# Kapitel 22

# Bytecode, JVM, Dynamische Compilierung

Am Beispiel des IBM Jalapeno-Compilers (besser als SUN!)[1]

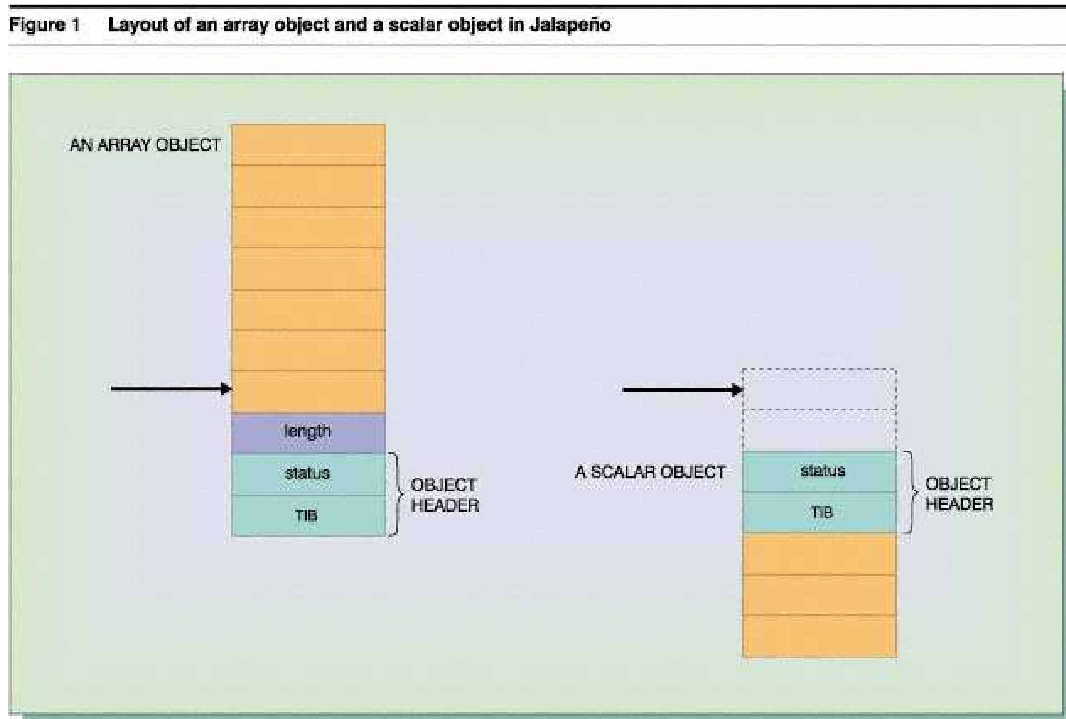
[1]Abbildungen aus: IBM Systems Journal, Vol 39 Nr 1

## 22.1 Laufzeitorganisation

Laufzeit-Datenbereiche der JVM:

- Program Counter: 1 PC-Register pro Thread. Enthält die Adresse der gerade ausgeführten Bytecode-Instruktion
- JVM Stack: jeder Thread hat eigenen Stack. Enthält die Activation Records der Methodenaufrufe
- Heap: enthält die Objekte. Alle Threads benutzen denselben Heap (shared memory)
- Method Area: enthält für jede Klasse Konstantentabelle sowie Bytecode für Methoden / Konstruktoren
- Operandenstack: zur Auswertung von (arithmetischen, logischen, ...) Ausdrücken

Objektlayout:




ersten 12 Byte: Länge (für Arrays; für Nicht-Arrays nicht belegt)

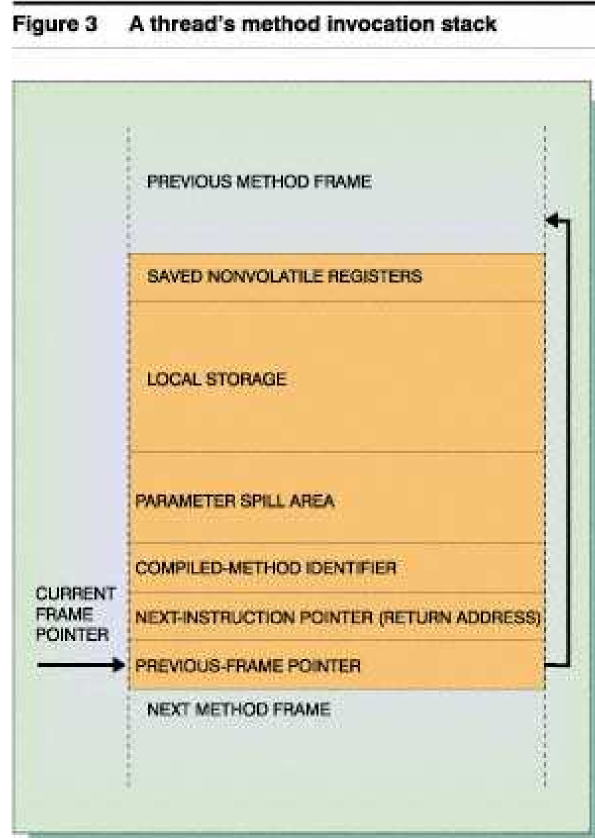
Status: Lock-Bits, Hash-Bits, Garbage-Collect-Bits

TIB: Type Information Block = vptr. JVM enthält in vtable zusätzlich Klassendeskriptor (vgl. Reflection-Interface)

Nullpointer-Zugriff erzeugt Hardware-Interrupt, da das length-Feld Offset -4 hat

Typische JVMs opfern Speicher, um Performance zu gewinnen!

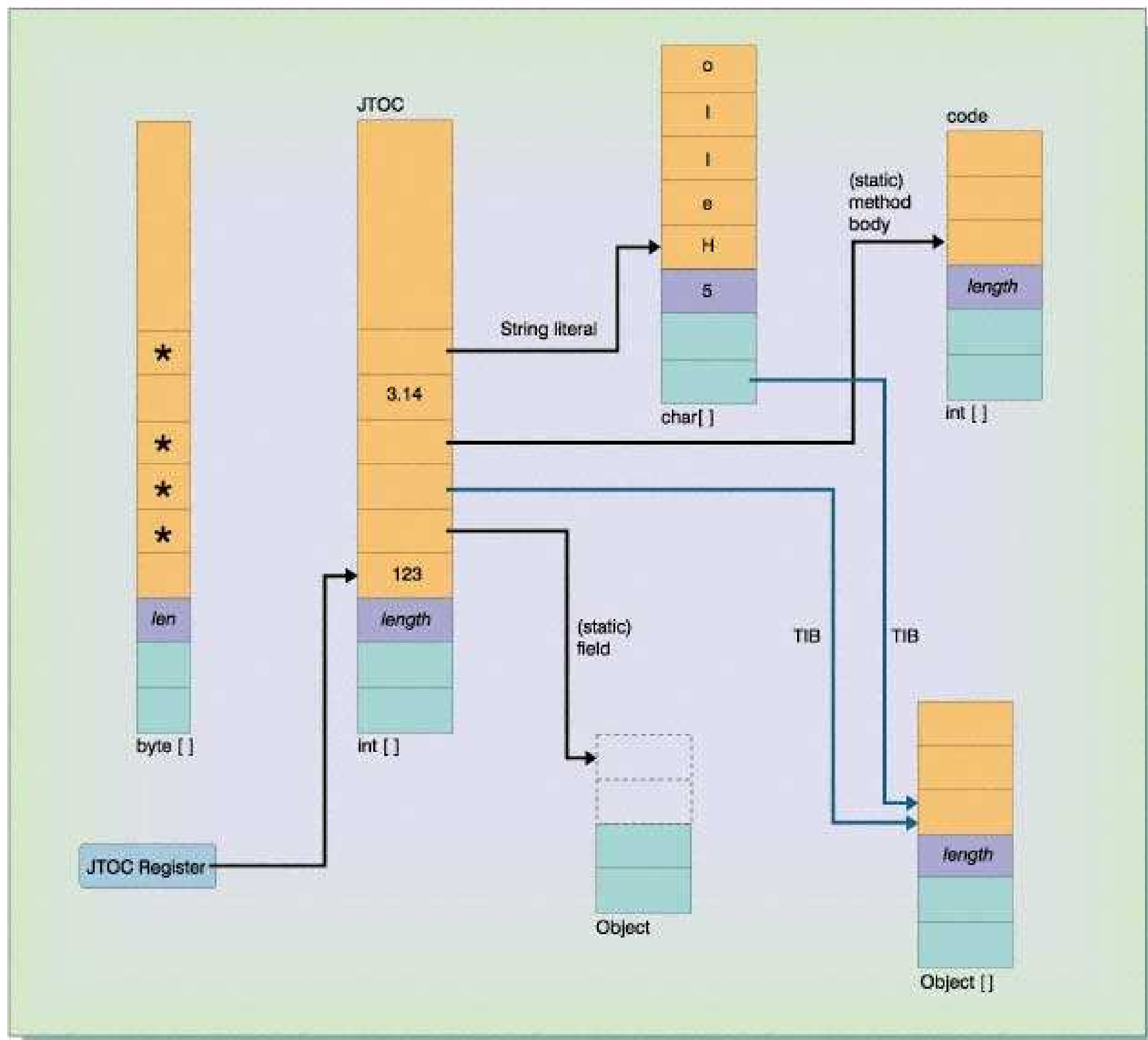
Aufbau des AR: analog C




JVM-Operandenstack wird in Hardwareregistern + Spillarea realisiert

Globale JTOC: Array mit (Verweisen auf) Konstanten + Klassendeskriptoren

**Figure 2** **The Jalepeño Table of Contents and other objects**



## 22.2 Bytecode

- typische abstrakte Stack-Maschine

  historisches Vorbild: Pascal P-Code

- Neben dem AR-Stack gibt es speziellen Operandenstack

  Arithmetische Codes: arbeiten auf Operanden-Stack.

- Unäre Operatoren (zB Typkonversion) wirken auf Topstack

  binäre verknüpfen die beiden obersten und
  schreiben Ergebnis wieder auf Stack.

  Ferner Lade/Speicherinstruktionen (push/pop)

- Alle Bytecodes kommen in verschiedenen typisierten Varianten:

| opcode | byte | short | int | long | float | double | char | reference |
|---|---|---|---|---|---|---|---|---|
| Tipush | bipush | sipush | | | | | | |
| Tconst | | | iconst | lconst | fconst | dconst | | aconst |
| Tload | | | iload | lload | fload | dload | | aload |
| Tstore | | | istore | lstore | fstore | dstore | | astore |
| Tinc | | | iinc | | | | | |
| Taload | baload | saload | iaload | laload | faload | daload | caload | aload |
| Tastore | bastore | sastore | iastore | lastore | fastore | dastore | castore | aastore |
| Tadd | | | iadd | ladd | fadd | dadd | | |
| Tsub | | | isub | lsub | fsub | dsub | | |
| Tmul | | | imul | lmul | fmul | dmul | | |
| Tdiv | | | idiv | ldiv | fdiv | ddiv | | |
| Trem | | | irem | lrem | frem | drem | | |
| Tneg | | | ineg | lneg | fneg | dneg | | |
| Tshl | | | ishl | lshl | | | | |
| Tshr | | | ishr | lshr | | | | |
| Tushr | | | iushr | lushr | | | | |
| Tand | | | iand | land | | | | |
| Tor | | | ior | lor | | | | |
| Txor | | | ixor | lxor | | | | |
| i2T | i2b | i2s | | i2l | i2f | i2d | | |
| l2T | | | l2i | | l2f | l2d | | |
| f2T | | | f2i | f2l | | f2d | | |
| d2T | | | d2i | d2l | d2f | | | |
| Tcmp | | | | lcmp | | | | |
| Tcmpl | | | | | fcmpl | dcmpl | | |
| Tcmpg | | | | | fcmpg | dcmpg | | |
| if_TcmpOP | | | if_icmpOP | | | | | if_acmpOP |
| Treturn | | | ireturn | lreturn | freturn | dreturn | | areturn |

*Beispiel 1*: Übersetzung von `x = x+y*z;`

int-Variablen `x`, `y`, `z` haben Offset 42, 43, 44 im AR

Bytecode:

```
iload 43
iload 44
imul
iload 42
iadd
istore 42
```

*Weitere Bytecodes*:

Objekterzeugung, Memberzugriff: `new, newarray, anewarray, multianewarray, getfiled, putfiled, getstatic, putstatic`

Arrayzugriff: `Taload, Tastore, arraylength`

Typetest: **`instanceof`**, `ckeckcast`

bedingte Sprünge: `ifeq, iflt, ifnull, if_icmpeq, if_acmpeq, ..., tableswitch, lookupswitch`

unbedingte Sprünge: `goto, goto_w`

Methodenaufruf: `invokevirtual, invokeinterface, invokespecial, invokestatic, Treturn`

Exceptions: `athrow, jsrm jsr_w, ret`

Synchronisation: `monitorenter, monitorexit`

Beispiel 2: Fibonnaci-Berechnung

```
static void calcSequence() {
    long fiboNum = 1;
    long a = 1;
    long b = 1;
    for (;;) {
        fiboNum = a + b;
        a = b;
        b = fiboNum;
    }
}
```

Bytecode:

```
 0 lconst_1 // Push long constant 1
 1 lstore_0 // Pop long into local vars 0 & 1:
            // long a = 1;
 2 lconst_1 // Push long constant 1
 3 lstore_2 // Pop long into local vars 2 & 3:
            // long b = 1;
 4 lconst_1 // Push long constant 1
 5 lstore 4 // Pop long into local vars 4 & 5:
            // long fiboNum = 1;
 7 lload_0  // Push long from local vars 0 & 1
 8 lload_2  // Push long from local vars 2 & 3
 9 ladd     // Pop two longs, add them, push result
10 lstore 4 // Pop long into local vars 4 & 5:
            // fiboNum = a + b;
12 lload_2  // Push long from local vars 2 & 3
13 lstore_0 // Pop long into local vars 0 & 1: a = b;
14 lload 4  // Push long from local vars 4 & 5
16 lstore_2 // Pop long into local vars 2 & 3:
            // b = fiboNum;
17 goto 7   // Jump back to offset 7: for (;;) {}
```

## 22.3 Methodenaufruf

1. Bezugsobjekt+aktuelle Parameter auf Operandenstack pushen
2. `invokevirtual`-Befehl ausführen:
3. neues AR anlegen (Länge statisch bekannt);
   Program Counter+1→ Return Address; Current-Frame-Ptr → Previous-Frame-ptr; Register retten
4. this-Ptr → Offset 0 im Local Storage; Parameter →Offset 1...; Operandenstack poppen
5. Einsprungadresse des Bytecode für Methodenrumpf aus vtable → Program Counter
6. Code für Rumpf ausführen
7. **`return`**-Befehl ausführen:
8. Return-Value auf Operandenstack pushen
9. Return-Adress → PC;
   Previous-Frame-Ptr → Frame-ptr; AR freigeben

## Beispiel 3: Methodenaufruf + Exceptions

```
class A {
   Object f() {
      return this;
   }
}
class B extends A {
   Object f() {
       throw new Error();
   }
}
class C {
   public static void main(String s[]) {
      A a;
      Object o;

      if(s[0].equals("A"))
        a = new A();
      else
        a = new B();

      try {
         o = a.f();
      } catch(Exception e) {
         o = e;
      }
   }
}
```

```
bytecode A.f(->java.lang.Object)
 0 aload 0
 1 areturn

bytecode A.<init>(->)
 0 aload 0
 1 invokespecial java.lang.Object.<init>(->)
 4 return

bytecode C.main(java.lang.String[]->)
EH [#1e,#23) #26 java.lang.Exception
 0 aload 0
 1 iconst 0
 2 aaload
 3 ldc string
 5 invokevirtual java.lang.String.equals(
                      java.lang.Object->.boolean)
 8 ifeq #16
 b new A
 e dup
 f invokespecial A.<init>(->)
12 astore 1
13 goto #1e
16 new B
19 dup
1a invokespecial B.<init>(->)
1d astore 1
1e aload 1
1f invokevirtual A.f(->java.lang.Object)
22 astore 2
23 goto #29
26 astore 3
27 aload 3
28 astore 2
29 return
```

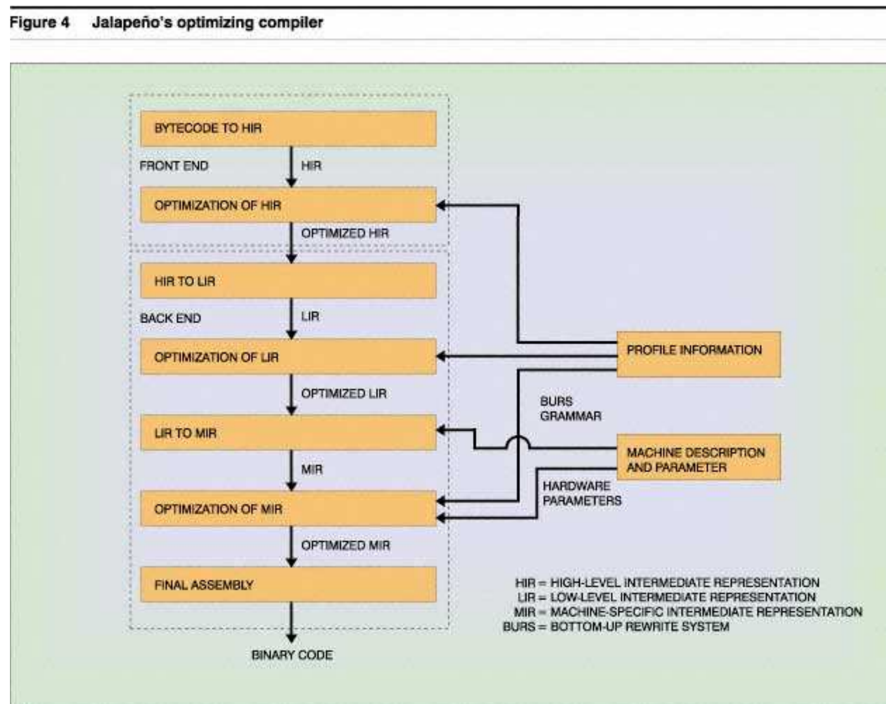
```
bytecode C.<init>(->)
 0 aload 0
 1 invokespecial java.lang.Object.<init>(->)
 4 return

bytecode B.<init>(->)
 0 aload 0
 1 invokespecial A.<init>(->)
 4 return

bytecode B.f(->java.lang.Object)
 0 new java.lang.Error
 3 dup
 4 invokespecial java.lang.Error.<init>(->)
 7 athrow
```

## 22.4 Just-in-Time Compiler

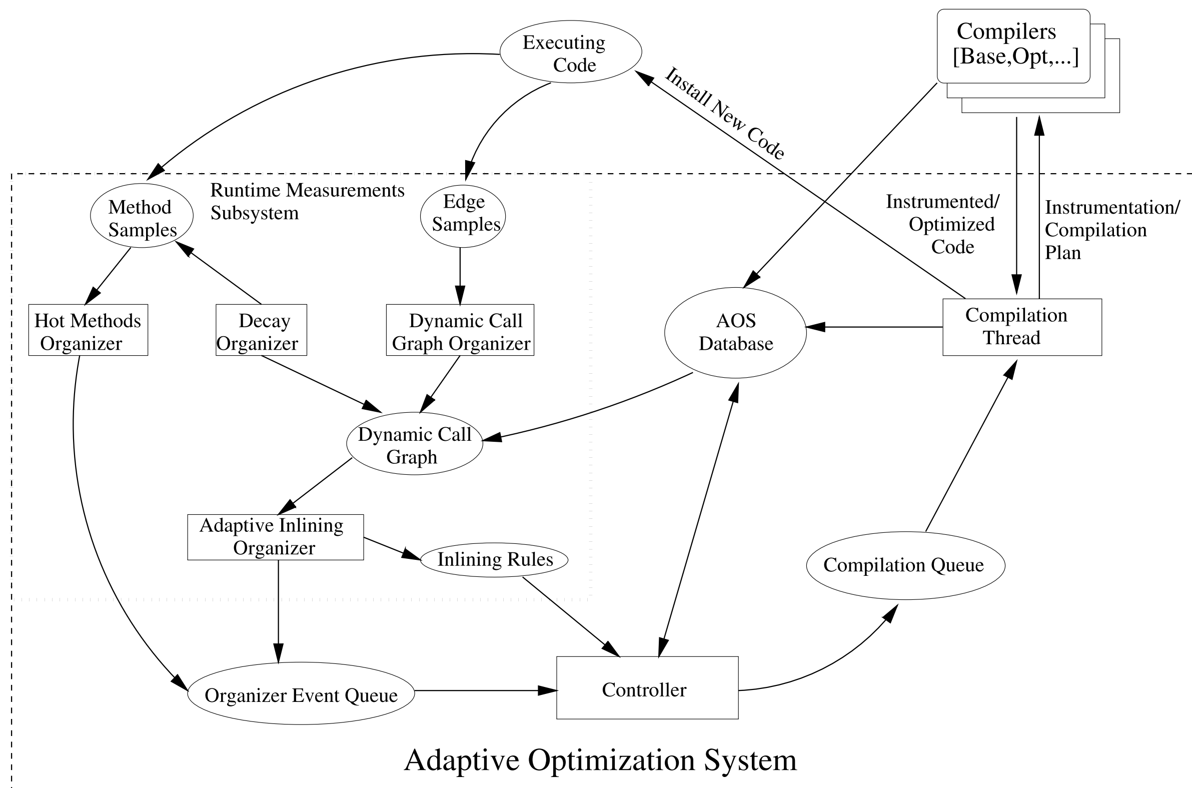
Compilervariationen:

- nur Bytecode-Generierung, JVM in C
- Just-in-time: Maschinencode für Methoden, sobald sie das erstemal aufgerufen werden
- Adaptive Compilation (Jalapeno): JVM größtenteils in Java, Generierung von Maschinencode und Optimierung aufgrund dynamischem Profiling

Grobaufbau des Jalapeno-Compilers: (Bytecode nach Maschinencode)



Struktur der Compileroptimierung
(Hotspot-Technologie)




Es wird sowohl Häufigkeit von Methodenausführungen als auch Zahl der Aufrufe $A.f() \to B.g()$ gemessen

Falls Schwellwert überschritten: Maschinencode; für Kanten im dynamischen Call Graph: Inlining

Schwellwerte sind heuristisch adaptiv; alte Werte „verfaulen"; Datenbank mit alten Messwerten

## 22.5 Bytecode Verifier

- prüft Bytecode auf Typsicherheit und Konsistenz

  Grund: Bytecode aus dem Internet (Applets) könnte „verseucht“ sein

- Prüfungen:

  1. Sprünge nur zu gültigen Bytecodeadressen (i.e. in der aktuellen Methode); Datenadressen nur im gültigen Bereich (i.e. ARs, Heap)

  2. Daten sind immer initialisiert; Pointer sind typsicher (Typcheck); Arithmetik nur mit konstanten Stackoffsets

  3. Zugriff auf „private“ Members wird strikt kontrolliert

- 1.+2. erfordern Datenflussanalyse und partielle Rekonstruktion von Stackinhalten (zB Typ von Stackelementen)

  3. wird dynamisch geprüft

  2. erlaubt effiziente Stackimplementierung in Registern $\leadsto$ Registerallokation

- es gibt Bytecode Verifier, die mit Maschinenbeweisern verifiziert sind