UNIVERSITÄT LEIPZIG

Mainframe Internet Integration

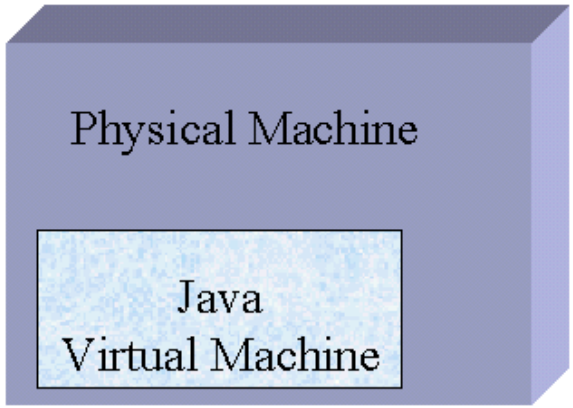
Prof. Dr. Martin Bogdan
Prof. Dr.-Ing. Wilhelm G. Spruth

SS2013

Java Enterprise Edition Teil 1

Java Virtual Machine (JVM)

Maschinensprache der Java Virtual Machine (JVM)



Im Zusammenhamg mit Java ist eine neue Hardware Prozessor Architektur entwickelt worden, vergleichbar zur System z, PowerPC oder x86 Architektur. Diese Architektur wird als Java Virtual Machine Architektur (JVMA) bezeichnet.

Wenn Java Quellcode kompiliert wird, entsteht Object Code. Dieser Objekt Code wird als „Byte Code“ bezeichnet.

Das Ausführungen eines Java Programms erfolgt im Prinzip auf speziellen Java-Prozessoren. Dies sind Mikroprozessoren, die Java-Bytecode als Maschinensprache verwenden. Sie wurden jedoch nie in größeren Stückzahlen gebaut. Statt dessen wird die Java Virtual Machine Architektur fast immer auf anderen Rechner Plattformen mit Hilfe einer Java Virtuellen Maschine (JVM) emuliert. Dies ist vergleichbar zur System z Emulation auf einem x86 Rechner mittels Hercules oder zPDT .

Jazelle DBX (Direct Bytecode eXecution) ermöglicht einigen ARM Processor Versionen die Ausführung von Java Bytecode in Hardware als Alternative zur normalen JVM Programm Ausführung. Die Dalvik virtuelle Maschine ist eine JVMA inkompatible Alternative zur JVM, die von Android für die Ausführung von Java Code benutzt wird.

Sicherheitsvorteile

Die Java Virtual Machine bietet neben der Plattformunabhängigkeit auch einen Gewinn an Sicherheit. Eine JVM überwacht zur Laufzeit die Ausführung des Programms, verhindert also z. B., dass ein Programm über Arraygrenzen hinweg liest oder schreibt. Im speziellen Fall von Java fällt diese Überwachung sehr einfach aus, da Java keine Zeiger unterstützt. Somit werden Pufferüberläufe verhindert, die vor allem bei den in C oder C++ geschriebenen Programmen vorkommen können.

Optimierungsverfahren

Um die Geschwindigkeit ("performance") der Programmausführung zu erhöhen, setzen die meisten JVM Implementierungen sogenannte JIT-Compiler ein, die unmittelbar während des Programmablaufs den Bytecode „Just In Time“ („Gerade zur rechten Zeit") in Maschinencode übersetzen. Eine Weiterentwicklung dieses Ansatzes, der Hotspot-Optimizer von Sun, behebt teilweise den Geschwindigkeitsnachteil der JVM, der hohe Speicherbedarf bleibt jedoch bestehen. Außerdem hat Java entwurfsbedingt einige Performance Nachteile, vor allem durch die automatische Speicherbereinigung (garbage collection). Einige JVM Implementierungen, zum Beispiel Insignia Jeode oder IBM J9, können diese Nachteile teilweise kompensieren.

BEA Jrockit, Sun Hotspot und IBM J9 dind die derzeitig führenden JVM Implementierungen.

Eine Java Anwendung, welche keine Probleme mit der begrenzten Hauptspeichergröße hat, läuft im 64 Bit Adressiermodus langsamer als im 31 Bit Adressiermodus !
http://www-03.ibm.com/systems/z/os/zos/tools/java/products/j6pcont64.html.

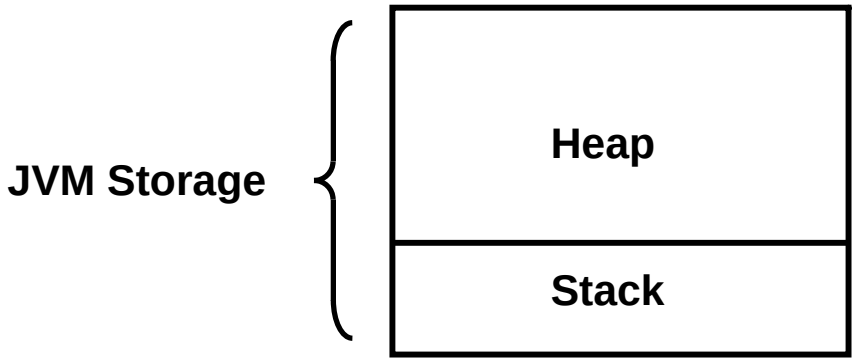
Abschottung der Threads

Innerhalb einer JVM können mehrere Prozesse (Threads) ablaufen. Servlets machen beispielsweise hiervon Gebrauch.

Die JVM schottet die Threads vom Betriebssystem ab. Dies hat zur Folge, dass es nicht mehr möglich ist, mit systemeigenen Mitteln Java Prozesse zu kontrollieren. Die JVM stellt aber auch keine eigenen Funktionen zur Prozesskontrolle und -steuerung bereit. Somit können diese auch nicht von außen beendet werden, wenn sie aufgrund eines Fehlers das Gesamtsystem stören. Im Fehlerfall muss die gesamte JVM beendet werden. Viele JVM Implementierungen erlauben deshalb das direkte Abbilden (Mappen) von dedizierten Java-Threads auf Betriebssystem-Prozesse (native Threads).

Heute ist das Mappen von Java-Threads in der Regel die Default-Einstellung, d. h. nur in Ausnahmefällen wie dem Verwenden einer älteren JVM erfolgt das Thread-Management nur durch die JVM und nicht durch das Mapping auf Threads des Betriebssystems.

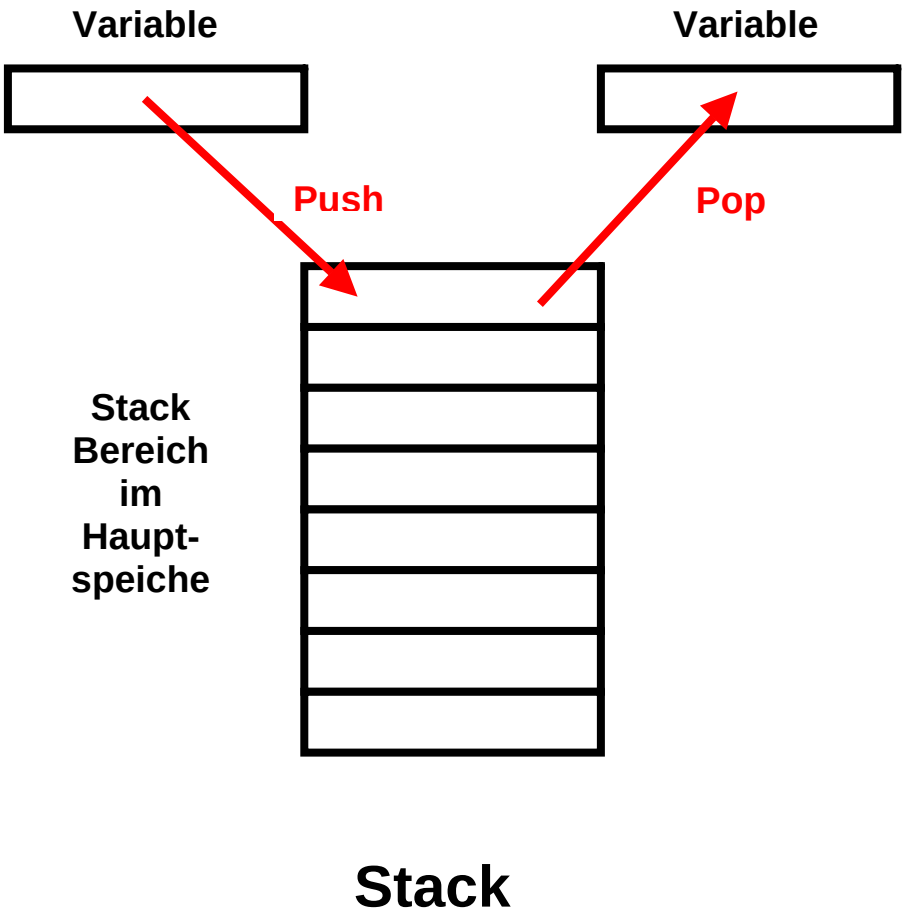
Die Isolation der Threads innerhalb der JVM untereinander ist unvollständig. Dies ist besonders kritisch, wenn mehrere Transaktionen als threads in der gleichen JVM verarbeitet werden. Hierauf wird später detailliert eingegangen. See http://publib.boulder.ibm.com/infocenter/javasdk/v1r4m2/index.jsp?topic=/com.ibm.java.doc.diagnostics.142/html/id1102.html.



JVM Memory Allocation

Der JVM Interpreter verwaltet einen Speicherbereich, der dem Anwendungsprogrammierer zugänglich ist, und der aus den beiden Teilen Heap und Stack besteht. Er enthält unterschiedliche Arten von Information während der Programmausführung:

- **Lokale Variablen und Stacks für die aktiven Methoden, und**
- **Arrays und Objekte, die während der Programm Ausführung erzeugt werden.**



Stack

Ein Stack ist ein Container, in dem man nur an der Spitze Elemente ablegen oder von oben wieder wegnehmen kann. Es besteht keine Möglichkeit, ein Element aus der Mitte zu entnehmen oder ein Element in die Mitte einzufügen. Diese Abarbeitungsreihenfolge wird auch als LIF0-Prinzip (last in–first out) bezeichnet. Mit der Methode `push()` kann ein Element auf einen Stack abgelegt werden. Mit der Methode `pop()` kann das oberste Element vom Stack wieder entnommen werden.

Ein Stack kann aus Variablen einfacher Datentypen aufgebaut werden - oder im Falle von Klassen auch aus Referenzen auf Objekte. Sind die Referenzen, die auf einem Stack gespeichert werden, vom Typ Object, so kann jede beliebige Referenz auf dem Stack abgelegt werden.

Der Java-Stack speichert lokale Variablen innerhalb von Methodenaufrufen, sowie Operanden für arithmetische Operationen. Bei jedem Methodenaufruf wird ein Stack-Frame auf den Stack gepusht, das Platz für die lokalen Variablen dieser Methode und einige Zusatzdaten bietet. Wird die Methode (durch `return` oder durch eine Exception) verlassen, wird der zugehörige Frame wieder vom Stack genommen.

Die JVM unterhält eine getrennten Stack für jeden Thread.

Heap

Der Heap existiert nur einmal pro Instanz der VM.

Der Heap dient zur Speicherung von dynamischen Informationen über konkrete Objekte. Für jedes Objekt sind dies sämtliche Instanzvariablen - sowohl die, die in seiner eigenen Klasse deklariert wurden, als auch die aus sämtlichen Vorfahren dieser Klasse.

Arrays und Objekte können nicht auf dem Stack gespeichert werden. Sie werden in dem Heap gespeichert.

Der new-Operator, der vom Anwendungsprogramm aufgerufen wird, um eine Variable auf dem Heap anzulegen, gibt dem Anwendungsprogramm eine Referenz auf die im Heap erzeugte dynamische Variable zurück. An welcher Stelle des Heaps die dynamische Variable angelegt wird, entscheidet nicht der Programmierer, sondern die virtuelle Maschine. Über diese Referenz kann man dann auf die dynamische Variable im Heap zugreifen.

Die Größe des Heaps ist beschränkt. Daher kann es zu einem Überlauf des Heaps kommen, wenn ständig nur Speicher angefordert und nichts zurückgegeben wird. Weiterhin wird mit zunehmendem Gebrauch der Heap zerstückelt, so dass der Fall eintreten kann, dass keine größeren Objekte mehr auf dem Heap angelegt werden können, obwohl in der Summe genügend freier Speicher vorhanden ist, aber eben nicht an einem Stück.

In Java werden Objekte im Heap nicht explizit freigegeben. Es wird vielmehr in unregelmäßigen Abständen durch die virtuelle Maschine der „Garbage Collector“ aufgerufen. Der Garbage Collector gibt den Speicherplatz, der nicht mehr referenziert wird, frei. Er ordnet ferner den Speicher neu (Defragmentierung), so dass auf dem Heap wieder größere homogene unbenutzte Speicherbereiche entstehen.

Class Loader

In Java wird kein ausführbares Programm durch einen Linker erzeugt. Statt dessen werden benötigte Klassen zur Laufzeit durch die JVM nachgeladen. Diese verwendet hierfür eine Klasse `ClassLoader` :

```
public abstract class ClassLoader extends Object
```

Ein ClassLoader erhält durch den Aufruf seiner Methode `loadClass()` die Aufforderung, Klassen aus .class-Dateien oder .jar-Archiven vom lokalen Dateisystem laden. Um diese Dateien zu lokalisieren, wertet der System-ClassLoader die Umgebungsvariable CLASSPATH aus.

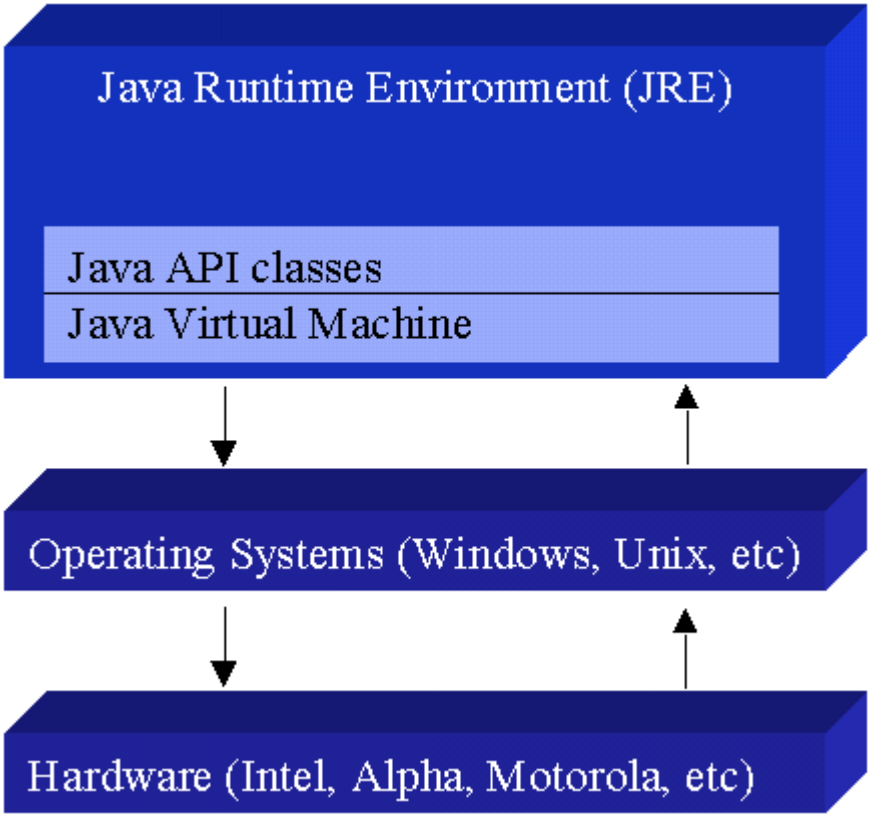
Ein Java Programm besteht aus einem losen Verbund von einzelnen `.class` Dateien, die bei Bedarf in die virtuelle Maschine geladen werden. Klassen, die eine `main()` Methode haben, können zum Starten einer Anwendung benutzt werden.

Jedes Klassen-Objekt enthält eine Reference zu dem ClassLoader der es definierte. Es existiert ein spezieller Bootstrap-ClassLoader für die Bootstrap-Klassen, und ein Erweiterungs-ClassLoader für die Klassen im lib/ext-Verzeichnis.

JDK

Der Java Development Kit (JDK) ist ein Subset des Software Development Kit (SDK). Der JDK ist zuständig für das Schreiben und Ausführen von Java Programmen. Der SDK enthält weitere Software, z.B. Debugger oder Dokumentation.

JDKs unterschiedlicher Hersteller befolgen grundsätzlich alle Java Specifikationen, können aber Funktionen enthalten, die in dem Java Standard nicht festgelegt sind. Beispiele sind Garbage Collection, Compilation Strategies und Optimierungsverfahren. Als Folge können bei der Portierung von Java Programmen Schwierigkeiten auftreten.



Java Runtime Environment (JRE)

Die Java Virtual Machine ist Teil eines größeren Systems, dem Java Runtime Environment (JRE). Jedes Betriebssystem und jede Hardware Architektur arbeitet mit einem unterschiedlichen JRE. Das JRE besteht aus einem Satz von Base Classes, welche eine JVM beinhalten wie auch eine Implementierung der Base Java API. Die JRE Implementierungen auf unterschiedlichen Plattformen ermöglichen die Portabilität von Java Programmen. Java Programme können auf einer bestimmten Plattform nur laufen, wenn dort ein JRE vorhanden ist.

Java verwendet Unicode

Java arbeitet intern ausschließlich mit Unicode UTF-16 Encoding.

Bei der Default-Codierung werden von der JVM Unicode-Zeichen bis \u00FF so gut wie möglich in die Code-Tabelle des Betriebssystem abgebildet.

Dies muss z.B. beachtet werden, wenn ein Java Programm auf eine DB2 oder IMS Datenbank zugreift.

Die “z/OS Support for Unicode Conversion Services” konvertieren alphanumerische Daten zwischen UTF-8, UTF-16, ASCII und EBCDIC.

Dies kann kompliziert werden, weil die JVM selbst häufig in C/C++ implementiert ist, und auf ASCII Daten zugreift.

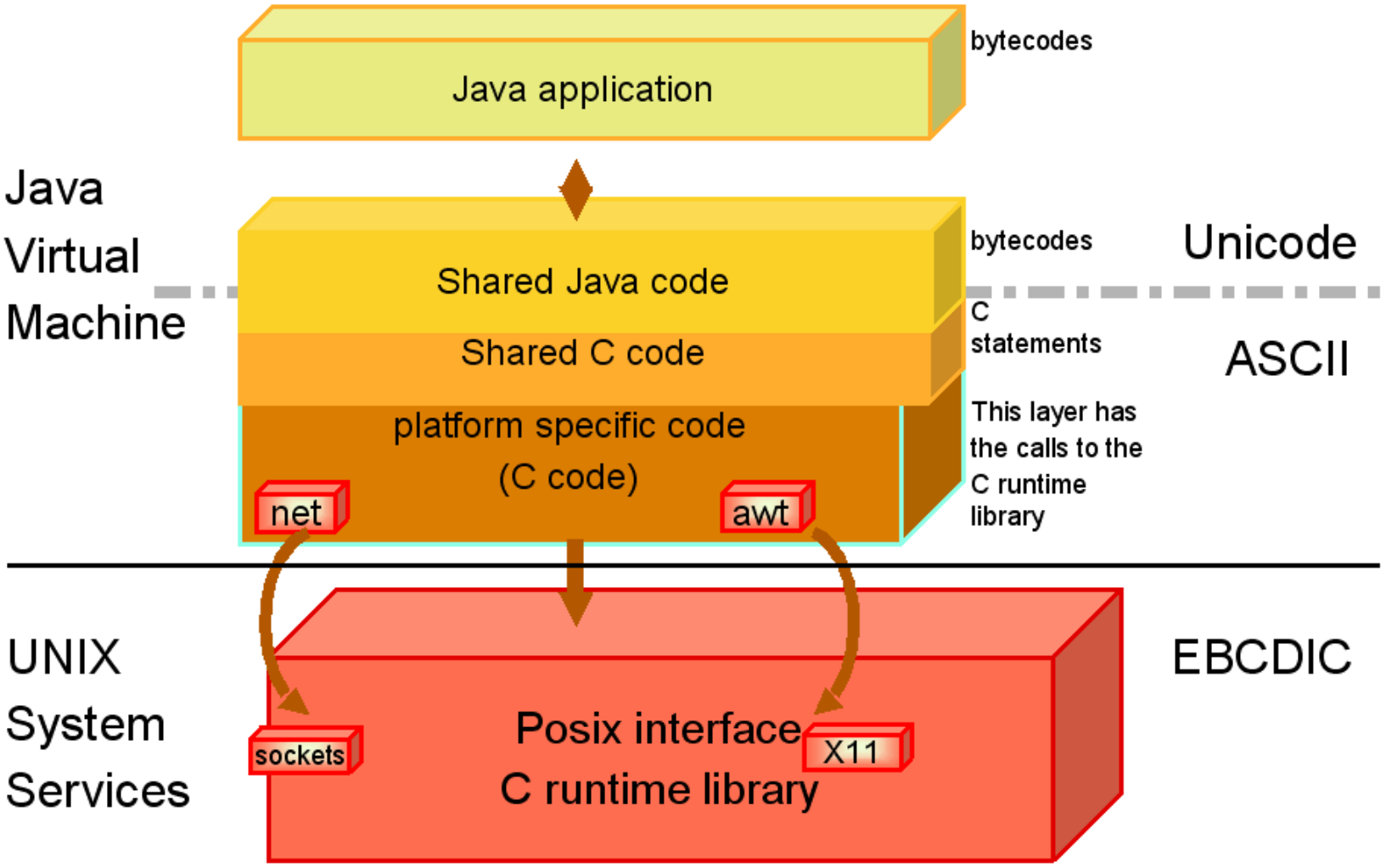
JVM unter z/OS und zLinux

Die JVM ist auf den meisten Plattformen teilweise in C/C++ implementiert, so auch unter zLinux. Die zLinux JVM geht davon aus, dass alle Daten in ASCII gespeichert sind; der Zugriff auf EBCDIC Daten erfordert zusätzliche Maßnahmen.

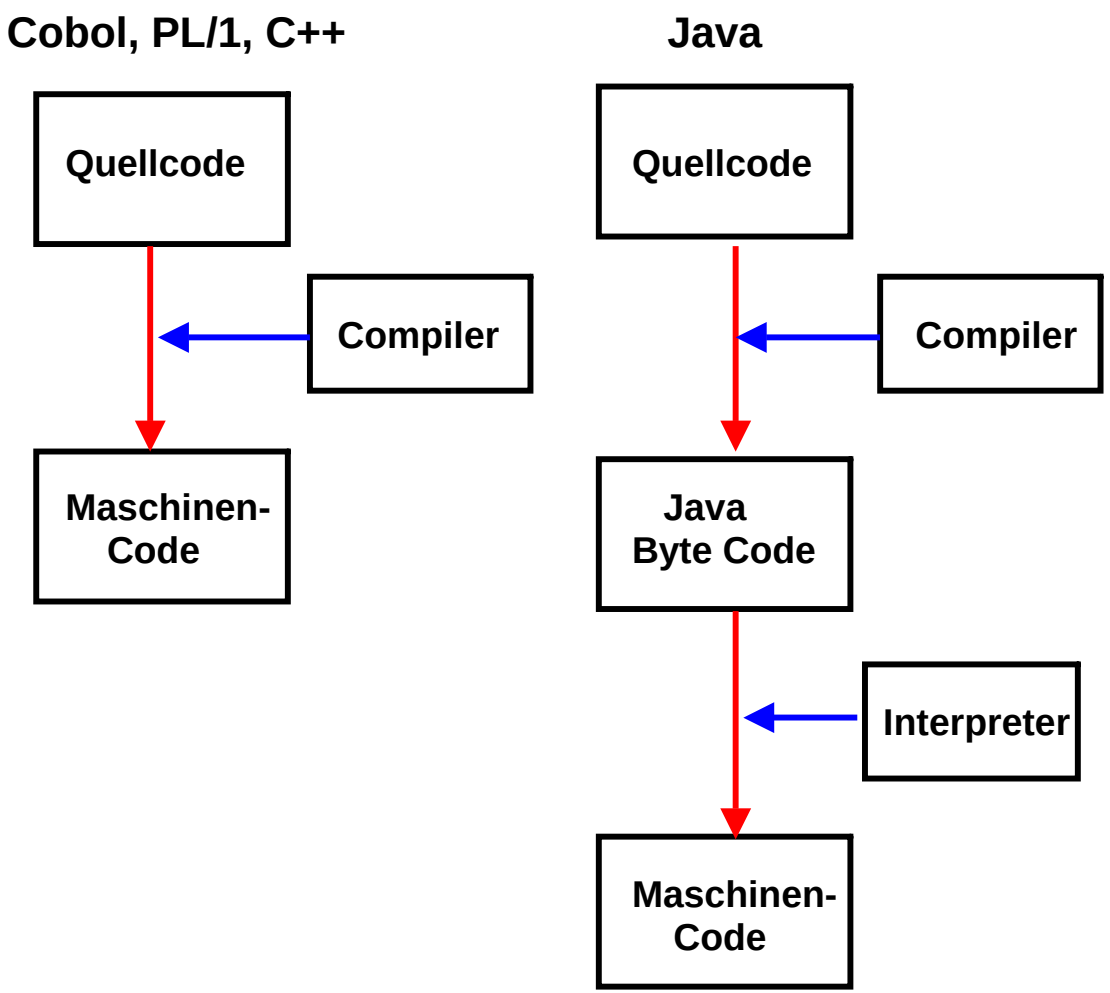
Unter z/OS ist es komplizierter. Die JVM ist als Teil der Unix System Services (USS) implementiert und enthält aus Kompatibilitätsgründen ebenfalls ASCII Support. Die allermeisten z/OS Daten sind jedoch in EBCDIC gespeichert. Die JVM bemüht sich, bei einer Kommunikation mit JVMs auf anderen Plattformen diesen Unterschied weitgehend unsichtbar zu machen.

Wird die Environment Variable IBM_JAVA_ENABLE_ASCII_FILETAG definiert, und File.Encoding für eine von z/OS unterstützte ASCII Code Page spezifiziert, dann werden alle neu angelegten Files mit einem "Coded Character Set Identifier" (CCSID) Tag versehen, das dieser Code Page entspricht. CCSID ist eine 16-bit Ziffer, die das Encoding spezifiziert.

Die folgende Abbildung zeigt die Struktur der JVM unter z/OS Unix System Services.



Implementierung der JVM unter z/OS Unix System Services



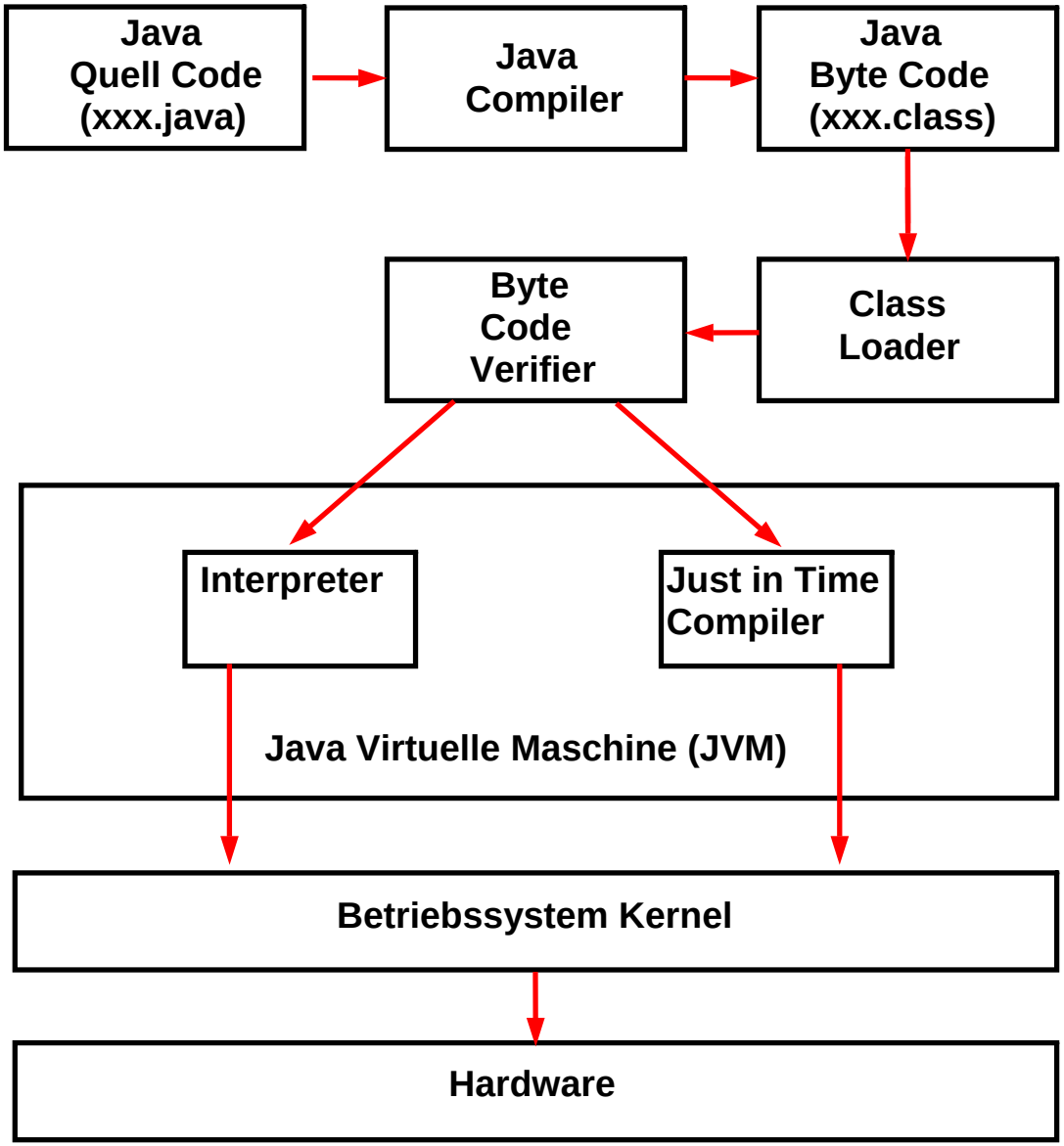
Java Programmausführung

Cobol, PL/1 oder C/C++ Quell Code wird durch einen optimierenden Compiler (optimising compiler) in Object Code übersetzt, aus dem nach dem Linking und Loading ausführbarer Maschinencode entsteht.

Bei Java erzeugt der Compiler statt dessen Plattform-unabhängigen Byte Code, der anschließend in einer JVM durch einen Interpreter oder Just-in-Time Compiler ausgeführt wird.

Die JVM selbst ist plattformabhängig. Sie ist typischerweise in C++ implementiert.

Optimizing Compiler werden für Sprachen wie Cobol, PL/1, C++ eingesetzt; sie erzeugen für die Ausführung besonders schnellen Maschinencode. Interpreter und Just in Time Compiler (JIT) sind in der Regel merklich langsamer.

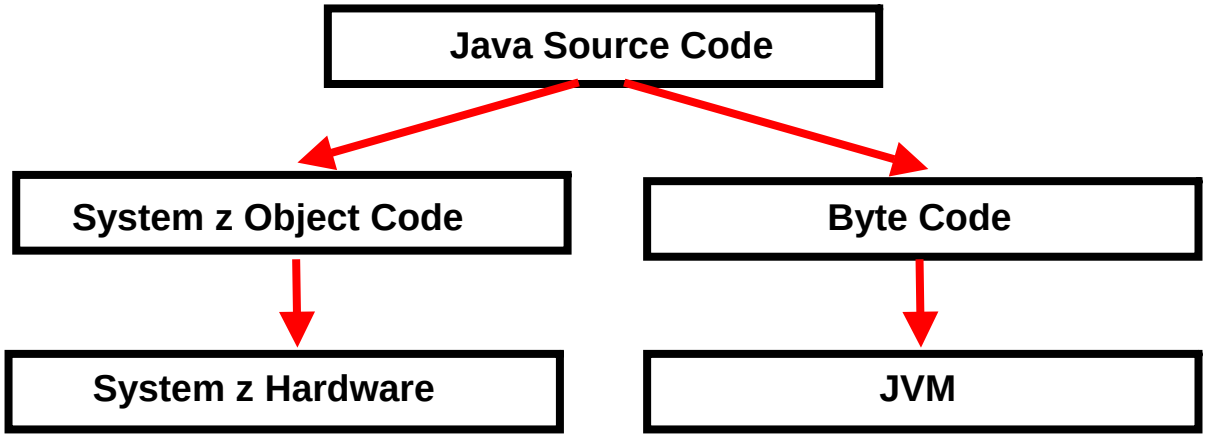


Java Virtuelle Maschine

Der Java-Compiler übersetzt Java Quellcode in Java-Objekt-Code (universell als Java Byte Code bezeichnet).

Der kompilierte Code wird mit Hilfe eines Class Loaders und eines Byte-Code Verifiers in die JVM zur Ausführung geladen.

Der Code wird mit Hilfe eines Interpreters oder eines Just-in-Time-Compiler ausgeführt.



Unter System z ist die Ausführung sowohl als System z Object Code als auch als Byte Code möglich.

The z/OS High Performance Java Compiler hat die Option, System z Object Code (binaries) an Stelle von Java Byte Code zu generieren. Der System z Object Code kann dann direkt in den Hauptspeicher ohne Benutzung einer JVM gelinked und geladen werden.

Der übersetzte Objekt Code ist nicht mehr portierbar, hat aber eine kürzere Ausführungszeit. Es ist möglich, im Compiler Durchlauf gleichzeitig Byte Code zu erzeugen, der dann auch in anderen Umgebungen ausgeführt werden kann.

Java Stand-alone Anwendungen unter z/OS

Die meisten Java Anwendungen laufen unter z/OS innerhalb eines Application Servers wie dem CICS Transaction Server oder dem WebSphere Application Server (WAS). Eine Alternative sind Java Stand-alone Anwendungen.

Wir definieren eine Java Stand-alone Anwendung als eine Anwendung, die nicht innerhalb eines Applikationsservers läuft. Ein Java Stand-alone Anwendung verfügt über ein Main-Methode, die ähnlich wie der Main entry Point eines traditionellen COBOL oder PL/1 Lademoduls arbeitet. Die Main Method ist die erste Methode, die ausgeführt wird wenn die Anwendung gestartet wird. Sie kann Parameter enthalten, die der Anwendung übergeben werden.

Ein Java Stand-alone Anwendung läuft in einer eigenen Java Virtual Machine (JVM). Diese muss gestartet werden, ehe die ersten Befehle der Anwendung ausgeführt werden.

Ein Java Stand-alone Anwendung ist autonom und benötigt einen Trigger um gestartet zu werden. Dieser Trigger kann sein:

- **Jemand gibt manuell einen Befehl auf der Unix System Services Command Line ein, um eine JVM mit der Anwendung zu starten.**
- **Ein Job Scheduler started ein JCL Script, das eine JVM mit der Anwendung startet.**

Zum Thema Java Stand-alone Anwendungen existieren zwei Redbooks:

http://www.redbooks.ibm.com/abstracts/sg247177.html
http://www.redbooks.ibm.com/abstracts/sg247291.html

Java Record I/O (JRIO) on z/OS

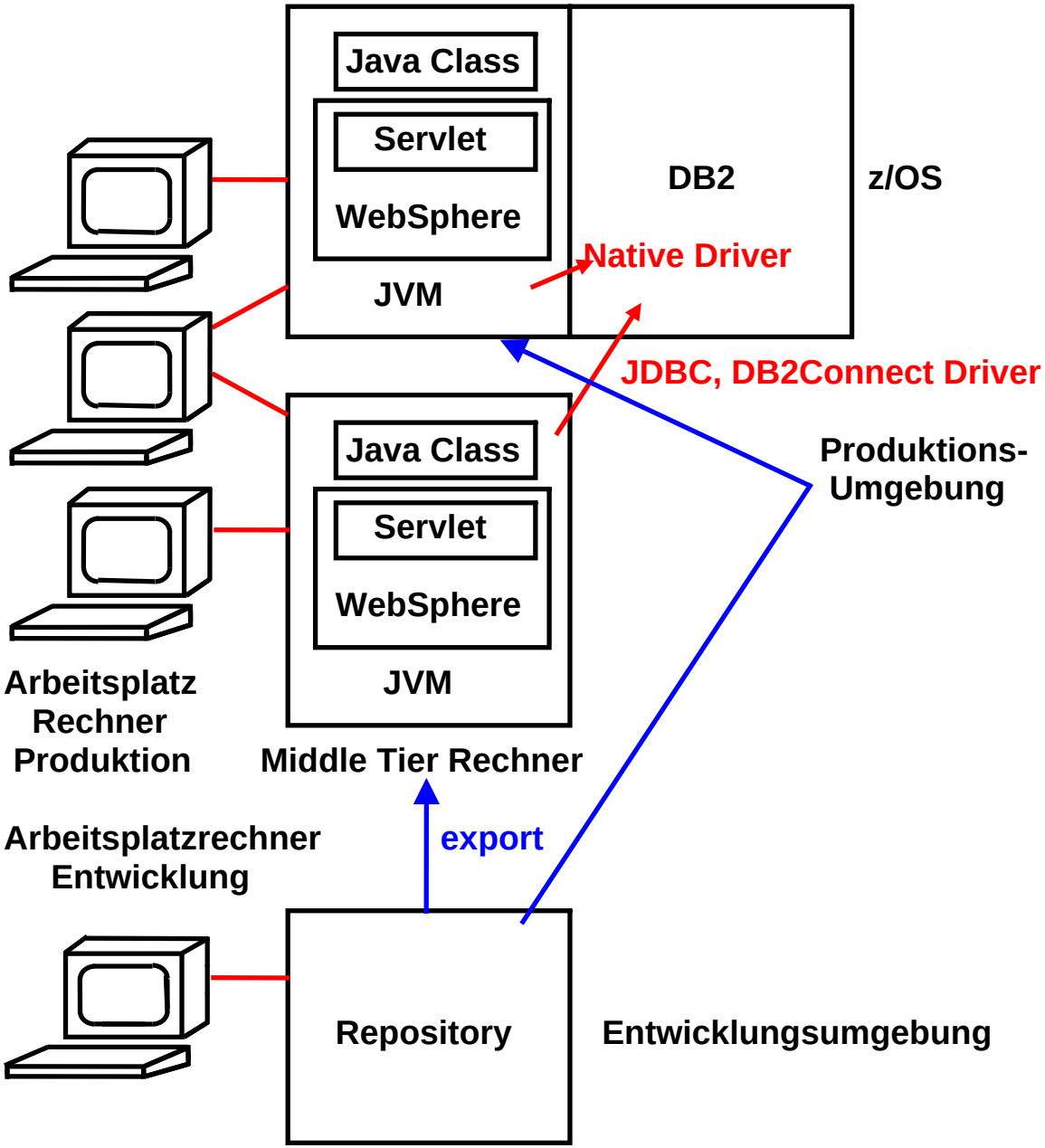
Eines der wichtigsten z/OS Funktionen ist die Verfügbarkeit von Record oriented Files (Data Sets) zusätzlich zu unstrukturierten Dateien.

JRIO ist eine Klassenbibliothek, ähnlich zu java.io. Während java.io Byte-orientierte oder feldorientierte Zugriffe auf Dateien ermöglicht, bietet JRIO einen Record-orientierten Zugang. Sie können mit JRIO auf VSAM-Datensätze und non-VSAM-Datensätze (PDS oder sequentiell) und HFS-Dateien zuzugreifen:

- **VSAM-Datensätze (nur KSDS)**
- **Non-VSAM Record orientierte Datensätze**
- **System Katalog**
- **Partitioniertee Data Set (PDS) Directory**
- **DDNAME Unterstützung**

In VSAM KSDS können Sie auf Datensätze in Entry Sequence Reihenfolge oder durch primäre eindeutigen Schlüssel zugreifen. JRIO bietet indizierten I/O-Zugriff auf Datensätze innerhalb eines VSAM KSDS mit z/OS nativer Unterstützung.

Die Java Record I/O Funktionalität ist als Teil des IBM JZOS Batch Toolkits verfügbar.



Trennung zwischen Entwicklung und Produktion

Die **Entwicklung** neuer Java Anwendungen kann mit Hilfe des JDK erfolgen. Sie erfolgt jedoch typischerweise in einer Entwicklungsumgebung (Integrated Development Environment, IDE).

Die **Ausführung** einer neu entwickelten Java Anwendung erfolgt häufig auf einem Web Application Server. Dieser befindet sich

- entweder auf einem Middle Tier Server, der mit dem Mainframe verbunden ist, oder
- unmittelbar auf dem Mainframe Rechner

Auf dem Mainframe läuft der Web Application Server entweder unter zLinux oder unter z/OS Unix System Services.

Entwicklungsumgebung

Integrated Development Environment (IDE)

Entwicklungsumgebungen sind Software Produkte, die von viele Herstellern erhältlich sind.

Weit verbreitet sind Eclipse , NetBeans, und IntelliJ IDEA

Eclipse und NetBeans sind Open Source. Eclipse kann heruntergeladen werden von www.eclipse.org . Ca. 350 MByte.

Eclipse (wie auch NetBeans und IntelliJ IDEA) ist sehr leistungsfähig und sehr beliebt, erfordert aber einen nicht unerheblichen Lernaufwand.

Eclipse hat eine sehr offene Architektur, und damit leichte Erweiterungsmöglichkeiten, die als Plug-ins bezeichnet werden.

Das Eclipse WebSphere Rational Application Developer für System z (RDz) Plug-in ist für z/OS Entwicklungen die wichtigste moderne Entwicklungsumgebung. RDz wurde wiederholt von IBM umbenannt: WSED → WDz → RDz, blieb aber im Wesentlichen das gleiche Software-Produkt.

Eclipse wurde ursprünglich für Java Programmierer entwickelt. Spezifisch das RDz Plug-in unterstützt aber auch Entwicklungen in COBOL, PL/I, C/C++ und Assembler.