Karlsruher Institut für Technologie

Praktikum Compilerbau
Sitzung 9 – Java Bytecode

Prof. Dr.-Ing. Gregor Snelting
Matthias Braun und Sebastian Buchwald

IPD Snelting, Lehrstuhl für Programmierparadigmen

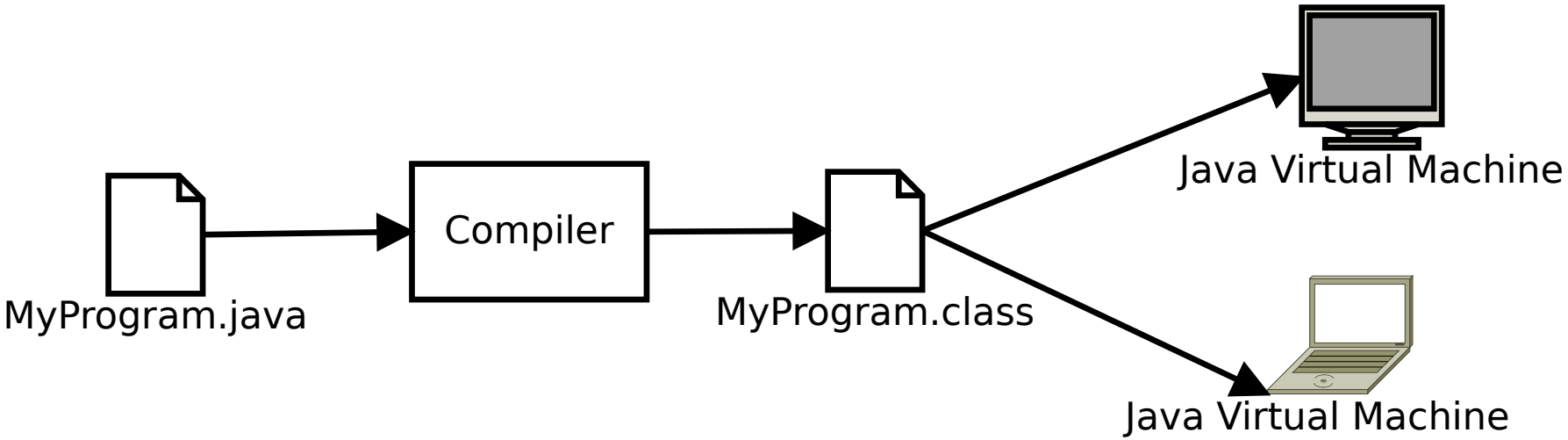
1. Letzte Woche
2. Java Bytecode
3. Jasmin Bytecode Assembler
4. Sonstiges

Letzte Woche

- Was waren die Probleme?
- Hat soweit alles geklappt?

1. Letzte Woche
2. Java Bytecode
3. Jasmin Bytecode Assembler
4. Sonstiges

Java Technologie



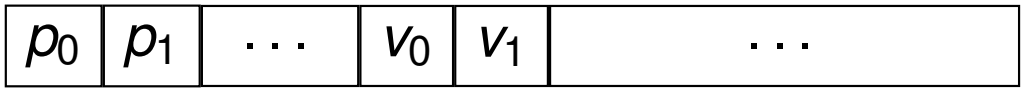
- Portable Zwischensprache: Bytecode
- Als virtuelle Maschine spezifiziert
- Umfangreiche Bibliothek
- Laufzeitsystem
- *The Java Virtual Machine Specification*
  `http://java.sun.com/docs/books/jvms/`

Virtuelle Maschine - Laufzeitsystem

- **Heap**: Speicher für Objektinstanzen. Getypt, automatische Speicherbereinigung (Garbage Collection), gemeinsamer Speicher für alle Threads.
- **Method Area**: Code für Methoden, nur lesbar.
- **Runtime Constant Pool**: Konstante Daten (Literale, Typinformationen, . . .)
- **Threads**: Je Thread:
  - **Program Counter**
  - **JVM Stack**: Activation Records (Stackframes)
  - **Native Method Stack**: Für Laufzeitsystem (meist in C/C++ geschrieben)

Activation Records (Stackframe)

- Rücksprungadresse
- dynamischer Vorgänger
- lokale Variablen:
  - Methodenparameter (Impliziter `this` Parameter an Position 0) p_0, $p_1, \ldots$
  - Lokale Variablen v_0, $v_1, \ldots$

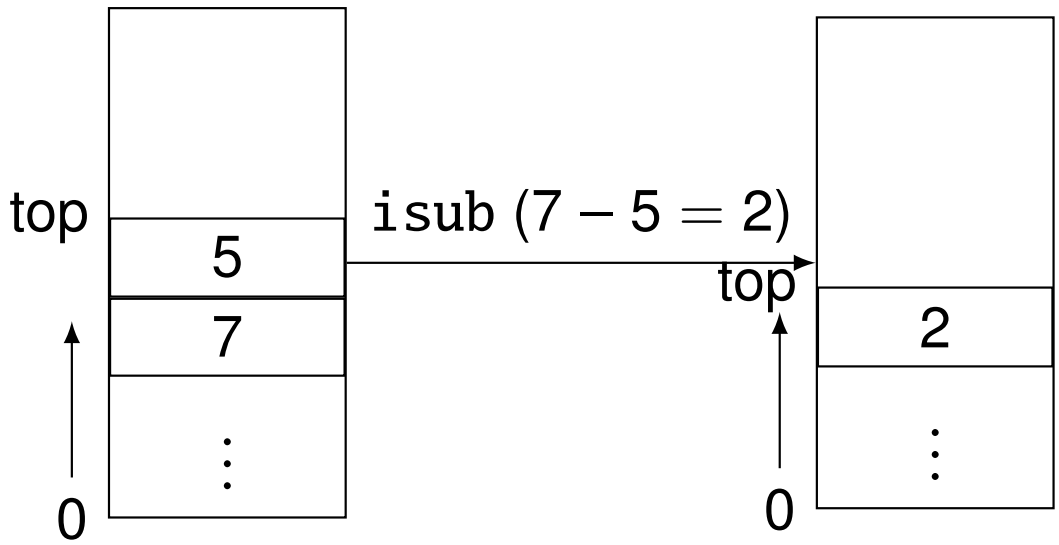


- Operandenstack

Bytecode, Operandenstack

- Stackbasierter Bytecode: Operanden und Rückgabewerte liegen auf Operandenstack.
- Kürzere Befehlscodierung da Operanden und Ziele nicht explizit.
- Maximale Stackgröße pro Methode im `.class`-File angegeben.

Beispiel:



Instruktionen

- Typen bekannt aus Java
- Instruktionen explizit typisiert: `iadd` (`int`), `fadd` (`float`)
- Instruktionsklassen:
  - Lesen/Schreiben von lokalen Variablen (`?load`, `?store <x>`, `...`)
  - Lesen/Schreiben von Feldern (`getfield`, `putfield`, `...`)
  - Sprungbefehle (`ifeq`, `ifnull`, `tableswitch`, `...`)
  - Methodenaufrufe (`invokevirtual`, `invokestatic`, `...`)
  - Objekterzeugung (`new`, `newarray`, ..)
  - Arithmetische Berechnungen (`?mul`, `?add`, `...`)

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
  int z = 6;
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	?	...

Stack:

Befehl:

```
// Lade Konstante 1
iconst_1
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
  int z = 6;
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	?	...

Stack: | ⊥ | | |

Befehl: | |

```
// Lade Konstante 1
iconst_1
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
  int z = 6;
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	?	...

Stack: | **1** | ⊥ | |

Befehl: `iconst_1`

```
// Lade Konstante 1
iconst_1
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
  int z = 6;
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	1	...

Stack: | ⊥ | | |

Befehl: istore_3

```
// Lade Konstante 1
iconst_1
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
  int z = 6;
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	1	...

Stack: | **5** | ⊥ | |

Befehl: `iload_2`

```
// Lade Konstante 1
iconst_1
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
  int z = 6;
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	1	...

Stack: | **6** | 5 | ⊥ |

Befehl: `iload_3`

```
// Lade Konstante 1
iconst_1
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
  int z = 6;
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	1	...

Stack: | **30** | ⊥ | |

Befehl: `imul`

```
// Lade Konstante 1
iconst_1
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
  int z = 6;
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	1	...

Stack: | **7** | 30 | ⊥ |

Befehl: `iload_1`

```
// Lade Konstante 1
iconst_1
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
  int z = 6;
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	1	...

Stack: | **37** | ⊥ | |

Befehl: `iadd`

```
// Lade Konstante 1
iconst_1
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
  int z = 6;
  x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	**37**	5	1	...

Stack: | ⊥ | | |

Befehl: istore_1

```
// Lade Konstante 1
iconst_1
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Fibonnaci-Berechnung

```
public int fib(int steps) {
  int last0 = 1;
  int last1 = 1;
  while (--steps > 0) {
    int t = last0 + last1;
    last1 = last0;
    last0 = t;
  }
  return last0;
}
```

```
iconst_1 // Konstante 1
istore_2 // in last0 (Var 2) schreiben
iconst_1 // Konstante 1
istore_3 // in last1 (Var 3) schreiben
iinc 1, −1 // steps (Var 1) dekrementieren
iload_1 // steps (Var 1) laden
ifle 24 // Falls <=0, springe
iload_2 // last0 (Var 2) laden
iload_3 // last1 (Var 3) laden
iadd // addiere (last0+last1)
istore 4 // in t (Var 4) schreiben
iload_2 // last0 (Var 2) laden
istore_3 // in last1 (Var 3) schreiben
iload 4 // t (Var 4 laden)
istore_2 // in last0 (Var 2) schreiben
goto 4 // springe zum Schleifenbeginn
iload_2 // last0 (Var 2) laden
ireturn // Verlassen mit Wert
```

Methodenaufrufe

1. Bezugsobjekt auf den Stack (falls nicht **static**)
2. Parameter auf den Stack
3. **invokevirtual** / **invokestatic** ausführen:
   Folgendes passiert vor / nach dem Aufruf automatisch:
   - 3.1 Array für Parameter und lokale Variablen anlegen (Größe ist angegeben)
   - 3.2 Returnadresse (Program Counter+1) und alten Framepointer sichern
   - 3.3 Neuen Framepointer setzen
   - 3.4 **this** Pointer und Parameter vom Stack ins Parameter Array kopieren
   - 3.5 Zu Methodenanfang springen und Code ausführen
   - 3.6 Returnwert auf den Stack
   - 3.7 Alten Framepointer setzen und zur Returnadresse springen
4. Returnwert vom Stack holen und weiterverarbeiten

Beispiel: Methodenaufruf

```
int bar() {
  return foo(42);
}

int foo(int i) {
  return i;
}
```

Konstantenpool

#2	Method	#3.#16
#3	class	#17
#11	Asciz	foo
#12	Asciz	(I)I
#16	NameAndType	#11:#12
#17	Asciz	Test

```
int bar();
   aload_0
   bipush 42
   invokevirtual #2
   ireturn

int foo(int);
   iload_1
   ireturn
```

Deskriptoren

Namen von Klassen, Feldern und Methoden müssen einem festgelegtem Schema entsprechen. (siehe JVMS 4.3)

- Klassennamen: `java.lang.Object` → `Ljava/lang/Object;`
- Typen: `int` → `I`, `void` → `V`, `boolean` → `Z`
- Methoden: `void foo(int, Object)` → `foo(ILjava/lang/Object;)V`
  Deskriptor: `(` *Parametertypen* `)` *Rückgabetyp*
  Identifiziert über "*Name* × *Deskriptor*"
- Felder: `boolean b` → `b:Z`
  Identifiziert nur über "*Name*"
- Konstruktoren: Name ist `<init>`, Static Initializer `<clinit>`

Objekt erzeugen & initialisieren

1. Objekt anlegen → Speicher reservieren
2. Objekt initialisieren → Konstruktor aufrufen

Hinweis: Jede Klasse braucht einen Konstruktor (Defaultkonstruktor)!

```
class Test {
  Test foo() {
    return new Test();
  }
}
```

#1	java/lang/Object.<init>()V
#2	Test
#3	Test.<init>()V

```
Test();
  aload_0
  invokespecial #1;
  return

Test foo();
  new #2;
  dup
  invokespecial #3;
  areturn
```

Beispiel: Array anlegen und darauf zugreifen

```
public void arr() {
    int[] array = new int[10];
    array[7] = 42;
}
```

```
bipush 10 // Konstante 10
newarray int // array anlegen vom Typ int
astore_1 // in variable array (var 1) speichern
aload_1 // variable array (var 1) laden
bipush 7 // Konstante 7
bipush 42 // Konstante 42
iastore // Wert (42) auf array index (7)
                    // von array("array") schreiben
return // Aus Funktion zurueckkehren
```

Beispiel: Auf Feld zugreifen

```
class field {
   public field field;
   public void setNull() {
      field = null;
   }
}
```

```
aload_0 // Parameter0 (this) auf Stack
aconst_null // null–Referenz auf den Stack
putfield field:Lfield; // Schreibe Wert (null)
   // auf Feld field:Lfield; von Objekt(this)
return // Aus Funktion zurueckkehren
```

1. Letzte Woche
2. Java Bytecode
3. Jasmin Bytecode Assembler
4. Sonstiges

Jasmin Assembler

Bytecode Assembler:

- Assemblersprache für Java Bytecode
- Leichter lesbar → debuggen
- Sprungmarken statt Bytecodepositionen
- Automatischer Aufbau des Konstantenpools
- `http://jasmin.sourceforge.net/`
- Aufruf: `java -jar jasmin.jar <Datei>`

Jasmin Klasse

- Header:

```
.class <Modifier> <ClassName>
.super <SuperClass>
```

- Methode:

```
.method <Modifier> <Name and Deskriptor>
<Code>
.end method
```

- Feld:

```
.field <Modifier> <FieldName> <Descriptor> [= <Value>]
```

Beispiel: Jasmin Code

```
.class Test
.super java/lang/Object

.method public <init>()V
   aload_0
   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

.method foo()LTest;
  .limit locals 1
  .limit stack 2
   new Test
   dup
   invokespecial Test/<init>()V
   areturn
.end method
```

Steuerfluss mit Sprungmarken

```
void foo(int z) {
  int i = 0;
  while (i < z) {
    i = i + 1;
  }
}
```

```
.method foo(I)V
  .limit locals 2
  .limit stack 3
  iconst_0
  istore_2
l1:
  iload_2
  iload_1
  if_icmpge l2
  iload_2
  iconst_1
  iadd
  istore_2
  goto l1
l2:
  return
.end method
```

Zum Schluss

- Anmerkungen?
- Probleme?
- Fragen?