

Einführung in den Compilerbau

Java Virtual Machine



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Einführung in den Compilerbau
WS 2017/18

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

- ▶ Java auf Platz 3 der populärsten Programmiersprachen ¹
- ▶ Jeder von Ihnen benutzt täglich Programme, die in Java geschrieben sind...
 - ▶ ...auf Ihrem Rechner
 - ▶ ...auf Ihrem mobilen Android-Device
 - ▶ ...in Ihrer Waschmaschine
 - ▶ ...in Einführung in den Compilerbau ;)

¹spectrum.ieee.org/computing/software/the-2017-top-programming-languages



Java ist auf vielen verschiedenen Plattformen verfügbar:

- ▶ Desktop- und Server-Maschinen
- ▶ Mobile Endgeräte, insbesondere Android-Plattform
- ▶ Eingebettete Systeme, z.B.
 - ▶ Kaffeemaschinen
 - ▶ Waschmaschinen
 - ▶ Roboter
 - ▶ Fahrzeuge

Überall wo ein Java-Programm ausgeführt werden soll, muss die Java Plattform vorhanden sein

Die sog. Java Plattform besteht aus 2 Hauptkomponenten

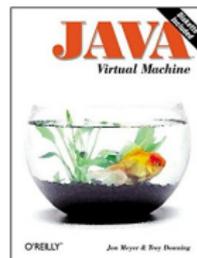
- ▶ Java API („Standardumgebung“)
 - ▶ Stellt Schnittstellen für grundlegende Funktionalitäten und Zugriff auf Ressourcen des Betriebssystems (Dateisystem, Netzwerk, etc.) bereit

- ▶ Java Virtual Machine (JVM)
 - ▶ Lädt vom Compiler erzeugte Klassen
 - ▶ Führt die geladenen Programme aus
 - ▶ Verwaltet und bereinigt den Speicher automatisch

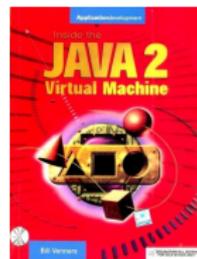


- ▶ Wie ist die JVM intern aufgebaut?
- ▶ Wie funktioniert die Ausführung eines Programms auf der JVM?
- ▶ Wie kann im Compiler Code für die JVM erzeugt werden?
- ▶ Welche Vor- und Nachteile bietet eine virtuelle Ausführungseinheit wie die JVM?

- ▶ Zwei etwas veraltete Lehrbücher
 - ▶ Meyer & Downing: *Java Virtual Machine*
 - ▶ Venners: *Inside the Java 2 Virtual Machine*



- ▶ Die offizielle Spezifikation:
<https://docs.oracle.com/javase/specs/jvms/se9/jvms9.pdf>
- ▶ Eine Version des MAVL-Compilers mit Backend für die Java Virtual Machine im Moodle





Java Virtual Machine



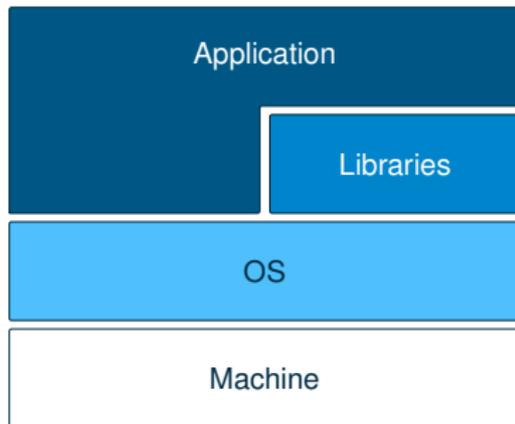
- ▶ Virtuelle Maschine: „Maschine auf der Maschine“
- ▶ Definiert eine Umgebung zur Ausführung von Bytecode unabhängig von zugrundeliegender Maschine
- ▶ Semantik und Funktionsweise der JVM durch Spezifikation vorgegeben²
- ▶ Verschiedene Implementierungen möglich, solange Spezifikation erfüllt wird
- ▶ Weitere Beispiele von JVM-Sprachen neben Java:
 - ▶ Scala
 - ▶ Groovy
 - ▶ Clojure
 - ▶ Kotlin

²<https://docs.oracle.com/javase/specs/jvms/se9/html/index.html>

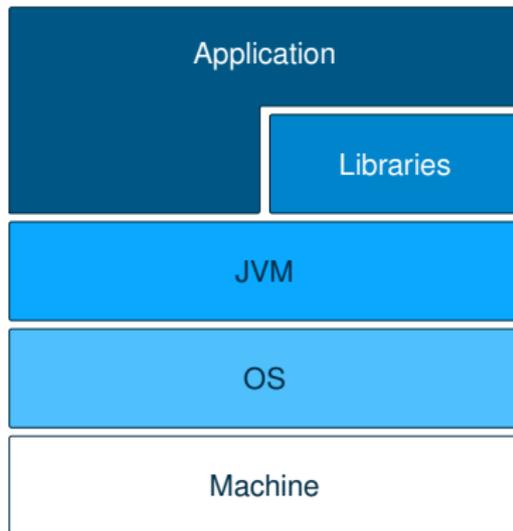
- ▶ Programme, die in einer JVM-Sprache verfasst sind, werden zu Bytecode kompiliert.
- ▶ Bytecode-Darstellung ist ein Mittelweg zwischen:
 - ▶ Interpretierten Sprachen (z.B. JavaScript)
 - ▶ Programm wird in seiner Textform von einem Interpreter ausgeführt, häufig verbunden mit vorheriger Verifikation und dynamischer Typprüfung
 - ▶ Maschinen-kompilierten Sprachen (z.B. C/C++)
 - ▶ Programm wird von Compiler und Assembler in Maschinencode für die zugrundeliegende Maschine übersetzt

- ▶ Deutlich schneller als interpretierte Sprachen
- ▶ Interpreter muss vor Ausführung das Programm parsen, verifizieren und Typ-Checks durchführen
- ▶ Bei Bytecode ist Betrachtung der aktuellen Instruktion ausreichend, viele Überprüfungen bereits im Compiler durchgeführt.
- ▶ Wird noch verstärkt durch JIT-Kompilierung:
 - ▶ 10-20% des Codes machen bis zu 80-90% der Laufzeit aus → Übersetze häufig ausgeführten Code zur Laufzeit in nativen Maschinencode
- ▶ Vorteil gegenüber Maschinen-kompilierten Sprachen: Portabilität des Bytecodes

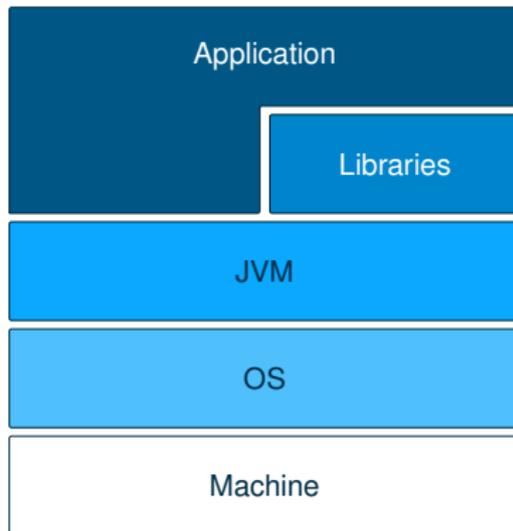
- ▶ Erzeugter Maschinencode und kompilierte Bibliotheken sind an die jeweils zugrundeliegende Maschine gebunden
- ▶ Portierung erfordert erneute Kompilierung
- ▶ Aufrufe zu Betriebssystem-funktionalitäten sind spezifisch für das jeweilige Betriebssystem und nicht portierbar



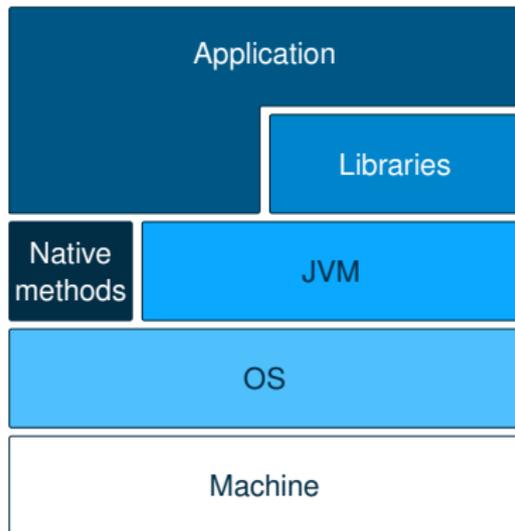
- ▶ JVM abstrahiert über zugrundeliegende Maschine und Betriebssystem
- ▶ Anwendung und Bibliotheken nutzen betriebssystem-unabhängige API
- ▶ Bytecode wird von JVM interpretiert bzw. auf jeweilige Maschine abgebildet



- ▶ Bitbreiten sind auf allen Plattformen identisch (im Gegensatz zu z.B. C/C++)
- ▶ Anwendungen und Bibliotheken sind portabel zwischen verschiedenen Maschinen
- ▶ Zur Erschließung neuer Plattformen ist lediglich eine Portierung der JVM notwendig



- ▶ Sonderfall: Java-Code nutzt nativen Code (z.B. C/C++) über das *Java Native Interface (JNI)*
- ▶ Native-Code ist wieder maschinenspezifisch und muss explizit portiert und verfügbar gemacht werden





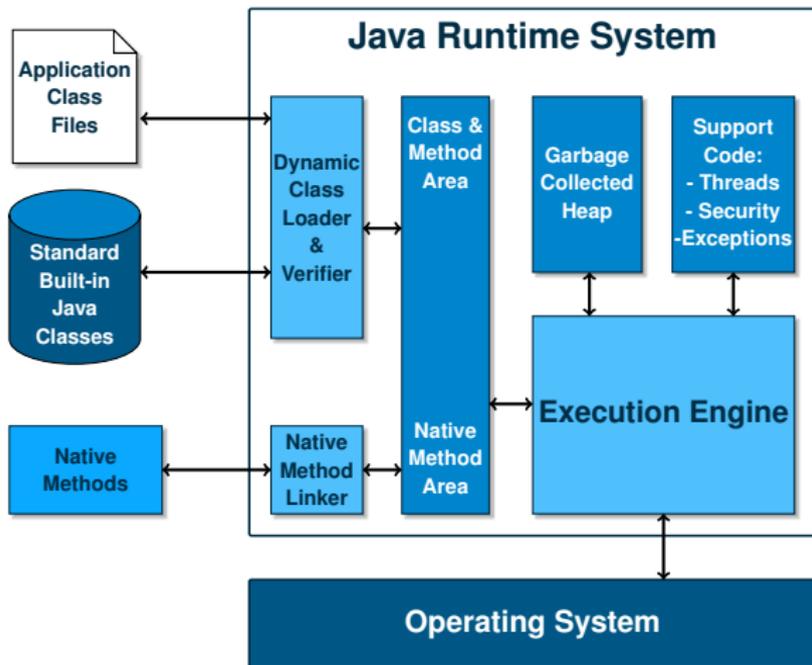
Weitere Vorteile von virtuellen Maschinen

- ▶ Implementierung von speziellen Operationen von unterstützten Sprachen möglich, z.B. `getField`, `multinewarray`
- ▶ Alle Anwendungen profitieren von einmaliger Optimierung der virtuellen Maschine

Nachteile von virtuellen Maschinen

- ▶ Zusätzliches Abstraktionslevel kann Ineffizienzen verursachen. Mit JIT-Compilation ist die Performance von Java aber besser als ihr Ruf.
- ▶ Zur Ausführung von Anwendungen muss die virtuelle Maschine immer verfügbar sein.

Interner Aufbau der Java Virtual Machine



Memory Manager

- ▶ Objekte und Arrays werden auf dem *Heap* abgelegt und automatisch gelöscht, wenn sie nicht mehr benötigt werden
- ▶ Methoden- und Klassendefinitionen werden in separatem Bereich abgelegt und ebenfalls automatisch verwaltet

Error- und Exception-Manager

- ▶ Errors und Exceptions können sowohl von der Java-Runtime (z.B. Out-of-memory) als auch von Nutzercode geworfen werden
- ▶ Exception-Manager handhabt Exceptions
- ▶ Von Methode abgefangene Exceptions werden in *Handler Table* definiert



Native Method Support

- ▶ Native Method: Inhalt der Funktion in anderer Programmiersprache (z.B. C/C++) verfasst und zu nativem Maschinencode kompiliert
- ▶ Native Method Linker lädt Code aus Bibliothek (.dll/.so)
- ▶ Bei Aufruf einer Native Method müssen Argumente und Return-Wert umgewandelt werden (*Marshalling*)

Thread Management

- ▶ JVM unterstützt parallele Ausführung mit mehreren Threads
- ▶ Nur zwei Spezialanweisungen, Thread-Management hauptsächlich über Methoden von `java.lang.Thread`

Class File

- ▶ Ein Class File enthält alle von der JVM benötigten Informationen zu einer Klasse
- ▶ Von einem JVM-Compiler (z.B. `javac`) wird ein Class File pro Klasse erzeugt
 - ▶ Bei geschachtelten Klassen werden mehrere Files erstellt, so entstehen aus dem folgenden Code

```
public class Test{  
    private class Inner{...}  
}
```

zwei Class Files: `Test.class` und `Test$Inner.class`

- ▶ Das Tool `javap` erlaubt die Untersuchung von Class Files



Repräsentation

- ▶ Effiziente Darstellung und Speicherung in binärem, byte-basiertem Format
- ▶ Elemente größer als 8 Bit werden als Folge von Bytes in *big-endian*-Reihenfolge dargestellt



Struktur

Interne Struktur besteht aus verschachtelten Tabellen:

- ▶ *Top-level Table*: Generelle Informationen zur Klasse (Name, Eltern-Klasse, Zugriffsrechte)
- ▶ *Constant Pool Table*
- ▶ *Interface Tabelle*: Liste aller von dieser Klasse implementierten Interfaces
- ▶ *Fields Table*: Liste aller Felder der Tabelle
- ▶ *Methods Table*: Liste aller Methoden
- ▶ *Attributes Table*: Aufzählung aller Attribute

Type Descriptors

JVM verwendet sogenannte *Type Descriptors* zur platzsparenden Speicherung von Typ-Informationen

Descriptor	Datentyp	Semantik
B	byte	Vorzeichenbehaftetes Byte
C	char	Character
D	double	Double-Precision IEEE-754 Gleitkommazahl
F	float	Single-Precision IEEE-754 Gleitkommazahl
I	int	Integer
J	long	Long Integer
S	short	Vorzeichenbehaftetes Short (16 Bit)
Z	boolean	true oder false

Klassentypen

- ▶ Zur Darstellung von Klassentypen wird ein String der Form „L<Klassenname>“ benutzt
- ▶ Im Package-Namen werden Punkte durch Schrägstriche ersetzt
- ▶ Beispiel: `java.lang.String` wird zu `Ljava/lang/String`;

Array-Typen

- ▶ Array-Typen werden mit einer Klammer gefolgt vom Element-Typen dargestellt:
 - ▶ `char[]` wird zu `[C`
 - ▶ `float[][]` wird zu `[[F`
 - ▶ `String[]` wird zu `[Ljava/lang/String`;

Methoden-Signaturen

- ▶ Zur Darstellung von Methoden-Signaturen wird ein String der Form „(<Argument-List><Return-Typ>“benutzt
- ▶ Beispielsweise wird **public int** foo(**char** c, **float** f, String s) zu (CFLjava/lang/String;)I
- ▶ Für Methoden mit Return-Typ **void** wird der spezielle Typdescriptor V verwendet, so wird **public void** baz() zu ()V



- ▶ Constant Pool enthält in seinen Einträgen eine ganze Reihe von Informationen:
 - ▶ Namen von Klassen, Methoden und Feldern
 - ▶ Konstante String-Literale
 - ▶ Konstanten der primitiven Typen **int**, **long**, **float** und **double**
- ▶ Zur Verwendung in Berechnungen bzw. in der Ausführung werden die Konstanten einfach mit ihrem Index referenziert
- ▶ Der Constant Pool spielt auch eine wichtige Rolle beim dynamischen Linken bzw. der dynamischen Auflösung von Referenzen zu anderen Klassen und deren Methoden



Einträge im Constant Pool von A für
Aufruf von B.baz:

1. String "baz"
2. String "B"
3. Type Descriptor (I)V
4. Klassen-Referenz für Klasse B,
referenziert 2.
5. NameAndType für Methode
baz, referenziert 1. und 3.
6. MethodenReferenz für baz,
referenziert 4. und 5.

```
public class A{  
    public static void foo(){  
        B.baz("Hello, World!");  
    }  
}
```

```
public class B{  
    public static void baz(int n){  
        ...  
    }  
}
```



```
[...]  
#2 = Methodref          #13.#14          // B.baz:(I)V  
[...]  
#13 = Class             #17              // B  
#14 = NameAndType      #18:#19         // baz:(I)V  
[...]  
#17 = Utf8              B  
#18 = Utf8              baz  
#19 = Utf8              (I)V
```

- ▶ Herzstück des dynamischen Linking in der JVM: Bei Aufruf von Methoden anderer Klassen wird der Class Loader zur Auflösung der Referenz verwendet
- ▶ Lokalisiert und lädt die Klassendefinition und macht sie in der JVM verfügbar
- ▶ Auflösen kann rekursive Aufrufe bedingen, z.B. Laden der Super-Klasse einer referenzierten Klasse
- ▶ Jede Klasse wird nur einmal aufgelöst, anschließend ist sie in der JVM verfügbar

Beim Laden einer Klasse für der Class Loader folgende Schritte aus:

1. Loading: Class File laden
2. Linking: Umwandlung des Class Files in JVM-interne Datenstrukturen (siehe Class- and Method-Area)
3. Verification: Überprüfung der Klassenstruktur und Sicherheitsprüfung durch den Security Manager
4. Preparation: Weitere Checks und Initialisierung der statischen Felder
5. Initialization: Aufruf der `<clinit>`-Methode nachdem alle Super-Klassen initialisiert wurden
6. Resolution: Auflösung von Einträgen im Constant Pool der gerade geladenen Klasse



- ▶ Die Execution Engine ist das Herzstück der JVM
- ▶ „Virtueller Prozessor“, pro Thread eine Execution Engine. Virtuelle Prozessoren werden auf CPU-Kerne der Maschine abgebildet
- ▶ Heap-Speicher (Objekte und Arrays) und Methodenspeicher zwischen verschiedenen Instanzen geteilt
- ▶ Hauptaufgabe: Ausführung der Bytecode-Instruktionen
- ▶ Häufig ausgeführter Code wird zur schnelleren Ausführung per JIT-Compilation in Maschinencode umgewandelt



Die Zahl der möglichen Opcodes in der JVM ist auf 256 limitiert (1 Byte). Daher gibt es nicht für jeden Java-Datentypen eine direkte Entsprechung in der JVM:

- ▶ **int**, **long**, **float**, **double** sind die grundlegenden arithmetischen Typen auf der JVM
 - ▶ Für diese Typen sind die arithmetischen Operationen explizit definiert
 - ▶ Wörter auf der JVM sind grundsätzlich 32 Bit breit, **long** und **double** belegen daher stets 2 Worte
- ▶ **short**, **char**, **byte** sind sogenannte *storage types*
 - ▶ Können als Typen von Feldern und Arrays verwendet werden
 - ▶ Nur Speicheroperationen zur effizienten Speicherung der Elemente definiert
 - ▶ Zur Verwendung in Berechnungen erfolgt immer eine Umwandlung in **int**
- ▶ **boolean** werden als **int** mit Wert 0 oder 1 gehandhabt

- ▶ Objekte und Arrays werden immer auf dem Heap abgelegt, Elemente des Typs `reference` verweisen auf Instanz
- ▶ In Java werden Objekte und Arrays immer als Referenzen übergeben, kein „call-by-value“ mit Objekten/Arrays
- ▶ Eine direkte Einbettung von Objekten in Objekten ist nicht möglich, immer nur per Referenz:

```
class Point{float x,y;};  
class Line{Point p1, p2;};
```

- ▶ Keine Pointer-Arithmetik mit `reference` möglich



Die Ausführung von Java-Programmen ist in Stack-Frames organisiert:

- ▶ Pro Thread/Execution Engine ein sogenannter *Java Stack*
- ▶ Bei Aufruf einer Methode wird ein neuer Stack-Frame auf dem Stack angelegt
- ▶ Nur der oberste Stack-Frame gilt als aktiv
- ▶ Beim Return wird der aktuellste Stack-Frame vom Stack entfernt
- ▶ Ein Stack-Frame erfasst den aktuellen Ausführungszustand in den folgenden Elementen
 - ▶ Register
 - ▶ Operand Stack
 - ▶ Local Variables



Register

- ▶ Keine allgemeinen Register, sondern Register für Zustandsvariablen
 - ▶ Stack-Top Index
 - ▶ Zustandsinformationen zum Thread
 - ▶ Pointer...
 - ▶ ...zur aktuellen Methode
 - ▶ ...zur aktuellen Klasse
 - ▶ ...dem Constant Pool der aktuellen Klasse
 - ▶ ...zum aktuellen Stack-Frame
 - ▶ Program Counter

- ▶ Keine Instruktionen zur direkten Manipulation der Register vorhanden



Operand Stack

- ▶ JVM ist eine stack-basierte Machine, alle Berechnungen werden auf dem Stack durchgeführt
- ▶ Elemente auf dem Stack sind 1 Wort (32 Bit) breit, **double** und **long** belegen 2 Elemente
- ▶ Operand Stacks sind pro Methode isoliert, kein Zugriff auf Stacks anderer Methoden



Local Variables

- ▶ Menge von nummerierten Local Variables
- ▶ Enthält:
 - ▶ **this**-Parameter (außer bei **static**-Methoden)
 - ▶ Parameter
 - ▶ Methoden-lokale Variablen
 - ▶ Häufig benutzte Werte/Berechnungen
- ▶ Implementierungen können Local Variables auf Hardwareregister der zugrundeliegenden Maschine abbilden



Exkurs: ASM-Framework



- ▶ Ergebnis der Code Generierung für die JVM ist eine binäre Byte-Datei
- ▶ Direkte Generierung dieser Datei möglich, aber kompliziert
- ▶ Neben den eigentlichen Instruktionen enthält Bytecode noch weitere Informationen, die hinzugefügt werden müssen, z.B.
 - ▶ Maximale Größe des Operandenstacks pro Funktion
 - ▶ Maximale Anzahl lokaler Variablen pro Funktion
 - ▶ *Stack Map Frames* zur schnelleren Verifikation

- ▶ Daher besser Verwendung eines Frameworks zur JVM Code Generierung

- ▶ In dieser Vorlesung verwenden wir daher das ASM-Framework³
- ▶ Verfügbar unter Open-Source Lizenz
- ▶ Ermöglicht Analyse, Transformation und Generierung von JVM-Bytecode
- ▶ Sehr benutzerfreundlich und effizient
- ▶ Zwei verschiedene APIs verfügbar
 - ▶ Event-based API: Analyse/Generierung als Sequenz von Events
 - ▶ Tree-based API: Darstellung und Manipulation als Baum von Objekten
- ▶ Wir verwenden die Event-based API in unseren Beispielen

³<http://asm.ow2.org/>

Erzeuge leere, **public** Klasse mit Default-Konstruktor, die von Object erbt:

```
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);

cw.visit(Opcodes.V1_6, Opcodes.ACC_PUBLIC, className, null,
        "java/lang/Object", null);

Method m = Method.getMethod("void <init> ()");
currentGenerator = new GeneratorAdapter(ACC_PUBLIC, m, null,
        null, cw);
currentGenerator.loadThis();
currentGenerator.invokeConstructor(Type.getType(Object.class),
        m);
currentGenerator.returnValue();
currentGenerator.endMethod();
...
cw.visitEnd();
```



Erzeuge neue **public static** Funktion für MAVL-Funktion mit Namen `m`

```
GeneratorAdapter currentGenerator = new  
    GeneratorAdapter(ACC_PUBLIC + ACC_STATIC, m, null, null,  
        cw);
```

...

```
currentGenerator.endMethod();
```



Code-Generierung für die JVM

- ▶ JVM Bytecode ist typisiert, d.h. es werden verschiedene Operationen für verschiedene Typen von Operanden verwendet
- ▶ Erlaubt z.B. Typprüfungen durch den Bytecode-Verifier
- ▶ Typ wird durch Präfix gekennzeichnet

Typ	Präfix	
int	i	
long	l	
float	f	
double	d	
byte	b	} Storage Types, nur für Load & Store
char	c	
short	s	
reference	a	} Verwendet für Objekte und Arrays



- ▶ Verwalten von Daten im Programm, umfasst Laden, Speichern und Verschieben von Daten
- ▶ Drei grundsätzliche Arten von Speicher
 - ▶ Operandenstack
 - ▶ Local Variables
 - ▶ Objekte und Arrays auf dem Heap
- ▶ Das Laden von Konstanten, auch aus dem Constant Pool, zählt ebenfalls zu den Aufgaben

- ▶ Allgemein: Laden von Konstanten aus dem Constant Pool mit **ldc**, **ldc_w**, **ldc2_w**
 - ▶ **ldc** lädt eine Single-Word Konstante aus den ersten 256 Einträgen
 - ▶ **ldc_w** verwendet breiteren Index und kann Single-Word Konstanten aus dem gesamten Pool laden
 - ▶ **ldc2_w** lädt Double-Word Konstanten (**long**, **double**)
- ▶ Spezialinstruktionen für arithmetische Konstanten, z.B.:
 - ▶ **bipush/sibush** für 1/2 Byte große, vorzeichenbehaftete Konstanten
 - ▶ **iconst_m1** für -1
 - ▶ **iconst_<n>** für Integer-Konstanten mit $0 \leq n \leq 5$
- ▶ Weitere Spezialinstruktionen für andere arithmetische Typen und **null**



- ▶ Das ASM-Framework trifft automatisch die Auswahl der passendsten Instruktion zum Laden einer Konstanten
- ▶ Folgender Code ist ausreichend für die Code Generierung von MAVL Integer Literalen:

```
@Override
public Instruction visitIntValue(IntValue intValue, Integer
    -)
{
    currentGenerator.push(intValue.getValue());
    return null;
}
```

Folgender Code wird dabei von ASM erzeugt:

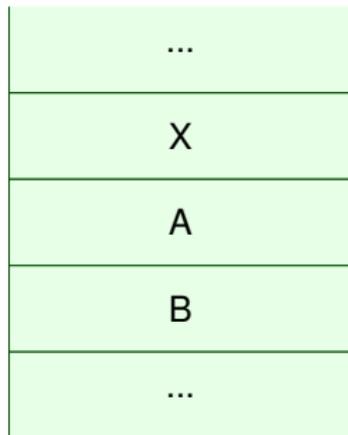
- ▶ Für das Integer-Literal 1: **iconst_1**
- ▶ Für das Integer-Literal 42: **bipush 42**
- ▶ Für das Integer-Literal -1337:
sipush 1337
ineg
- ▶ Für das Integer-Literal 45123:
Constant Pool:
#17 = Integer 45123
...
ldc #17



Verschiedene Operationen zur Manipulation von Elementen auf dem Operandenstack

- ▶ **pop**: Entferne das oberste Wort vom Stack
- ▶ **pop2**: Entferne die beiden obersten Worte vom Stack
- ▶ **dup**: Dupliziere das oberste Wort
- ▶ **dup2**: Dupliziere die obersten beiden Worte
- ▶ **swap**: Vertausche die obersten beiden Worte

Auf Double-Words (**long**, **double**) dürfen nur pop2 und dup2 angewendet werden!



- ▶ Um die Daten in lokalen Variablen zu modifizieren, müssen diese immer zuerst auf/vom den Operandenstack geladen/gespeichert werden. Eine direkte Bearbeitung der lokalen Variablen ist nicht möglich
 - ▶ Einzige Ausnahme: **inc** erlaubt das direkte Inkrementieren von Integer-Variablen
- ▶ Grundsätzlich zwei Operationen, wobei **<p>** eines der Präfixe **i, l, d, f, a** sein muss:
 - ▶ **<p>load** zum Laden von Werten auf den Stack
 - ▶ **<p>store** zum Speichern von Werten vom Stack|
- ▶ Lokale Variablen werden dabei per Index identifiziert

Das ASM-Framework stellt einfache Methoden bereit, um Code für den Zugriff auf lokale Variablen zu generieren:

- ▶ Anlegen einer neuen lokalen Variable vom Typ `type`:
`int offset = currentGenerator.newLocal(type);`
- ▶ Laden eines primitiven Funktionsarguments `arg`:
`currentGenerator.loadArg(arg);`
- ▶ Laden einer lokalen Variablen mit Typ `type` an Index `offset`:
`currentGenerator.loadLocal(offset, type);`
- ▶ Speichern in eine primitive lokale Variable mit Typ `type` an Index `offset`:
`currentGenerator.storeLocal(offset, type);`



Aus dem folgenden MAVL-Code

```
function void foo(int a){  
    val int b = 5;  
    var int c;  
    c = a + b;  
}
```

erzeugt der Compiler diesen Bytecode:

```
0: iconst_5  
1: istore_1  
2: iload_0  
3: iload_1  
4: iadd  
5: istore_2  
6: return
```

- ▶ Arrays werden auf dem Heap verwaltet, lokale Variable enthält nur *Referenz*
- ▶ Spezielle Anweisungen zum Erzeugen neuer Arrays:
 - ▶ **newarray** Erzeugt ein neues Array mit primitiven Elementen
 - ▶ **anewarray** Erzeugt ein neues Array von Objekten/Arrays
 - ▶ **multianewarray** Erzeugt ein neues mehrdimensionales Array
- ▶ Unser MAVL-JVM-Compiler bildet Vektoren und Matrizen auf Arrays ab (Matrizen werden zu Arrays von Arrays)
- ▶ Folgender Code aus dem ASM-Framework erzeugt ein neues Array vom Typ `type`:
`currentGenerator newArray(type);`



Für MAVL-Vektoren `var vector<int>[5] b:`

```
0: iconst_5
1: newarray      int
3: astore_1
```

Für MAVL-Matrizen `var matrix<int>[3][3] A:`

Constant Pool:

```
#11 = Utf8      [I
#12 = Class    #11
...
4: iconst_3
5: anewarray   #12
```

MAVL-Matrix wird anschließend noch mit Referenzen auf die Arrays für die einzelnen Zeilen befüllt:

```
15: dup           // Duplizieren der Referenz auf die Matrix
16: iload_3      // Laden des aktuellen Index (Counter)
17: iconst_3     // Laden der Konstante 3
18: newarray int // Erzeuge neues Array fuer Zeile
20: aastore      // Speichere neues Array an Index in Matrix
```

Das Ganze geschieht in einer Schleife, hier nur ein Ausschnitt des Schleifenkörpers gezeigt



- ▶ Lesender/schreibender Zugriff auf Elemente eines Arrays mit **<p>a**load**** bzw. **<p>a**store****, wobei **<p>** eines der Präfixe **i, l, f, d, a, b, c, s** sein muss
- ▶ Lokale Variable speichert nur Referenz, Laden daher dreistufiger Zugriff:
 1. Laden der Array-Referenz aus lokaler Variable
 2. Ablegen des gewünschten Index auf dem Stack
 3. **<p>a**load**** konsumiert die beiden Werte und legt stattdessen den gewünschten Wert auf den Stack
- ▶ Analoges Vorgehen beim Schreiben, vor Aufruf wird der zu schreibende Wert abgelegt



Ausgehend von folgenden MAVL-Deklarationen

```
var vector<int>[5] b;  
var vector<int>[3] c;
```

ergibt sich aus `c[1]=b[3]` der folgende Bytecode:

```
8: aload_1    // Laden von Referenz auf b  
9: iconst_3   // Index 3  
10: iaload    // Laden von b[3]  
11: aload_2   // Laden von Referenz auf c  
12: swap     // b[3] und Referenz tauschen, b[3] jetzt oben  
13: iconst_1  // Index 1  
14: swap     // b[3] und Index tauschen, neue Reihenfolge:  
           // b[3], Index(1), Referenz auf c  
15: iastore   // Speichern in c[1]
```



- ▶ Für alle mathematischen Operatoren in Java gibt es eine Entsprechung auf der JVM
- ▶ Alle arithmetischen Operationen arbeiten auf dem Stack
 - ▶ Operanden müssen vor der arithmetischen Operation in richtiger Reihenfolge auf dem Stack abgelegt werden
 - ▶ Operation konsumiert die entsprechende Zahl von Stackelementen und legt das Ergebnis wiederum als oberstes Stackelement ab
- ▶ Integer-Division durch 0 führt zu Exception
- ▶ Integer-Rechnung kann zu Overflow/Underflow führen
- ▶ Keine Exceptions bei Float-Rechnung, stattdessen Verwendung der IEEE-754-Sonderwerte **(-)NaN**, **(-)Inf**, etc.



Für die vier arithmetischen Typen auf der JVM (**int**, **long**, **float**, **double**) sind die folgenden binären arithmetischen Operationen definiert

- ▶ **<p>add**
- ▶ **<p>sub**
- ▶ **<p>mul**
- ▶ **<p>div**
- ▶ **<p>rem**

Zusätzlich ist noch die unäre Operation

- ▶ **<p>neg**

definiert, um einen Wert (mathematisch) zu negieren.

Dabei muss **<p>** eines der Präfixe **i**, **l**, **f**, **d** sein



Für die beiden ganzzahligen arithmetischen JVM-Typen **int** und **long** sind zusätzlich die folgenden logischen Operationen definiert:

- ▶ **<p>shl** Links-Shift
- ▶ **<p>shr** Arithmetischer Rechts-Shift
- ▶ **<p>ushr** Logischer Rechts-Shift
- ▶ **<p>and** Bitweises UND
- ▶ **<p>or** Bitweises ODER
- ▶ **<p>xor** Bitweises exklusives ODER

Hier ist die Menge der Präfixe für **<p>** auf **i** und **l** beschränkt



Der folgende Code dient zur Erzeugung einer arithmetischen/logischen Operation op auf dem Typ type:

```
currentGenerator.math(op, type);
```

Erzeugter Code:

```
var int a;
```

```
var int b;
```

```
var float c;
```

```
var float d;
```

```
val int x = a * b;
```

```
val float z = c + d;
```

0: **iload_1**

1: **iload_2**

2: **imul**

3: **istore** 5

5: **fload_3**

6: **fload** 4

8: **fadd**

9: **fstore** 6



Im Gegensatz zu Java, wo die Addition eines Float mit einem Int erlaubt ist, gibt es auf der JVM keine impliziten Typ-Konversionen. Deshalb müssen Typen vor einer Rechnung explizit konvertiert werden, dafür stehen die folgenden Operationen zur Verfügung:

- ▶ **<s>2<t>** für Konversionen vom Typ **<s>** zu Typ **<t>**, definiert für alle Kombinationen der vier arithmetischen Typen **int**, **long**, **float**, **double** mit den Typ-Bezeichnern **i, l, f, d**
- ▶ **i2<t>** für das Abschneiden von Integer-Werten auf einen der drei Storage-Typen **byte**, **char**, **short** mit Typ-Bezeichnern **b, c, s**

Bei Konversion von Gleitkommazahlen zu ganzzahligen Werten kommt der sog. *Round-towards-Zero*-Modus zum Einsatz, d.h. $9.6 \rightarrow 9$

- ▶ Die JVM verfügt über bedingte und unbedingte (**goto**) Sprung-Operationen
- ▶ Die Abbildung von Kontrollfluss-Konstrukten wie Verzweigungen oder Schleifen funktioniert ähnlich wie Sie das bereits für die TAM kennengelernt haben
- ▶ Zur einfacheren Code-Generierung unterstützt das ASM-Framework das Einfügen von Sprungmarken (Labels)

Bedingte Sprünge

Die JVM verfügt über zwei Arten von bedingten Sprüngen, beide ausschließlich definiert für **int**-Werte.

Die erste Art von bedingten Sprüngen entfernt jeweils nur den obersten Wert vom Stack und betrachtet diesen für die Sprungbedingung:

Anweisung	Beschreibung
ifeq	Springe wenn 0
ifnull	Springe wenn null
iflt	Springe wenn < 0
ifle	Springe wenn ≤ 0
ifne	Springe wenn $\neq 0$
ifnonnull	Springe wenn nicht null
ifgt	Springe wenn > 0
ifge	Springe wenn ≥ 0

Die zweite Art von bedingten Sprüngen vergleicht die beiden obersten Werte auf dem Stack für die Sprungbedingung

Anweisung	Beschreibung
if_icmpeq	Springe wenn $a == b$
if_icmpne	Springe wenn $a \neq b$
if_cmplt	Springe wenn $a < b$
if_cmple	Springe wenn $a \leq b$
if_cmpgt	Springe wenn $a > b$
if_cmpge	Springe wenn $a \geq b$

Die JVM verfügt abgesehen von den bedingten Sprüngen über keine weiteren Vergleichsoperationen für **int**-Werte. Die Berechnung eines Vergleichs wie im folgenden Code muss daher über einen „Umweg“ auf die JVM abgebildet werden.

```
var int a;  
var int b;  
...  
val bool x = a < b;
```

```
0: iload_1  
1: iload_2  
2: if_icmplt      9  
5: iconst_0  
6: goto          10  
9: iconst_1  
10: istore_3
```



Für die Datentypen **long**, **float** und **double** verfügt die JVM über spezielle Vergleichsoperationen (z.B. **lcmp**, **fcmpl**, **dcmpl**), die zwei Werte vergleichen und als Ergebnis einen der Werte **0**, **-1** oder **1** ablegen. Die Berechnung eines Vergleichs wie im folgenden Code muss auch hier über einen „Umweg“ auf die JVM abgebildet werden.

```
var float c;  
var float d;  
...  
val bool z = c >= d;
```

12: fload_1	
13: fload_2	
15: fcmpl	
16: ifge	23
19: iconst_0	
20: goto	24
23: iconst_1	
24: istore	6

- ▶ Wie die Auflösung von Methoden-Referenzen funktioniert, haben Sie bereits im Kapitel zum Constant Pool kennengelernt
- ▶ Die JVM verfügt über 4 Spezialinstruktionen für den Aufruf von Methoden
 - ▶ **invokevirtual**
 - ▶ **invokespecial**
 - ▶ **invokeinterface**
 - ▶ **invokestatic**
- ▶ Hier Fokus auf **invokestatic**, da MAVL-Methoden auf statische Funktionen abgebildet werden
- ▶ JVM-Methoden sind nicht zu verwechseln mit *Subroutines*, die mit **jsr** und **ret** aufgerufen/verlassen werden. Diese werden v.a. für Exception Handling benutzt (hier nicht behandelt)

Das JVM-Routinenprotokoll ist dem Routinenprotokoll der TAM, das Sie bereits kennengelernt haben, sehr ähnlich.

- ▶ Die Parameter werden in der richtigen Reihenfolge auf dem Stack abgelegt
- ▶ Der Aufruf konsumiert die Parameter vom Stack
- ▶ Im Anschluss an den Aufruf liegt ggf. der Return-Wert als oberstes Element auf dem Stack
 - ▶ Für den Return steht die Operationen **<p>return** mit den möglichen Präfixen **i, l, f, d, a** und dem leeren Präfix für **void**-Returns zur Verfügung

Einziger größerer Unterschied zwischen JVM und TAM ist, dass Parameter in der JVM automatisch in den Local Variables verfügbar gemacht werden

```
function void bar(int x,  
    float y, vector<int>[3]  
    z)
```

```
function void foo(){  
    var int a; var float c;  
    var vector<int>[3] v;  
    bar(a, c, v);  
}
```

Constant pool:

#1 = Utf8	A
#2 = Class	#1
#11 = Utf8	bar
#12 = Utf8	(IF[IV)
#13 = NameAndType	#11:#12
#14 = Methodref	#2.#13

```
public static void foo(int);
```

```
...
```

```
0: iconst_3
```

```
1: newarray      int
```

```
3: astore_3
```

```
4: iload_1
```

```
5: fload_2
```

```
6: aload_3
```

```
7: invokestatic #14
```

```
10: return
```

```
public static void bar(int,  
float, int[]);
```

```
flags: ACC_PUBLIC, ACC_STATIC
```



Zusammenfassung



- ▶ Die Java Virtual Machine ist das Herzstück der Java Plattform
- ▶ Immer wenn ein Programm in einer der JVM-Sprachen ausgeführt werden soll, muss eine JVM vorhanden sein
- ▶ Die Verwendung einer virtuellen Maschine bietet Vorteile gegenüber rein interpretierten oder maschinen-kompilierten Sprachen, z.B. Portabilität und Effizienz
- ▶ Zentrale Repräsentation von Programmen ist der JVM Bytecode

- ▶ Bei der Generierung von Bytecode kann ein Framework wie das ASM-Framework extrem hilfreich sein
- ▶ Die drei zentralen Datenstrukturen, auf denen die Operationen arbeiten, sind
 - ▶ Local Variables
 - ▶ Heap
 - ▶ Operandenstack
- ▶ Das Vorgehen bei Berechnungen, Kontrollfluss und Methodenaufrufen ähnelt in sehr vielen Aspekten der TAM, da es sich bei der JVM ebenfalls um eine Stack-Maschine handelt