Java – Erster Überblick

Stephan Karrer

Autor: Stephan Karrer

Designziele von Java, Teil 1

- **Einfach und vertraut:**
  Orientiert an C/C++, aber drastisch gesäubert! Konzepte aus Smalltalk, Ada, ...
- **Objektorientiert:**
  Kapselung, Geheimnisprinzip, Vererbung, Polymorphismus
- **Basis für verteilte Client-Server-Systeme:**
  Bereitstellung umfangreicher Klassen-Bibliotheken und Mechanismen für die Erstellung verteilter Anwendungen
- **Robust:**
  Strenge Typprüfung zur Übersetzungszeit durch den Java-Compiler, keine Zeigerarithmetik, automatische Speicherbereinigung (Garbage Collection), Laufzeitprüfung (Ausnahmebehandlung)
- **Sicher:** Zugriff auf Ressourcen außerhalb der Laufzeitumgebung ist steuerbar

Designziele von Java, Teil 2

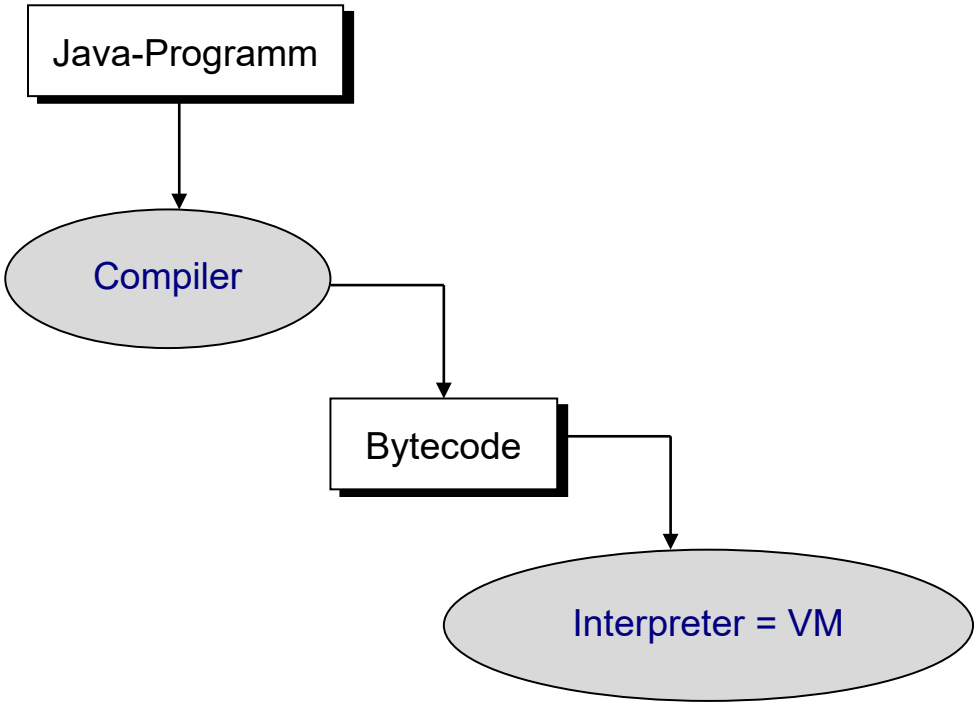
- **Architekturneutral und portabel:**
  - verteilt wird Byte-Code, der an der Client-Maschine interpretiert wird (architekturneutral)
  - Ein integer in Java ist auf jedem System eine 32-Bit-Zahl (in C nicht ...)
  - Unicode für die Codierung von Zeichen, ...

- **Leistungsfähig:**
  - Interpreter verlässt sich auf gewisse Prüfungen und wird damit schneller
  - Automatische Speicherbereinigung läuft als Hintergrund-Thread mit niedriger Priorität: Speicher ist vorhanden, wenn man ihn braucht
  - Schnittstelle zu nativem Code ist vorhanden (dieser wird üblicherweise in C programmiert)

- **Multi-Threading-fähig:**
  Java unterstützt Threads auf Sprachebene (Thread-Klasse), im Laufzeitsystem (Bausteine zur Synchronisation) und in den Bibliotheken (Thread-sichere Routinen).

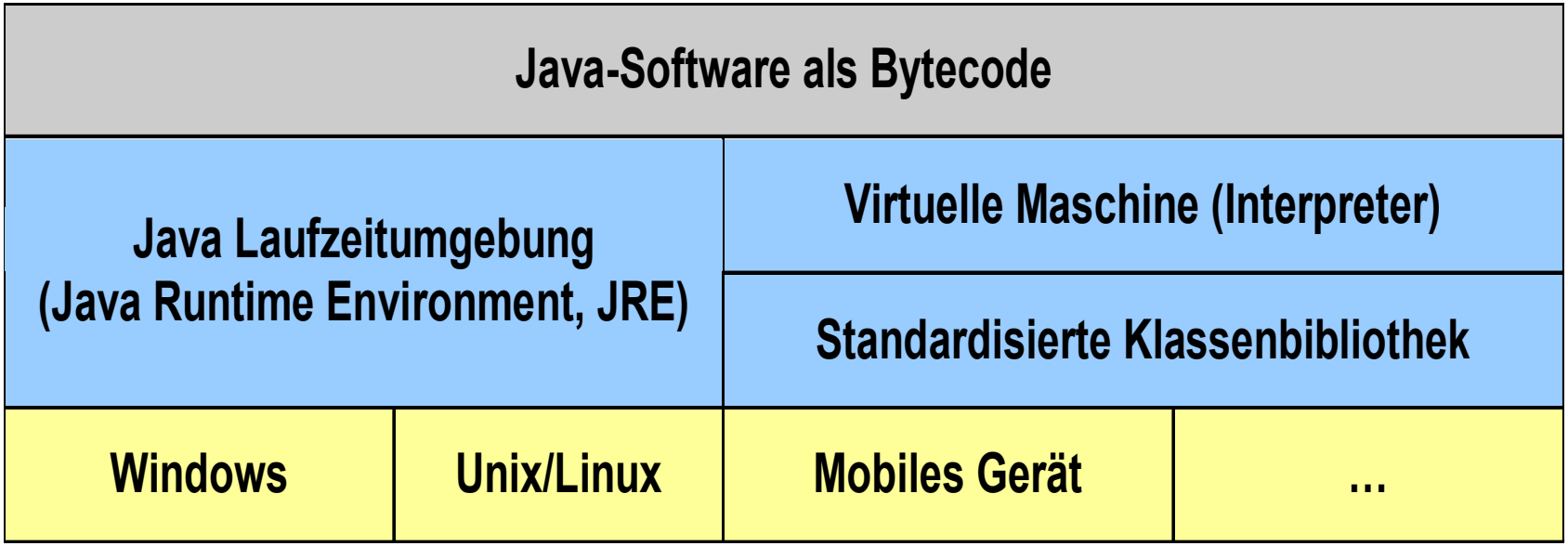
Grundgedanke bei Java für die Übersetzung und Ausführung

- Sowohl die Sprache Java als auch der Bytecode + Funktionen der VM sind standardisiert.
- Selbstverständlich gibt es diverse Implementierungen von Compiler und VM.
- Das Zielsystem mit der VM ist in der Regel nicht identisch mit dem Entwicklungssystem.
  - Entscheidend ist somit die Java-Version des Zielsystems !
- Das Zielsystem besitzt üblicherweise neben der VM auch die standardisierte Klassenbibliothek .



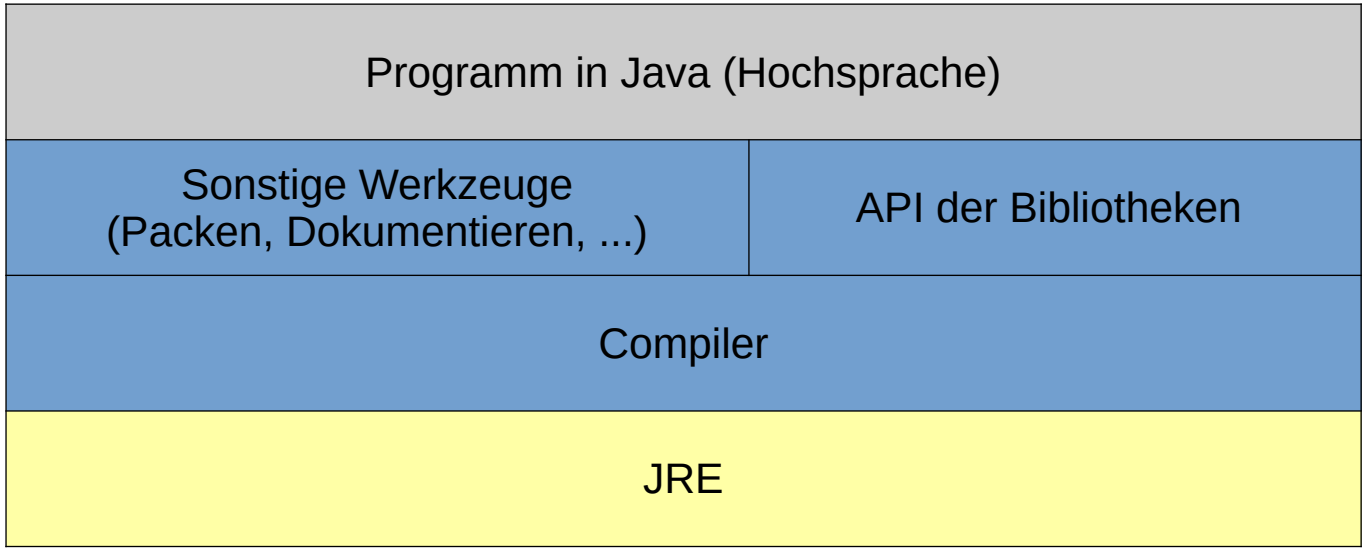
Write Once, Run Everywhere

Java Runtime Environment (JRE)



- Die mit der SE standardisierte Klassenbibliothek umfasst Methoden für den Zugriff auf Betriebssystem-Ressourcen (Dateien, I/O, Netzwerk, …)
  - Somit sind für den Zugriff systemspezifische Implementierungen (nativer Code) auf der unteren Ebene erforderlich.
  - JRE ist somit von der Betriebssystem-Plattform abhängig.

Java Development Kit (JDK)



- Das JDK umfasst das JRE und stellt zusätzlich die Werkzeuge (Kommandozeile) und Bibliotheken für die Entwicklung in der Sprache Java zur Verfügung.

Java Technologie: Spezifikation

- seit 1998 werden die Spezifikationen im sogenannten Java Community Process (JCP) entwickelt.
- Es werden 3 verschiedene Plattformen (Frameworks mit bestimmten Umfang) definiert:
  - **JSE** (Java Platform Standard Edition)
  - **JEE** (Java Platform Enterprise Edition), nur bis JEE 8, dann Spezifikation durch Jakarta
  - **JME** (Java Platform Micro Edition), da tut sich in den letzten Jahren nichts
- Diverse Hersteller/Konsortien, insbesondere Oracle (früher Sun) liefern Umsetzungen der Spezifikation als Java Development Kits (JDKs) bzw. Java Runtime Environments (JREs) für die verschiedenen Plattformen:
  - Hauseigene Systeme (z.B. Oracle für Oracle DBMS, Solaris)
  - Linux, Windows, macOs
  - iOS, Android
- Die Firma Sun als Schöpfer von Java hat bereits vor der Übernahme durch Oracle das OpenJDK Projekt ins Leben gerufen, welches von Oracle fortgeführt wird.

Java Standardisierung: JCP, JEP, JESP

- Die einzelnen Spezifikationen in Form von Java Specification Requests (JSRs) werden durch den Community Process definiert.
- Hier haben sowohl kommerzielle, nichtkommerzielle Organisationen und auch einzelne Personen Zugang.
- Daneben existiert unter dem Open JDK ein Prozess bzgl. des JDK in Form JDK Enhancement -Proposals (JEPs)
- Nachdem sich Oracle aus der JEE-Spezifikation weitestgehend zurückgezogen hat, wird die JEE-Version in Form von Jakarta EE durch den Jakarta EE Specification Process (JESP) definiert.

Java Technologie: Umsetzung der SE

- OpenJDK (Open Java Development Kit) ist eine freie, Open-Source-Implementierung der Java Platform, Standard Edition (Java SE).
- Dies ist seit Java SE 7 die Referenz-Implementierung und damit das Maß der Dinge.
- Wichtige Hersteller (Oracle, Red Hat, IBM, Apple, SAP) unterstützen das OpenJDK Projekt mit Ressourcen bzgl. Entwicklung und Organisation.
- Es gibt mittlerweile eine Vielzahl von Umsetzungen auf Basis des OpenJDK mit unterschiedlichen Support-Modellen.
- Das hindert natürlich Hersteller nicht daran für ihre Implementierung bzw. Support Lizenzgebühren zu nehmen.
- Insbesondere Oracle versucht sich unter Aspekten wie Tool-Unterstützung, Support, Updates und Patches seit Java 8 an kostenpflichtigen Lizenzmodellen.
  - Das hat zu einem massiven Anstieg der Nutzung von OpenJDK als Alternative zum klassischen Oracle Java geführt.
  - Oracle selbst stellt eine freie Implementierung des OpenJDK zur Verfügung und deren proprietäre Variante ist dazu kompatibel.

siehe auch: https://en.wikipedia.org/wiki/Java_version_history

Entwicklung der SE-Variante

1991 – 1995	Entwicklung der ersten Java-Version	
1996	JSE 1.0	
1997	JSE 1.1	
1998	JSE 1.2	
2000	JSE 1.3	
2002	JSE 1.4	
2004	Java 5.0	
2006	Java 6.0	
2011	Java 7	
2014	Java 8	
2017 (Sep.)	Java 9	ab jetzt halbjährliche Releases
...		
2018 (Sep.)	Java 11 LTS	
…		
2021 (Sep.)	Java 17 LTS	
…		
2023 (Sep.)	Java 21 LTS	

Stand der Dinge

Aktuelle Java-Versionen:

JSE 21 (ab Sep. 2023)

JSE 17 (ab Sep. 2021)

JSE 11 (ab Sep. 2018)

JSE 8.x (ab März 2014)

JEE 8 (2017) bzw.
Jakarta EE 8 (2019)

Jakarta EE 9.x (ab Sep. 2020)

Jakarta EE 10.x (ab Sep. 2022)

Plattformen:

- macOS, iOS (Apple)
- Winxxx (Microsoft)
- Linux/UNIX (Diverse)
- Android (Google)
- ...

Funktionen der VM: Dynamisches Binden und Laden

- Statt Linker --> Klassenlader
- Laden ist inkrementell, leichtgewichtig
- Laden ist angepaßt an Netzwerkumgebung
- Laden ist ideal bei häufigen Änderungen
- Java-Compiler löst Referenzen nicht bis zu numerischen Werten (Offsets) auf
  (Java-Interpreter löst Referenzen einmalig zu Offsets auf beim Einbinden der Klasse)
- Speicheranordnung der Objekte bestimmt der Interpreter (nicht: Compiler)

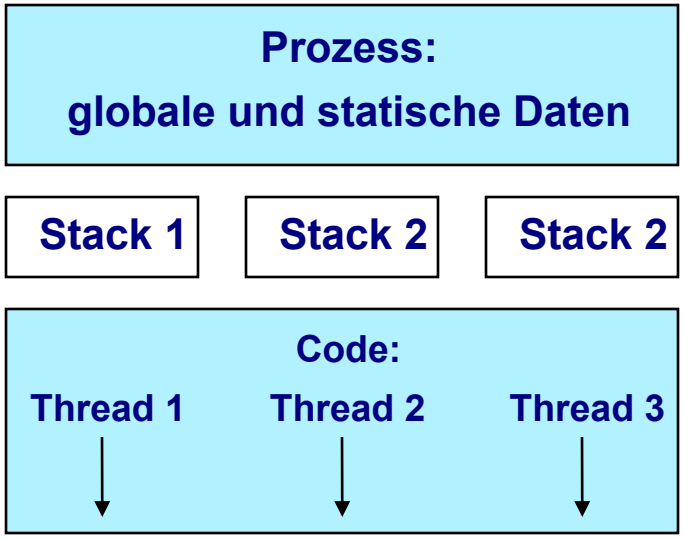
Funktionen der VM: Informationen zur Laufzeit

- Jedes Objekt enthält Verweis auf ein Class-Objekt
- Zur Laufzeit abfragbar: Klassenname, -typ u.ä.
- Über das Reflection-API können weitere Informationen zu den Klassen und Methoden zur Laufzeit abgefragt werden
  - dies ermöglicht leistungsfähige Debugging- , Monitoring- und Konfigurations-Möglichkeiten
  - kann genutzt werden um dynamisch zur Laufzeit Objekte zu erzeugen und auf Methoden zuzugreifen (die vorab noch nicht bekannt waren)
  - viele Frameworks, insbesondere aus dem JEE-Umfeld, nutzen Reflection

Funktionen der VM: Speicherverwaltung

- Alle Objekte werden dynamisch angelegt (Heap)
- Daten der Basistypen werden ebenfalls dynamisch angelegt
- Keine explizite Speicher-Allokierung (wie malloc, calloc o.ä. in C/C++) sondern Objekt-Instantiierung mit new
- Automatische Speicherbereinigung (Garbage Collection)
  - "Wenn keiner mehr darauf verweist: vernichten"
  - "Wenn eine Verweisvariable ihren Gültigkeitsbereich verläßt: Verweis löschen"
  - Ansonsten obliegt es dem Entwickler, die Referenzen auf Objekte frei zu geben!
- Garbage Collector
  - Thread niedriger Priorität

Funktionen der VM: Thread-Konzept



Vorteile:

- effizienter Kontextwechsel
- gut für Serverprozesse
- gut abbildbar auf symmetrische Multiprozessor-Architektur
- effiziente Interthread-Kommunikation

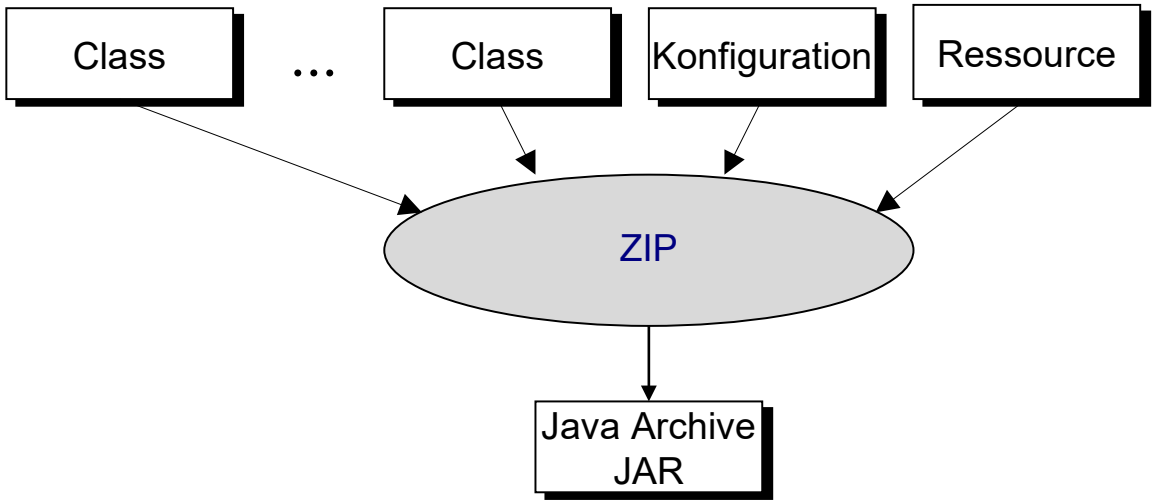
Nachteile:

- weniger robust als Multiprozessansatz
- Synchronisation ist eigenes Thema
- Abbildung von "User Level Threads" auf "Kernel Threads" kann auch suboptimal sein
- löst auch nicht die Frage "Was ist parallelisierbar?"

Funktionen der VM: Thread-Konzept

- Multithreading in die Sprache eingebaut.
- Muß nicht notwendigerweise durch das Betriebssystem unterstützt werden, wäre aber günstig.
- Zur Verfügung stehen:
  - Klassen Thread, ThreadGroup
  - Operationen start(), interrupt(), join(), yield(), setPriority(), ...
  - Schlüsselwort synchronized
  - Methoden notify() und wait()
- Programmierung paralleler Verarbeitungsschritte und deren Synchronisation ist generell eine Herausforderung.

Deployment (im Kleinen)



- Statt einzelne Bytecode-Klassen zu verteilen, werden diese samt Konfigurations-dateien und sonstigen Ressourcen in vorgegebenen Verzeichnisstrukturen organisiert, via ZIP gepackt und als Java Archive bereit gestellt.

- Die Standards definieren hierzu entsprechende Formate (JAR, WAR, EAR, …)

Bibliotheken

- Die SE-Bibliothek bietet mit ca. 4000 – 6000 (je nach Version) Klassen eine grundlegende Basis.
- In der Praxis wird das ergänzt durch eine Vielzahl weiterer Bibliotheken und Frameworks, von denen in der Regel viele frei verfügbar sind.
- Ein typisches Java-Projekt benutzt neben der SE-Bibliothek zusätzlich 5 – 20 weitere externe Bibliotheken.
- Das alle Versionen zusammen passen ist eine Herausforderung!
- Sowohl auf dem Entwicklungssystem als auch dem Zielsystem muss das Laden funktionieren, d.h. wir sollten den gleichen Bestand an API-Bibliotheken haben.
  - Entweder die Anwendung bringt die zusätzlichen Bibliotheken mit oder aber das Zielsystem stellt diese schon bereit (siehe auch JEE).