiges Werkzeug, um komplexe, verschachtelte Probleme mit sehr wenigen Zeilen Code zu lösen.

Sie haben nun gelernt, wie Sie Ihren Code strukturieren, eigene Befehle erschaffen und Daten sicher in Funktionen hinein- und wieder herausreichen. Bisher haben wir unseren Variablen jedoch immer nur exakt einen einzigen Wert zugewiesen. Was ist, wenn wir hundert Temperaturwerte oder tausend Kundennummern speichern wollen?

Teil V: Datenmengen verwalten (Arrays)

9 Arrays

In unserer Werkstatt haben wir bisher gelernt, wie wir einzelne Materialien in maßgeschneiderten Kisten (Variablen) ablegen. Wenn wir eine Temperatur speichern wollten, haben wir ein int angelegt. Wenn wir einen Namen brauchten, haben wir einen String genutzt.

Doch stellen Sie sich vor, Sie sollen für eine Wetterstation die Temperaturen eines ganzen Jahres auswerten – also 365 Werte. Wenn Sie mit unserem bisherigen Wissen arbeiten, müssten Sie 365 einzelne Variablen anlegen (temp1, temp2, temp3 ... bis temp365). Allein das Eintippen dieser Variablen würde Stunden dauern, und eine Schleife könnte nicht automatisch über sie hinweglaufen, da jede Variable einen völlig eigenständigen Namen hat.

Für genau dieses Problem bietet Java ein geniales Konzept: **Arrays** (zu Deutsch: Felder oder Reihungen). Ein Array ist wie ein großer Setzkasten oder ein Regal mit vielen durchnummerierten Schubladen. In diesem Kapitel lernen wir, wie wir solche Regale bauen, befüllen und systematisch auswerten.

9.1 Motivation und Anmerkungen

Ein Array ist eine Datenstruktur, die eine *feste Anzahl* von Elementen desselben Datentyps speichert.

Drei wichtige Anmerkungen vorab:

1. **Homogenität:** Sie können in einem Array für ganze Zahlen (int[]) nur ganze Zahlen ablegen. Sie können dort nicht plötzlich ein Wort (String) oder eine Kommazahl (double) hineinquetschen. Jede Schublade im Regal hat exakt dieselbe Form.
2. **Feste Größe:** Wenn Sie das Regal einmal gebaut haben (z. B. mit 10 Schubladen), können Sie es im Nachhinein nicht mehr vergrößern oder verkleinern. Sie müssen sich beim Erschaffen des Arrays entscheiden, wie viel Platz Sie benötigen.
3. **Referenzdatentyp:** Arrays sind, genau wie String, keine einfachen (primitiven) Datentypen. Sie sind komplexe Gebilde. Das hat Auswirkungen darauf, wie Java sie im Speicher verwaltet.

9.2 Array-Erzeugung und Array-Referenzvariablen

Das Erstellen eines Arrays erfolgt in Java in der Regel in zwei Schritten, die man oft in einer einzigen Zeile zusammenfasst: der Deklaration der **Referenzvariablen** und der

eigentlichen **Erzeugung** des Arrays.

1. Zuerst sagen wir dem Computer, dass wir ein Array planen. Wir hängen dafür einfach eckige Klammern [] an den gewünschten Datentyp an (z. B. int[]). Die Variable, die wir so erschaffen, enthält noch nicht das Array selbst! Sie ist lediglich ein Zettel, auf dem später steht, wo das Regal in unserer Werkstatt (im Arbeitsspeicher) steht. Man nennt dies eine Referenz.
2. Dann rufen wir den Handwerker, um das Regal physisch zu bauen. Dafür nutzen wir in Java zwingend das Schlüsselwort new. Wir sagen new int[5], um ein Regal mit 5 Schubladen für ganze Zahlen zu zimmern.

```
public class ArrayErzeugungBeispiel {
    public static void main(String[] args) {
        // 1. Deklaration der Referenzvariable (der Zettel)
        int[] temperaturen;

        // 2. Erzeugung des Arrays (das Regal wird gebaut)
        // und die Adresse wird auf den Zettel geschrieben (= Zuweisung)
        temperaturen = new int[5];

        // Beides in einer Zeile (die absolut übliche Schreibweise):
        double[] preise = new double[3];

        // Wenn wir die Werte schon vorher wissen, gibt es eine Abkürzung
        // (direkte Initialisierung ohne das Wort 'new'):
        String[] wochentage = {"Montag", "Dienstag", "Mittwoch"};

        System.out.println("Arrays wurden erfolgreich im Speicher " +
                           "angelegt!");
    }
}
```

Wenn wir new int[5] aufrufen, füllt Java das neue Regal netterweise sofort mit Standardwerten auf, damit keine alten, fehlerhaften Daten aus dem RAM darin liegen. Bei Zahlen ist dieser Standardwert immer 0, bei boolean ist er false.

9.3 Zugriff auf Array-Elemente

Nun steht unser Regal bereit. Wie greifen wir auf die einzelnen Schubladen zu? Wir nutzen dafür den Namen des Arrays und geben in eckigen Klammern die Nummer der gewünschten Schublade an. Diese Nummer nennt man **Index**.

Hier lauert die mit Abstand gefährlichste Falle für Programmieranfänger: **Computer fangen immer bei 0 an zu zählen!** Wenn Ihr Array eine Größe von 5 hat, heißen die Schubladen nicht 1 bis 5. Sie heißen 0, 1, 2, 3 und 4. Die 5 existiert als Index nicht!

```java
public class ArrayZugriffBeispiel {
    public static void main(String[] args) {
        // Wir bauen ein Array für 4 Spieler-Punkte
        int[] punkte = new int[4];

        // Wir schreiben Werte in die einzelnen Schubladen (Index 0 bis 3)
        punkte[0] = 150; // Der 1. Spieler
        punkte[1] = 200; // Der 2. Spieler
        punkte[2] = 50;  // Der 3. Spieler
        punkte[3] = 99;  // Der 4. Spieler

        System.out.println("Punkte von Spieler 2 (Index 1):");
        System.out.println(punkte[1]); // Gibt 200 aus

        // Wir können auch mit den Elementen rechnen
        int summe = punkte[0] + punkte[1];
        System.out.println("Summe der ersten beiden Spieler: " + summe);

        // Jedes Array kennt seine eigene Größe!
        // Das Attribut .length liefert die Anzahl der Schubladen (hier 4)
        System.out.println("Größe des Arrays: " + punkte.length);

        // FALSCH: Programmabbruch mit ArrayIndexOutOfBoundsException!
        // Der folgende Code würde das Programm sofort abstürzen lassen,
        // da es den Index 4 (die 5. Schublade) nicht gibt:
        // punkte[4] = 1000;
    }
}
```

9.4 Array-Zerstörung (Garbage Collection)

Wie wir in Kapitel 1.4 gelernt haben, ist der Arbeitsspeicher (RAM) begrenzt. Wenn wir in einem riesigen Programm ständig neue Arrays mit new erzeugen, wäre der Speicher irgendwann voll.

Glücklicherweise müssen wir uns in Java nicht manuell um den Abriss alter Regale kümmern. Java besitzt einen vollautomatischen Müllabfuhrmechanismus: die **Garbage Collection**.

Die Regel der Garbage Collection ist simpel: Sobald es keinen Zettel (Referenzvariable) mehr gibt, der auf ein Regal zeigt, geht Java davon aus, dass wir das Regal nicht mehr brauchen. Der Speicherplatz wird automatisch freigegeben. Wir können diesen Prozess erzwingen, indem wir unserer Variablen das spezielle Schlüsselwort null (nichts) zuweisen.

```
public class GarbageCollectionBeispiel {
    public static void main(String[] args) {
        // Wir bauen ein riesiges Array (braucht viel Speicher)
        double[] messdaten = new double[1000000];

        messdaten[0] = 42.5;
        System.out.println("Daten gespeichert.");

        // Wir brauchen die Daten nicht mehr.
        // Wir zerschneiden den Zettel, der auf das Regal zeigt:
        messdaten = null;

        // Ab diesem Moment ist das Array "verwaist".
        // Die Java Garbage Collection wird es bei nächster Gelegenheit
        // automatisch aus dem RAM löschen.

        // Wenn wir jetzt versuchen würden, auf messdaten[0] zuzugreifen,
        // bekämen wir einen Fehler (NullPointerException), da der Zettel
        // leer ist!
    }
}
```

9.5 Die for-each-Schleife

Arrays entfalten ihre wahre Magie erst in Kombination mit Schleifen. Wir können eine reguläre for-Schleife (siehe Kapitel 7) nutzen, um den Index von 0 bis array.length - 1 hochzuzählen und so jede Schublade nacheinander zu öffnen.

Weil das Auslesen *aller* Elemente eines Arrays extrem oft vorkommt, bietet Java eine komfortable Abkürzung: die **for-each-Schleife** (zu Deutsch: Für-jedes-Schleife). Sie kommt ganz ohne fehleranfälliges Hochzählen von Index-Variablen aus.

```
public class ForEachBeispiel {

    // Eine Hilfsfunktion zur Demonstration
    public static void druckeNamen(String[] namensListe) {

        System.out.println("--- Die traditionelle for-Schleife ---");
        for (int i = 0; i < namensListe.length; i++) {
            System.out.println("Hallo " + namensListe[i]);
        }

        System.out.println("--- Die moderne for-each-Schleife ---");
        // Lies: "Für jeden String 'name' in der 'namensListe' mache
        // Folgendes:"
        for (String name : namensListe) {
            System.out.println("Hallo " + name);
```

```
        }
    }

    public static void main(String[] args) {
        String[] kunden = {"Anna", "Ben", "Carla", "David"};
        druckeNamen(kunden);
    }
}
```

Die for-each-Schleife kopiert bei jedem Durchlauf automatisch den Inhalt der nächsten Schublade in die Hilfsvariable (hier name), bis das Array komplett durchlaufen ist. *Hinweis:* Sie eignet sich nur zum *Lesen* von Arrays. Wenn Sie Werte im Array *verändern* wollen, müssen Sie weiterhin die traditionelle for-Schleife mit dem Index [i] verwenden.

9.6 Mehrdimensionale Arrays

Manchmal reicht ein einfaches, eindimensionales Regal nicht aus. Wenn wir ein Schachbrett, ein Tic-Tac-Toe-Spielfeld oder eine Tabelle wie in Excel programmieren wollen, brauchen wir Zeilen *und* Spalten.

Java erlaubt es uns, Arrays in Arrays zu packen. Wir nennen dies **mehrdimensionale Arrays** (meistens 2D-Arrays). Um ein 2D-Array zu erzeugen, nutzen wir einfach zwei Paar eckige Klammern [][]. Die erste Klammer steht für die Zeile (Y-Achse), die zweite für die Spalte (X-Achse).

Lassen Sie uns ein kleines Spielfeld erschaffen und es mit Nullen (leer) und Einsen (Wände) füllen:

```
public class MehrdimensionalBeispiel {
    public static void main(String[] args) {
        // Wir bauen ein 2D-Array: 3 Zeilen und 4 Spalten
        int[][] spielfeld = new int[3][4];

        // Wir setzen manuell ein paar Wände (Zahl 1)
        spielfeld[0][1] = 1; // 1. Zeile, 2. Spalte
        spielfeld[1][1] = 1; // 2. Zeile, 2. Spalte
        spielfeld[2][3] = 1; // 3. Zeile, 4. Spalte

        // Wir nutzen verschachtelte for-Schleifen, um das Spielfeld zu
        // zeichnen
        System.out.println("Unser Spielfeld:");

        // Die äußere Schleife geht Zeile für Zeile durch (.length liefert
        // Anzahl der Zeilen)
        for (int zeile = 0; zeile < spielfeld.length; zeile++) {
```

```
            // Die innere Schleife geht in der aktuellen Zeile durch jede
            // Spalte
            for (int spalte =0; spalte < spielfeld[zeile].length; spalte++) {

                // Print (ohne 'ln') druckt in derselben Zeile weiter!
                System.out.print(spielfeld[zeile][spalte] + " ");
            }
            // Nach jeder Zeile machen wir einen Zeilenumbruch
            System.out.println();
        }
    }
}
```

Die Ausgabe dieses Programms sieht wie ein echtes, kleines Gitter aus:

```
Unser Spielfeld:
0 1 0 0
0 1 0 0
0 0 0 1
```

Sie haben soeben das Fundament für unser großes Praxisprojekt – das Spiel "Dungeon-Schatzsuche" – gelegt!

Teil VI: Die Brücke zur Objektorientierung

10 Referenzdatentypen

In den vergangenen Kapiteln haben wir zwei völlig unterschiedliche Arten von Datenstrukturen genutzt, ohne den fundamentalen Unterschied in ihrer Mechanik tiefgründig zu beleuchten. Wir hatten auf der einen Seite die primitiven (einfachen) Datentypen wie int oder boolean. Auf der anderen Seite hatten wir komplexe Konstrukte wie Arrays (int[]) oder Zeichenketten (String), die wir oft mit dem geheimnisvollen Wort new ins Leben rufen mussten.

Warum dieser Unterschied? Warum verhalten sich Arrays und Strings beim Zuweisen und Vergleichen manchmal völlig anders, als wir es von einfachen Zahlen gewohnt sind?

Die Antwort lautet: Es handelt sich um **Referenzdatentypen**. In diesem Kapitel werfen wir einen Blick unter die Motorhaube von Java. Wir schauen uns an, wie der Computer seinen Speicher wirklich organisiert. Wenn Sie dieses Kapitel verstanden haben, werden Sie nie wieder versehentlich Daten überschreiben oder sich über unlogische Gleichheitsprüfungen wundern.

10.1 Speicherverwaltung und Schema von Referenzvariablen

Um Referenzdatentypen zu verstehen, müssen wir unsere Werkstatt-Metapher um ein Detail erweitern. Der Arbeitsspeicher (RAM) eines Java-Programms ist grob in zwei getrennte Bereiche unterteilt: den **Stack** (Stapelspeicher) und den **Heap** (Halden- oder Objektspeicher).

1. **Der Stack (Die Werkbank):** Hier landen alle lokalen Variablen unserer Funktionen. Wenn Sie int alter = 25; schreiben, baut Java direkt auf der Werkbank eine Kiste, schreibt "alter" darauf und legt die Zahl 25 hinein. Die Kiste *enthält* den Wert.
2. **Der Heap (Die große Lagerhalle):** Arrays und andere komplexe Objekte können riesig werden (Erinnern Sie sich an unser Array mit einer Million Messdaten?). Auf der kleinen Werkbank ist dafür kein Platz. Wenn wir das Wort new benutzen, baut Java das Regal drüben in der riesigen Lagerhalle auf.

Wie finden wir nun unser Array in der Lagerhalle wieder?

Wenn wir int[] zahlen = new int[5]; schreiben, passieren zwei Dinge:

- Das new int[5] baut das Regal in der *Lagerhalle*.
- Die Variable zahlen wird als kleine Kiste auf unserer *Werkbank* abgestellt.

In dieser Kiste namens zahlen liegen nun aber nicht die fünf Zahlen! In der Kiste liegt

lediglich ein **Zettel mit der Adresse** des Regals in der Lagerhalle (die sogenannte Referenz oder der Verweis). Eine Referenzvariable *enthält* also nicht die Daten selbst, sie *zeigt* nur auf sie.

```
public class SpeichermodellBeispiel {
    public static void main(String[] args) {
        // Primitiver Datentyp: Der Wert 42 liegt DIREKT in der Variable
        int einfacheZahl = 42;

        // Referenzdatentyp: Das Array liegt im Heap (Lagerhalle).
        // Die Variable 'meinArray' enthält nur die ADRESSE dorthin.
        int[] meinArray = new int[3];
        meinArray[0] = 10;
        meinArray[1] = 20;
        meinArray[2] = 30;

        System.out.println("Einfache Zahl: " + einfacheZahl);

        // Was passiert, wenn wir versuchen, den "Zettel" direkt
        // auszudrucken?
        System.out.println("Der Inhalt der Referenzvariablen: " + meinArray);
        // Ausgabe ist kryptisch, z.B.: [I@7a81197d (Das ist die
        // Speicheradresse!)
    }
}
```

10.2 Lokale Variablen, Literale und Zuweisung

Dieses "Zettel"-Prinzip (Referenz-Schema) hat dramatische Auswirkungen auf die Zuweisung mit dem Gleichheitszeichen (=).

Wenn wir bei einfachen Datentypen int a = b; schreiben, kopiert Java den *Inhalt* (die Zahl) von Kiste b in Kiste a. Beide Kisten sind danach völlig unabhängig. Ändern wir a, bleibt b unberührt.

Wenn wir bei Referenzdatentypen int[] a = b; schreiben, kopiert Java ebenfalls den Inhalt der Kiste. Aber der Inhalt ist ja nur der Zettel mit der Adresse! Wir kopieren also nicht das riesige Regal in der Lagerhalle. Wir schreiben nur einen zweiten Zettel mit derselben Adresse. **Beide Variablen zeigen nun auf exakt dasselbe Regal im Speicher.** Das Literal für einen "leeren Zettel" kennen wir bereits: Es lautet null.

```
public class ReferenzZuweisungBeispiel {
    public static void main(String[] args) {
        // Wir bauen EIN Regal in der Lagerhalle
        int[] regalA = new int[3];
        regalA[0] = 5;
```

```
        // GEFAHR: Wir kopieren NICHT das Regal, sondern nur den
        // Adress-Zettel!
        int[] regalB = regalA;

        System.out.println("Wert in regalA an Index 0: " + regalA[0]);
        // Gibt 5
        System.out.println("Wert in regalB an Index 0: " + regalB[0]);
        // Gibt 5

        // Wir ändern nun das Array über den Zettel 'regalB'
        System.out.println("Wir überschreiben regalB[0] mit der Zahl 99...");
        regalB[0] = 99;

        // Da beide Zettel auf dasselbe Regal zeigen, hat sich auch regalA
        // "geändert"!
        System.out.println("Wert in regalA an Index 0: " + regalA[0]);
        // Gibt plötzlich 99!

        // Wir werfen den Zettel B weg (wir weisen das leere Literal zu)
        regalB = null;

        // Das Regal ist aber noch da, denn Zettel A zeigt noch darauf!
        System.out.println("Regal existiert noch über A: " + regalA[0]);
    }
}
```

10.3 Operatoren und Gleichheit

Eine weitere große Fehlerquelle, die aus dem Referenz-Schema resultiert, ist das Prüfen auf Gleichheit mit dem Operator ==.

Der Operator == macht in Java immer exakt das Gleiche: Er vergleicht stur den Inhalt der beiden Variablen-Kisten auf der Werkbank (dem Stack).

- Bei int vergleicht er die Zahlen. 5 == 5 ist true.
- Bei Arrays oder Strings vergleicht er die *Zettel* (die Speicheradressen)! Er geht *nicht* in die Lagerhalle und prüft, ob in den Regalen zufällig dieselben Dinge liegen.

Dies führt zu dem berühmten Phänomen, dass zwei Arrays mit identischem Inhalt für Java nicht "gleich" sind. Man nennt dies den Unterschied zwischen **Identität** (selbes Objekt im Speicher) und **Gleichheit** (gleicher Inhalt).

```
public class GleichheitBeispiel {
    public static void main(String[] args) {
        // Wir bauen ZWEI verschiedene Regale mit exakt demselben Inhalt
```

```
        int[] array1 = {1, 2, 3};
        int[] array2 = {1, 2, 3};

        // Wir schreiben einen Zettel, der auf dasselbe Regal zeigt wie
        // array1
        int[] array3 = array1;

        System.out.println("Sind array1 und array2 identisch (==)?");
        // Liefert FALSE! Es sind zwei verschiedene Adressen im Speicher.
        System.out.println(array1 == array2);

        System.out.println("Sind array1 und array3 identisch (==)?");
        // Liefert TRUE! Beide Zettel zeigen auf exakt dieselbe Adresse.
        System.out.println(array1 == array3);
    }
}
```

Anmerkung zu String: Da String ebenfalls ein Referenzdatentyp ist, dürfen Sie Zeichenketten in Java niemals mit == vergleichen! Dafür gibt es eine spezielle Funktion namens equals, die wir im nächsten Buch zur Objektorientierung im Detail behandeln werden.

10.4 Parameter und Funktionsrückgabewerte

Im Kapitel 8 (Funktionen) haben wir gelernt, dass Java bei der Übergabe von Parametern immer *Call by Value* (Wertübergabe) anwendet. Das heißt, die Parameter-Variable in der Funktion ist eine reine Kopie. Das ist korrekt – aber bei Referenztypen kopieren wir eben wieder nur den Adress-Zettel!

Wenn wir ein Array an eine Funktion übergeben, übergeben wir die Adresse. Die Funktion kann das Array in der Lagerhalle betreten und die Werte darin dauerhaft für alle verändern (man nennt das einen *Seiteneffekt*).

Ebenso kann eine Funktion in ihrem Inneren ein komplett neues Array mit new erschaffen und am Ende den Zettel (die Referenz) über return an die main-Methode zurückgeben.

```
public class ReferenzParameterBeispiel {

    // 1. Funktion, die ein Array als Parameter bekommt und ES VERÄNDERT
    public static void verdoppleWerte(int[] daten) {
        // 'daten' enthält die Adresse des originalen Arrays aus main!
        for (int i = 0; i < daten.length; i++) {
            daten[i] = daten[i] * 2; // Wir manipulieren das Original im Heap
        }
```

```
    }

    // 2. Funktion, die ein NEUES Array erschafft und zurückgibt
    public static int[] baueZahlenReihe(int laenge) {
        // Wir bauen ein neues Regal im Heap
        int[] neuesArray = new int[laenge];
        for (int i = 0; i < laenge; i++) {
            neuesArray[i] = i + 1; // Füllt mit 1, 2, 3...
        }
        // Wir geben den Adress-Zettel an den Aufrufer zurück
        return neuesArray;
    }

    public static void main(String[] args) {
        // --- Test zu Parameter-Seiteneffekten ---
        int[] meineZahlen = {10, 20, 30};
        System.out.println("Vor der Funktion: " + meineZahlen[0]); // Gibt 10

        // Wir übergeben die Adresse
        verdoppleWerte(meineZahlen);

        // Das Original wurde von der Funktion dauerhaft verändert!
        System.out.println("Nach der Funktion: " + meineZahlen[0]);
        // Gibt 20

        // --- Test zur Rückgabe von Referenzen ---
        // Wir fangen den neuen Adress-Zettel auf
        int[] frischGebauteReihe = baueZahlenReihe(5);

        System.out.println("Letzte Zahl der neuen Reihe: " +
                           frischGebauteReihe[4]); // Gibt 5
    }
}
```

Damit haben Sie das Geheimnis der Speicherverwaltung gelüftet! Sie wissen nun, dass komplexe Daten im Heap (der Lagerhalle) wohnen und wir sie über Zettel (Referenzvariablen) im Stack fernsteuern. Sie verstehen, warum sich eine Änderung einer Variable manchmal scheinbar magisch auf eine andere auswirkt und warum == nicht immer das tut, was man naiv erwarten würde.

Dieses Wissen ist das perfekte Sprungbrett für den letzten theoretischen Schritt dieses Buches. Was ist, wenn uns Arrays nicht mehr reichen? Ein Array kann immer nur denselben Datentyp (z. B. nur int) aufnehmen. Wenn wir aber einen "Spieler" speichern wollen, der einen Namen (String), Lebenspunkte (int) und einen Status "vergiftet" (boolean) hat, brauchen wir einen eigenen, maßgeschneiderten Datentyp.

11 Klassen und Objekte

In unserer Werkstatt haben wir bisher gelernt, wie wir einzelne Materialien in Variablen lagern (int, String) und wie wir viele Materialien der *gleichen* Sorte in großen Regalen, den Arrays (int[]), strukturieren.

Doch was ist, wenn die echte Welt komplexer wird? Stellen Sie sich vor, Sie bauen ein Computerspiel und möchten einen "Spieler" im Speicher ablegen. Ein Spieler besteht nicht nur aus einer einzelnen Zahl. Er hat einen Namen (String), eine Anzahl an Lebenspunkten (int) und einen Zustand, ob er gerade vergiftet ist (boolean).

Wir können diese drei völlig unterschiedlichen Datentypen nicht in ein einziges Array quetschen, da Arrays absolut homogen sein müssen (sie fassen nur gleiche Typen). Wir müssten drei separate Variablen anlegen (spielerName, spielerLebenspunkte, spielerVergiftet). Wenn wir 100 Spieler haben, versinken wir im Variablen-Chaos.

Die Lösung ist die Erschaffung eines eigenen, völlig neuen Datentyps. In Java tun wir dies mithilfe von **Klassen und Objekten**. In diesem Kapitel lernen wir, wie wir eigene Baupläne zeichnen und daraus komplexe Daten-Pakete schnüren.

Wichtiger Vorab-Hinweis: In der modernen Softwareentwicklung sind Klassen noch viel mächtiger – sie können nicht nur Daten speichern, sondern auch eigenes Verhalten (Methoden) besitzen. Dieses Verhalten ist der Kern der "Objektorientierten Programmierung", der wir ein komplett eigenes, weiterführendes Buch widmen. In diesem Kapitel nutzen wir Klassen ausschließlich als **reine Datencontainer**, um Informationen logisch zu bündeln.

11.1 Motivation und Klassendefinition

Um einen eigenen Datentyp zu erschaffen, müssen wir Java zunächst beibringen, wie dieser Datentyp überhaupt aussieht. Wir zeichnen einen Bauplan. In der Programmiersprache nennt man diesen Bauplan eine **Klasse** (class).

Eine Klassendefinition reserviert noch *kein* einziges Byte im Arbeitsspeicher für konkrete Daten. Sie ist wirklich nur die Zeichnung auf dem Papier, die sagt: *"Wenn* wir später jemals einen Spieler bauen, *dann* wird er aus einem Text, einer Zahl und einem Wahrheitswert bestehen."

```
// Dies ist unser eigener Bauplan!
// Eine Klasse wird normalerweise in einer eigenen Datei namens Spieler.java
// gespeichert.
```

```
public class Spieler {

    // Die Eigenschaften (Attribute) unseres neuen Datentyps:
    String name;
    int lebenspunkte;
    boolean istVergiftet;

}
```

Die Variablen, die wir direkt innerhalb der Klasse (aber außerhalb von Funktionen) deklarieren, nennt man **Attribute** (oder auch Instanzvariablen/Eigenschaften).

11.2 Objekterzeugung und Objekt-Referenzvariablen

Der Bauplan ist fertig. Nun wollen wir ihn nutzen, um in unserer Werkstatt einen echten Spieler zu erschaffen. Das eigentliche, physische Ding, das nach dem Bauplan im Speicher (im Heap / der Lagerhalle) gebaut wird, nennt man ein **Objekt** (oder auch eine Instanz).

Erinnern Sie sich an Kapitel 10? Genau wie bei Arrays nutzen wir zum Bauen von komplexen Dingen in der Lagerhalle zwingend das Schlüsselwort new. Und genau wie bei Arrays erhalten wir beim Bauen kein riesiges Objekt für unsere Werkbank, sondern lediglich einen kleinen Adress-Zettel (eine Referenzvariable), der auf das fertige Objekt in der Lagerhalle zeigt.

```
// Dies ist unsere ausführende Hauptklasse
public class ObjektErzeugungBeispiel {
    public static void main(String[] args) {

        // 1. Wir deklarieren eine Referenzvariable vom Typ 'Spieler'
        // (unserem neuen Bauplan)
        // Das ist unser leerer Zettel.
        Spieler held;

        // 2. Wir rufen den Handwerker mit 'new' und dem Namen der Klasse.
        // Das baut das Objekt im Heap und schreibt die Adresse auf den
        // Zettel.
        held = new Spieler();

        // Beides kompakt in einer Zeile (wie wir es auch bei Arrays gemacht
        // haben):
        Spieler feind = new Spieler();

        System.out.println("Zwei Spieler-Objekte wurden im Speicher " +
                           "erschaffen!");
        System.out.println("Held-Referenz: " + held);
```

```
        // Gibt kryptische Speicheradresse aus
    }
}
```

11.3 Attribute deklarieren und nutzen

Wir haben nun zwei Objekte (held und feind) im Speicher. Aktuell sind diese Objekte noch mit den Standardwerten von Java gefüllt (Zahlen sind 0, Strings sind null, Wahrheitswerte sind false).

Wie kommen wir an die einzelnen Kisten *innerhalb* des Objekts heran, um sie mit echten Daten zu füllen? Dafür nutzen wir die sogenannte **Punkt-Notation**.

Wir schreiben den Namen unserer Referenzvariablen (den Zettel), gefolgt von einem Punkt (.) und dann dem Namen des gewünschten Attributs. Der Punkt bedeutet so viel wie: *"Gehe zu der Adresse auf dem Zettel und greife dort in die Kiste namens...".*

```
public class AttributZugriffBeispiel {
    public static void main(String[] args) {
        // Erschaffe einen neuen Spieler
        Spieler ritter = new Spieler();

        // Fülle die Attribute mit Leben (Schreiben)
        ritter.name = "Arthur";
        ritter.lebenspunkte = 100;
        ritter.istVergiftet = false;

        // Wir erschaffen ein ZWEITES, völlig unabhängiges Objekt
        Spieler ork = new Spieler();
        ork.name = "Grumsh";
        ork.lebenspunkte = 45;
        ork.istVergiftet = true;

        // Werte wieder auslesen (Lesen)
        System.out.println("Kampf-Status:");
        System.out.println("Held: " + ritter.name + " (" +
                           ritter.lebenspunkte + " HP)");
        System.out.println("Gegner: " + ork.name + " (" + ork.lebenspunkte +
                           " HP)");

        // Wir können auch mit den Attributen ganz normal rechnen
        System.out.println("Der Ork greift an!");
        ritter.lebenspunkte = ritter.lebenspunkte - 15;

        System.out.println("Neue HP von Arthur: " + ritter.lebenspunkte);
    }
```

```
}
```

11.4 Arrays mit Objekten

Was ist, wenn wir nicht nur einen Feind haben, sondern eine ganze Armee? Wir können Arrays und Objekte wunderbar kombinieren. Wir bauen einfach ein Array, dessen Datentyp unsere eigene Klasse ist!

Hier lauert jedoch ein klassischer Anfängerfehler, der fast unweigerlich zu einer NullPointerException (Programmabsturz) führt. Wenn wir ein Objekt-Array mit new bauen, bauen wir *nur das Regal für die Zettel*. Die Schubladen sind anfangs alle leer (null). Wir müssen danach zwingend jede einzelne Schublade aufziehen und dort ein *eigenes, neues Objekt* hineinlegen!

```
public class ObjektArrayBeispiel {
    public static void main(String[] args) {

        // 1. Wir bauen ein Regal für 3 Spieler-Zettel
        // ACHTUNG: Hier werden noch KEINE Spieler-Objekte gebaut!
        Spieler[] orkArmee = new Spieler[3];

        // Wenn wir jetzt versuchen würden, auf orkArmee[0].name zuzugreifen,
        // würde das Programm abstürzen, weil in Schublade 0 noch 'null'
        // liegt!

        // 2. Wir füllen das Regal mithilfe einer for-Schleife mit echten
        // Objekten
        for (int i = 0; i < orkArmee.length; i++) {

            // JEDE Schublade bekommt ein BRANDNEUES Spieler-Objekt
            orkArmee[i] = new Spieler();

            // Jetzt können wir die Attribute füllen
            orkArmee[i].name = "Ork-Krieger Nr. " + (i + 1);
            orkArmee[i].lebenspunkte = 50;
            orkArmee[i].istVergiftet = false;
        }

        // Test-Ausgabe des zweiten Orks (Index 1)
        System.out.println("Wer steht in Reihe 2?");
        System.out.println(orkArmee[1].name); // Gibt "Ork-Krieger Nr. 2" aus
    }
}
```

11.5 Arrays und Objekte als Attribute (Verschachtelung)

Wir können das Konzept der Baupläne noch weiter auf die Spitze treiben. Ein Attribut innerhalb einer Klasse muss nicht zwangsläufig ein einfacher Datentyp (int) sein. Ein Bauplan kann als Bauteil auch andere Baupläne oder Arrays vorschreiben! So können wir extrem komplexe, verschachtelte Datenstrukturen erzeugen.

Stellen Sie sich vor, jeder Spieler in unserem Spiel soll ein eigenes Inventar (Rucksack) haben, in dem er bis zu 5 gefundene Gegenstände (Strings) speichern kann.

Zusätzlich hat der Spieler eine Waffe. Eine Waffe ist wiederum so komplex (Name, Schaden), dass sie einen eigenen Bauplan benötigt!

Schritt 1: Wir schreiben den Bauplan für die Waffe.

```
public class Waffe {
    String bezeichnung;
    int schaden;
}
```

Schritt 2: Wir erweitern unseren Spieler-Bauplan um die Verschachtelung.

```
public class KomplexerSpieler {
    String name;

    // Verschachtelung 1: Ein Array als Attribut
    String[] rucksack;

    // Verschachtelung 2: Ein anderes Objekt als Attribut
    Waffe aktuelleWaffe;
}
```

Schritt 3: Wir erwecken dieses gigantische Konstrukt in unserer main-Methode zum Leben:

```
public class VerschachtelungBeispiel {
    public static void main(String[] args) {
        // 1. Spieler erschaffen
        KomplexerSpieler held = new KomplexerSpieler();
        held.name = "Lancelot";

        // 2. Array für den Rucksack erschaffen und dem Attribut zuweisen
        held.rucksack = new String[5];
        held.rucksack[0] = "Heiltrank";
        held.rucksack[1] = "Fackel";
```

```
        // 3. Waffe erschaffen und befüllen
        Waffe schwert = new Waffe();
        schwert.bezeichnung = "Excalibur";
        schwert.schaden = 99;

        // 4. Die Waffe in die Hand des Spielers geben (Zuweisung der
        // Referenz)
        held.aktuelleWaffe = schwert;

        // --- Zugriff auf die tief verschachtelten Daten ---

        // Wir ketten die Punkt-Notation einfach aneinander!
        // "Gehe zum held, nimm seine aktuelleWaffe und lies dort die
        // bezeichnung"
        System.out.println("Held: " + held.name);
        System.out.println("Bewaffnet mit: " +
                           held.aktuelleWaffe.bezeichnung);
        System.out.println("Macht Schaden: " + held.aktuelleWaffe.schaden);

        // Zugriff auf das Rucksack-Array innerhalb des Objekts
        System.out.println("Erster Gegenstand im Rucksack: " +
        held.rucksack[0]);
    }
}
```

Herzlichen Glückwunsch! Sie haben sich das komplette Werkzeugset der imperativen Programmierung in Java angeeignet. Sie beherrschen den Kontrollfluss (Schleifen und Verzweigungen), können Aufgaben in Funktionen auslagern, Daten in Arrays verwalten und nun sogar Ihre eigenen, verschachtelten Datentypen konstruieren.

Alle theoretischen Vorlesungen sind hiermit abgeschlossen. Es wird Zeit für Ihre Meisterprüfung. Wir verlassen die Trockenübungen und bauen ein echtes, lauffähiges und interaktives Programm, das *all* diese Konzepte kombiniert.

Teil VII: Die Meisterprüfung (Praxisprojekt)

12 Praxisprojekt: Das Spiel "Dungeon-Schatzsuche"

Herzlichen Glückwunsch! Sie haben alle grundlegenden Werkzeuge der imperativen Programmierung kennengelernt. Sie wissen, wie man Daten speichert, Bedingungen prüft, Abläufe wiederholt und eigene Datentypen konstruiert. In der Werkstatt-Metapher gesprochen: Sie kennen nun jede Säge, jeden Hammer und jede Holzart.

Doch ein guter Handwerker zeichnet sich nicht dadurch aus, dass er seine Werkzeuge theoretisch benennen kann, sondern dadurch, dass er am Ende ein Möbelstück baut, das nicht zusammenbricht. In diesem finalen Kapitel legen wir die Theorie beiseite. Wir bauen Ihr erstes großes, interaktives Programm: Das textbasierte Konsolenspiel **"Dungeon-Schatzsuche"**.

12.1 Spielidee, Spielfeld und Vorbereitung

Bevor wir wild drauflos tippen, müssen wir – wie in Kapitel 1 gelernt – unser Problem analysieren und entwerfen.

Die Spielidee:

Der Spieler bewegt sich durch ein dunkles Verlies (den Dungeon). Er muss den verborgenen Schatz finden, ohne dabei in die Wände zu laufen oder von einem Monster besiegt zu werden. Das Spiel läuft rundenbasiert: Der Spieler tippt eine Richtung ein, die Figur bewegt sich, und das Spielfeld wird neu auf dem Bildschirm gezeichnet.

Das Spielfeld:

Wir repräsentieren den Dungeon als ein zweidimensionales Array (char[][]), das aus einzelnen Zeichen besteht.

- Ein Punkt . steht für einen leeren Boden.
- Eine Raute # steht für eine massive Wand.
- Ein X markiert den Schatz.

Wir legen in unserer IDE ein neues Projekt an und beginnen mit der Hauptklasse für unser Spiel:

```
public class DungeonSpiel {

    // Eine Hilfsfunktion, um uns ein fertiges 2D-Array (den Dungeon) zu
    // liefern
    public static char[][] baueDungeon() {
```

```
        char[][] feld = {
            {'#', '#', '#', '#', '#', '#', '#'},
            {'#', '.', '.', '.', '#', 'X', '#'},
            {'#', '.', '#', '.', '#', '.', '#'},
            {'#', '.', '.', '.', '.', '.', '#'},
            {'#', '#', '#', '#', '#', '#', '#'}
        };
        return feld;
    }

    // Wir bereiten schon einmal die main-Methode vor
    public static void main(String[] args) {
        System.out.println("Willkommen in der Dungeon-Schatzsuche!");
        char[][] spielfeld = baueDungeon();
        // Hier kommt später unsere Hauptschleife hin
    }
}
```

12.2 Definition der Datenstrukturen (Klassen)

Unser Spielfeld aus Wänden und leeren Feldern ist starr. Der Spieler und das Monster hingegen sind dynamisch: Sie bewegen sich, sie haben Namen und Lebenspunkte. Wir könnten versuchen, ihre X- und Y-Koordinaten mühsam als einzelne int-Variablen zu verwalten, aber als angehende Handwerksmeister nutzen wir unseren eigenen Datentyp (eine Klasse).

Wir erstellen eine neue Datei in unserem Projekt und nennen sie Spielfigur.java. Dieser Bauplan dient uns sowohl für unseren Helden als auch für das Monster.

```
// Die Klasse Spielfigur (reiner Datencontainer)
public class Spielfigur {
    String name;

    // Die aktuelle Position im 2D-Array (Zeile und Spalte)
    int x; // Spalte
    int y; // Zeile

    int lebenspunkte;
    char symbol; // Welches Zeichen repräsentiert die Figur auf dem Feld?
                 // (z.B. 'P' oder 'M')
}
```

12.3 Implementierung der Spiellogik (Funktionen)

Um zu verhindern, dass unsere main-Methode zu einem unlesbaren, tausend Zeilen

langen Chaos verkommt, lagern wir jede logische Aufgabe in eine eigene statische Funktion aus (Prozedurale Zerlegung). Wir fügen diese Funktionen in unsere Datei DungeonSpiel.java über der main-Methode ein.

Schritt 1: Das Spielfeld zeichnen

Wir brauchen eine Funktion, die das 2D-Array auf der Konsole ausdruckt. Der Clou: Wir wollen auch den Helden und das Monster sehen! Wir drucken also nicht stur das Array, sondern prüfen bei jedem Feld, ob zufällig gerade eine Figur darauf steht.

```
    public static void zeichneSpielfeld(char[][] feld, Spielfigur held,
                                         Spielfigur monster) {
        System.out.println("-------------------------");

        for (int zeile = 0; zeile < feld.length; zeile++) {
            for (int spalte = 0; spalte < feld[zeile].length; spalte++) {

                // Steht der Held genau auf dieser Koordinate?
                if (held.y == zeile && held.x == spalte) {
                    System.out.print(held.symbol + " ");
                }
                // Steht das Monster hier?
                else if (monster.y == zeile && monster.x == spalte) {
                    System.out.print(monster.symbol + " ");
                }
                // Ansonsten zeichne das normale Spielfeld (Boden, Wand oder
                // Schatz)
                else {
                    System.out.print(feld[zeile][spalte] + " ");
                }
            }
            System.out.println(); // Zeilenumbruch nach jeder Reihe
        }

        System.out.println("HP " + held.name + ": " + held.lebenspunkte);
        System.out.println("-------------------------");
    }
```

Schritt 2: Die Bewegung des Spielers (Verzweigungen)

Der Spieler drückt 'w' (oben), 'a' (links), 's' (unten) oder 'd' (rechts). Bevor wir die X- oder Y-Koordinate unserer Figur verändern, müssen wir zwingend prüfen, ob das Zielfeld eine Wand (#) ist. Wenn ja, blockieren wir die Bewegung.

```
    public static void bewegeFigur(Spielfigur figur, char eingabe,
                                   char[][] feld) {
        int zielX = figur.x;
        int zielY = figur.y;

        // Berechne die Zielkoordinate basierend auf der Eingabe
        switch (eingabe) {
            case 'n': zielY = zielY - 1; break; // Hoch (Zeile minus 1)
            case 's': zielY = zielY + 1; break; // Runter (Zeile plus 1)
            case 'w': zielX = zielX - 1; break; // Links (Spalte minus 1)
            case 'o': zielX = zielX + 1; break; // Rechts (Spalte plus 1)
            default: System.out.println("Ungültige Taste!"); return;
                                                 // Bricht die Funktion ab
        }

        // Kollisionsabfrage: Ist das Zielfeld eine Wand?
        if (feld[zielY][zielX] == '#') {
            System.out.println("Bumm! Du bist gegen eine Wand gelaufen.");
        } else {
            // Der Weg ist frei, wir überschreiben die alten Koordinaten
            figur.x = zielX;
            figur.y = zielY;
        }
    }
```

12.4 Zusammenbau und finales Testen

Wir haben das Feld, wir haben die Figuren, wir können zeichnen und wir können uns bewegen. Jetzt bauen wir in der main-Methode die "Game Loop" – die alles entscheidende while-Schleife, die das Spiel am Leben hält.

Studieren Sie diesen Code genau. Er ist das Konzentrat aus allem, was Sie in diesem Buch gelernt haben: Zuweisungen, Objekterzeugung mit new, Aufruf von Funktionen (Methoden), Scanner-Eingaben und if-Abfragen.

```
    import java.util.Scanner;

    public static void main(String[] args) {
        Scanner tastatur = new Scanner(System.in);
        char[][] spielfeld = baueDungeon();

        // 1. Objekte erschaffen und initialisieren
        Spielfigur spieler = new Spielfigur();
        spieler.name = "Ritter Lancelot";
        spieler.symbol = 'P'; // P für Player
        spieler.x = 1; // Start in Spalte 1
        spieler.y = 3; // Start in Zeile 3
```

```
        spieler.lebenspunkte = 100;

        Spielfigur feind = new Spielfigur();
        feind.name = "Goblin";
        feind.symbol = 'M'; // M für Monster
        feind.x = 3;
        feind.y = 1;
        feind.lebenspunkte = 30;

        boolean spielLaeuft = true;

        // 2. Die Hauptschleife (Game Loop)
        while (spielLaeuft) {

            // Alles auf den Bildschirm zeichnen
            zeichneSpielfeld(spielfeld, spieler, feind);

            // Prüfen, ob der Spieler gewonnen hat (Steht er auf dem 'X'?)
            if (spielfeld[spieler.y][spieler.x] == 'X') {
                System.out.println("Gewonnen! Du hast den Schatz gefunden!");
                spielLaeuft = false; // Bricht die Schleife ab
                break;
            }

            // Prüfen, ob der Spieler in das Monster gelaufen ist
            if (spieler.x == feind.x && spieler.y == feind.y) {
                System.out.println("Ein wildes Monster greift an! " +
                                   "Du verlierst 50 HP.");
                spieler.lebenspunkte = spieler.lebenspunkte - 50;

                if (spieler.lebenspunkte <= 0) {
                    System.out.println("Du bist gestorben. GAME OVER.");
                    spielLaeuft = false;
                    break;
                }
            }

            // Eingabe des Nutzers abwarten
            System.out.print("Wohin willst du gehen? (n/s/w/o): ");

            // Nimmt das allererste Zeichen aus der Eingabe
            char eingabe = tastatur.next().charAt(0);

            // Spiellogik aufrufen
            bewegeFigur(spieler, eingabe, spielfeld);

        } // Ende der while-Schleife

        System.out.println("Danke fürs Spielen!");
    }
```

Wie das Spiel abläuft:

Kopieren Sie diesen kompletten Code in Ihre IDE und starten Sie ihn. Sie werden das Gitter sehen, das von Wänden # umrandet ist. Ihr Charakter P steht unten links, das Monster M oben in der Mitte, und der Schatz X wartet oben rechts. Sie können n (hoch), o (rechts) usw. eintippen und Enter drücken. Sie können nicht durch Wände gehen, und wenn Sie das Monster berühren, verlieren Sie Lebenspunkte. Wenn Sie das X erreichen, endet das Programm mit einer Siegesmeldung.

Ihre erste Software!

Spielen Sie damit herum! Ändern Sie das Layout im Array baueDungeon(). Geben Sie dem Goblin mehr Lebenspunkte. Lassen Sie die Wände aus einem anderen Zeichen bestehen. Sie sind jetzt nicht länger nur ein Anwender, der konsumiert, was andere geschrieben haben. Sie sind der Handwerker. Sie haben die volle Kontrolle über die Maschine.

12.5 Epilog: Das Fundament steht – Wie Ihre Reise weitergeht

Herzlichen Glückwunsch! Sie haben es geschafft. Mit der Fertigstellung Ihres ersten eigenen Dungeon-Spiels endet das erste Buch unserer Reihe.

Lassen Sie uns für einen Moment innehalten und zurückblicken, was Sie erreicht haben. Als Sie dieses Buch aufschlugen, war Programmieren vielleicht noch ein Mysterium für Sie. Heute beherrschen Sie das absolute Fundament der Softwareentwicklung: das **imperative Handwerk**. Sie können dem Computer präzise Anweisungen erteilen. Sie wissen, wie man Daten in Variablen und Arrays speichert, wie man den Kontrollfluss mit if-Abfragen und Schleifen lenkt und wie man wiederkehrende Aufgaben in statische Methoden auslagert. Sie sind vom reinen Anwender zum Schöpfer geworden – Sie haben die Kontrolle über die Maschine übernommen.

Doch am Ende unseres Praxisprojekts haben Sie vermutlich auch eine architektonische Grenze gespürt.

Je mehr Code wir geschrieben haben, desto schwieriger wurde es, den Überblick zu behalten. Unsere Spielfigur, die Monster und die Wände bestanden aus unzähligen, lose verknüpften Arrays und Variablen. Unsere statischen Funktionen haben wild auf diese wehrlosen Datencontainer zugegriffen. In kleinen Programmen wie unserem Dungeon lässt sich dieses Chaos noch im Kopf beherrschen. Aber stellen Sie sich vor, Sie

programmieren ein System mit Millionen von Codezeilen – eine Bankensoftware oder ein globales Logistiknetzwerk. Mit rein imperativen Mitteln bricht ein solches Projekt unweigerlich zusammen.

Hier endet das Handwerk und die Architektur beginnt. Ihr Weg zum Meister-Programmierer führt Sie nun durch die drei verbleibenden Bände dieser Reihe, die jeweils einen gewaltigen Paradigmenwechsel bedeuten:

- **Band 2: Die Architektur der Objekte (Objektorientierte Programmierung)**
  Im direkt folgenden Buch lösen wir das Chaos der losgelösten Daten und Funktionen. Wir werden unseren Klassen beibringen, sich selbst zu verteidigen. Sie werden lernen, wie man Daten und Verhalten zu intelligenten, eigenständigen Einheiten verschmilzt: den *Objekten*. Sie erlernen die Geheimniskrämerei (Kapselung), geben Wissen an andere Klassen weiter (Vererbung) und bauen flexible, hochgradig wartbare Systeme. Sie steigen vom Handwerker zum Software-Architekten auf.
- **Band 3: Der Fluss der Daten (Funktionale Programmierung)**
  Wenn Sie die objektorientierte Welt gemeistert haben, werden Sie feststellen, dass riesige Mengen an veränderlichen Objekten neue Probleme mit sich bringen. Im dritten Band verlassen wir das imperative Befehlsdiktat ("Wie mache ich es?") und lernen die *deklarative Programmierung* ("Was will ich haben?"). Sie werden lernen, wie man unveränderliche Daten wie auf einem Fließband durch intelligente Pipelines (Streams) schickt, ohne jemals wieder fehleranfällige Schleifen schreiben zu müssen.
- **Band 4: Die Choreografie der Threads (Parallele Programmierung)**
  Den krönenden Abschluss bildet die Meisterklasse der modernen Informatik. Bisher lief all unser Code gemütlich nacheinander ab. Aber moderne Prozessoren besitzen dutzende Rechenkerne. In diesem Band lernen Sie, wie Sie Ihr Programm in hunderte *Threads* aufteilen, die alle gleichzeitig arbeiten. Sie werden das Chaos geteilter Ressourcen bändigen und Hochleistungs-Software entwerfen, die die Grenzen der Hardware sprengt.

Sie haben das Fundament gegossen. Der Beton ist fest. Es ist an der Zeit, darauf Wolkenkratzer zu errichten.