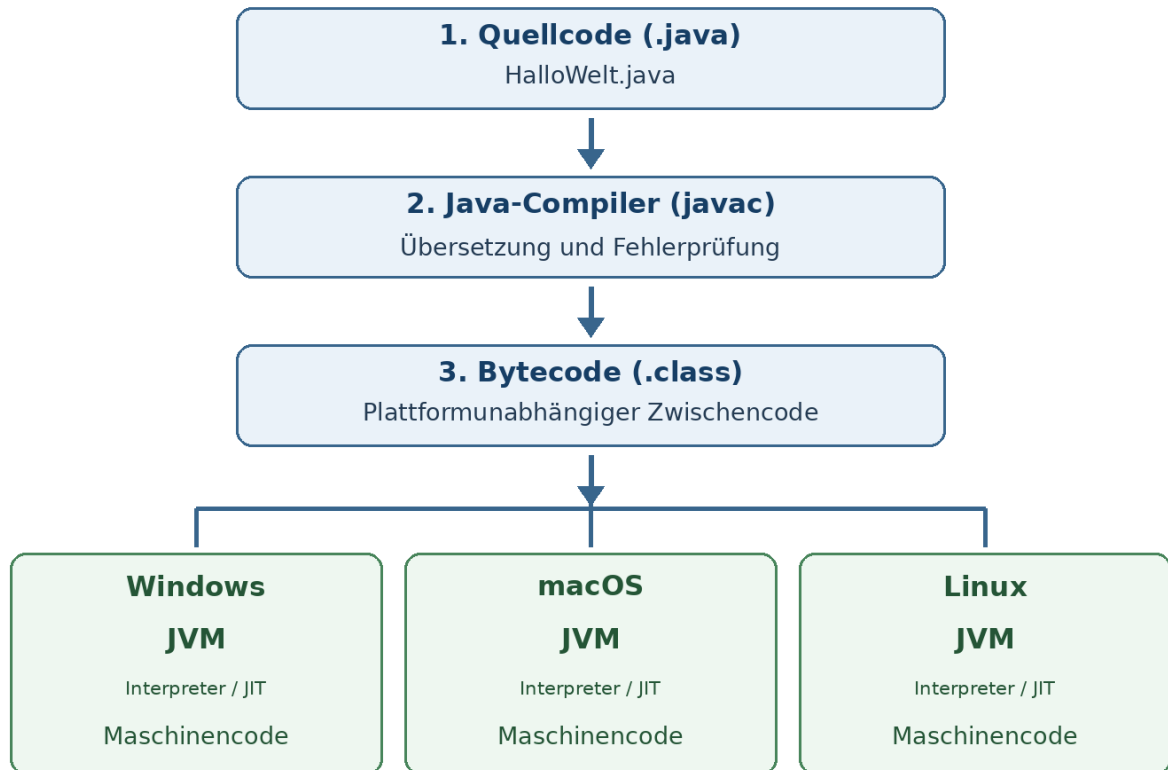


Lösung - Grundlagen der Java-Programmierung

Ausführliche Lehrkraftfassung | Java, Compiler, Interpreter und Fehlerarten

1. Vom Java-Programm zur Ausführung



Plattformunabhängigkeit: derselbe Bytecode, passende JVM je System

JVM führt Bytecode durch Interpretation und/oder JIT-Kompilierung aus.

Erläuterung der einzelnen Schritte

1. Quellcode erstellen: Der Programmierer schreibt Java-Anweisungen in einer Entwicklungsumgebung und speichert sie als Datei mit der Endung .java.

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

2. Kompilierung: javac prüft den Quellcode unter anderem auf Syntax- und Typfehler und übersetzt ihn in Java-Bytecode. Aufruf: javac HalloWelt.java. Bei Fehlern entsteht normalerweise keine erfolgreich kompilierte Klasse.

3. Bytecode: Die Datei HalloWelt.class enthält plattformneutralen Zwischencode, keinen unmittelbar für Windows, macOS oder Linux bestimmten Maschinencode.

4. JVM: Die Java Virtual Machine lädt und prüft Klassen und führt Bytecode aus. Sie kann Bytecode interpretieren oder häufig ausgeführte Programmteile per Just-in-Time-Compiler (JIT) in nativen Maschinencode übersetzen.

5. Ausführung: Mit `java HalloWelt` startet die Anwendung. Die Ausgabe lautet: `Hallo Welt!`

Kernaussage: `javac` erzeugt im Regelfall Bytecode. Die konkrete JVM sorgt für die Ausführung auf dem jeweiligen System.

2. Compiler und Interpreter vergleichen

Kriterium	Compiler	Interpreter
Funktionsweise	Übersetzt Programmcode in eine andere Codeform.	Verarbeitet und führt Anweisungen während der Laufzeit aus.
Zeitpunkt	Separate Übersetzungsphase vor Ausführung.	Während der Programmausführung.
Ergebnis	Zum Beispiel nativer Code oder Bytecode.	Nicht zwingend eine gespeicherte ausführbare Datei.
Geschwindigkeit	Nativer kompilierter Code oft schnell.	Interpretation kann langsamer sein.
Fehlersuche	Viele Fehler früh beim Übersetzen erkennbar.	Manche Fehler erst beim Erreichen der Anweisung.
Vorteile	Frühe Prüfungen, Optimierung, Tempo.	Flexibilität, unmittelbares Ausprobieren.
Nachteile	Zusätzliche Kompilierungszeit.	Möglicher Laufzeit-Overhead.
Beispiele	C/C++ (typisch native Kompilierung).	Klassische Python-Interpreter (auch Python nutzt Bytecode).

Welche Rolle spielen beide Prinzipien bei Java?

Compilerprinzip: `javac` übersetzt `HalloWelt.java` in `HalloWelt.class`. Interpreterprinzip: Die JVM kann Bytecode schrittweise ausführen. JIT-Kompilierung: Die JVM übersetzt häufig ausgeführte Abschnitte zur Laufzeit in nativen Maschinencode. So verbindet Java Portabilität und Leistungsfähigkeit.

`HalloWelt.java` → `javac` → `HalloWelt.class` → JVM → Ausführung

Wichtig: Ein Compiler muss nicht zwingend Maschinencode erzeugen. Auch die Übersetzung in Bytecode ist eine Kompilierung.

3. Fehler erkennen

A. Syntaxfehler

Syntaxfehler verstoßen gegen die grammatikalischen Regeln von Java und werden beim Kompilieren erkannt.

```
int preis = 50
System.out.println(preis);
```

Fehler: Nach 50 fehlt das Semikolon. Korrektur:

```
int preis = 50;
System.out.println(preis);
```

B. Laufzeitfehler

Das Programm lässt sich kompilieren, aber eine bestimmte Operation löst bei der Ausführung eine Exception aus.

```
int umsatz = 1000;
int anzahl = 0;
int durchschnitt = umsatz / anzahl;
System.out.println(durchschnitt);
```

Ursache: Ganzzahldivision durch null. Java löst eine `ArithmeticException` aus. Ohne Behandlung bricht der betroffene Programmablauf ab. Korrektur:

```
if (anzahl != 0) {
    int durchschnitt = umsatz / anzahl;
    System.out.println(durchschnitt);
} else {
    System.out.println("Berechnung nicht möglich.");
}
```

C. Logischer Fehler

Das Programm wird kompiliert und läuft, berechnet aber fachlich das Falsche.

```
double netto = 100.0;
double brutto = netto * 0.19;
System.out.println(brutto); // 19.0 statt 119.0
```

Fehler: Es wird nur der Steuerbetrag berechnet. Korrektur:

```
double netto = 100.0;
double brutto = netto * 1.19;
System.out.println(brutto); // 119.0
```

Merkmal	Syntaxfehler	Laufzeitfehler	Logischer Fehler
Kompilierung	Scheitert	Gelingt grundsätzlich	Gelingt
Start	Nicht in dieser Form	Ja	Ja
Fehlermeldung	Compiler	Häufig Exception	Meist keine
Beispiel	Semikolon fehlt	Division durch 0	Falscher Faktor
Erkennung	Compilerdiagnose	Ausführung / Test	Soll-Ist-Vergleich

Merksatz: Syntaxfehler verhindern die Übersetzung; Laufzeitfehler treten bei der Ausführung auf; logische Fehler erzeugen falsche Ergebnisse trotz scheinbar funktionierendem Programm.

4. Plattformunabhängigkeit

Der Java-Compiler übersetzt den Quellcode in Bytecode für eine virtuelle Maschine. Dieser Bytecode ist nicht unmittelbar an ein bestimmtes Betriebssystem oder einen Prozessortyp gebunden. Deshalb kann dieselbe `.class`-Datei grundsätzlich auf Windows, macOS oder Linux ausgeführt werden.

Voraussetzung

Auf jedem Zielsystem muss eine passende Java-Laufzeitumgebung mit JVM vorhanden sein. Sie muss die verwendete Bytecode-Version unterstützen. Zusätzlich dürfen keine unvereinbaren plattformspezifischen Abhängigkeiten vorliegen.

Kaufmännisches Beispiel

Ein Unternehmen erstellt und kompiliert eine Java-Anwendung zur Verwaltung von Kundenbestellungen unter Windows. Die erzeugte `Bestellverwaltung.class` kann auf einem Linux-Server mit geeigneter Java-Laufzeitumgebung ausgeführt werden, ohne den Java-Quellcode erneut kompilieren zu müssen.

Vorteile für Unternehmen

- Geringerer Entwicklungsaufwand durch eine gemeinsame Codebasis.
- Einfachere Wartung und Aktualisierung.

- Flexibler Einsatz auf unterschiedlichen Betriebssystemen.
- Mögliche Kosteneinsparungen bei Entwicklung und Betrieb.

Grenzen

Plattformspezifische Bibliotheken, Dateipfade, Hardwarezugriffe oder andere native Abhängigkeiten können die Portabilität einschränken.

Merksatz: „Write Once, Run Anywhere“ – einmal kompilierter Bytecode kann grundsätzlich auf verschiedenen Systemen mit geeigneter JVM laufen.