Compilerbau

Martin Plümicke
Andreas Stadelmeier

SS 2025

Beschreibung

In der Vorlesung werden anwendungsnahe Konzepte und Techniken zu Programmiersprachen und Compilerbau vermittelt. Konkret werden zunächst die Phasen des Compilerbaus an Hand eines Java-Compilers vorgestellt. Als Implementierungstechnik wird die funktionale Programmiersprache Haskell verwendet. Dazu werden die notwendigen Grundlagen der funktionalen Programmierung aufbauend auf den Kenntissen der Grundvorlesung vermittelt. Im 2. Teil der Lehrveranstaltungen werden die Studierenden in Gruppenarbeit einen Mini-Java-Compiler mit den gelernten Techniken implementieren.

Literatur

Bauer and Höllerer.
Übersetzung objektorientierter Programmiersprachen.
Springer-Verlag, 1998, (in german).

Alfred V. Aho, Ravi Lam, Monica S.and Sethi, and Jeffrey D. Ullman.
Compiler: Prinzipien, Techniken und Werkzeuge.
Pearson Studium Informatik. Pearson Education Deutschland, 2. edition, 2008.
(in german).

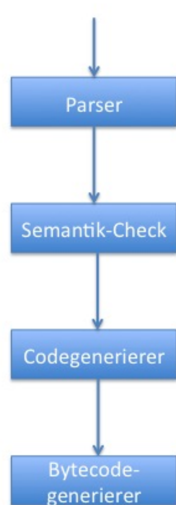
Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers Principles, Techniques and Tools.
Addison Wesley, 1986.

Reinhard Wilhelm and Dieter Maurer.
Übersetzerbau.
Springer-Verlag, 2. edition, 1992.
(in german).

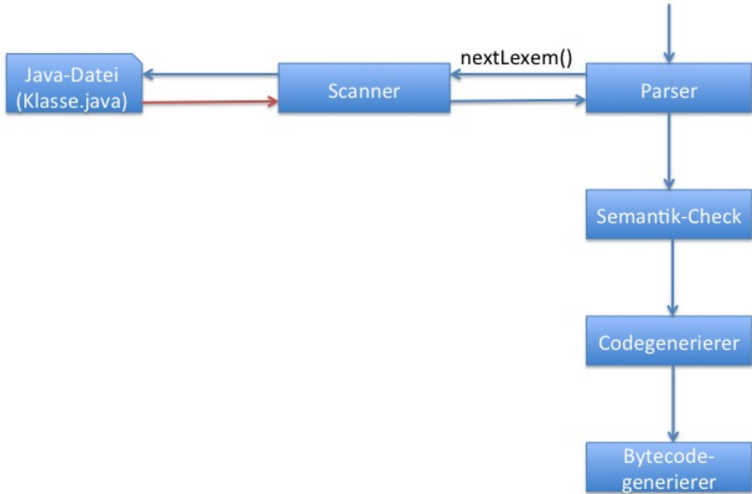
Literatur II

- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.
  The Java® Language Specification.
  The Java series. Addison-Wesley, Java SE 8 edition, 2014.
- Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley.
  The Java® Virtual Machine Specification.
  The Java series. Addison-Wesley, Java SE 8 edition, 2014.
- Bryan O'Sullivan, Donald Bruce Stewart, and John Goerzen.
  Real World Haskell.
  O'Reilly, 2009.
- Peter Thiemann.
  Grundlagen der funktionalen Programmierung.
  Teubner, 1994.

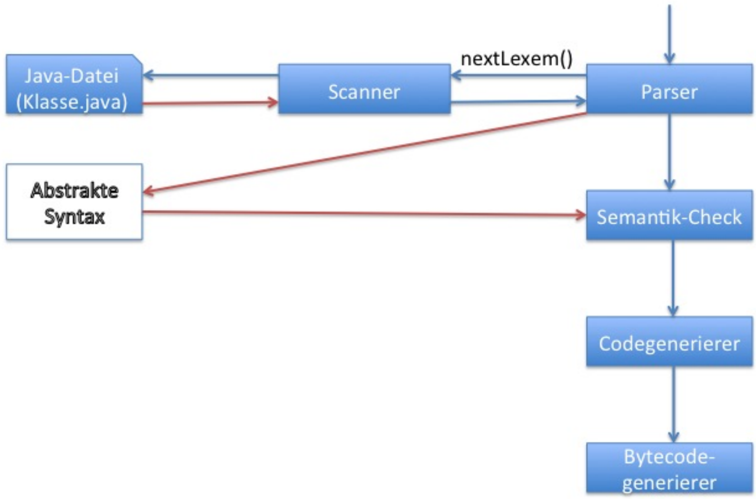
Compiler Überblick



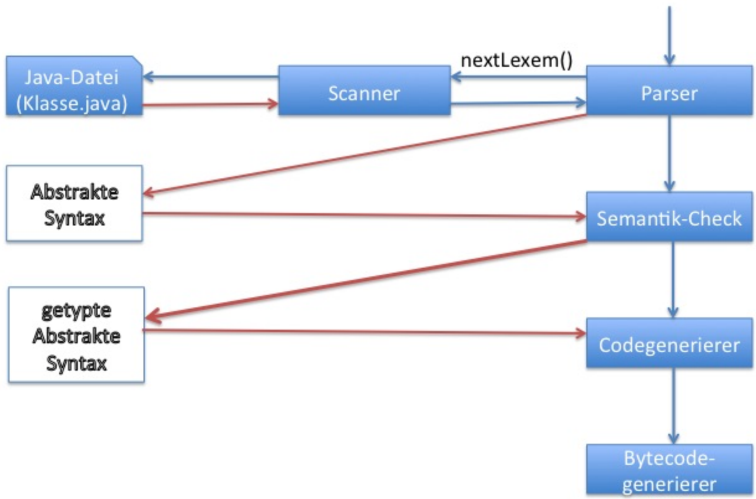
Compiler Überblick



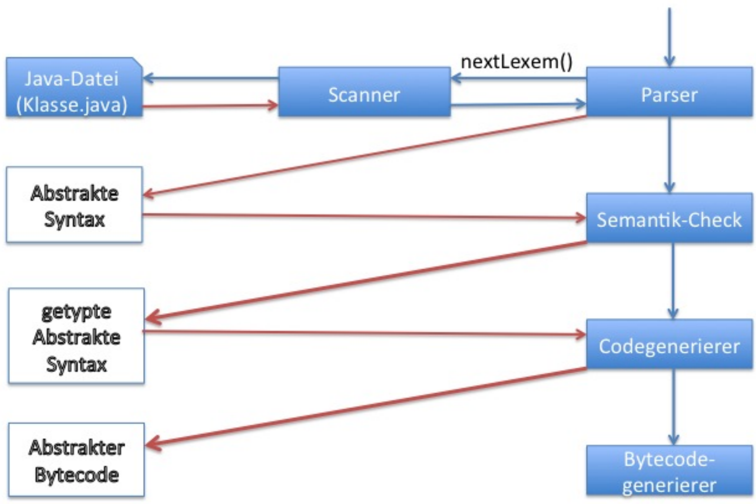
Compiler Überblick



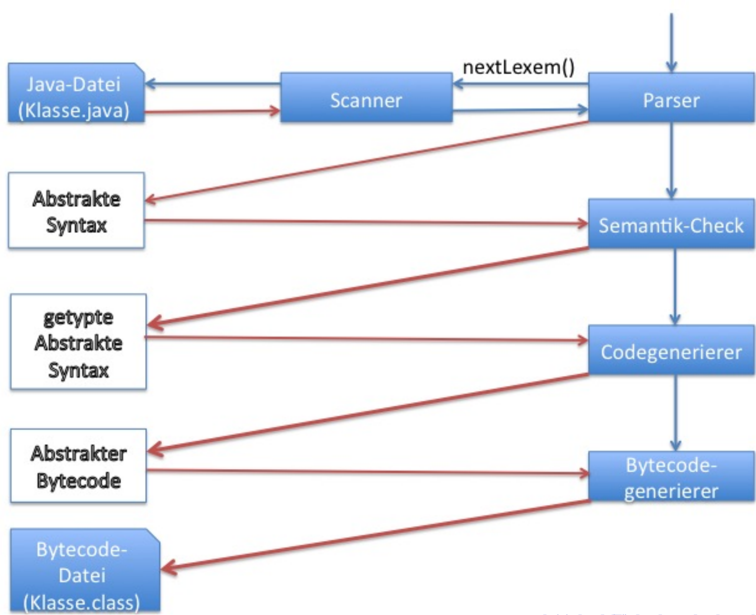
Compiler Überblick



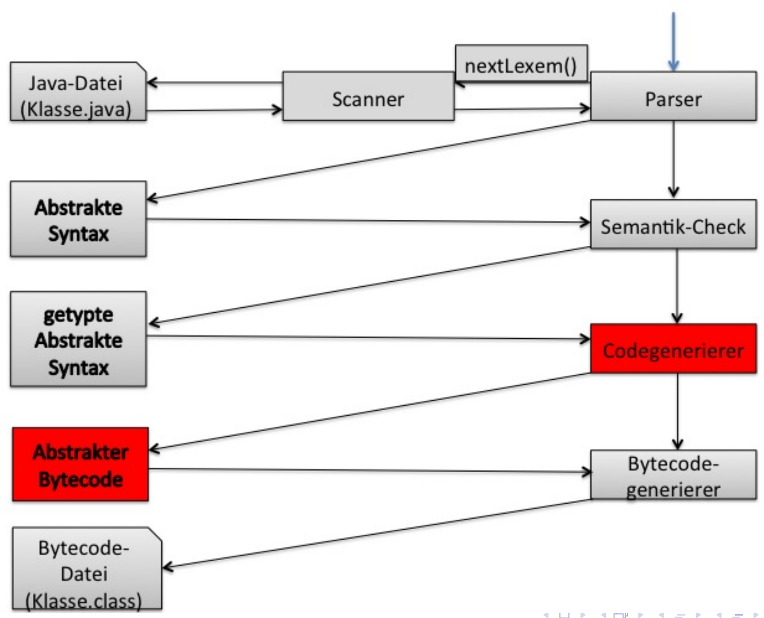
Compiler Überblick



Compiler Überblick



Codegenerierung



Codegenerierung

Typisierte abstrakte Syntax -> Abstrakten Bytecode

Abstrakter Bytecode (HackageDB)[4]

```
data ClassFile =
ClassFile { magic            :: Magic                -- CAFEBABE
          , minver           :: MinorVersion         -- Versionen
          , maxver           :: MajorVersion
          , count_cp         :: ConstantPool_Count   -- Anz. Eintr. Konst.p
          , array_cp         :: CP_Infos             -- Konstantenpool
          , acfg             :: AccessFlags          -- Berechtigungen
          , this             :: ThisClass            -- This-Klasse
          , super            :: SuperClass           -- Super-Klasse
          , count_interfaces :: Interfaces_Count     -- Anz. Interfaces
          , array_interfaces :: Interfaces           -- Interfaces
          , count_fields     :: Fields_Count         -- Anzahl Fields
          , array_fields     :: Field_Infos          -- Fields
          , count_methods    :: Methods_Count        -- Methoden
          , array_methods    :: Method_Infos         -- Methoden
          , count_attributes :: Attributes_Count     -- Anz. Attribute
          , array_attributes :: Attribute_Infos      -- Attribute
          }
```

```
type CP_Infos        = [CP_Info]
type Interfaces      = [Interface]
type Field_Infos     = [Field_Info]
type Method_Infos    = [Method_Info]
type Attribute_Infos = [Attribute_Info]

data Magic = Magic

data MinorVersion = MinorVersion {
                        numMinVer :: Int
                    }

data MajorVersion = MajorVersion {
                        numMaxVer :: Int
                    }
```

Konstantenpool–Einträge I

```
data CP_Info =
        Class_Info
              { tag_cp                :: Tag
              , index_cp              :: Index_Constant_Pool
              , desc                  :: String                }
      | FieldRef_Info
              { tag_cp                :: Tag
              , index_name_cp         :: Index_Constant_Pool
              , index_nameandtype_cp  :: Index_Constant_Pool
              , desc                  :: String                }
      | MethodRef_Info
              { tag_cp                :: Tag
              , index_name_cp         :: Index_Constant_Pool
              , index_nameandtype_cp  :: Index_Constant_Pool
              , desc                  :: String                }
      | InterfaceMethodRef_Info
              { tag_cp                :: Tag
              , index_name_cp         :: Index_Constant_Pool
              , index_nameandtype_cp  :: Index_Constant_Pool
              , desc                  :: String                }
```

Konstantenpool–Einträge II

```
| String_Info
        { tag_cp                :: Tag
        , index_cp              :: Index_Constant_Pool
        , desc                  :: String              }
| Integer_Info
        { tag_cp                :: Tag
        , numi_cp               :: Int
        , desc                  :: String              }
| Float_Info
        { tag_cp                :: Tag
        , numf_cp               :: Float
        , desc                  :: String              }
| Long_Info
        { tag_cp                :: Tag
        , numi_l1_cp            :: Int
        , numi_l2_cp            :: Int
        , desc                  :: String              }
| Double_Info
        { tag_cp                :: Tag
        , numi_d1_cp            :: Int
        , numi_d2_cp            :: Int
        , desc                  :: String              }
```

Konstantenpool–Einträge III

```
        | NameAndType_Info
                { tag_cp                :: Tag
                , index_name_cp         :: Index_Constant_Pool
                , index_descr_cp        :: Index_Constant_Pool
                , desc                  :: String              }
        | Utf8_Info
                { tag_cp                :: Tag
                , tam_cp                :: Int
                , cad_cp                :: String
                , desc                  :: String              }

data Tag = TagClass
         | TagFieldRef
         | TagMethodRef
         | TagInterfaceMethodRef
         | TagString
         | TagInteger
         | TagFloat
         | TagLong
         | TagDouble
         | TagNameAndType
         | TagUtf8
```

Field_Info und Method_Infos

```
data Field_Info = Field_Info
        { af_fi          :: AccessFlags
        , index_name_fi  :: Index_Constant_Pool     -- name_index
        , index_descr_fi :: Index_Constant_Pool     -- descriptor_index
        , tam_fi         :: Int                     -- count_attributte
        , array_attr_fi  :: Attribute_Infos
        }

data Method_Info = Method_Info
        { af_mi          :: AccessFlags
        , index_name_mi  :: Index_Constant_Pool     -- name_index
        , index_descr_mi :: Index_Constant_Pool     -- descriptor_index
        , tam_mi         :: Int                     -- attributes_count
        , array_attr_mi  :: Attribute_Infos
        }
```

Attribute_Info

```
data Attribute_Info =
    ...
  | AttributeCode
    { index_name_attr       :: Index_Constant_Pool   -- attribute_name_index
    , tam_len_attr          :: Int                   -- attribute_length
    , len_stack_attr        :: Int                   -- max_stack
    , len_local_attr        :: Int                   -- max_local
    , tam_code_attr         :: Int                   -- code_length
    , array_code_attr       :: ListaInt              -- code como array
                                                     -- de bytes
--, array_code_attr       :: [Code]                -- code array (altern.)
    , tam_ex_attr           :: Int                   -- exceptions_length
    , array_ex_attr         :: Tupla4Int             -- no usamos
    , tam_atrr_attr         :: Int                   -- attributes_count
    , array_attr_attr       :: Attribute_Infos
    }
  | ...
```

Beispiel: Byte–Code

```
class bct {

    Integer i;

}
```

```
> javac -g:none bct.java
```

```
magic = 0x CAFEBABE
minor_version = 0
major_version = 52
constant_pool_count = 12
constant_pool =
{
1| tag = CONSTANT_Methodref, class_index = 3, name_and_type_index = 9
2| tag = CONSTANT_Class, name_index = 10
3| tag = CONSTANT_Class, name_index = 11
4| tag = CONSTANT_Utf8, length = 1, bytes = i
5| tag = CONSTANT_Utf8, length = 19, bytes = Ljava/lang/Integer;
6| tag = CONSTANT_Utf8, length = 6, bytes = <init>
7| tag = CONSTANT_Utf8, length = 3, bytes = ()V
8| tag = CONSTANT_Utf8, length = 4, bytes = Code
9| tag = CONSTANT_NameAndType, name_index = 6, descriptor_index = 7
10| tag = CONSTANT_Utf8, length = 3, bytes = bct
11| tag = CONSTANT_Utf8, length = 16, bytes = java/lang/Object
}
access_flags = 32  // ACC_SUPER
this_class = #2  // bct
super_class = #3  // java/lang/Object
interfaces_count = 0
interfaces = {}
```

```
fields_count = 1
fields [0] =
{
access_flags = 0
name_index = #4  // i
descriptor_index = #5  // Ljava/lang/Integer;
attributes_count = 0
attributes = {}
}
methods_count = 1
methods [0] =
{
access_flags = 0
name_index = #6  // <init>
descriptor_index = #7  // ()V
```

```
attributes_count = 1
attributes [0] =
{
attribute_name_index = #8  // Code
attribute_length = 17
max_stack = 1, max_locals = 1
code_length = 5
code =
{
    0  aload_0
    1  invokespecial #1  // java/lang/Object.<init> ()V
    4  return
}
exception_table_length = 0
exception_table = {}
attributes_count = 0
attributes = {}
}
}
attributes_count = 0
attributes = {}
```

JVM – Zur Laufzeit

Frame: Für jeden Methodenaufruf wird ein Frame erzeugt:

- Array von lokalen Variablen
- Operanden-Stack
- Referenz zum zugehörigen Konstantenpool

JVM – Zur Laufzeit

Frame: Für jeden Methodenaufruf wird ein Frame erzeugt:

- Array von lokalen Variablen
- Operanden-Stack
- Referenz zum zugehörigen Konstantenpool

Stack: Auf dem Stack liegen die Frames, der aufgerufenen Methoden

JVM – Zur Laufzeit

Frame: Für jeden Methodenaufruf wird ein Frame erzeugt:

- Array von lokalen Variablen
- Operanden-Stack
- Referenz zum zugehörigen Konstantenpool

Stack: Auf dem Stack liegen die Frames, der aufgerufenen Methoden

Heap: Speicher für alle Objekte

JVM – Zur Laufzeit

Frame: Für jeden Methodenaufruf wird ein Frame erzeugt:

- Array von lokalen Variablen
- Operanden-Stack
- Referenz zum zugehörigen Konstantenpool

Stack: Auf dem Stack liegen die Frames, der aufgerufenen Methoden

Heap: Speicher für alle Objekte

Method Area: Speicher für alle Methoden

Code–Übersetzung
Standard–Konstruktor

```
class bct {
  bct() {
    super();
  }
}
```

führt zu

```
0   aload_0
1   invokespecial #1  // java/lang/Object.<init> ()V
4   return
```

aload_<n>, aload

Code: aload_<n>

Format: aload_<n>

Bechreibung: Lädt die Referenz der *n*-ten lokalen Variablen auf den Stack

Formen: aload_0 = 42 (0x2a)
aload_1 = 43 (0x2b)
aload_2 = 44 (0x2c)
aload_3 = 45 (0x2d)

aload_<n>, aload

Code: aload_<n>

Format: aload_<n>

Bechreibung: Lädt die Referenz der *n*-ten lokalen Variablen auf den Stack

Formen: aload_0 = 42 (0x2a)
aload_1 = 43 (0x2b)
aload_2 = 44 (0x2c)
aload_3 = 45 (0x2d)

Code: aload

Format: aload *index*

Bechreibung: Lädt die Variable *index*-te Variable auf den Stack

Formen: aload = 25 (0x19)

invokespecial

Code: invokespecial

Format: invokespecial indexbyte1 indexbyte2

Bechreibung: Ruft die Instanzmethode auf, die durch die Referenz *indexbyte1* $<< 8 \mid$ *indexbyte2* in den Konstantenpool bestimmt wird. (Ohne dynamische Bindung!!!)

- Wenn die Methode nicht vorhanden ist wird in Superklassen gesucht.
- Die Argumente der Methoden müssen ebenfalls auf den Stack liegen.
- Am Ende werden die Argumente und das Objekt vom Stack gelöscht und das Ergebnis drauf gelegt.

Formen: invokespecial = 183 (0xb7)

Opcode: return

Code: return

Format: return

Bechreibung: Gibt `void` von einer Methode zurück.
Gibt die Kontrolle an die aufgerufende Methode zurück.

Formen: return = 177 (0xb1)

Variablen beschreiben

```
class bct {
  int i;

  void m () {
     i = 1;
  }
}
```

führt zu

```
0  aload_0       //this
1  iconst_1
2  putfield #2  // bct.i I
5  return
```

iconst_<n>

Code: iconst_<i>

Format: iconst_<i>

Bechreibung: Push int Konstante auf den Stack

Formen: iconst_m1 = 2 (0x2)
iconst_0 = 3 (0x3)
iconst_1 = 4 (0x4)
iconst_2 = 5 (0x5)
iconst_3 = 6 (0x6)
iconst_4 = 7 (0x7)
iconst_5 = 8 (0x8)

bipush

Code: bipush

Format: bipush byte

Bechreibung: Push *byte*

Formen: bipush = 16 (0x10)

putfield

Code: putfield

Format: putfield indexbyte1 indexbyte2

Bechreibung: Ordnet das Oberste Element des Stacks dem Attribut *indexbyte1* $<< 8 \mid$ *indexbyte2* des Konstantenpools im Objekt des zweiobersten Elements des Stacks zu, löscht die beiden obersten Elemente des Stacks.

Formen: putfield = 181 (0xb5)

Variablen auslesen und beschreiben

```
class bct {
  int i;

  void m () {
     i = i + 1;
  }
}
```

führt zu

```
 0  aload_0   //this
 1  aload_0   //this
 2  getfield #2   // bct.i I
 5  iconst_1
 6  iadd
 7  putfield #2   // bct.i I
10  return
```

getfield

Code: getfield

Format: getfield indexbyte1 indexbyte2

Bechreibung: Liest das Attribut *indexbyte1* $<< 8 \mid$ *indexbyte2* im Konstantenpool des Obersten Element des Stacks aus, löscht das oberste Element und legt den gelesenen Wert auf den Stack.

Formen: getfield = 180 (0xb4)

iadd

Code: iadd

Format: iadd

Bechreibung: Addiert die beiden obersten Elemente des Stacks und nimmt sie vom Stack und legt das Ergebnis drauf.

Formen: iadd = 96 (0x60)

iadd

Code: iadd

Format: iadd

Bechreibung: Addiert die beiden obersten Elemente des Stacks und nimmt sie vom Stack und legt das Ergebnis drauf.

Formen: iadd = 96 (0x60)

Anlog: *isub*, *imul*, *idiv*, *iand*, *ior*, *ixor*, *ineg*, ...

Inkrement/Dekrement

```
class bct {

  void m () {
     int i = 0;
     i++;
     i--;
  }
}
```

führt zu

```
0  iconst_0
1  istore_1
2  iinc 1 1    //Variable 1 (+1)
5  iinc 1 255    //Variable 1 (-1)
8  return
```

istore_<n>

Code: istore_<n>

Format: istore_<n>

Bechreibung: Schreibt den Integerwert des obersten Elements des Operanden–Stacks in die *n*-te lokale Variable und löscht das oberste Element des Operanden–Stacks.

Formen: istore_0 = 59 (0x3b)
istore_1 = 60 (0x3c)
istore_2 = 61 (0x3d)
istore_3 = 62 (0x3e)

istore

Code: istore

Format: istore index

Bechreibung: Schreibt den Integerwert des obersten Elements des Operanden–Stacks in die *index*-te lokale Variable und löscht das oberste Element des Operanden–Stacks.

Formen: istore = 54 (0x36)

iinc

Code: iinc

Format: iinc index const

Bechreibung: Inkrement der lokalen Variable *index* durch vozeichenbehaftete 8-Bit *const*.

Formen: iinc = 132 (0x84)

New

```
class A { }

class bct {

  void m () {
     A aa = new A();
  }
}
```

führt zu

```
0  new #2  // A
3  dup
4  invokespecial #3  // A.<init> ()V
7  astore_1
8  return
```

Opcode: new

Code: new

Format: new indexbyte1 indexbyte2

Bechreibung: Erzeugt ein neues Objekt der Klasse *indexbyte1* $<< 8 \mid$ *indexbyte2* des Konstantenpool auf dem Heap und initialisiert die Attribute. Eine Referenz auf den Speicher im Heap wird auf den Stack gelegt.

Formen: new = 187 (0xbb)

dup

Code: dup

Format: dup

Bechreibung: Dubliziert das oberste stack Element

Formen: dup = 89 (0x59)

astore_<n>, astore

Code: astore_<n>

Format: astore_<n>

Bechreibung: Schreibt die oberste Referenz des Stacks in die *n*-te lokale Variable und löscht die Referenz vom Stack

Formen: astore_0 = 75 (0x4b)
astore_1 = 76 (0x4c)
astore_2 = 77 (0x4d)
astore_3 = 78 (0x4e)

astore_<n>, astore

Code: astore_<n>

Format: astore_<n>

Bechreibung: Schreibt die oberste Referenz des Stacks in die *n*-te lokale Variable und löscht die Referenz vom Stack

Formen: astore_0 = 75 (0x4b)
astore_1 = 76 (0x4c)
astore_2 = 77 (0x4d)
astore_3 = 78 (0x4e)

Code: astore

Format: astore index

Bechreibung: Schreibt die oberste Referenz des Stacks in die *index*-te lokale Variable und löscht die Referenz vom Stack

Formen: astore = 58 (0x3a)

Methodenaufruf

```
class A { int m2(int a)  return a;  }
class bct {
  void m () {
      int j = 0;
      A aa = new A();
      int i = aa.m2(j); }
}
```

führt zu

```
...
 2  new #2  // A
 5  dup
 6  invokespecial #3  // A.<init> ()V
 9  astore_2
10  aload_2
11  iload_1
12  invokevirtual #4  // A.m2 (I)I
15  istore_3
16  return
```

iload_<n>, iload

Code: iload_<n>

Format: iload_<n>

Bechreibung: Lädt den Integerwert aus der *n*-ten lokalen Variable auf den Stack.

Formen: iload_0 = 26 (0x1a)
iload_1 = 27 (0x1b)
iload_2 = 28 (0x1c)
iload_3 = 29 (0x1d)

iload_<n>, iload

Code: iload_<n>

Format: iload_<n>

Bechreibung: Lädt den Integerwert aus der *n*-ten lokalen Variable auf den Stack.

Formen: iload_0 = 26 (0x1a)
iload_1 = 27 (0x1b)
iload_2 = 28 (0x1c)
iload_3 = 29 (0x1d)

Code: iload

Format: iload index

Bechreibung: Lädt den Integerwert aus der *index*-ten lokalen Variable auf den Stack.

Formen: iload = 21 (0x15)

invokevirtual

Code: invokevirtual

Format: invokevirtual indexbyte1 indexbyte2

Bechreibung: Ruft die Instanzmethode *abhängig von der aktuellen Receiver–Klasse* auf, die durch die Referenz *indexbyte1* $<< 8 \mid$ *indexbyte2* in dem Konstantenpool bestimmt wird.

- Wenn die Methode nicht vorhanden ist wird in Superklassen gesucht.
- Die Argumente der Methoden müssen ebenfalls auf dem Stack liegen.
- Am Ende werden die Argumente und das Objekt vom Stack gelöscht und das Ergebnis drauf gelegt.

Formen: invokevirtual = 182 (0xb6)

Dynamische Bindung I

```
class Saeugetier {
    void steckbrief() { System.out.println("Ich bin ein Säugetier."); }
}

class Gepard extends Saeugetier {
    void steckbrief() { System.out.println("Ich bin ein Gepard."); }
}

class Elefant extends Saeugetier {
    void steckbrief() { System.out.println("Ich bin ein Elefant."); }
}
public class DynamischeBindungBeispiel {
    public static void main(String[] args) {
        List<Saeugetier> tiere = new ArrayList<Saeugetier>();
        tiere.add(new Saeugetier());
        tiere.add(new Gepard());
        tiere.add(new Elefant());

        for (Saeugetier tier: tiere) {
            tier.steckbrief(); } } }
```

Dynamische Bindung II

```
    public static void main(String[] args) {
        List<Saeugetier> tiere = new ArrayList<Saeugetier>();
        tiere.add(new Saeugetier());
        tiere.add(new Gepard());
        tiere.add(new Elefant());

        for (Saeugetier tier: tiere) {
            tier.steckbrief(); } } }
```

Ausgabe

```
Ich bin ein Säugetier.
Ich bin ein Gepard.
Ich bin ein Elefant.
```

Dynamische Bindung III

```
class Superclass {

    void interestingMethod() {
        System.out.println("Superclass's interesting method."); }

    void exampleMethod() {
        interestingMethod(); }
}

class Subclass extends Superclass {

    void interestingMethod() {
        System.out.println("Subclass's interesting method."); }

    public static void main(String args[]) {
        Subclass me = new Subclass();
        me.exampleMethod(); }
}
```

Dynamische Bindung III

```
class Superclass {

    void interestingMethod() {
        System.out.println("Superclass's interesting method."); }

    void exampleMethod() {
        interestingMethod(); }
}

class Subclass extends Superclass {

    void interestingMethod() {
        System.out.println("Subclass's interesting method."); }

    public static void main(String args[]) {
        Subclass me = new Subclass();
        me.exampleMethod(); }
}
```

Erg: Subclass's interesting method.

Dynamische Bindung (Bytecode)

```
class Superclass {

    void interestingMethod() {
        System.out.println("Superclass's interesting method."); }

    void exampleMethod() {
        interestingMethod(); }
}
```

führt zu

```
0  aload_0
1  invokevirtual #5  // interestingMethod ()V (dynamisch)
4  return
```

Statische Bindung

```
class Superclass {

    private void interestingMethod() {
        System.out.println("Superclass's interesting method."); }

    void exampleMethod() {
        interestingMethod(); }
}

class Subclass extends Superclass {

    void interestingMethod() {
        System.out.println("Subclass's interesting method."); }

    public static void main(String args[]) {
        Subclass me = new Subclass();
        me.exampleMethod(); }
}
```

Statische Bindung

```
class Superclass {

    private void interestingMethod() {
        System.out.println("Superclass's interesting method."); }

    void exampleMethod() {
        interestingMethod(); }
}

class Subclass extends Superclass {

    void interestingMethod() {
        System.out.println("Subclass's interesting method."); }

    public static void main(String args[]) {
        Subclass me = new Subclass();
        me.exampleMethod(); }
}
```

Erg: Superclass's interesting method.

Statische Bindung (Bytecode)

```
class Superclass {

    private void interestingMethod() {
        System.out.println("Superclass's interesting method."); }

    void exampleMethod() {
        interestingMethod(); }
}
```

führt zu

```
0  aload_0
1  invokespecial #5  // Superclass.interestingMethod ()V
4  return
```

Return

```
class bct {
   bct b;

    int m1() { return 1; }

    bct m2() { return b; }
}
```

führt zu

```
0  iconst_1       //m1
1  ireturn

0  aload_0        //m2
1  getfield #2    //bct.b Lbct;
4  areturn
```

areturn, ireturn

Code: areturn

Format: areturn

Bechreibung: Rückgabe des obersten Elements des (Operanden-)Stacks als Referenz auf ein Objekt. Legt die Referenz auf den (Operanden-)Stack des Frames der aufrufenden Methode.

Formen: areturn = 176 (0xb0)

areturn, ireturn

Code: areturn

Format: areturn

Bechreibung: Rückgabe des obersten Elements des (Operanden-)Stacks als Referenz auf ein Objekt. Legt die Referenz auf den (Operanden-)Stack des Frames der aufrufenden Methode.

Formen: areturn = 176 (0xb0)

Code: ireturn

Format: ireturn

Bechreibung: Rückgabe des obersten Elements des (Operanden-)Stacks als `int`, `boolean`, `byte`, `short`, `char` und legt Element auf den (Operanden-)Stack des Frames der aufrufenden Methode.

Formen: ireturn = 172 (0xb0)

While

```
class bct {

    void m () {
        int i = 0;
        while (i == 1) { i++; }
    }
}
```

führt zu

```
 0  iconst_0
 1  istore_1
 2  iload_1
 3  iconst_1
 4  if_icmpne 0 9
 7  iinc 1 1     //Variable 1 (+1)
10  goto 255 248
13  return
```

if_icmp<cond>

Code: if_icmp<cond>

Format: if_icmp<cond> branchbyte1 branchbyte2

Bechreibung: relativer Sprung nach 16 bit vorzeichenbehaftete Zahl
branchbyte1 $<< 8 \mid$ *branchbyte2* wenn jeweiliger Integervergleich des
zweitobersten mit den obersten Stack Element erfolgreich ist

Formen: if_icmpeq = 159 (0x9f)
if_icmpne = 160 (0xa0)
if_icmplt = 161 (0xa1)
if_icmpge = 162 (0xa2)
if_icmpgt = 163 (0xa3)
if_icmple = 164 (0xa4)

goto

Code: goto

Format: goto branchbyte1 branchbyte2

Bechreibung: relativer Sprung nach 16 bit vorzeichenbehaftete Zahl
$branchbyte1 << 8 \mid branchbyte2$

Formen: goto = 167 (0xa7)

If

```
class bct {

    void m () {
        int i = 0;
        if (i == 1) { i++; }
    }
}
```

führt zu

```
 0  iconst_0
 1  istore_1
 2  iload_1
 3  iconst_1
 4  if_icmpne 0 6
 7  iinc 1 1     //Variable 1 (+1)
10  return
```

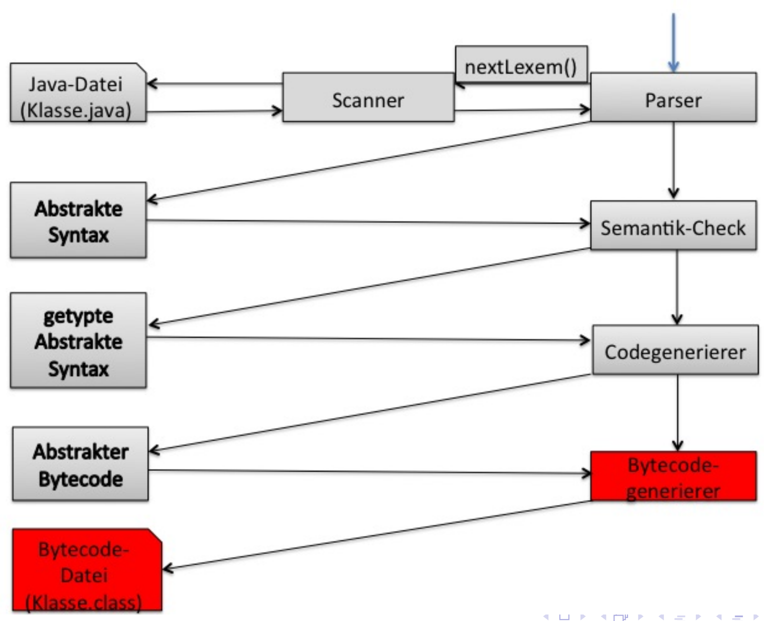
If_else

```
class bct {
    void m () {
        int i = 0;
        if (i == 1) { i++; }
        else { i--; }
    }
}
```

führt zu

```
 0  iconst_0
 1  istore_1
 2  iload_1
 3  iconst_1
 4  if_icmpne 0 9
 7  iinc 1 1     //Variable 1 (+1)
10  goto 0 6
13  iinc 1 255   //Variable 1 (-1)
16  return
```

Bytecodegenerierung



Abstrakter Bytecode -> Bytecode

Analyse des Binär-Files

Hexadezimal:

```
od -tx1 bct.class

0000000   ca fe ba be 00 00 00 34 00 0c 0a 00 03 00 09 07
0000020   00 0a 07 00 0b 01 00 01 69 01 00 13 4c 6a 61 76
0000040   61 2f 6c 61 6e 67 2f 49 6e 74 65 67 65 72 3b 01
0000060   00 06 3c 69 6e 69 74 3e 01 00 03 28 29 56 01 00
0000100   04 43 6f 64 65 0c 00 06 00 07 01 00 03 62 63 74
0000120   01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a
0000140   65 63 74 00 20 00 02 00 03 00 00 00 01 00 00 00
0000160   04 00 05 00 00 00 01 00 00 00 06 00 07 00 01 00
0000200   08 00 00 00 11 00 01 00 01 00 00 00 05 2a b7 00
0000220   01 b1 00 00 00 00 00 00
0000230
```

```
magic = 0x CAFEBABE
minor_version = 0
major_version = 52

ca  fe  ba  be  00  00  00  34
```

```
constant_pool_count = 12
constant_pool =
{
1| tag = CONSTANT_Methodref, class_index = 3, name_and_type_index = 9
2| tag = CONSTANT_Class, name_index = 10
3| tag = CONSTANT_Class, name_index = 11
4| tag = CONSTANT_Utf8, length = 1, bytes = i
5| tag = CONSTANT_Utf8, length = 19, bytes = Ljava/lang/Integer;
6| tag = CONSTANT_Utf8, length = 6, bytes = <init>
7| tag = CONSTANT_Utf8, length = 3, bytes = ()V
8| tag = CONSTANT_Utf8, length = 4, bytes = Code
9| tag = CONSTANT_NameAndType, name_index = 6, descriptor_index = 7
10| tag = CONSTANT_Utf8, length = 3, bytes = bct
11| tag = CONSTANT_Utf8, length = 16, bytes = java/lang/Object
}
```

```
                                00 0c 0a 00 03 00 09 07
00 0a 07 00 0b 01 00 01 69 01 00 13 4c 6a 61 76
61 2f 6c 61 6e 67 2f 49 6e 74 65 67 65 72 3b 01
00 06 3c 69 6e 69 74 3e 01 00 03 28 29 56 01 00
04 43 6f 64 65 0c 00 06 00 07 01 00 03 62 63 74
01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 4f 62 6a
65 63 74
```

```
access_flags = 32  // ACC_SUPER
this_class = #2  // bct
super_class = #3  // java/lang/Object
interfaces_count = 0
interfaces = {}
```

00 20 00 02 00 03 00 00

```
fields_count = 1
fields [0] =
{
access_flags = 0
name_index = #4  // i
descriptor_index = #5  // Ljava/lang/Integer;
attributes_count = 0
attributes = {}
}
```

```
00  01  00  00  00 04  00  05  00  00
```

```
methods_count = 1
methods [0] =
{
access_flags = 0
name_index = #6  // <init>
descriptor_index = #7  // ()V

00  01  00  00  00  06  00  07
```

```
attributes_count = 1
attributes [0] =
{
attribute_name_index = #8  // Code
attribute_length = 17
max_stack = 1, max_locals = 1
code_length = 5
code =
{
    0  aload_0
    1  invokespecial #1  // java/lang/Object.<init> ()V
    4  return
}

00  01  00  08  00  00  00  11  00  01 00  01  00  00  00  05
2a  b7  00  01  b1
```

```
exception_table_length = 0
exception_table = {}
attributes_count = 0
attributes = {}
}
}
```

00 00 00 00

```
exception_table_length = 0
exception_table = {}
attributes_count = 0
attributes = {}
}
}
```

```
00  00  00  00
```

```
attributes_count = 0
attributes = {}
```

```
00  00
```

Dezimal (8 Bit)

```
od -td1 bct.class

0000000   -54  -2 -70 -66   0   0   0  52   0  12  10   0   3   0   9   7
0000020     0  10   7   0  11   1   0   1 105   1   0  19  76 106  97 118
0000040    97  47 108  97 110 103  47  73 110 116 101 103 101 114  59   1
0000060     0   6  60 105 110 105 116  62   1   0   3  40  41  86   1   0
0000100     4  67 111 100 101  12   0   6   0   7   1   0   3  98  99 116
0000120     1   0  16 106  97 118  97  47 108  97 110 103  47  79  98 106
0000140   101  99 116   0  32   0   2   0   3   0   0   0   1   0   0   0
0000160     4   0   5   0   0   0   1   0   0   0   6   0   7   0   1   0
0000200     8   0   0   0  17   0   1   0   1   0   0   0   5  42 -73   0
0000220     1 -79   0   0   0   0   0   0
0000230
```

ASCII

```
od -ta bct.class

0000000                                  nul  ff  nl nul etx nul  ht bel
0000020 nul  nl bel nul  vt soh nul soh   i soh nul dc3   L   j   a   v
0000040   a   /   l   a   n   g   /   I   n   t   e   g   e   r   ; soh
0000060 nul ack   <   i   n   i   t   > soh nul etx   (   )   V soh nul
0000100 eot   C   o   d   e  ff nul ack nul bel soh nul etx   b   c   t
0000120 soh nul dle   j   a   v   a   /   l   a   n   g   /   O   b   j
0000140   e   c   t nul  sp nul stx nul etx nul nul nul soh nul nul nul
0000160 eot nul enq nul nul nul soh nul nul nul ack nul bel nul soh nul
0000200  bs nul nul nul dc1 nul soh nul soh nul nul nul enq
0000220 soh     nul nul nul nul nul nul
0000230
```

Haskell: Codegen

```
codegen :: Class -> ClassFile
-- bildet abstrakte Syntax in abstrakten Bytecode ab

compiler :: String -> ClassFile
compiler = codegen . typecheck . parse . alexScanTokens

encodeClassFile :: FilePath -> ClassFile -> IO()
-- erzeugt Bytecode--Datei

decodeClassFile :: FilePath -> IO ClassFile
--liest Bytecode--Datei und erzeugt Class daraus
```

Haskell: Codegen

```
codegen :: Class -> ClassFile
-- bildet abstrakte Syntax in abstrakten Bytecode ab

compiler :: String -> ClassFile
compiler = codegen . typecheck . parse . alexScanTokens

encodeClassFile :: FilePath -> ClassFile -> IO()
-- erzeugt Bytecode--Datei

decodeClassFile :: FilePath -> IO ClassFile
--liest Bytecode--Datei und erzeugt Class daraus

main = do
  s <- readFile "name.java"
  encodeClassFile "name.class" (compiler s)
  cf <- decodeClassFile "name.class"
  print cf
```

javap

`javap` disassembliert Classfiles

Optionen:

-v Verbose (zusäztliche Informationen)

-p Zeigt auch private Klassen, Methoden und Attribute

javap -v -p bct.class I

```
class bct
  minor version: 0
  major version: 50
  flags: (0x0020) ACC_SUPER
  this_class: #2                          // bct
  super_class: #3                         // java/lang/Object
  interfaces: 0, fields: 1, methods: 1, attributes: 0
Constant pool:
   #1 = Methodref          #3.#9          // java/lang/Object."<init>":()
        V
   #2 = Class              #10            // bct
   #3 = Class              #11            // java/lang/Object
   #4 = Utf8               i
   #5 = Utf8               Ljava/lang/Integer;
   #6 = Utf8               <init>
   #7 = Utf8               ()V
   #8 = Utf8               Code
   #9 = NameAndType        #6:#7          // "<init>":()V
  #10 = Utf8               bct
  #11 = Utf8               java/lang/Object
```

javap -v -p bct.class II

```
{
  java.lang.Integer i;
    descriptor: Ljava/lang/Integer;
    flags: (0x0000)

  bct();
    descriptor: ()V
    flags: (0x0000)
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1  // Method java/lang/Object."<init>":()V
         4: return
}
```

Bytecode Generierung mit ASM

Fayez Abu Alia

javap

javap

- Disassemblierung von Klassendatei

Option	Aufgabe
`-c`	zeigt den erzeugten Bytecode
`-p/-private`	zeigt alle Methoden und Elemente der Klasse
`-s`	zeigt die Deskriptoren
`-v/-verbose`	zeigt ausführliche Informationen wie z.B. Konstantenpool

- Keine Option => zeigt nur die Struktur der Klassendatei

```java
public class Bytecode {

   public void m(){
      System.out.println("Bytecode");
   }

}
```

```
~/Bytecode/examples » javap -c Bytecode
Compiled from "Bytecode.java"
public class Bytecode {
  public Bytecode();
    Code:
       0: aload_0
       1: invokespecial #1   // Method java/lang/Object."<init>":()V
       4: return

  public void m();
    Code:
       0: getstatic     #2   // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3   // String Bytecode
       5: invokevirtual #4   // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

```java
public class Bytecode {

    public void m(){
        System.out.println("Bytecode");
    }

}
```

```
~/Bytecode/examples » javap -c Bytecode
Compiled from "Bytecode.java"
public class Bytecode {
  public Bytecode();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
       4: return

  public void m();
    Code:
       0: getstatic     #2    // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3    // String Bytecode
       5: invokevirtual #4    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

this auf den Stack laden

```java
public class Bytecode {

   public void m(){
      System.out.println("Bytecode");
   }

}
```

```
~/Bytecode/examples » javap -c Bytecode
Compiled from "Bytecode.java"
public class Bytecode {
  public Bytecode();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
       4: return

  public void m();
    Code:
       0: getstatic     #2    // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3    // String Bytecode
       5: invokevirtual #4    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

<init> auf *this* aufrufen

```java
public class Bytecode {

    public void m(){
        System.out.println("Bytecode");
    }

}
```

```
~/Bytecode/examples » javap -c Bytecode
Compiled from "Bytecode.java"
public class Bytecode {
  public Bytecode();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
       4: return

  public void m();
    Code:
       0: getstatic     #2    // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3    // String Bytecode
       5: invokevirtual #4    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

Konstruktor verlassen

```java
public class Bytecode {

    public void m(){
        System.out.println("Bytecode");
    }

}
```

```
~/Bytecode/examples » javap -c Bytecode
Compiled from "Bytecode.java"
public class Bytecode {
  public Bytecode();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
       4: return

  public void m();
    Code:
       0: getstatic     #2    // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3    // String Bytecode
       5: invokevirtual #4    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

Statischer Feld auslesen

```java
public class Bytecode {

    public void m(){
        System.out.println("Bytecode");
    }

}
```

```
~/Bytecode/examples » javap -c Bytecode
Compiled from "Bytecode.java"
public class Bytecode {
  public Bytecode();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
       4: return

  public void m();
    Code:
       0: getstatic     #2    // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3    // String Bytecode
       5: invokevirtual #4    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

„Bytecode“ auf den Stack laden

```java
public class Bytecode {

   public void m(){
      System.out.println("Bytecode");
   }

}
```

```
~/Bytecode/examples » javap -c Bytecode
Compiled from "Bytecode.java"
public class Bytecode {
  public Bytecode();
    Code:
       0: aload_0
       1: invokespecial #1   // Method java/lang/Object."<init>":()V
       4: return

  public void m();
    Code:
       0: getstatic     #2   // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3   // String Bytecode
       5: invokevirtual #4   // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

Methode „println“ aufrufen

```java
public class Bytecode {

    public void m(){
        System.out.println("Bytecode");
    }

}
```

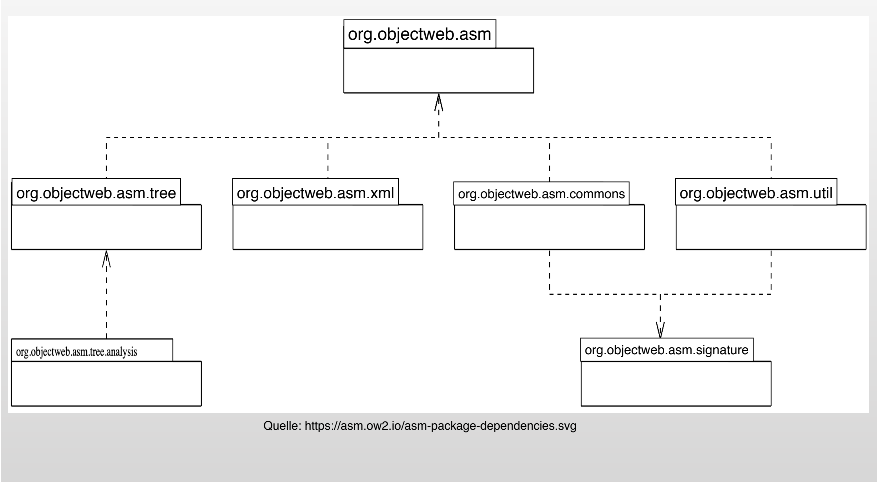
```
~/Bytecode/examples » javap -c Bytecode
Compiled from "Bytecode.java"
public class Bytecode {
  public Bytecode();
    Code:
       0: aload_0
       1: invokespecial #1   // Method java/lang/Object."<init>":()V
       4: return

  public void m();
    Code:
       0: getstatic     #2   // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3   // String Bytecode
       5: invokevirtual #4   // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

Methode verlassen
Keine Rückgabe

ASM

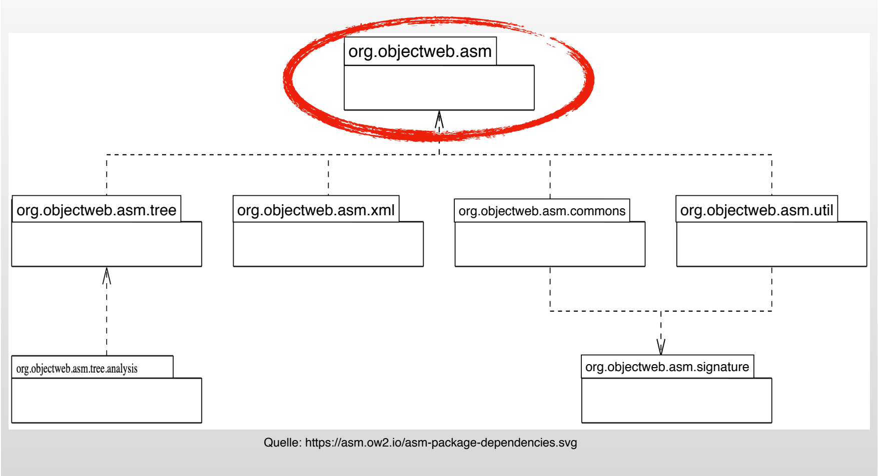
- ein Java Framework für Bearbeitung und Analyse von Bytecode
- Generierung und Transformation von Kompilierten Klassen
- Website:
  - https://asm.ow2.io/
  - Documentation
- Aktuelle Version 7.1

ASM



Quelle: https://asm.ow2.io/asm-package-dependencies.svg

ASM



Quelle: https://asm.ow2.io/asm-package-dependencies.svg

`org.objectweb.asm`

- Das core Package
- bietet:
  - ClassReader - parst kompilierten Java Klassen
  - ClassWriter - generiert Bytecode

ClassWriter

```
public class ClassWriter extends ClassVisitor {

      public ClassWriter(int flags);

      public void visit(int version, int access, String name,
            String signature,String superName, String[] interfaces);

      public FieldVisitor visitField(int access, String name,
                String descriptor, String signature, Object value);

      public MethodVisitor visitMethod(int access, String name,
                String descriptor, String signature, String[] exceptions);

      public void visitEnd();

      public byte[] toByteArray();
                      ⋮
}
```

MethodVisitor

```
public abstract class MethodVisitor {

      public void visitCode();

      public void visitInsn(int opcode);

      public void visitVarInsn(int opcode, int operand);

      public void visitLdcInsn(Object cst);

      public void visitFieldInsn(int opc, String owner, String name,
                              String desc);

      public void visitMethodInsn(int opc, String owner, String name,
                              String desc);

      public void visitLabel(Label label);

      public void visitMaxs(int maxStack, int maxLocals);

      public void visitEnd();
                            ⋮
}
```

Opcodes

- Interface enthält Konstanten:
  - Java Versions: `V1_5, V1_6, V1_7,…`
  - Zugriffsrechte: `ACC_PUBLIC, ACC_PRIVATE,…`
  - Bytecodes: `ICONST_0, ALOAD, IADD,…`

Prozess

- Erstelle ClassWriter-Instanz
- Rufe für jedes Element der Klasse die entsprechende visit-Methode auf
- Schreibe Bytecode aus

Prozess

```
ClassWriter cw = ...;
cw.visit(…);

MethodVisitor mv1 = cw.visitMethod(..., "m1", ...);
MethodVisitor mv2 = cw.visitMethod(..., "m2", ...);
// Bytecode Generierung für „m1“
mv1.visitCode();
mv1.visitInsn(...);
...
mv1.visitMaxs(...);
mv1.visitEnd();
// Bytecode Generierung für „m2“
mv2.visitCode();
mv2.visitInsn(...);
...
mv2.visitMaxs(...);
mv2.visitEnd();

cw.visitEnd();
```

Beispiel 1

```
public class Bytecode {

  public void m(){
    System.out.println("Bytecode");
  }

}
```

Beispiel 1

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

- ClassWriter.***COMPUTE_MAXS*** **:** Automatische Berechnung von der maximalen Anzahl der lokalen Variablen und Größe der Stack
- ClassWriter.***COMPUTE_FRAMES*** **:** Automatische Berechnung von Stackmap Frames

Beispiel 1

```
cw.visit(Opcodes.V1_8,
         Opcodes.ACC_PUBLIC,
         "Bytecode",
         null,
         "java/lang/Object",
         null);
```

- definiert Klassenkopf

Beispiel 1

```
MethodVisitor constructor =
        cw.visitMethod(Opcodes.ACC_PUBLIC,
                       "<init>",
                       "()V",
                       null,
                       null);

MethodVisitor method =
        cw.visitMethod(Opcodes.ACC_PUBLIC,
                       "m",
                       "()V",
                       null,
                       null);
```

Beispiel 1

```
constructor.visitCode();

// aload_0
constructor.visitVarInsn(Opcodes.ALOAD, 0);

// invokespecial
constructor.visitMethodInsn(Opcodes.INVOKESPECIAL,
        "java/lang/Object", "<init>", "()V", false);

// return
constructor.visitInsn(Opcodes.RETURN);

constructor.visitMaxs(0, 0);

constructor.visitEnd();;
```

Beispiel 1

```
method.visitCode();
// getstatic
method.visitFieldInsn(Opcodes.GETSTATIC,
                      "java/lang/System", "out",
                      "Ljava/io/PrintStream;");
// ldc "Bytecode"
method.visitLdcInsn("Bytecode");
// invokevirtual
method.visitMethodInsn(Opcodes.INVOKEVIRTUAL,
                    "java/io/PrintStream", "println",
                    "(Ljava/lang/String;)V", false);
// return
method.visitInsn(Opcodes.RETURN);

method.visitMaxs(0, 0);

method.visitEnd();
```

Beispiel 2

```
public void m() {
    int i = 0;
    while(i<5) {
      System.out.println("Bytecode");
      i++;
    }
}
```

```
public void m() {
    int i = 0;
    while(i<5) {
      System.out.println("Bytecode");
      i++;
    }
}
```

```
start:
  int i = 0

loop:
  if i>=5 goto end
  print("Bytecode")
  i = i + 1
  goto loop

end:
  return
```

```
start:
  int i = 0

loop:
  if i>=5 goto end

print("Bytecode")
  i = i + 1
  goto loop

end:
  return
```

```
Label start = new Label();
Label loop = new Label();
Label end = new Label();
```

```
start:
  int i = 0
```

```
Label start = new Label();
Label loop = new Label();
Label end = new Label();

method.visitLabel(start);
method.visitInsn(Opcodes.ICONST_0);
method.visitVarInsn(Opcodes.ISTORE, 1);
```

```
start:
  int i = 0
```

```
loop:
  if i>=5 goto end
  print("Bytecode")
  i = i + 1
  goto loop
```

```
Label start = new Label();
Label loop = new Label();
Label end = new Label();

method.visitLabel(start);
method.visitInsn(Opcodes.ICONST_0);
method.visitVarInsn(Opcodes.ISTORE, 1);
```

```
method.visitLabel(loop);
method.visitVarInsn(Opcodes.ILOAD, 1);
method.visitInsn(Opcodes.ICONST_5);
method.visitJumpInsn(Opcodes.IF_ICMPGE, end);
```

```
start:
  int i = 0
```

```
loop:
  if i>=5 goto end
  print("Bytecode")
  i = i + 1
  goto loop
```

```java
Label start = new Label();
Label loop = new Label();
Label end = new Label();

method.visitLabel(start);
method.visitInsn(Opcodes.ICONST_0);
method.visitVarInsn(Opcodes.ISTORE, 1);
```

```java
method.visitLabel(loop);
method.visitVarInsn(Opcodes.ILOAD, 1);
method.visitInsn(Opcodes.ICONST_5);
method.visitJumpInsn(Opcodes.IF_ICMPGE, end);

method.visitFieldInsn(Opcodes.GETSTATIC,
          "java/lang/System", "out",
          "Ljava/io/PrintStream;");
method.visitLdcInsn("Bytecode");
method.visitMethodInsn(Opcodes.INVOKEVIRTUAL,
          "java/io/PrintStream", "println",
          "(Ljava/lang/String;)V", false);
```

```
start:
  int i = 0
```

```
Label start = new Label();
Label loop = new Label();
Label end = new Label();

method.visitLabel(start);
method.visitInsn(Opcodes.ICONST_0);
method.visitVarInsn(Opcodes.ISTORE, 1);
```

```
loop:
  if i>=5 goto end
  print("Bytecode")
  i = i + 1
  goto loop
```

```
method.visitLabel(loop);
method.visitVarInsn(Opcodes.ILOAD, 1);
method.visitInsn(Opcodes.ICONST_5);
method.visitJumpInsn(Opcodes.IF_ICMPGE, end);

method.visitFieldInsn(Opcodes.GETSTATIC,
          "java/lang/System", "out",
          "Ljava/io/PrintStream;");
method.visitLdcInsn("Bytecode");
method.visitMethodInsn(Opcodes.INVOKEVIRTUAL,
          "java/io/PrintStream", "println",
          "(Ljava/lang/String;)V", false);

method.visitIincInsn(1, 1);
method.visitJumpInsn(Opcodes.GOTO, loop);
```

```
start:
  int i = 0
```

```java
Label start = new Label();
Label loop = new Label();
Label end = new Label();

method.visitLabel(start);
method.visitInsn(Opcodes.ICONST_0);
method.visitVarInsn(Opcodes.ISTORE, 1);
```

```
loop:
  if i>=5 goto end
  print("Bytecode")
  i = i + 1
  goto loop
```

```java
method.visitLabel(loop);
method.visitVarInsn(Opcodes.ILOAD, 1);
method.visitInsn(Opcodes.ICONST_5);
method.visitJumpInsn(Opcodes.IF_ICMPGE, end);

method.visitFieldInsn(Opcodes.GETSTATIC,
           "java/lang/System", "out",
           "Ljava/io/PrintStream;");
method.visitLdcInsn("Bytecode");
method.visitMethodInsn(Opcodes.INVOKEVIRTUAL,
           "java/io/PrintStream", "println",
           "(Ljava/lang/String;)V", false);

method.visitIincInsn(1, 1);
method.visitJumpInsn(Opcodes.GOTO, loop);
```

```
end:
  return
```

```java
method.visitLabel(end);
method.visitInsn(Opcodes.RETURN);
```

Literatur

- https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html
- https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html
- https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javap.html
- Offizielle ASM-Webseite: https://asm.ow2.io/

codeGen–Methode

Ablauf: Lauf über alle Fields, alle Methoden, alle Statements und alle Expressions der Klasse:

- Aufruf der jeweiligen `visit`–Funktionen.

Beispiel: Klasse `Class`

```
class Class {
  Type name;
  Vector<Field> fields
  Vector<Method> meth;

  void typeCheck() { ... }

  void codeGen() {
    ClassWriter cw = new ClassWriter( ClassWriter.COMPUTE_FRAMES |
                                      ClassWriter.COMPUTE_MAXS);
    cw.visit(Opcodes.V1_8, Opcodes.ACC_PUBLIC, name, null,
             "java/lang/Object", null);
    for(field: fields) { field.codeGen(cw); }
    for(m: meth) { m.codeGen(cw); }
    writeClassfile(cw); }
}
```