

1 Java Virtual Machine

Die **Java Virtual Machine (JVM)** ist das Herzstück der Java-Plattform. Sie fungiert als abstrakte Rechenmaschine, die eine Laufzeitumgebung zur Ausführung von Bytecode bereitstellt, unabhängig von der zugrunde liegenden Hardware oder dem Betriebssystem.

1.1 Architektur und Konzept

Die Java-Plattform besteht aus zwei Hauptkomponenten:

- **Java API:** Standardbibliothek für Zugriff auf Ressourcen (IO, Netzwerk, etc.).
- **Java Virtual Machine (JVM):** Lädt Klassen, führt Programme aus und verwaltet den Speicher (Garbage Collection).

Konzept der Virtuellen Maschine

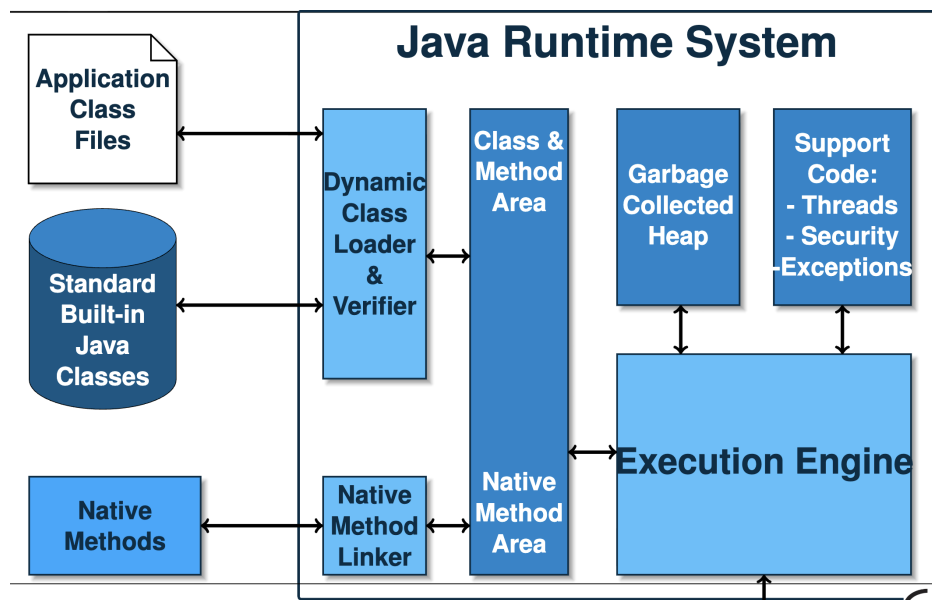
Eine **Virtuelle Maschine** ist eine "Maschine auf der Maschine". Sie definiert eine Spezifikation für die Ausführung von Programmen. Solange eine Implementierung (z.B. HotSpot) die Spezifikation erfüllt, ist das Verhalten des Programms auf jeder physischen Maschine identisch.

1.1.1 Bytecode und Kompilierung

Java-Code wird nicht direkt in Maschinencode, sondern in **Bytecode** übersetzt (.class Dateien). Dieser Ansatz ist ein Hybrid:

- **Interpretiert:** Bytecode kann direkt interpretiert werden (langsamer).
- **JIT-Kompilierung (Just-In-Time):** Häufig ausgeführte Code-Pfade ("Hotspots") werden zur Laufzeit in nativen Maschinencode übersetzt. Dies ermöglicht eine Leistung nahe an nativ kompilierten Sprachen (C++), bei gleichzeitiger Portabilität.

Portabilität: Da die JVM über das Betriebssystem abstrahiert, sind Datentypen (z.B. die Bitbreite von `int`) auf allen Plattformen identisch. Ein Programm muss nur einmal zu Bytecode kompiliert werden und läuft überall, wo eine JVM installiert ist ("Write Once, Run Anywhere").



1.2 Interner Aufbau der JVM

Das **Java Runtime System** besteht aus mehreren Subsystemen:

1.2.1 Class Loader Subsystem

Der Class Loader ist für das dynamische Laden, Linken und Initialisieren von Klassen zuständig. Der Prozess läuft in folgenden Phasen ab:

1. **Loading:** Laden des binären Class Files.
2. **Linking:** Überführung in JVM-interne Strukturen.
 - *Verification:* Prüfung der Struktur und Sicherheit (Bytecode Verifier).
 - *Preparation:* Speicherreservierung für statische Felder (Standardwerte).
 - *Resolution:* Auflösen symbolischer Referenzen im Constant Pool (kann verzögert geschehen).
3. **Initialization:** Ausführung statischer Initialisierer (z.B. `<clinit>`).

1.2.2 Speicherbereiche (Runtime Data Areas)

Die JVM verwaltet den Speicher in verschiedenen Bereichen:

- **Method Area:** Speichert Klassendefinitionen, Konstanten, statische Variablen und Code.
- **Heap:** Hier werden alle **Objekte** und **Arrays** zur Laufzeit angelegt. Der Speicher wird durch den **Garbage Collector** automatisch bereinigt.
- **Java Stack:** Speichert Stack Frames (lokale Variablen, Operandenstack) für jeden Methodenaufruf. Jeder Thread hat seinen eigenen Stack.
- **PC Register:** Zeigt auf die aktuelle Instruktion.
- **Native Method Stack:** Für Aufrufe von nativem Code (via JNI, z.B. C++ Bibliotheken).

1.3 Class File Format und Typen

Ein Class File enthält alle Informationen einer Klasse in einem binären Format (Big-Endian). Wichtige Komponenten sind:

- **Constant Pool:** Eine Tabelle, die Literale (Strings, Zahlen) und symbolische Referenzen (Klassen-, Methoden-, Feldnamen) enthält. Der Bytecode referenziert Werte oft über einen Index in diesen Pool, was das dynamische Linken ermöglicht.
- **Methods Table:** Enthält den Bytecode der Methoden.
- **Fields Table:** Beschreibt die Variablen der Klasse.

1.3.1 Type Descriptors

Die JVM verwendet kompakte Strings, um Typen zu beschreiben. Dies ist essentiell für das Verständnis von Methodensignaturen im Bytecode.

Deskriptor	Datentyp	Bemerkung
B	byte	Vorzeichenbehaftet (8 Bit)
C	char	Unicode Character (16 Bit)
D	double	64-Bit Gleitkomma
F	float	32-Bit Gleitkomma
I	int	32-Bit Integer
J	long	64-Bit Integer
S	short	Vorzeichenbehaftet (16 Bit)
Z	boolean	true/false
V	void	Nur als Rückgabebetyp
L<Klasse>;	Object	z.B. Ljava/lang/String;
[Array	z.B. [I (int[]), [[F (float[])]

Beispiel Methodensignatur: Java: `public int foo(char c, float f, String s)` JVM-Deskriptor: `(CFLjava/lang/Stri`

1.4 Ausführungsmodell: Stack Frames

Die JVM ist eine **stack-basierte Maschine**. Es gibt keine allgemeinen Register zur Berechnung. Operationen finden auf dem Operandenstack statt. Bei jedem Methodenaufruf wird ein neuer **Stack Frame** erzeugt, der enthält:

Bestandteile eines Stack Frames

- **Local Variables:** Ein Array von lokalen Variablen (inkl. Methodenparameter). Index 0 ist bei Instanzmethoden `this`.
- **Operand Stack:** Ein LIFO-Speicher für Zwischenergebnisse. Operationen (z.B. `iadd`) nehmen Werte vom Stack und legen das Ergebnis zurück.
- **Frame Data:** Referenzen auf den Constant Pool, Return-Adressen etc.

Wichtig: Die JVM arbeitet intern mit 32-Bit Slots ("Wörtern").

- `long` und `double` belegen **zwei** Slots (sowohl im Stack als auch in den Local Variables).
- Typen wie `short`, `byte`, `char` werden für Berechnungen implizit in `int` umgewandelt (sind sogenannte *Storage Types*).

1.5 Bytecode Instruktionen

Bytecode-Instruktionen (Opcodes) sind meist typisiert. Der erste Buchstabe (Präfix) gibt den Typ an:

- `i`: `int` (auch `byte`, `char`, `short`, `boolean`)
- `l`: `long`
- `f`: `float`
- `d`: `double`
- `a`: reference (Objekte, Arrays)

1.5.1 Daten laden und speichern

- **Konstanten laden:**
 - `iconst.<n>`: Lädt kleine Integers (-1 bis 5) effizient.
 - `bipush` / `sipush`: Lädt Byte/Short Konstanten.
 - `ldc`: Lädt Konstanten (Strings, große Zahlen) aus dem Constant Pool.
- **Lokale Variablen:**
 - `<p>load <index>`: Lädt Wert aus lokaler Variable auf den Stack.

- `<p>store <index>`: Speichert Wert vom Stack in lokale Variable.

1.5.2 Stack-Manipulation

Da man Register nicht direkt adressieren kann, muss der Stack oft manipuliert werden:

- `pop` / `pop2`: Löscht oberstes Element (1 oder 2 Wörter).
- `dup` / `dup2`: Dupliziert oberstes Element.
- `swap`: Vertauscht die obersten zwei Elemente.

1.5.3 Arrays

Arrays sind Objekte auf dem Heap. Zugriff erfolgt in drei Schritten:

1. Referenz auf das Array laden.
2. Index laden.
3. Operation ausführen (Wert wird geladen/gespeichert).

Instruktionen:

- `newarray`: Erstellt Array primitiver Typen.
- `anewarray`: Erstellt Array von Referenzen.
- `<p>aload`: Lädt Wert aus Array auf Stack (`ArrRef`, `Index` -> `Value`).
- `<p>astore`: Speichert Wert in Array (`ArrRef`, `Index`, `Value` -> `_`).

1.5.4 Arithmetik und Logik

Arithmetische Operationen konsumieren Operanden vom Stack und legen das Ergebnis ab.

- `<p>add`, `<sub>`, `<mul>`, `<div>`, `<rem>`, `<neg>`
- Logisch (nur `int/long`): `shl`, `shr`, `ushr`, `and`, `or`, `xor`.

Es gibt keine implizite Typumwandlung. Explizite Konversion nötig: `i2f` (`int` zu `float`), `i2i`, etc.

1.5.5 Kontrollfluss

Die JVM bietet Sprungbefehle (`goto`) und bedingte Sprünge.

1. Integer-Vergleiche:

- Unär (vergleicht Stack-Top mit 0): `ifeq` (`==0`), `iflt` (`<0`), etc.
- Binär (vergleicht zwei oberste Werte): `if_icmpeq`, `if_icmpge`, etc.

2. Float/Double/Long Vergleiche (Wichtig!): Die JVM hat keine direkten Sprungbefehle für diese Typen (außer Vergleich gegen 0). Der Vergleich erfolgt über einen "Umweg" in zwei Schritten:

1. Eine Vergleichsoperation (`lcmp`, `fcmlpl`, `dcmpl`) vergleicht zwei Werte und legt das Ergebnis als `int` (-1, 0, 1) auf den Stack.
2. Ein Standard-Integer-Branch (z.B. `ifge`) wertet dieses Ergebnis aus.

1.5.6 Methodenaufrufe

- `invokestatic`: Aufruf statischer Methoden.
- `invokevirtual`: Aufruf von Instanzmethoden (dynamischer Dispatch).

Protokoll: Argumente werden auf den Stack gelegt. Die aufgerufene Methode findet diese in ihren lokalen Variablen. Der Rückgabewert (falls vorhanden) liegt nach Rückkehr auf dem Stack des Aufrufers.

1.6 ASM Framework

Für die Code-Generierung im Compilerbau wird oft das ASM-Framework genutzt. Es abstrahiert die binäre Erzeugung des Class Files.

- Arbeitet Event-basiert (Visitor Pattern) oder Tree-basiert.
- Nimmt dem Entwickler die Berechnung von Sprung-Offsets und Constant Pool Indizes ab.
- Beispiel: `currentGenerator.push(5)` erzeugt automatisch `iconst_5`.