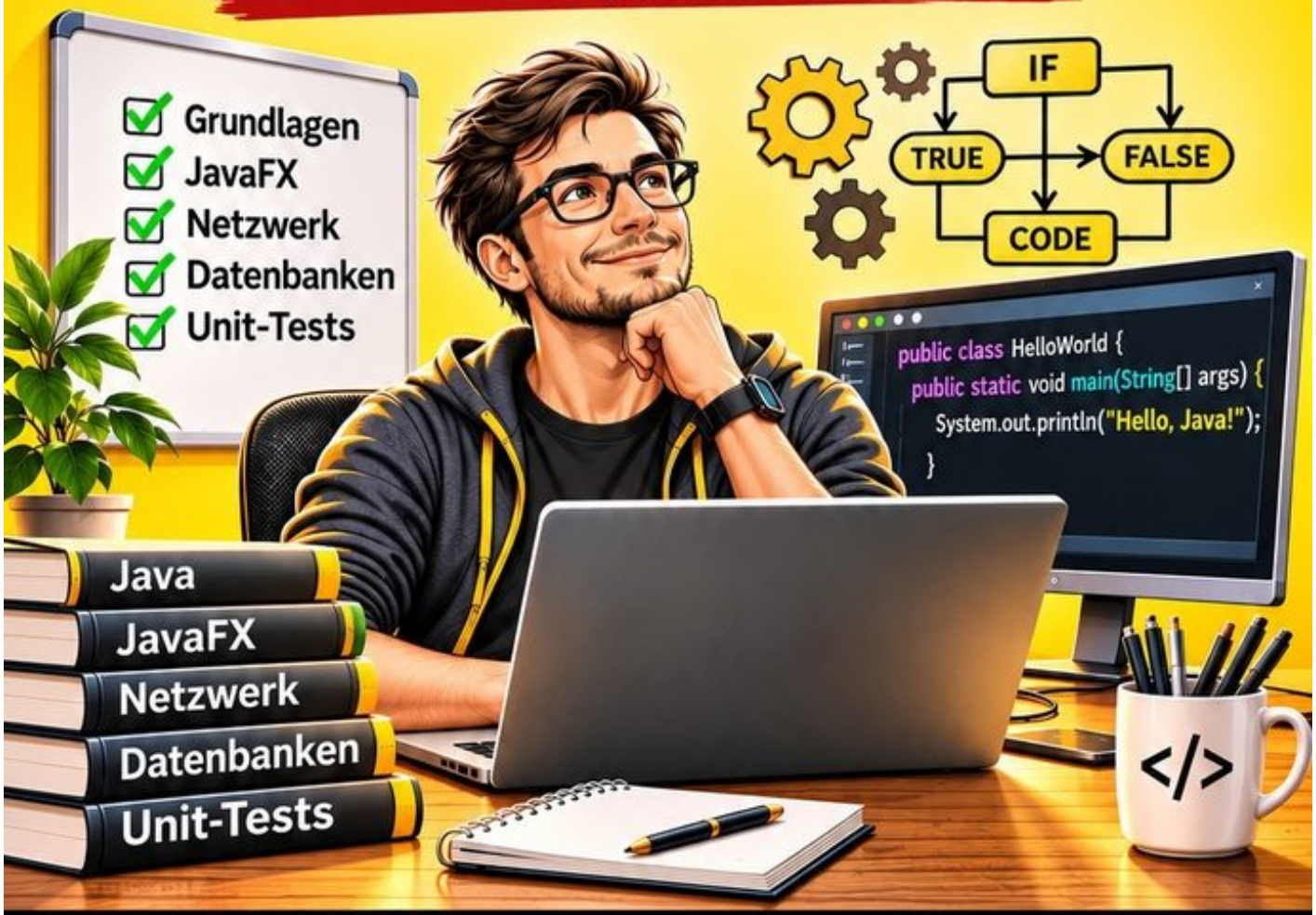
GANZ EINFACH

Softwareentwicklung mit Java

Von den Grundlagen bis zu JavaFX, Netzwerkprogrammierung, Datenbanken und Unit-Tests

Ein unabhängiges Praxisbuch.



Danilo Sieren

Danilo Sieren

Ganz Einfach

Softwareentwicklung mit Java

Programmieren verstehen und anwenden
Vom ersten Programm zum eigenen Projekt

Überarbeitete und erweiterte
Neuauflage 2026

Willkommen

Java hilft dir, Programme zu bauen, deren Daten und Abläufe auch bei wachsendem Umfang nachvollziehbar bleiben. Du kannst damit kleine Werkzeuge für den eigenen Rechner schreiben, Dateien auswerten, Desktopoberflächen entwickeln und mit Datenbanken oder Webdiensten arbeiten. Dieses Buch führt dich von einem ersten Konsolenprogramm zu einer geprüften Bestandsverwaltung. Es setzt keine Erfahrung mit einer anderen Programmiersprache voraus.

Du lernst nicht nur Schreibweisen. Zu jedem wichtigen Baustein gehört die Frage, welche Eingaben er akzeptiert, welches Ergebnis er verspricht und wie er sich bei einem Fehler verhält. Eine Liste ist erst dann das passende Werkzeug, wenn du weißt, ob Reihenfolge und doppelte Einträge erlaubt sind. Eine Datei ist erst dann zuverlässig gespeichert, wenn du auch einen fehlgeschlagenen Schreibvorgang bedacht hast. Diese Zusammenhänge machen aus einzelnen Anweisungen ein verständliches Programm.

W.1 So arbeitest du mit dem Buch

Lege für jedes Kapitel einen eigenen Übungsordner an. Vollständige Java-Dateien sind mit ihrem Dateinamen bezeichnet. Speichere sie als reinen Text mit der Endung .java. Übernimm gerade Anführungszeichen aus dem Code; typografische Anführungszeichen aus einer Textverarbeitung sind kein Ersatz. Terminalbefehle gehören in die Eingabeaufforderung, PowerShell oder das Terminal deines Betriebssystems.

Die ersten Beispiele verwenden bewusst die ausführliche Form mit benannter Klasse und public static void main(String[] args). Dadurch erkennst du später dieselbe Struktur in größeren

Projekten wieder. Seit Java 25 gibt es zusätzlich kompakte Quelldateien und Instanzmethoden als Programmeinstieg. Du lernst beide Formen kennen und kannst entscheiden, welche zur Aufgabe passt. Die ausführliche Form ist weiterhin gültig. [D04]

Führe ein Beispiel zunächst unverändert aus. Notiere die Ausgabe und ändere danach eine einzelne Voraussetzung. Bei einer Bestandsberechnung kann das eine Menge von null, eine negative Menge oder ein fehlender Artikel sein. Sage vor dem Start voraus, was passieren sollte. Wenn Ergebnis und Erwartung voneinander abweichen, hast du einen konkreten Ausgangspunkt für die Fehlersuche.

W.2 Versionsstand und Lernpfad

Redaktionsstand ist der 2. November 2026. Java 25 ist eine LTS-Linie bei Anbietern, die dafür langfristige Wartung anbieten; Java 27 ist die aktuelle halbjährliche Hauptversion. LTS bezeichnet eine Wartungszusage des jeweiligen Anbieters, keine zusätzliche Spracheigenschaft. Wähle für neue Lernprojekte ein aktuell gepflegtes JDK 25. Die gesonderte Einordnung neuerer Versionen erläutert auch Funktionen, die noch Preview-Status besitzen. [D01, D02]

Der Kern des Buches verwendet stabile Sprach- und Bibliotheksfunktionen. Die meisten Beispiele lassen sich bereits mit Java 21 übersetzen. Wo Java 25 erforderlich ist, steht das unmittelbar beim Beispiel. Preview-Funktionen bilden keine Voraussetzung für das Abschlussprojekt. Eine neue Versionsnummer allein sagt weder etwas über die Qualität deines Programms noch über die Verfügbarkeit sämtlicher Zusatzbibliotheken aus.

Die Kapitel bauen fachlich aufeinander auf. Zunächst richtest du das JDK ein und lernst Werte, Entscheidungen und Methoden kennen. Danach folgen Objekte, Collections, Generics und Streams. Die zweite Hälfte behandelt Dateien, Tests,

Nebenläufigkeit, Oberflächen, Netzwerke und Datenbanken. Im Anhang verbindest du die Teile zu einem vollständigen Projekt. Zusätzliche Werkstätten behandeln Grenzen und Fehlersuche, damit du die Beispiele auf eigene Aufgaben übertragen kannst.

Du brauchst weder einen Cloudzugang noch ein kostenpflichtiges Konto. Das Abschlussprojekt arbeitet mit selbst erzeugten lokalen Beispieldaten. Der Netzwerkteil beginnt ebenfalls mit einem lokalen Dienst. Für JUnit und das JDBC-Beispiel benötigst du zusätzliche Bibliotheken. Deren Versionen werden im Projekt ausdrücklich festgehalten, statt stillschweigend die jeweils neueste Ausgabe zu laden.

Inhalt

Willkommen 2
1 Java einrichten und den ersten Start verstehen 7
2 Werte Typen und genaue Berechnungen 12
3 Texte Eingaben und nachvollziehbare Entscheidungen 18
4 Schleifen Methoden und kleine Verträge 23
5 Klassen Objekte und gültiger Zustand 28
6 Records Enums und unveränderliche Modelle 33
7 Vererbung Interfaces und austauschbares Verhalten 38
8 Arrays Listen und sichere Momentaufnahmen 43
9 Mengen Maps und verlässliche Schlüssel 47
10 Generics und verständliche Typschnittstellen 51
11 Lambdas Streams und kontrollierte Auswertungen 56
12 Rekursion Algorithmen und gemessener Aufwand 60
13 Ausnahmen Ressourcen und verständliche Fehler 64
14 Dateien Pfade und ein sicheres Austauschformat 68
15 Projekte Pakete und reproduzierbare Builds 72
16 Tests mit JUnit und begründeten Erwartungen 75
17 Zeit Konfiguration und nachvollziehbare Protokolle 80
18 Threads gemeinsamer Zustand und Abbruch 84
19 Executor virtuelle Threads und begrenzte Arbeit 88
20 Annotationen Reflection und Modulgrenzen 93
21 Eine bedienbare Oberfläche mit Swing 97
22 JavaFX und die Wahl einer Desktoparchitektur 101
23 HTTP mit dem Standardclient verstehen 104
24 Ein lokaler Dienst und klare Protokollgrenzen 108
25 Relationale Daten und JDBC 113

26 Transaktionen konkurrierende Änderungen und Migrationen...118
27 Persistenzschichten und ein wartbarer Entwurf 124
28 Programme ausliefern und über längere Zeit pflegen 127
29 Modernes Java bis Version 27 einordnen 130
30 Debugging von der Beobachtung zur Ursache 133
A Praxisprojekt eine geprüfte Bestandsverwaltung 137
B Vertiefende Werkstatt mit vollständigen Bausteinen 157
C Lösungen und begründete Erwartungen 168
D Fehler gezielt diagnostizieren 178
E Nachschlagen und eigene Aufgaben planen 184
F Weitere vollständige Anwendungen 189
Detailliertes Inhaltsverzeichnis 204
Sachregister 214
Quellen und weiterführende Dokumentation 221
Impressum.....227

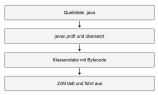
1 Java einrichten und den ersten Start verstehen

1.1 JDK JVM und Quelltext unterscheiden

Ein Java-Programm entsteht aus Textdateien. Der Compiler javac prüft deren Syntax und Typbeziehungen und erzeugt Klassendateien mit Bytecode. Die Java Virtual Machine lädt diesen Bytecode und führt ihn aus. Eine konkrete JVM kann dabei interpretieren und häufig ausgeführte Teile in Maschinencode übersetzen. Diese Arbeitsweise erlaubt Optimierungen zur Laufzeit; sie bedeutet nicht, dass jede Anweisung bei jedem Aufruf erneut vollständig übersetzt wird. [D05]

Das Java Development Kit enthält die Laufzeit und Entwicklungswerkzeuge, darunter java, javac, jar, javadoc und jshell. Für das Buch installierst du ein JDK. Eine alleinige Laufzeit reicht zum Start vorhandener Programme, stellt aber nicht automatisch einen Compiler bereit. Die Bezeichnungen Java SE und JDK hängen zusammen: Java SE beschreibt unter anderem Sprache und Standardbibliothek, ein JDK liefert eine konkrete Implementierung und Werkzeuge.

Bytecode macht ein Programm nicht von jeder Umgebung unabhängig. Eine Anwendung kann bestimmte Betriebssystemfunktionen, native Bibliotheken oder eine passende Datenbank voraussetzen. Auch die benötigte Java-Version zählt. Eine Klasse, die für eine neue Zielversion übersetzt wurde, lässt sich nicht ohne Weiteres auf einer älteren JVM laden. Portabilität musst du deshalb mit den tatsächlich unterstützten Zielumgebungen prüfen.



1.2 Eine passende Distribution installieren

Eclipse Temurin ist eine mögliche OpenJDK-Distribution. Lade das JDK von der offiziellen Projektseite und wähle Betriebssystem, Prozessorarchitektur und die gewünschte Hauptversion aus. Für einen Mac mit Apple Silicon ist das AArch64 beziehungsweise ARM64; auf einem Intel-Mac brauchst du die entsprechende x64-Ausgabe. Unter Windows und Linux muss die Architektur ebenfalls zum Rechner passen. Ein Archiv entpackst du an einen selbst gewählten Ort, ein Installationspaket führt durch die Einrichtung. [D03]

Vermeide parallele, unübersichtliche Installationen am Anfang. Prüfe nach der Installation in einem neuen Terminal java --version und javac --version. Beide Befehle sollen zur vorgesehenen Hauptversion gehören. Wenn nur java funktioniert, findet die Befehlsuche möglicherweise eine ältere Laufzeit. JAVA_HOME bezeichnet üblicherweise das JDK-Verzeichnis, während PATH bestimmt, wo die Shell ausführbare Dateien sucht. Eine Änderung an JAVA_HOME allein ändert die Befehlsuche nicht zwingend.

Unter macOS kann /usr/libexec/java_home -V registrierte Installationen anzeigen. Unter Windows helfen Get-Command java und Get-Command javac in PowerShell bei der Suche nach den tatsächlich gestarteten Programmen. Unter Linux zeigt command -v java den gefundenen Pfad. Prüfe zusätzlich die ausgegebene Version. Ein symbolischer Link kann auf ein anderes Programmziel verweisen, als der sichtbare Pfad zunächst vermuten lässt.

1.3 Hallo Java vollständig ausführen

Speichere die folgende Datei als Hallo.java in deinem Übungsordner. Der Name der öffentlichen Klasse stimmt

einschließlich Großschreibung mit dem Dateinamen überein. Die Klammern begrenzen Klasse und Methode. Eine Anweisung endet hier mit einem Semikolon.

Vollständige Datei Hallo.java:

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

Öffne ein Terminal genau in diesem Ordner. Übersetze die Datei mit javac -encoding UTF-8 Hallo.java und starte anschließend java Hallo. Beim zweiten Befehl steht weder .java noch .class hinter dem Klassennamen. Der Launcher sucht die übersetzte Klasse und ruft deren Einstiegsmethode auf. Die erwartete Ausgabe ist Hallo Java.

Du kannst die kleine Quelldatei auch unmittelbar mit java Hallo.java starten. Der Quelltextmodus übersetzt für die Ausführung, ohne dass du vorher selbst javac aufrufen musst. Halte die beiden Arbeitsweisen zunächst auseinander. java Hallo verlangt eine bereits auffindbare Klasse, java Hallo.java bezeichnet eine Quelldatei. Die Dateiendung verändert hier die Bedeutung des Aufrufs. [D06]

1.4 Fehlermeldungen als Arbeitsmaterial lesen

Ein Compilerfehler enthält gewöhnlich einen Dateinamen, eine Zeilennummer und eine Beschreibung. Lies zuerst die erste Meldung. Ein vergessenes Anführungszeichen kann viele Folgemeldungen auslösen, weil der Compiler den Rest der Datei anders versteht. Korrigiere die Ursache und übersetze erneut, bevor du sämtliche späteren Meldungen einzeln bearbeitest.

Eine Meldung über eine nicht gefundene Hauptklasse betrifft häufig den Startpfad oder Klassennamen. Prüfe, ob die .class-Datei tatsächlich im erwarteten Ordner liegt und ob du dort

startest. Eine Meldung über eine nicht unterstützte Klassenversion deutet auf unterschiedliche Compiler- und Laufzeitversionen. Das behebst du durch eine passende Toolchain oder eine bewusst gesetzte Zielversion, nicht durch Umbenennen der Klassendatei.

Mit javac --release 21 kannst du beispielsweise eine Java-21-Zielplattform anfordern, wenn dein Quelltext ohne neuere Funktionen auskommt. Diese Option begrenzt auch die verfügbare Standard-API. Nur eine ältere Bytecodeversion auszuwählen, während du neue Bibliotheksmethoden verwendest, wäre keine zuverlässige Kompatibilitätsprüfung. [D07]

1.5 IDE und Terminal zusammen nutzen

Eine Entwicklungsumgebung bietet Navigation, automatische Ergänzung, Refactorings und einen Debugger. Richte dort dasselbe JDK ein, das du im Terminal verwendest. Projekteinstellungen, Buildwerkzeug und IDE können jeweils eine eigene Java-Auswahl besitzen. Ein grüner Startknopf beweist daher noch nicht, dass dein Projekt außerhalb der IDE übersetzbar ist.

Für die ersten Schritte genügt ein Texteditor. Später übernimmt Maven den gemeinsamen Build. Behalte einen funktionierenden Terminalaufruf bei, damit du Probleme eingrenzen kannst. Wenn ein Programm im Terminal läuft, aber in der IDE nicht, vergleiche zunächst JDK, Arbeitsverzeichnis, Umgebungsvariablen und Argumente. Ändere erst danach den Fachcode.

1.6 Werkstatt für einen nachvollziehbaren Start

Erzeuge absichtlich einen Syntaxfehler, indem du das Semikolon hinter println entfernst. Lies die Meldung und stelle die ursprüngliche Datei wieder her. Ändere danach nur den

Ausgabetext und übersetze erneut. Wenn beim Start der alte Text erscheint, läuft wahrscheinlich eine andere oder veraltete Klassendatei.

Notiere in einer kleinen Projektdatei die verwendete JDK-Hauptversion und die beiden Startbefehle. Gib den Ordner an eine zweite Person oder kopiere ihn in einen anderen eigenen Ordner. Ein erfolgreicher Start mit denselben Befehlen ist ein erster praktischer Nachweis, dass dein Programm nicht zufällig von einer versteckten IDE-Einstellung abhängt.

2 Werte Typen und genaue Berechnungen

2.1 Was eine Variable festlegt

Eine Variable besitzt einen Typ, einen Namen und einen Wert. int menge = 3 legt eine lokale Variable für eine 32-Bit-Ganzzahl an. Der Typ bestimmt erlaubte Werte und Operationen. Er beschreibt keine beliebige Schublade, in die du später Text statt einer Zahl legen kannst. Eine Zuweisung wie menge = 5 verändert den Wert derselben Variable.

Die acht primitiven Typen heißen byte, short, int, long, float, double, char und boolean. Klassen und Arrays gehören zu den Referenztypen. Eine Referenzvariable enthält einen Referenzwert, über den ein Objekt erreichbar sein kann, oder null. Du arbeitest in Java nicht mit frei berechenbaren Speicheradressen. Diese Trennung wird beim Kopieren und Vergleichen von Objekten wichtig. [D08]

Lokale Variablen müssen vor ihrer Verwendung eindeutig initialisiert sein. Felder eines neu erzeugten Objekts und Elemente eines neuen Arrays erhalten dagegen definierte Standardwerte, beispielsweise null für Referenzen. Verlasse dich im eigenen Modell nicht unnötig auf implizite Werte. Ein sinnvoller Konstruktor macht die gültigen Anfangsbedingungen sichtbar.

2.2 Ganzzahlen und Überläufe

int reicht von minus 2 hoch 31 bis 2 hoch 31 minus 1. long besitzt einen größeren, aber ebenfalls endlichen Bereich. Gewöhnliche Ganzzahlarithmetik wirft bei einem Überlauf nicht automatisch eine Ausnahme. Deshalb kann eine Rechnung einen formal gültigen, fachlich falschen Wert erzeugen. Math.addExact und

Math.multiplyExact machen einen Überlauf durch ArithmeticException sichtbar.

Vollständige Datei Kapazitaet.java:

```
public class Kapazitaet {
  public static void main(String[] args) {
    long dateien = 80_000L;
    long bytesProDatei = 30_000_000L;
    long gesamt = Math.multiplyExact(
      dateien, bytesProDatei);
    System.out.println(gesamt);
    System.out.println(gesamt / 1_000_000_000L);
  }
}
```

Die erste Ausgabe lautet 2400000000000, die zweite 2400. Die zweite Rechnung verwendet die dezimale Einheit Gigabyte. Für Gibibyte würdest du durch 1024 hoch 3 teilen und das Ergebnis ausdrücklich GiB nennen. Einheiten sind Teil der Fachlogik. Ein korrekter Operator kann eine falsche Einheit nicht ausgleichen.

Ganzzahldivision schneidet den Bruchteil in Richtung null ab. Daher ergibt -7 / 3 den Wert -2. Bei einer mathematischen Abrundung nach unten wäre das Ergebnis -3; dafür existiert Math.floorDiv. Prüfe bei negativen Werten bewusst, welche Regel deine Aufgabe benötigt. Der Restoperator % ist ebenfalls an die Java-Divisionsregeln gebunden.

2.3 Gleitkommazahlen und Geldbeträge

Ein double stellt viele Dezimalbrüche nur näherungsweise dar. Das ist für Messwerte oft sinnvoll. Es bedeutet jedoch, dass 0.1 + 0.2 nicht zwingend exakt denselben double-Wert wie 0.3 besitzt. Vergleiche Messwerte mit einer zur Aufgabe passenden absoluten oder relativen Toleranz. Eine feste Toleranz von 0.0001 ist keine allgemeingültige Regel für jede Größenordnung.

Für dezimale Geldrechnungen eignet sich BigDecimal, wenn du Rundung und Skala ausdrücklich festlegst. Erzeuge Dezimalwerte vorzugsweise aus Text oder mit valueOf. Der Konstruktor mit double übernimmt dessen bereits vorhandene binäre Näherung. Eine Division ohne Rundungsregel kann bei einer nicht endlichen Dezimaldarstellung eine Ausnahme auslösen. [D09]

Vollständige Datei Preis.java:

```
import java.math.BigDecimal;
import java.math.RoundingMode;

public class Preis {
  public static void main(String[] args) {
    var einzel = new BigDecimal("19.95");
    var anzahl = BigDecimal.valueOf(3);
    var brutto = einzel.multiply(anzahl);
    var netto = brutto.divide(
      new BigDecimal("1.19"),
      2, RoundingMode.HALF_UP);
    System.out.println(brutto.toPlainString());
    System.out.println(netto.toPlainString());
  }
}
```

Das Beispiel gibt 59.85 und 50.29 aus. Die Rundungsregel ist eine Annahme dieses Beispiels, keine universelle Vorgabe für Rechnungen oder Steuern. Lege in einer echten Anwendung fest, ob pro Position oder erst für die Gesamtsumme gerundet wird. Unterschiedliche fachliche Regeln können trotz gleicher Ausgangsbeträge verschiedene Ergebnisse liefern.

2.4 var und final richtig einordnen

var erlaubt bei lokalen Variablen eine Typableitung aus dem Initialisierer. Aus var name = "Ada" wird eine String-Variable, keine dynamisch typisierte Variable. Ein späteres name = 42 bleibt unzulässig. Nutze var, wenn der Typ aus der rechten Seite gut erkennbar ist. Bei einem unklar benannten Methodenaufruf kann eine ausgeschriebene Typangabe verständlicher sein.

final verhindert eine erneute Zuweisung an dieselbe Variable. Bei einer Referenz verhindert es keine Änderungen am referenzierten Objekt. Eine final List kann weiterhin Elemente aufnehmen, sofern die konkrete Liste veränderlich ist. Unveränderliche Werte und unveränderliche Referenzen sind verschiedene Eigenschaften. Du wirst sie beim Entwurf von Records und Collections wieder unterscheiden.

2.5 Sichtbarkeit und Lebensdauer auseinanderhalten

Ein lokaler Name gilt nur in seinem lexikalischen Bereich. Nach dem Ende eines Blocks kannst du ihn außerhalb nicht mehr verwenden. Daraus folgt nicht, dass ein referenziertes Objekt genau in diesem Moment gelöscht wird. Wenn andere Referenzen es weiterhin erreichen, bleibt es erreichbar. Selbst bei unerreichbaren Objekten bestimmt die Garbage Collection den Zeitpunkt der Speicherfreigabe.

Garbage Collection ersetzt auch kein Schließen einer Datei oder Datenbankverbindung. Ein Objekt kann eine externe Ressource besitzen, deren rechtzeitige Freigabe für die Anwendung wesentlich ist. Solche Ressourcen behandelst du mit ausdrücklich geregelter Lebensdauer. Das nächste Kapitel über Methoden und später try-with-resources liefern dafür die passenden Strukturen.

2.6 Werkstatt mit Grenzwerten

Berechne 50000 mal 50000 zunächst mit int und danach mit Math.multiplyExact. Erkläre, weshalb ein negativer Wert beim ersten Versuch möglich ist und der zweite Versuch stattdessen fehlschlägt. Ändere die Operanden schließlich in long. Achte darauf, dass die Multiplikation bereits im größeren Typ stattfindet; ein nachträglicher Cast eines schon übergelaufenen int repariert das Ergebnis nicht.

Vergleiche außerdem new BigDecimal("2.0") und new BigDecimal("2.00") mit equals und compareTo. equals berücksichtigt auch die Skala, compareTo die numerische Ordnung. Schreibe auf, welcher Vergleich für eine Preisgrenze und welcher für eine genau definierte Darstellung gebraucht wird. Die erwartete Regel gehört in den Test, bevor du die Implementierung auswählst.

2.7 Dezimaldarstellung und Eingabesprache

Eine technische Datendatei kann den Punkt als Dezimaltrennzeichen festlegen, während eine Oberfläche eine deutschsprachige Eingabe mit Komma erlaubt. Diese beiden Regeln gehören an verschiedene Grenzen. Ersetze nicht pauschal jedes Komma durch einen Punkt, wenn auch Tausendertrennzeichen oder mehrere Zahlen im Text vorkommen können.

Für eine lokalisierte Zahleneingabe wählst du einen Parser mit passender Locale und prüfst, ob die gesamte Eingabe verarbeitet wurde. Ein teilweise erfolgreich gelesenes Präfix soll nicht unbemerkt den restlichen falschen Text verschwinden lassen. Die interne Rechnung verwendet anschließend einen eindeutig festgelegten Zahlentyp. Die Darstellung am Bildschirm folgt wieder der vorgesehenen Sprache.

Bewahre Geldbetrag und Währung getrennt und ausdrücklich auf. 19.95 ohne Währung und Rundungsregel ist für einen Preisvergleich unvollständig. Auch ein BigDecimal allein verhindert nicht, dass du Euro und Dollar versehentlich addierst. Ein kleines eigenes Wertmodell kann diese Zusammenhänge besser festhalten als mehrere lose Variablen.

3 Texte Eingaben und nachvollziehbare Entscheidungen

3.1 Text ist mehr als eine Zeichenfolge im Alltag

String ist unveränderlich. Methoden wie strip und replace liefern einen neuen String; sie verändern den vorhandenen Wert nicht. Wenn du das Ergebnis behalten willst, musst du es zuweisen. Diese Eigenschaft vereinfacht das gemeinsame Verwenden von Texten, hebt aber die Verantwortung für passende Eingabegrenzen nicht auf. Ein riesiger fremder Text bleibt auch als unveränderlicher Wert ein Ressourcenproblem. [D10]

Ein char ist eine UTF-16-Codeeinheit und nicht immer ein vollständiges Unicode-Zeichen. Ein Zeichen außerhalb der grundlegenden mehrsprachigen Ebene benötigt zwei solche Einheiten. String.length zählt Codeeinheiten. codePointCount zählt Unicode-Codepunkte; sichtbare Schriftzeichen können wiederum aus mehreren Codepunkten bestehen. Entscheide daher ausdrücklich, ob eine Begrenzung Speicher, Codepunkte oder sichtbare Darstellung betrifft.

Bei Benutzereingaben kann Unicode-Normalisierung sinnvoll sein, etwa für den Vergleich gleich aussehender Namen. Verändere Originaldaten nicht ohne eine begründete Regel. Ein Suchschlüssel darf normalisiert werden, während du den ursprünglichen Anzeigenamen separat bewahrst. Für technische Schlüssel sind oft eng begrenzte, dokumentierte Zeichenmengen einfacher als eine scheinbar universelle Umwandlung.

3.2 Gleichheit und null behandeln

Bei Referenzen vergleicht == die Identität der referenzierten Objekte beziehungsweise null. Für den Inhalt von Strings verwendest du equals. Dass zwei Stringliterale zufällig dieselbe Referenz besitzen, macht == nicht zum richtigen Inhaltsvergleich. Objects.equals(a, b) ist hilfreich, wenn einer der Werte null sein darf. [D11]

Ein leerer String und null bedeuten nicht dasselbe. Der leere String ist ein vorhandener Text mit Länge null. null bezeichnet eine fehlende Referenz. Du kannst außerdem entscheiden, dass ein Name aus Leerzeichen fachlich ungültig ist. Diese Regeln müssen an einer Eingabegrenze überprüft werden, bevor späterer Code sich auf einen gültigen Namen verlässt.

3.3 Eine Entscheidung in einer Methode ausdrücken

Das folgende Programm entscheidet anhand eines Bestands und einer Mindestmenge. Negative Werte gelten als ungültige Eingabe. Danach wird eine Menge von null als leer gemeldet. Erst anschließend wird die Mindestgrenze geprüft. Die Reihenfolge ist Teil der fachlichen Bedeutung.

Vollständige Datei Bestandsstatus.java:

```
public class Bestandsstatus {
  static String status(int menge, int minimum) {
    if (menge < 0 || minimum < 0) {
      throw new IllegalArgumentException(
        "Negative Menge");
    }
    if (menge == 0) {
      return "leer";
    }
    if (menge < minimum) {
      return "knapp";
    }
    return "ausreichend";
  }

  public static void main(String[] args) {
    System.out.println(status(0, 3));
    System.out.println(status(2, 3));
    System.out.println(status(3, 3));
  }
}
```

Die Ausgaben lauten leer, knapp und ausreichend. Bei minimum gleich null bleibt ein Bestand von null trotzdem leer. Das ist eine ausdrückliche Entscheidung des Modells. Wenn deine Fachaufgabe null Mindestbestand anders behandelt, änderst du Regel und Tests gemeinsam. Verstecke diese Bedeutung nicht in einer zufälligen Reihenfolge mehrerer Bedingungen.

3.4 Kurzschlussauswertung nutzen

Bei a && b wird b nur ausgewertet, wenn a wahr ist. Bei a || b wird b nur ausgewertet, wenn a falsch ist. Damit lässt sich beispielsweise text != null && !text.isBlank() sicher auswerten. Der zweite Teil wird bei null nicht aufgerufen. Vertauschst du die Teile, kannst du eine NullPointerException erzeugen.