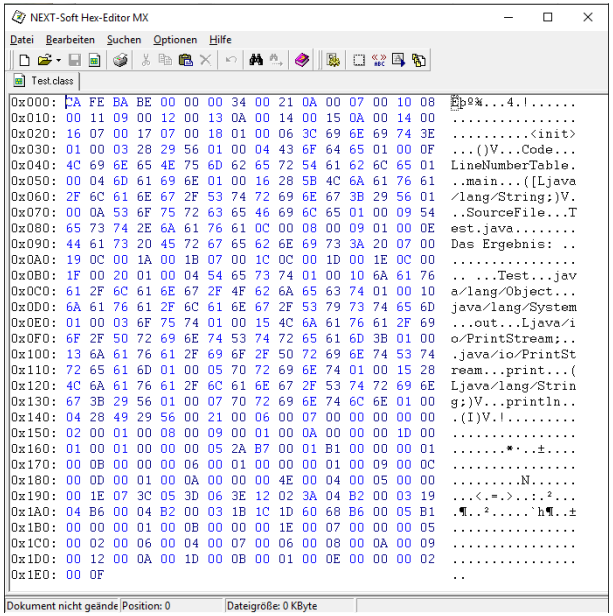
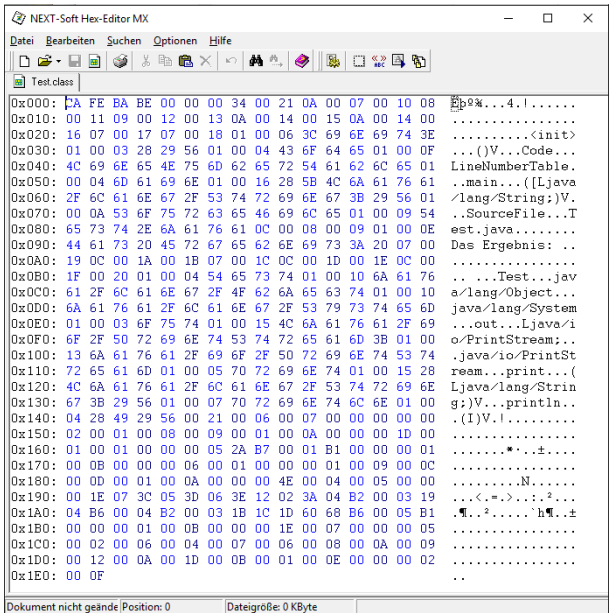
Exkurs: Java Bytecode

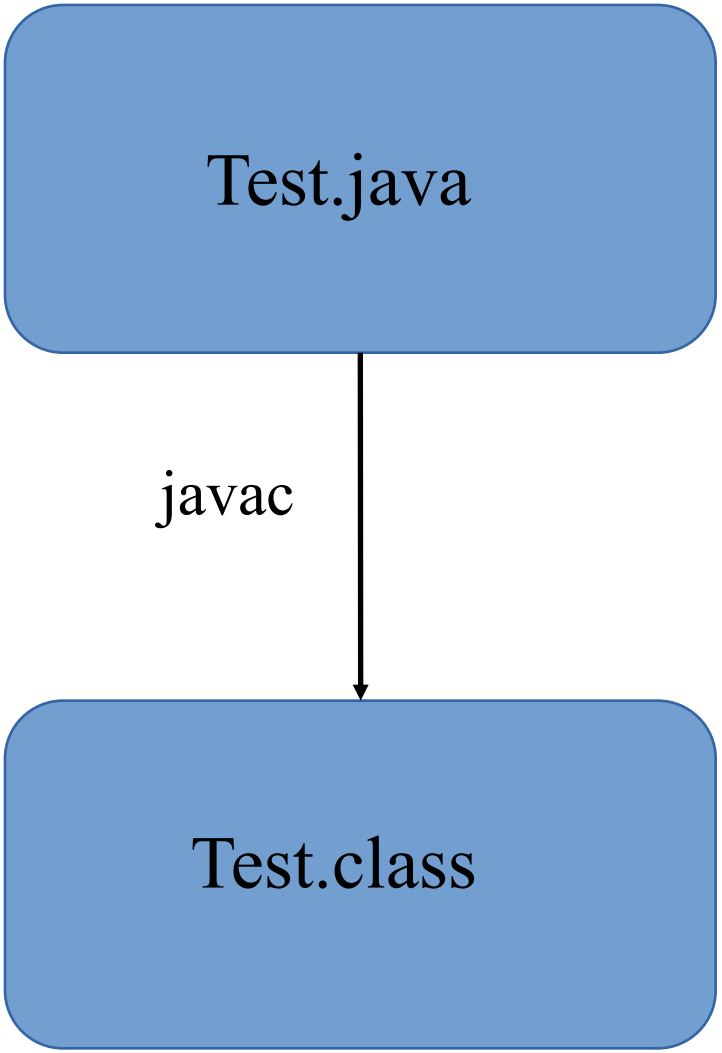



```
public class Test {
  public Test();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_4
       1: istore_1
       2: iconst_2
       3: istore_2
       4: iconst_3
       5: istore_3
       6: ldc           #2                  // String Das Ergebnis:
       8: astore        4
      10: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
      13: aload         4
      15: invokevirtual #4                  // Method java/io/PrintStream.print:(Ljava/lang/String;)V
      18: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
      21: iload_1
      22: iload_2
      23: iload_3
      24: iadd
      25: imul
      26: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      29: return
}
```

Java Compiler

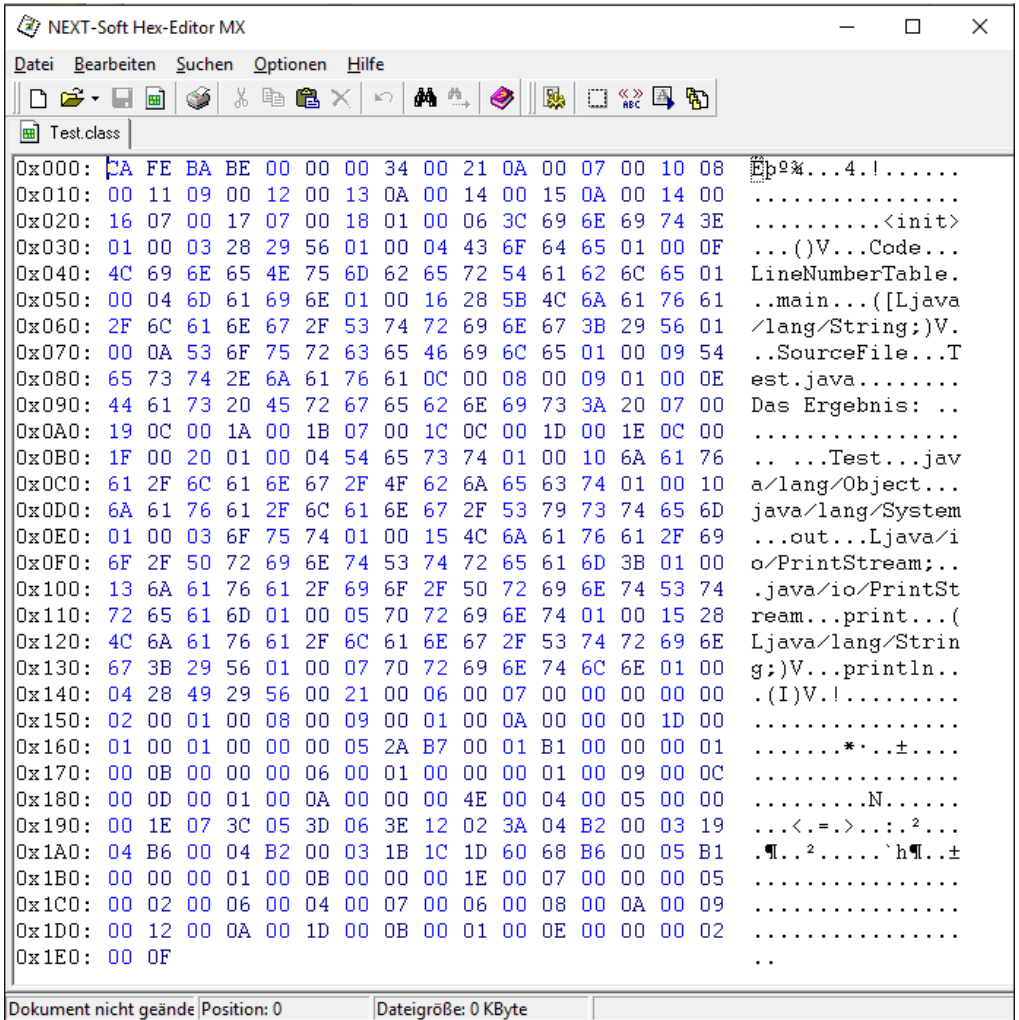
```java
public class Test
{
    public static void main(String[] args)
    {
        int x = 4;
        int y = 2;
        int z = 3;
        String s = "Das Ergebnis: ";
        System.out.print(s);
        System.out.println(x*(y+z));
    }
}
```





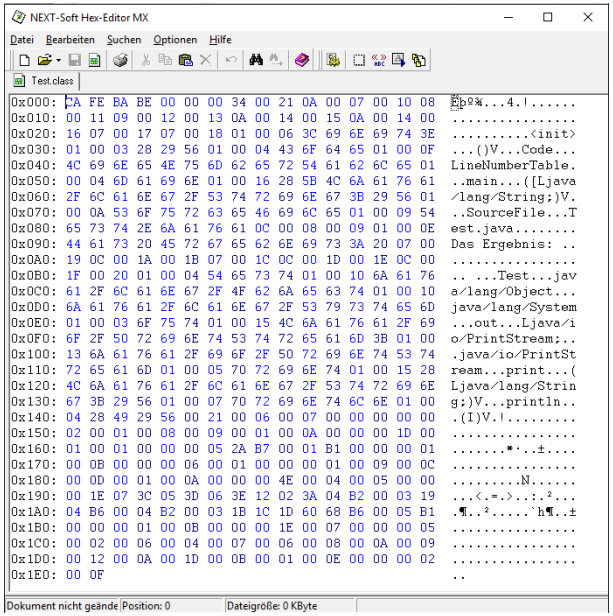
Java Class Dateien

Classdateien sind für die Programmierung ungeeignet. Für die Codegenerierung (javac) schwierig!



- Start immer mit CAFEBABE
- Eine Class Datei enthält mehr als nur Bytecode
- Signaturen werden gespeichert: z.B. **([Ljava/lang/String;)V** oder **<init>** für den Konstruktor
- Stringkonstanten werden gespeichert. z.B. "**Das Ergebnis:**"
- Bytecodes werden Hexadezimal kodiert: z.B. **1B 1C 1D 60 68**

Java Class Dateien



Test.class

javap -c Test > Test.txt

Test.txt

```
public class Test {
  public Test();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_4
       1: istore_1
       2: iconst_2
       3: istore_2
       4: iconst_3
       5: istore_3
       6: ldc           #2                  // String Das Ergebnis:
       8: astore        4
      10: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
      13: aload         4
      15: invokevirtual #4                  // Method java/io/PrintStream.print:(Ljava/lang/String;)V
      18: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
      21: iload_1
      22: iload_2
      23: iload_3
      24: iadd
      25: imul
      26: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      29: return
}
```

Java Class Dateien

Diese Ausgabe ist leserlicher, aber immer noch nicht ohne Weiteres verständlich.

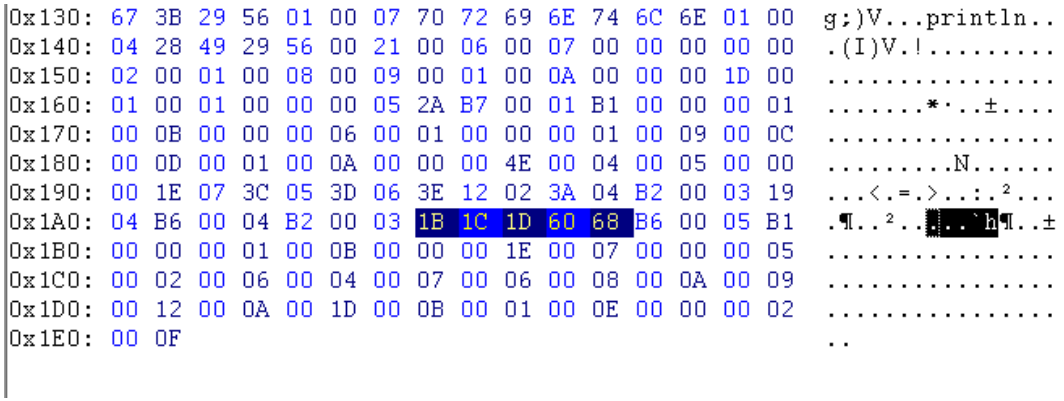
```
public class Test {
  public Test();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: iconst_4
       1: istore_1
       2: iconst_2
       3: istore_2
       4: iconst_3
       5: istore_3
       6: ldc           #2                  // String Das Ergebnis:
       8: astore        4
      10: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
      13: aload         4
      15: invokevirtual #4                  // Method java/io/PrintStream.print:(Ljava/lang/String;)V
      18: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
      21: iload_1
      22: iload_2
      23: iload_3
      24: iadd
      25: imul
      26: invokevirtual #5                  // Method java/io/PrintStream.println:(I)V
      29: return
}
```

Bytecode Kodierung

```
Code:
   0: iconst_4
   1: istore_1
   2: iconst_2
   3: istore_2
   4: iconst_3
   5: istore_3
   6: ldc           #2
   8: astore        4
  10: getstatic     #3
  13: aload         4
  15: invokevirtual #4
  18: getstatic     #3
  21: iload_1
  22: iload_2
  23: iload_3
  24: iadd
  25: imul
  26: invokevirtual #5
  29: return
```

Beispiele:

iconst_4:	0x07
ldc:	0x12
iload_1:	0x1b
iload_2:	0x1c
iload_3:	0x1d
iadd:	0x60
imul:	0x68



Mehr in der Spezifikation:
http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

Konstantenpool

```
Code:
   0: iconst_4
   1: istore_1
   2: iconst_2
   3: istore_2
   4: iconst_3
   5: istore_3
   6: ldc           #2
   8: astore        4
  10: getstatic     #3
  13: aload         4
  15: invokevirtual #4
  18: getstatic     #3
  21: iload_1
  22: iload_2
  23: iload_3
  24: iadd
  25: imul
  26: invokevirtual #5
  29: return
```

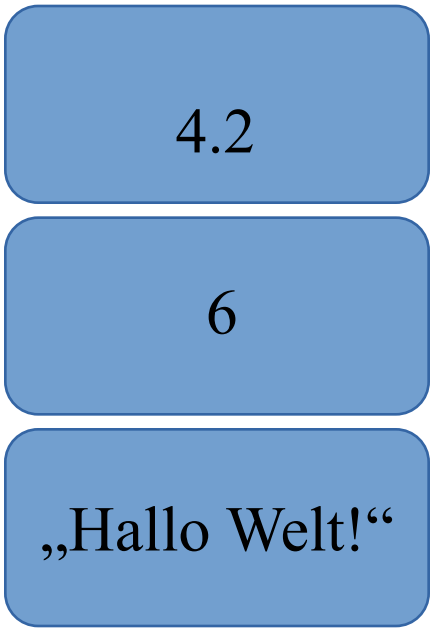
```
#1 Method java/lang/Object."<init>":()V

#2 String ''Das Ergebnis:''

#3 Field
java/lang/System.out:Ljava/io/PrintStream
;

#4 Method
java/io/PrintStream.print:(Ljava/lang/Str
ing;)V
```

Konstantenpool: Eine Tabelle mit allen Konstanten

```
#5 Method
java/io/PrintStream.println:(I)V
```

JVM Berechnungsmodel (Vereinfacht)

- Bytecodes manipulieren den Stack.
- Der Operandenstack enthält Werte unterschiedlichen Typs
- Instruktionen wie store (pop) und load (push) bewegen Daten vom/auf den Stack



Stack

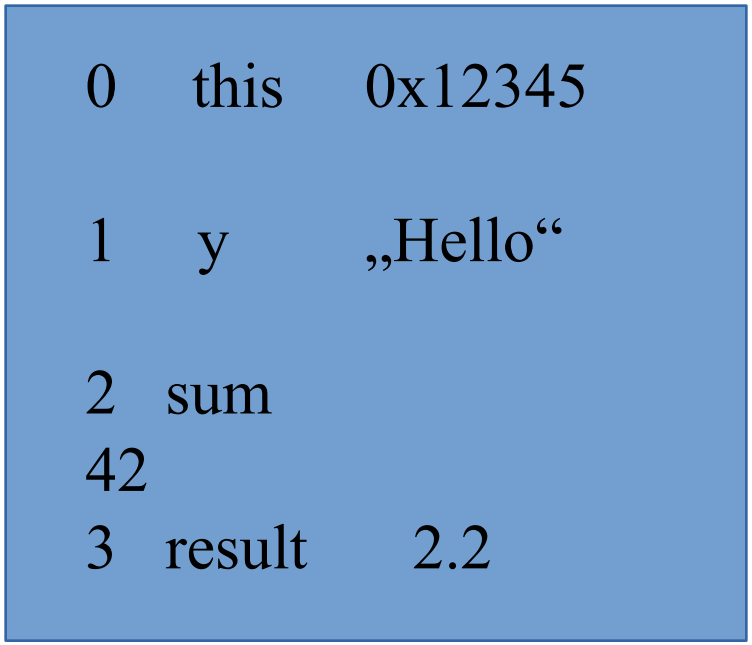
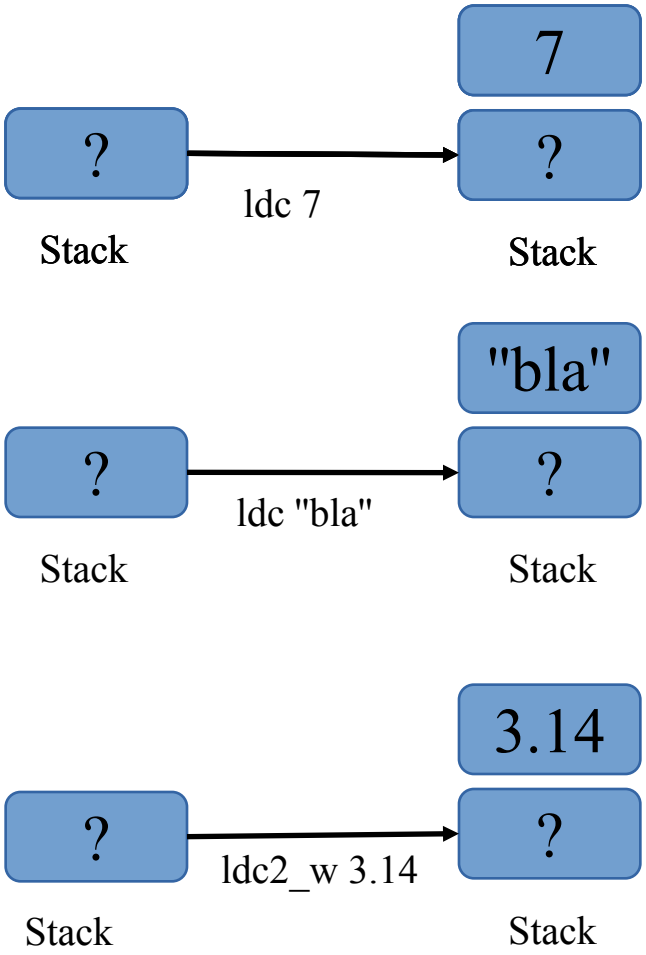


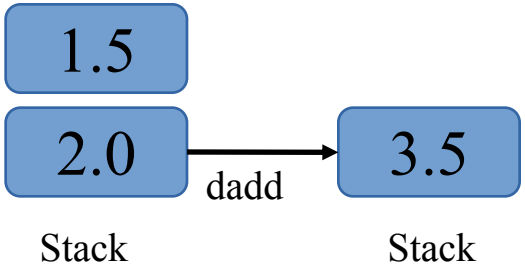
Tabelle der Variablen

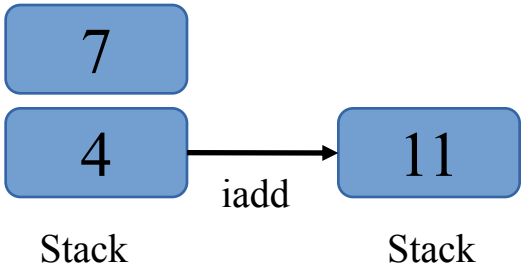
Bytecode: Konstanten

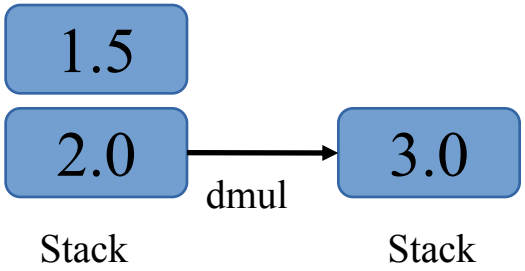


Unterschiedliche Bytecode je nach Konstantentyp

Bytecode: Arithmetik

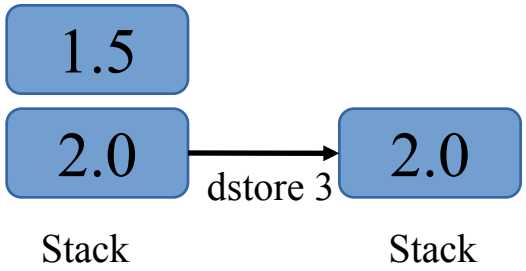


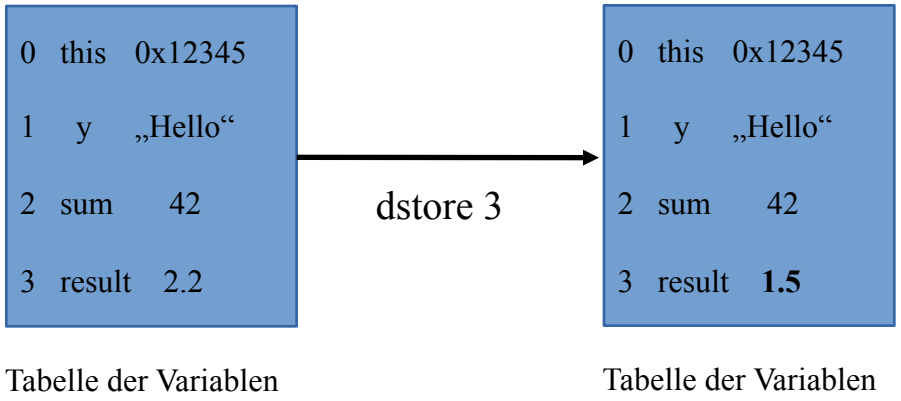


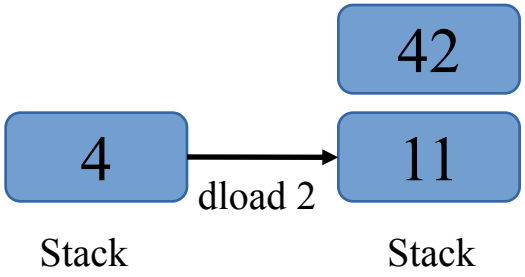


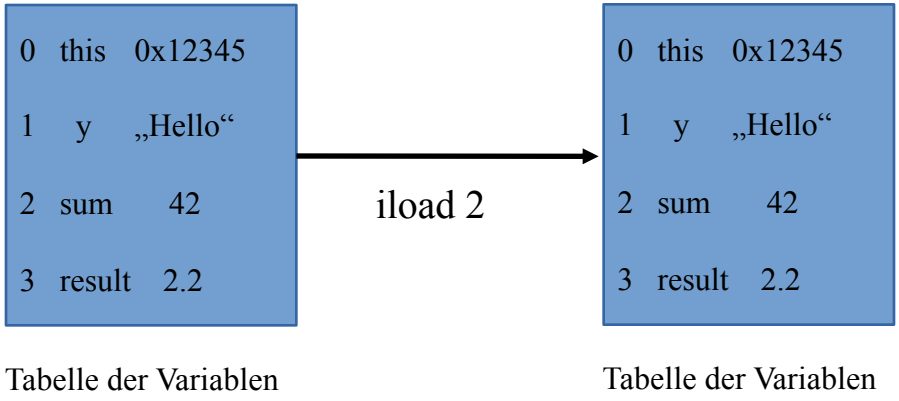
Bytecodes sind typisiert:
z.B. : **i**add für integer
dmul für double

Bytecode: Push und Pop



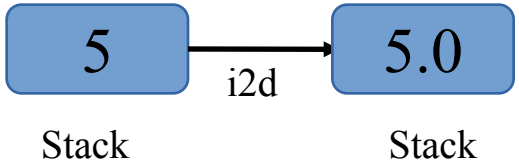


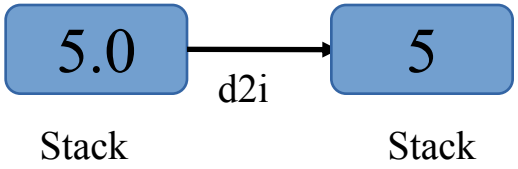




Integer mit iload/istore, Objekte mit aload/astore usw.

Bytecode: Casting



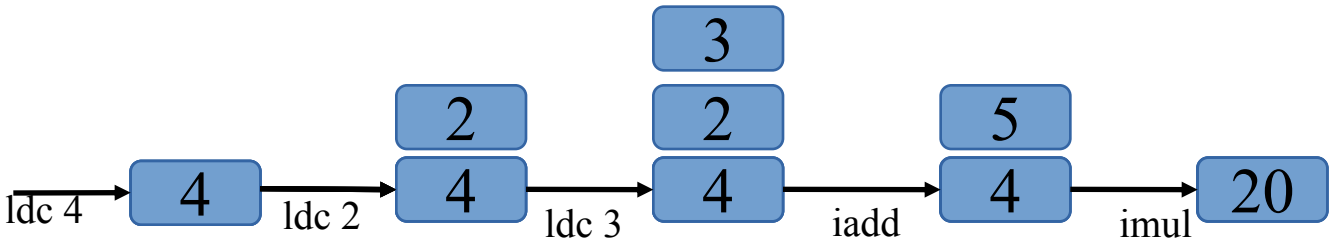


Bytecode: Kontrollfluss

Abarbeitung wie gewohnt von oben nach unten.

Beispiel:

0: ldc 4
1: ldc 2
2: ldc 3
3: iadd
4: imul

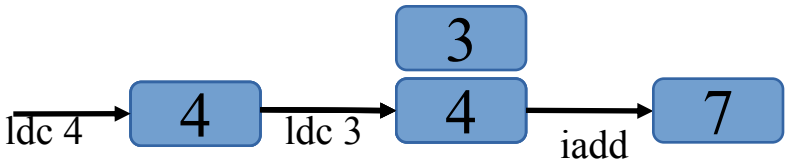


Bytecode: Kontrollfluss 2

Sprunganweisungen verändern die Befehlsabfolge

Beispiel:

0: ldc 4
1: **goto 3**
2: ldc 2
3: ldc 3
4: iadd
5: imul



Bedingte Sprünge mit: *`if_Xcmpne LABEL`*

Bytecode: Arrays 1

Ein Array erstellen:

Größe auf den Stack; newarray typ; astore x

Beispiel:
ldc 5
newarray int
astore 1

Bytecode: Arrays 2

Einem Array Werte zuweisen:

Referenz auf den Stack; Index auf den Stack; Wert auf den Stack; Xastore

Beispiel:
aload 1
ldc 0
ldc 42
iastore

Bytecode: Arrays 3

Werte aus einem Array lesen:

Referenz auf den Stack; Index auf den Stack; Xaload

Beispiel:
aload 1
ldc 0
iaload

Bytecode: (statische) Methodenaufrufe

Argumente auf den Stack; invokestatic NameSpace/Signatur

Beispiel:
ldc 4
ldc 5
invokestatic MethodStuff/f(II)I

Bytecode: JVM Spezifikation

iadd ***iadd***

Operation Add `int`

Format | *iadd* |

Forms *iadd* = 96 (0x60)

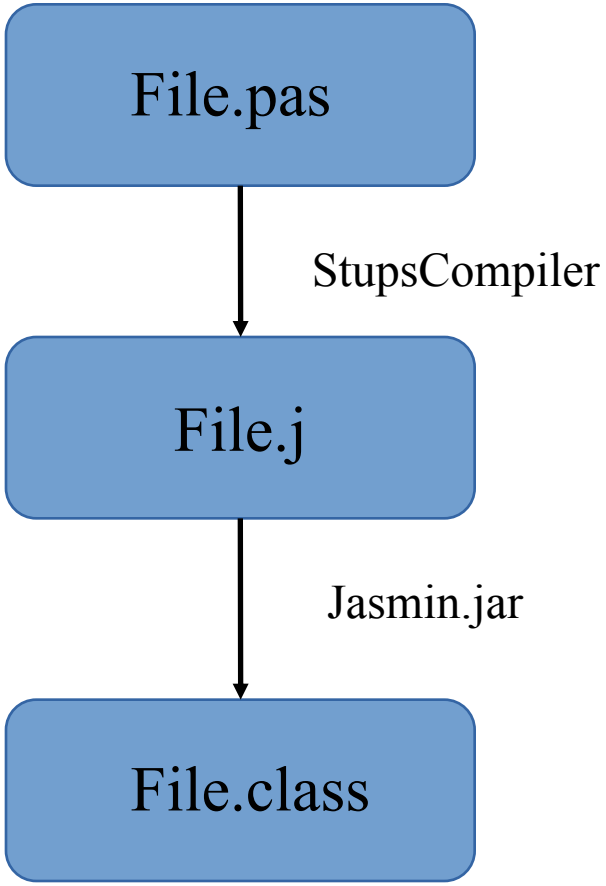
Operand Stack ..., *value1*, *value2* →
 ..., *result*

Description Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a run-time exception.

Jasmin Assembler



Java File

Aufbau Jasmindatei

.bytecode 49.0
.source Filename.pas
.class public FileName
.super java/lang/Object
.method public <init>()V
<bytecode>
.end method

.method public static main([Ljava/lang/String;)V
.limit stack X
.limit locals Y
<bytecode>
.end method

Aufbau Jasmindatei 2

Jasmin Derektiven sind Anweisungen an den Assembler.
Hier eine Auswahl:

.bytecode <number>
-Bytecode Version
.source <Name>
- (optional) Quelldateiname für Debuginfos.
.class public **<object-name>**
- Name der Klasse
.super **<object-name>**
- Name der Superklasse
.method **<signature>** und .end method
- Beginn und Ende einer Methode
.limit stack X
- Maximalgröße des Operandenstacks
.limit locals Y
- Größe der Tabelle lokaler Variablen

Mehr auf: http://jasmin.sourceforge.net/guide.html

Aufbau Jasmindatei 3

Namen und Siganturen:
- java.lang.Object wird zu java/lang/Object
- byte: B
- short:S
- int: I
- float: F
- double: D
- boolean: Z
- void: V
- Referenzen: L<Name>; z.B. Ljava/lang/String;
- Arrays: [Type z.B. [I

Beispiel:
([Ljava/lang/String;)V

Aufbau Jasmindatei 4

Globale (statische) Variablen:

- Deklaration: .field static name typ
- Lesen: getstatic namespace/name typ
- Schreiben: putstatic namespace/name typ

Beispiel (boolean b):

.field static b Z

putstatic Foo/b Z

getstatic Foo/b Z

Jasmin Beispiele

```
.bytecode 49.0
.source HelloWorld.pas
.class public HelloWorld
.super java/lang/Object
.method public <init>()V
    .limit stack 1
    .limit locals 1
    .line 1
        aload_0
        invokespecial java/lang/Object/<init>()V
        return
.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 10
    .limit locals 2

    ; s = "HelloWorld"
        ldc "HelloWorld"
        astore 1
        getstatic java/lang/System/out Ljava/io/PrintStream;

    ; System.out.println(s);
        aload 1
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
        return
.end method
```

Jasmin Beispiele

```
.bytecode 49.0
.source DoubleStuff.pas
.class public DoubleStuff
.super java/lang/Object
.field public static d D
.method public <init>()V
    .limit stack 1
    .limit locals 1
    .line 1
        aload_0
        invokespecial java/lang/Object/<init>()V
        return
.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 10
    .limit locals 3
        ldc2_w 3.141592654
        putstatic DoubleStuff/d D
        getstatic java/lang/System/out Ljava/io/PrintStream;
        getstatic DoubleStuff/d D
        invokevirtual java/io/PrintStream/println(D)V

        getstatic java/lang/System/out Ljava/io/PrintStream;
        ldc2_w 1.1
        ldc2_w 2.0
        dadd
        invokevirtual java/io/PrintStream/println(D)V


        getstatic java/lang/System/out Ljava/io/PrintStream;
        ldc2_w 1.1
        ldc  2
        i2d
        dadd
        invokevirtual java/io/PrintStream/println(D)V
        return
.end method
```

Jasmin Beispiele

```
.bytecode 49.0
.source MethodStuff.pas
.class public MethodStuff
.super java/lang/Object
.field public static d D
.method public <init>()V
    .limit stack 1
    .limit locals 1
    .line 1
        aload_0
        invokespecial java/lang/Object/<init>()V
        return
.end method

.method public static f(II)I
    .limit stack 10
    .limit locals 3
    ldc 42
    istore 2
    iload 0
    iload 1
    iadd
    ireturn
.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 10
    .limit locals 3
        getstatic java/lang/System/out Ljava/io/PrintStream;
        ldc 4
        ldc 5
        invokestatic MethodStuff/f(II)I
        invokevirtual java/io/PrintStream/println(I)V
        return
.end method
```

Zusammenfassung

- Java VM:
  - Class Dateien, Stack, Bytecode, Konstanten, Variablen
- Typen von Bytecodes
  - Arithmetik, Kontrollfluss
- Jasmin Assembler
  - Direktiven, Bytecode

http://jasmin.sourceforge.net
https://docs.oracle.com/javase/specs/jvms/se7/html/