Compilerkonstruktion

Wintersemester 2015/16

Prof. Dr. R. Parchmann

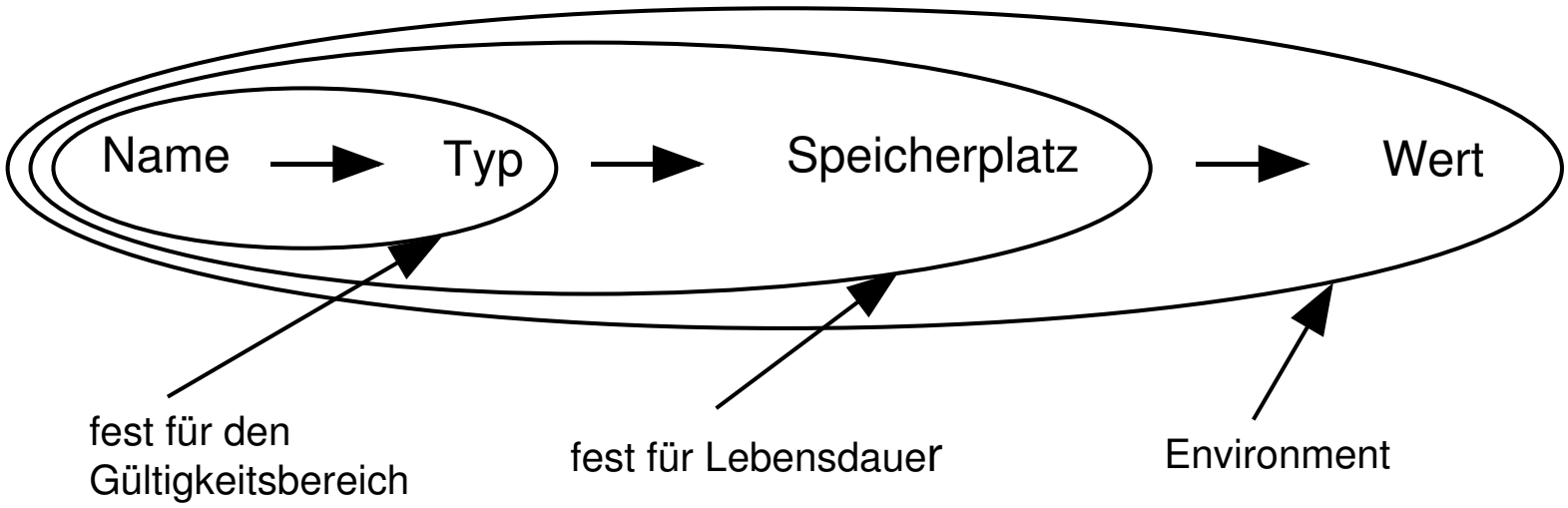
10. November 2015

Das Laufzeitsystem

Gültigkeitsbereich und Lebensdauer

- Der Gültigkeitsbereich (Scope) einer Deklaration der Variablen `abc` umfasst den Teil des Programms, in dem sich eine Verwendung des Namens `abc` auf diese Deklaration beziehen würde.
  - statischer (lexikographischer) Gültigkeitsbereich, d.h. unabhängig von einem Programmlauf, allein durch den Programmtext, festgelegt.
  - dynamischer Gültigkeitsbereich, letzte abgearbeitete Deklaration ist gültig

- Die Lebensdauer (Extend) des durch die Variabel `abc` bezeichneten Objekts beginnt mit der Speicherzuordnung für das Objekt und endet mit der Freigabe des Speichers.
  - Speicherzuordnung beim Eintritt und Freigabe beim Verlassen einer Prozedur oder eines Blocks.
  - Speicherzuordnung durch expliziten Aufruf eines Konstruktors und Freigabe des Speichers durch den Aufruf eines Destruktors
  - automatische Garbage-Collector Verfahren, bei denen nicht mehr referenzierte Objekte identifiziert und der zugeordnete Speicherplatz wieder freigegeben wird.


Name
Typ
Speicherplatz
Wert
fest für den
Gültigkeitsbereich
fest für Lebensdauer
Environment

Aktivierungs-Record, statische und dynamische Verkettung

Beispiel

Beispiel für Schachtelung von Blöcken und Prozeduren

```
A: begin
     real x,y;
     procedure P();
     begin real z;
         x := x + z;
     end P;
  B: begin
       boolean x,y;
          ...
       C: P();
          ...
       D: begin
             integer x,y;
               ...
             E:  P();
               ...
          end D;
     end B;
   end A;
```

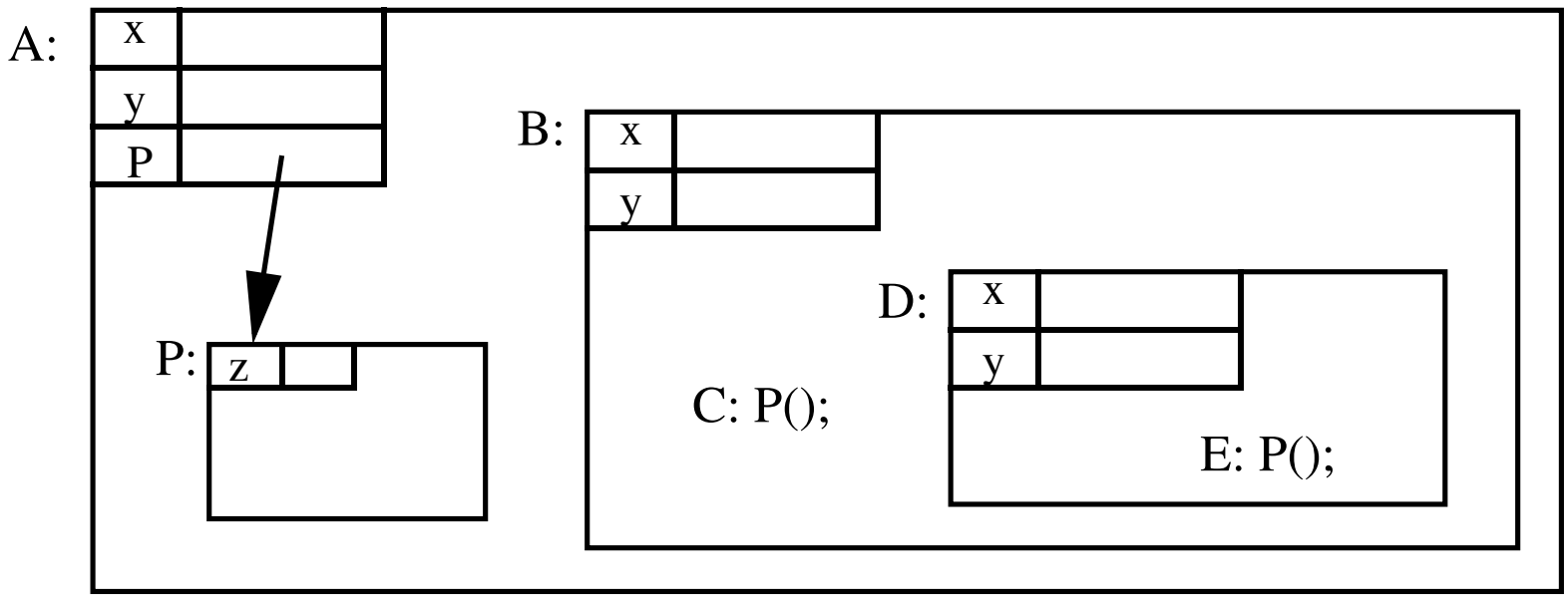
Um die Position einzelner Blöcke bzw. Prozedurrümpfe in der Schachtelungshierarchie zu beschreiben, verwendet man die sogenannte Stufenzahl.

- Alle Deklarationen im äußersten, umfassenden Block sind auf Stufe 1.
- Bei jedem Blockeintritt erhöht sich die Stufenzahl um 1, bei jedem Blockaustritt vermindert sie sich um 1.
- Bei einer Prozedur auf Stufe i sind die Deklarationen im Rumpf sowie die Deklarationen der Formalparameter auf Stufe i, der Name der Prozedur ist jedoch noch auf Stufe i-1 deklariert.

Schachtelungsdiagramm

Beispiel

Darstellung der Schachtelung im vorigen Beispiel durch ein Schachtelungsdiagramm:

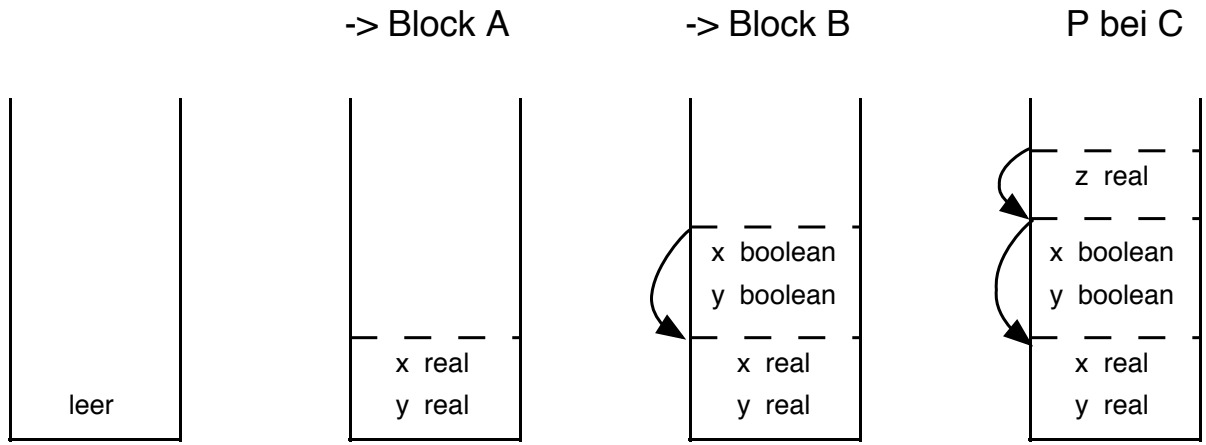


- Die großen Rechtecke stellen jeweils einen Block oder einen Prozedurrumpf dar.
- In der linken oberen Ecke eines derartigen Rechtecks sind die in diesem Block deklarierten Namen mit symbolischen Platzhaltern für die jeweiligen Werte vermerkt.
- Die Stufenzahl wird dann durch die Tiefe der Verschachtelung der Rechtecke repräsentiert.
- Jeder Blockrand ist halb durchlässig - man kann innerhalb eines Blocks nach außen sehen, jedoch nicht von außen in einen Block hinein.

Der Speicher für die deklarierten Variablen eines Blocks wird üblicherweise in einem Aktivierungs-Record (stack-frame) bereitgestellt.
Dieser enthält alle für **eine** Aktivierung einer ausführbaren Programmeinheit (eines Blocks) notwendigen Daten.
Die Aktivierungs-Records werden üblicherweise auf einem Stack, dem sogenannten Laufzeitstack (run-time stack) abgelegt.

Beispiel

Der Laufzeitstack würde sich für das Beispielprogramm dann wie folgt entwickeln:



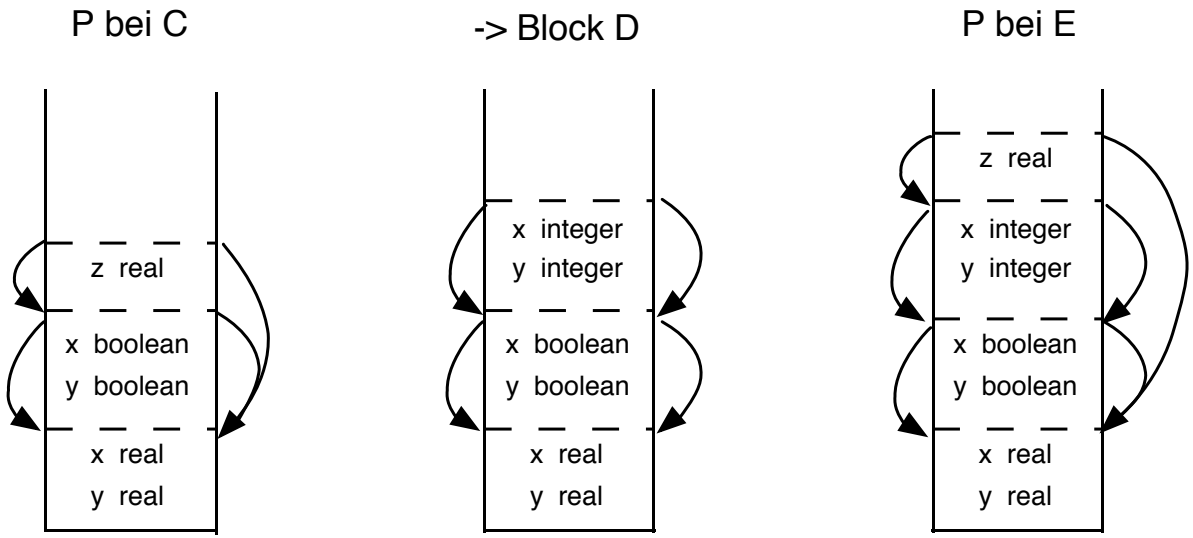
wobei jeder einzelne Aktivierungs-Records einen zusätzlichen Zeiger besitzt (dynamischer Link, durch Zeiger auf der linken Seite des Stacks angedeutet), der auf den Anfang des darunter liegenden Records auf dem Stack zeigt. Damit kann nach Verlassen eines Blocks der zugehörige Aktivierungs-Record wieder gelöscht werden.

Zwei Möglichkeiten der Interpretation bei der Abarbeitung des Befehls `x := x + z` in `P`:

1. Durchsuchen des Stacks von „oben nach unten“, um den ersten Aktivierungs-Record finden, in dem ein x auftritt. (dynamischen Gültigkeitsbereich). Man folgt der Verkettung der Aktivierungs-Records (dem dynamischen Link) und würde die Deklaration von `x` bei `B:` finden.
2. Durchsuchen des Stacks von „oben nach unten“, um den ersten Aktivierungs-Record eines `P` umfassenden Blocks zu finden, der eine Deklaration von `x` enthält (lexikaler Gültigkeitsbereich). Um diesen Record zu finden benötigt man eine weitere Verkettung der Aktivierungs-Records (statischer Link. In diesem Fall würde die Deklaration von `x` bei `A:` gefunden.

Beispiel

Die weitere Entwicklung des Laufzeit-Stacks mit Angabe der statischen Verkettung der Aktivierungs-Records wäre dann wie folgt:



Befindet man sich zur Laufzeit des Programms in einem Block oder einer Prozedur, die auf Stufe i definiert wurde, so hat die Kette der statischen Links die Länge $i - 1$. Eine nicht lokale Größe, die j Stufen zurück definiert wurde, ist in einem Aktivierungs-Record zu finden, der j Schritte in der Kette der statischen Links zurück liegt.

Also werden Variablen 2-dimensional adressiert:

- die Stufenzahl der Deklaration wird benötigt, um den richtigen Aktivierungs-Record zu finden, und
- ein vom Compiler festgelegter Abstand vom Anfang des Aktivierungs-Records (Offset) liefert die endgültige Adresse.

Dies bedeutet aber auch, dass die Variablennamen *nicht* auf dem Aktivierungsstack abgespeichert werden müssen!

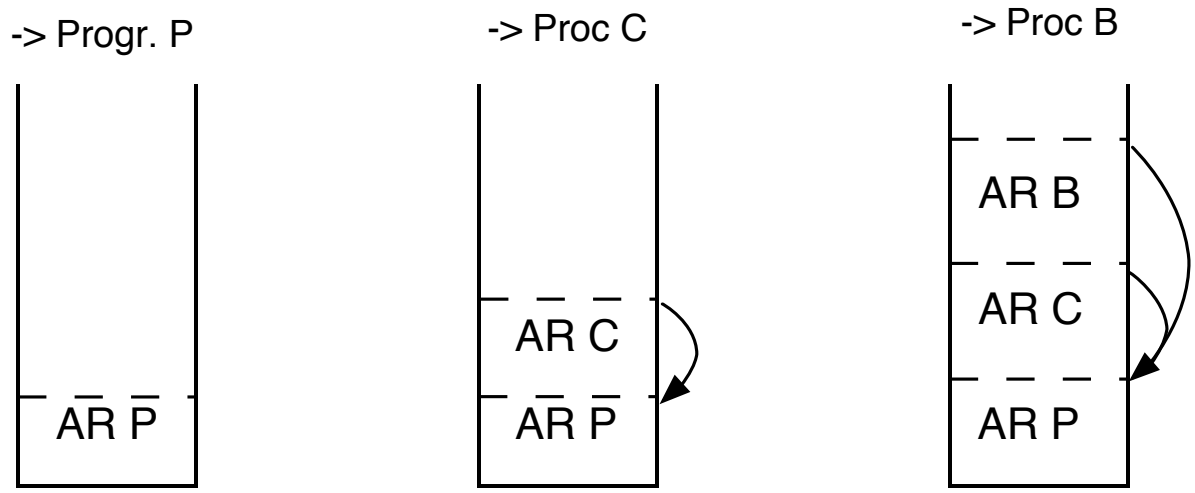
Prozeduren als Parameter oder als Rückgabewerte

Beispiel

Man betrachte das folgende Programmfragment:

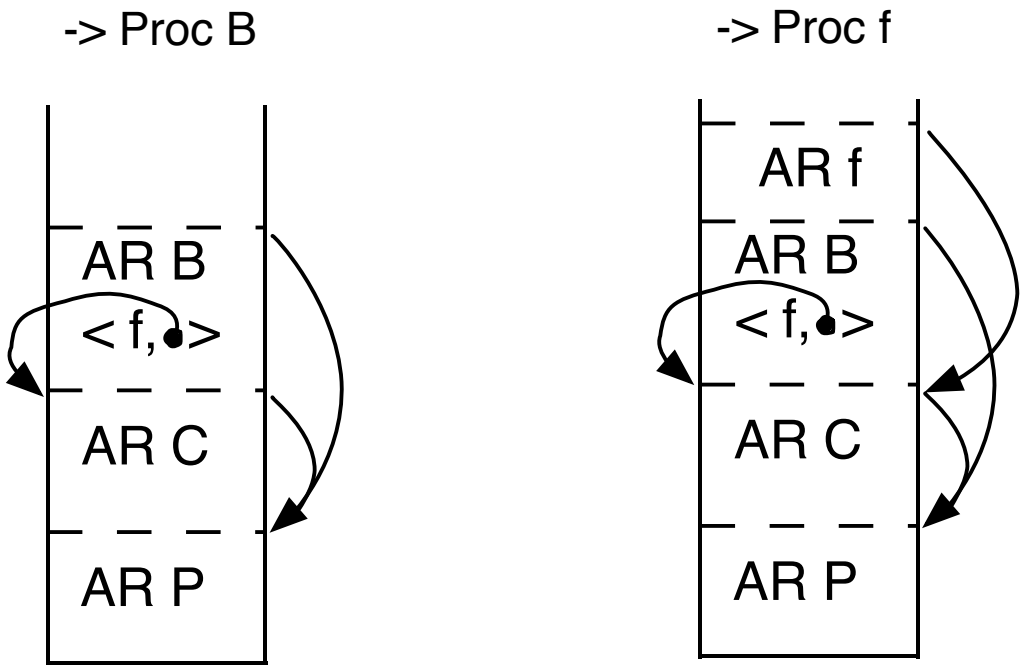
```
program P
    procedure B (function h(n : integer) : integer);
        begin
            writeln (h(2))
        end B;
    procedure C
        var m : integer;
        function f (n:integer) : integer);
            begin
                f := m+n;
            end f;
        begin
            m := 0;
            B(f);
        end C;
    begin
        C;
    end P;
```

Nach dem Aufruf von `C` und danach von `B` ergibt sich folgendes Bild:



In `B` wird jetzt die als Parameter übergebene Funktion `f` aufgerufen. Allerdings tritt im Gültigkeitsbereich von `B` die Variable `m` *nicht* auf. Es ist also wichtig, dass die Funktion `f` ihr Environment mitbringt, d.h. als Parameter für den Aufruf der Prozedur `B` wird sowohl die Startadresse der Prozedur als auch ein Zeiger auf das Environment des Aufrufs, also einen Zeiger auf den Aktivierungs-Record von `C` übergeben. Man nennt ein derartiges Paar auch Closure.

Damit ergibt sich folgendes Bild des Laufzeitstacks:



Ähnliche Probleme treten bei nicht-lokalen Sprüngen auf!

Speicherorganisation

Üblicherweise ordnet das Betriebssystem einem Prozess zumindest die folgenden drei Speichersegmente zu, wobei jedes Speichersegment einen zusammenhängenden, sequentiellen Adressraum darstellt:

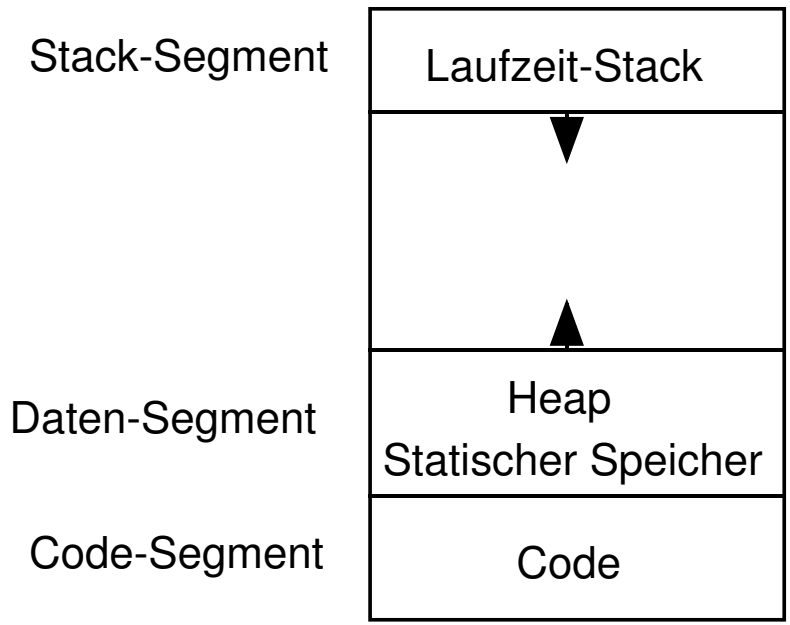
- Das *Code-Segment*, das das ausführbare Programm enthält und üblicherweise schreibgeschützt ist.
- Das *Stack-Segment*, das den Laufzeit-Stack für die Aktivierungs-Records enthält.
- Das *Daten-Segment*, das den statischen Speicher und den dynamischen Speicher (auch Heap genannt) enthält.

Der Laufzeit-Stack enthält den Speicher für Datenobjekte, deren Lebensdauer an die Aktivierung von Prozeduren bzw. Blöcken gebunden ist. Für jede Aktivierung wird im allgemeinen ein Speicherplatz einer festen Größe zugeordnet (Teil des Aktivierungs-Records). Die Adressierung der Objekte erfolgt meist indirekt über eine für alle Aktivierungs-Records gleiche Startposition, die z.B. über einen sogenannten *Frame-Pointer* festgelegt wird, und einen Offset, der bereits zur Übersetzungszeit bekannt ist.

Der statische Speicher enthält Speicherplatz für alle Datenobjekte, deren Lebensdauer sich vom Start des Programms bis zu dessen Ende erstreckt. Diese Speicherzuordnung kann bereits während der Übersetzungsphase erfolgen und die Adressierung kann damit direkt erfolgen.

Der dynamische Speicher enthält Speicherplatz für alle Datenobjekte, deren Lebensdauer nicht durch die Ablaufstruktur der Programmiersprache vorhersagbar ist. Typischerweise sind dies Datenobjekte, die vom Programmierer selbst erzeugt werden können und deren Speicher, je nach Programmiersprache, auch explizit wieder freigegeben werden muss. In C gibt es dazu die Standardaufrufe `malloc` und `mfree`. Auch Konstruktoren in objektorientierten Programmiersprachen erzeugen Objekte, die in diesem Speicherbereich abgelegt werden.

Typische Verteilung der drei Segmente in einem hier angenommenen linearen und zusammenhängenden Adressraum.



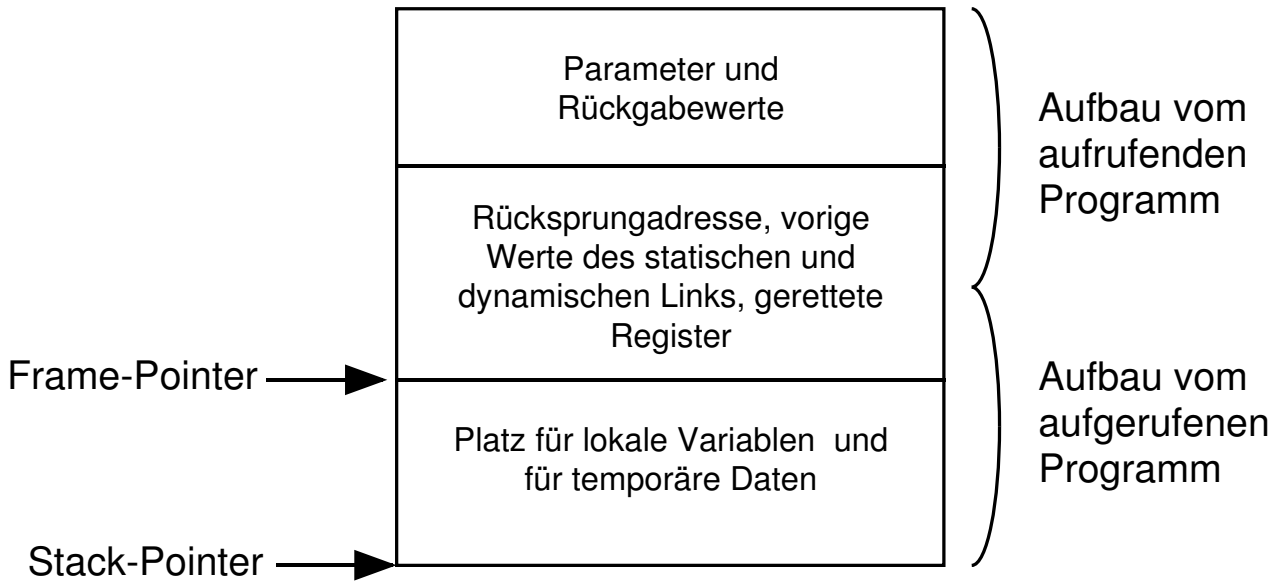
Das Stack-Segment wächst dabei von oben, d.h. zu den kleineren Adressen hin, während der Heap bei Bedarf nach oben, also zu den größeren Adressen hin wächst.

Aufbau eines Aktivierungs-Records

Beim Aufruf einer Prozedur werden

- zunächst die Aktualparameter auf den Stack geschrieben und Platz für eventuelle Rückgabewerte geschaffen.
- danach die momentanen Werte des des dynamischen und statischen Links gerettet.
- Beim Sprung zur Prozedur wird dann die Rücksprungadresse ebenfalls auf den Stack geschrieben.
- Dann werden zunächst die Registerinhalte gerettet und danach, durch Setzen des Stack- und Frame-Pointers, der Aufbau des Aktivierungs-Records beendet.

Möglicher Aufbau eines Aktivierungs-Records



Parameterübergabe

Es gibt mehrere Methoden der Parameterübergabe, da es meist mehrere Möglichkeiten gibt, die Bedeutung eines Aktualparameters zu interpretieren.
Zum Beispiel kann ein Aktualparameter `a[i]` interpretiert werden als

1. der Wert des i-ten Elementes eines Feldes a (*r-Wert*), oder
2. der Ort, an dem das i-te Element eines Feldes a abgespeichert ist (*l-Wert*), oder aber auch
3. eine Zeichenkette, die immer dann einzusetzen ist, wenn der zugeordnete Formalparameter in der Prozedur auftritt (Substitution).

Daraus ergeben sich die folgenden Möglichkeiten der Parameterübergabe:

call-by-value

Formalparameter sind wie initialisierte lokale Variablen zu betrachten - die Aktualparameter werden im Environment des Aufrufs ausgewertet und die r-Werte werden zur Initialisierung der Formalparameter benutzt.

call-by-reference

Hat der Aktualparameter einen l-Wert, so wird dieser übergeben. Ansonsten wird der Aktualparameter ausgewertet und der Wert in einen temporären Speicherplatz, dessen Adresse übergeben wird, abgelegt.

call-by-copy-restore

Die Aktualparameter werden ausgewertet und die r-Werte werden übergeben. Beim Verlassen der Prozedur werden die r-Werte der Formalparameter in die l-Werte der Aktualparameter (soweit vorhanden) zurückgespeichert.

call-by-name

Jedes Auftreten eines Formalparameters wird im entsprechenden Aufruf der Prozedur durch den Aktualparameter textuell substituiert. Das entspricht bei einer Variablen als Aktualparameter einem call-by-reference. Bei komplizierteren Ausdrücken als Aktualparameter wird dieser in eine parameterlose Prozedur (*thunk*) umgewandelt und jedesmal aufgerufen, wenn der Formalparameter benötigt wird.

Beispiel

```
procedure sub (var a, b, c, d: integer);
    begin
        b := a + a;
        d := a + c;
    end;
```

wird aufgerufen in:

```
program parameter-test;
    var  x, y, z: integer;
    begin
        x := 1;
        y := 2;
        z := 7;
        sub (x, x, x+y, z);
        print z;
    end
```

Ergebnis

Übergabemethode	Ausgabewert
call-by-value	7
call-by-reference	5
copy-restore	4
call-by-name	6

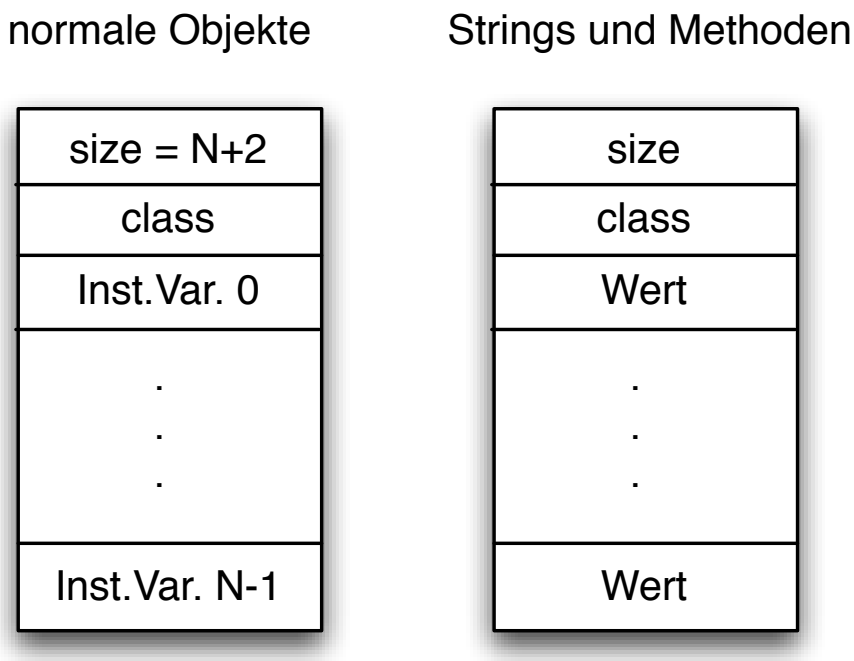
Speicherverwaltung in objektorientierten Systemen

Problem: Die Lebensdauer von Objekten ist nicht an das Eintreten bzw. Verlassen eines Blocks gekoppelt.

Als Beispiel soll hier die Laufzeitumgebung für Smalltalk-80 beschrieben werden, die charakteristisch für ein auf einer virtuellen Maschine ablaufendes objektorientiertes System ist. Fast alle im Smalltalk-System existierenden Objekte sind im Heap gespeichert.

Aufbau von Objekten

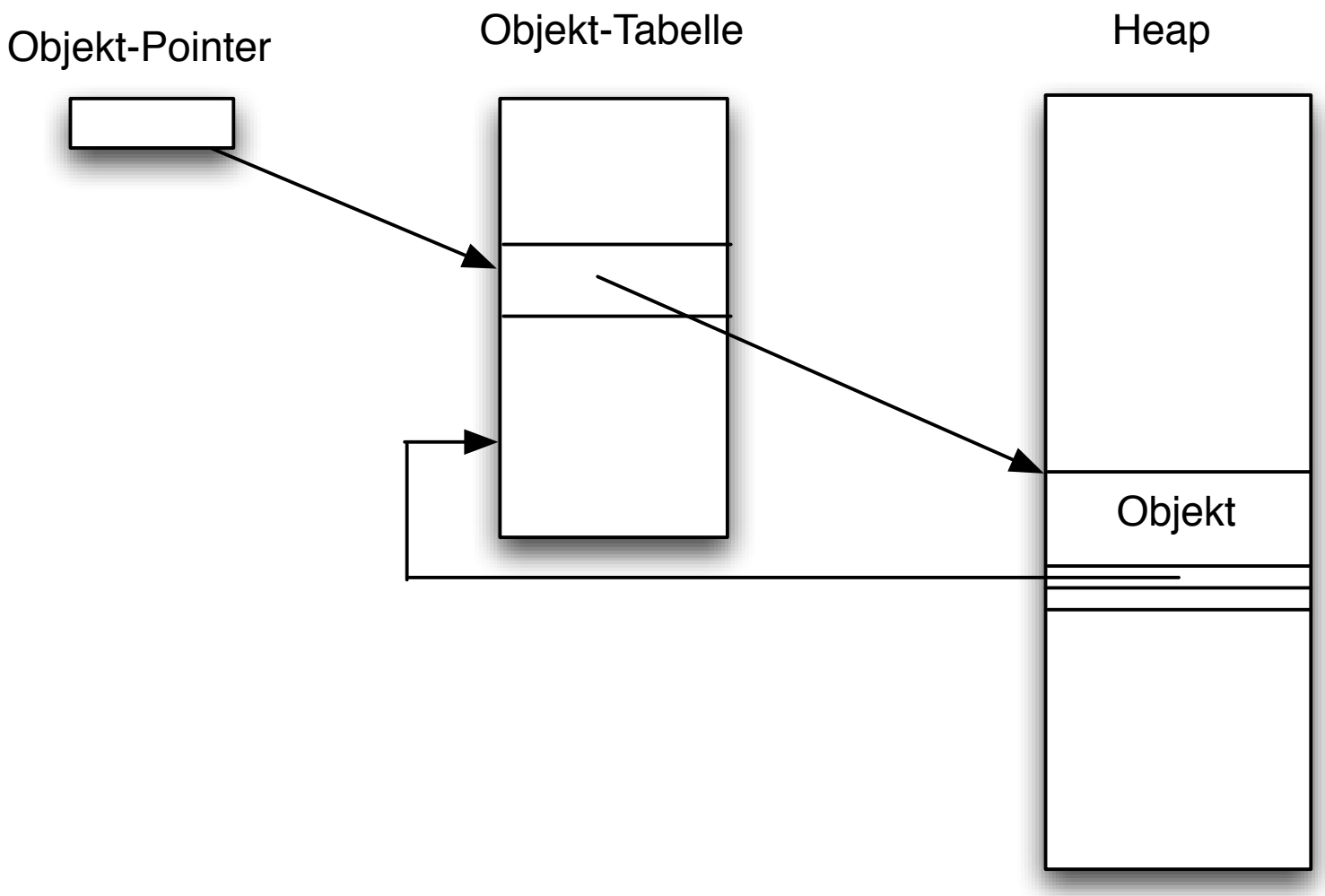
Der Standardaufbau eines Objekts im Heap ist der folgende:



Zeichenketten (Strings), Integerobjekte und übersetzte Methoden werden abweichend dargestellt.

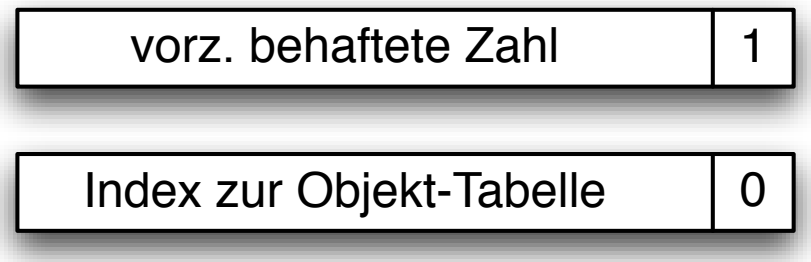
Die Objekt-Tabelle

Die im Heap befindlichen Objekte werden nicht direkt, sondern über eine Objekt-Tabelle indirekt adressiert. Für jedes im Heap befindliche Objekt gibt es einen Eintrag in der Objekt-Tabelle und nur dort ist, neben anderen Informationen, die Adresse verzeichnet, ab der sich das Objekt im Speicher befindet. Eine Referenz auf dieses Objekt (ein *Objekt-Pointer*), etwa in einer Variablen oder einer Instanzenvariablen, ist nur ein Index in die Objekt-Tabelle.


Objekt-Pointer
Objekt-Tabelle
Heap
Objekt

Darstellung von Integer-Objekten

Ist das unterste Bit eines Objekt-Pointers eine 1, so werden die restlichen Bits als vorzeichenbehaftete Zahl, also als ein Objekt der Klasse Integer, interpretiert.
Ist das unterste Bit eines Objekt-Pointers dagegen eine 0, so werden die restlichen Bits als Index in die Objekt-Tabelle betrachtet



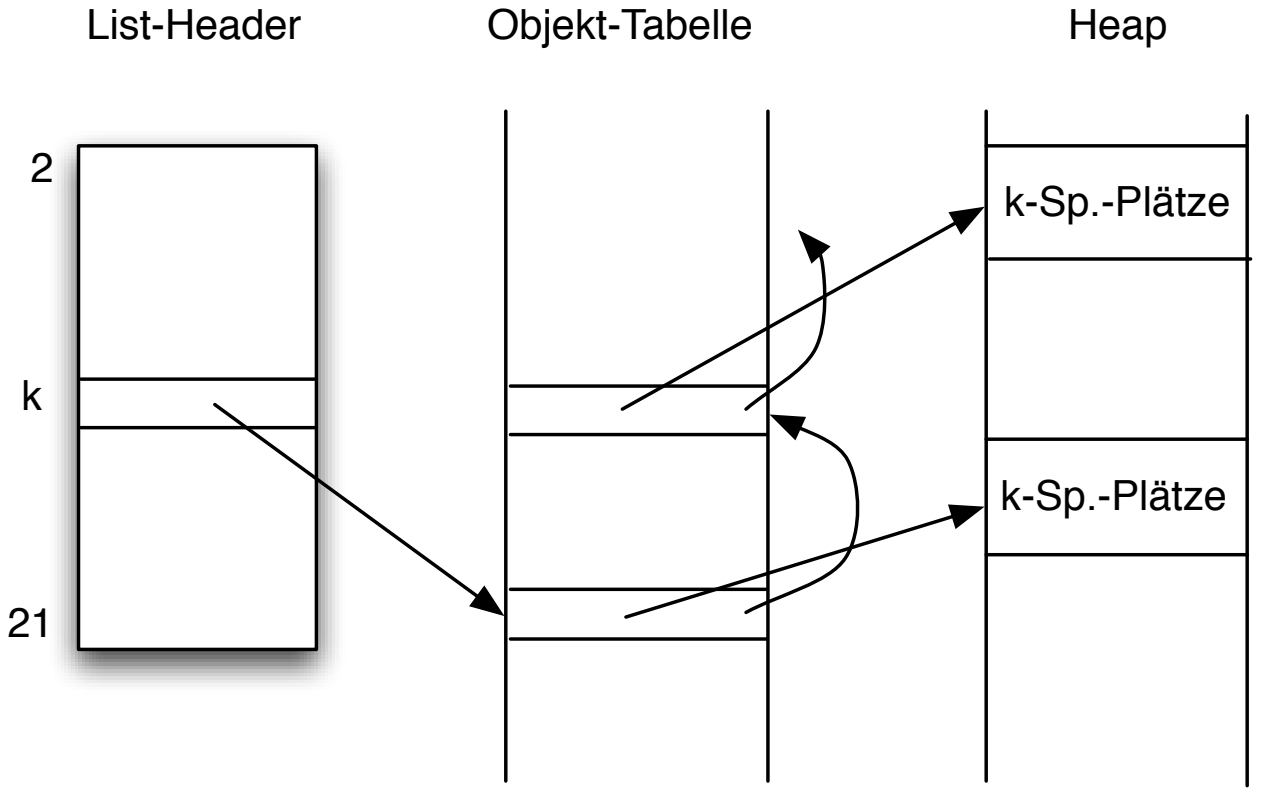
Einträge in der Objekt-Tabelle

Einträge bestehen aus zwei Speicherplätzen, die Verwaltungsinformationen und den Ort enthalten, wo sich das Objekt im Heap befindet.
Freie Einträge in der Objekt-Tabelle werden durch das F-Bit markiert. Diese Einträge sind verkettet und über eine globale Variable erreichbar.

- Ref-Count - Reference Count
- F-Flag - freier Eintrag in Objekt-Tabelle
- P-Flag und O-Flag für speziell dargestellte Objekte (String und Methoden)

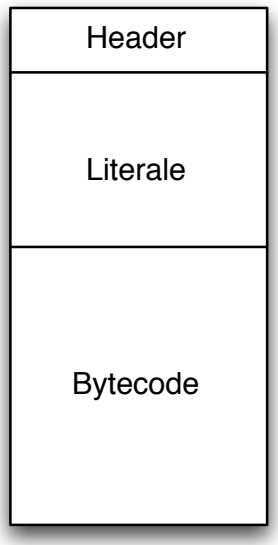
Verwaltung des freien Speichers

Verkettung freier Speicherbereiche mit jeweils k (≤ 20) Speicherplätzen über einen List-Header k, der auf eine verkettete Liste von Einträgen in der Objekt-Tabelle verweist, die jeweils auf frei Speicherplätze der Größe k im Heap verweisen.
Alle größeren freien Speicherbereiche (im Original > 20 Worte) sind ebenfalls auf diese Weise verkettet.



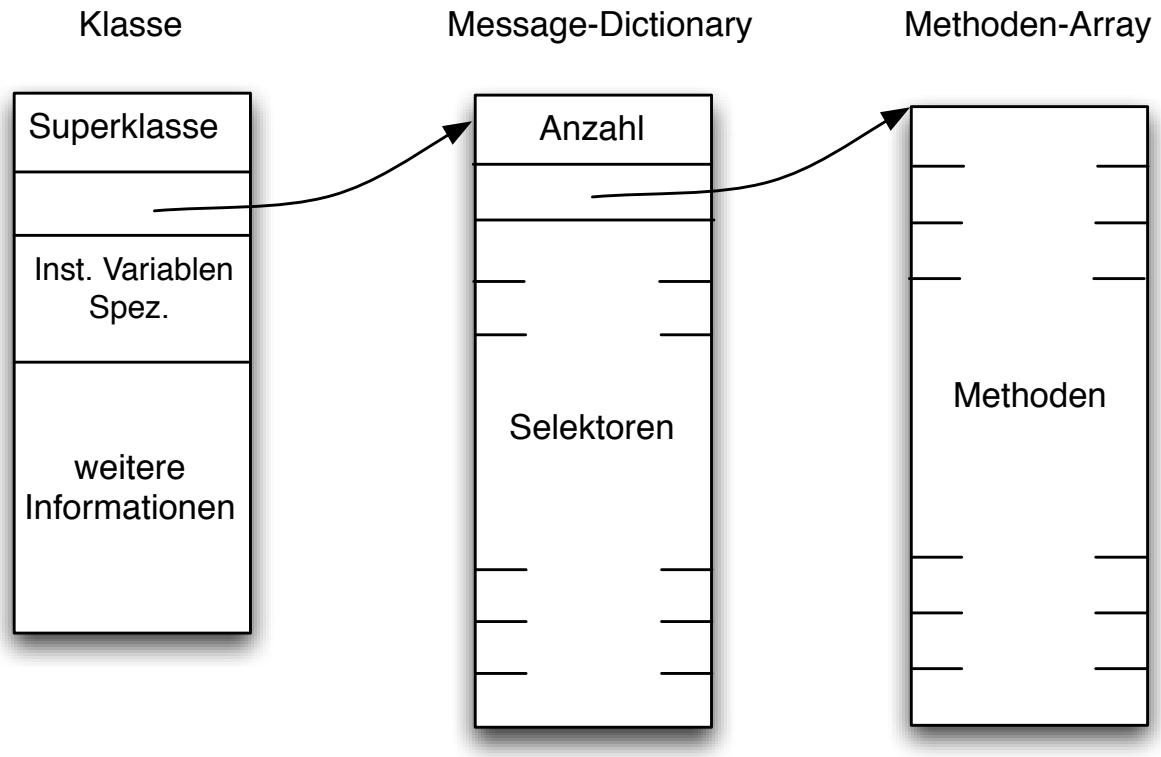
Übersetzte Methoden

Übersetzte Methoden sind natürlich ebenfalls Objekte und befinden sich daher im Heap. Allerdings werden sie etwas anders gespeichert. Der Header enthält Informationen über die Anzahl der Argumente und die Anzahl der Literale der Methode. Ferner ist die Zahl der benötigten temporären Werte inklusive der Argumente vermerkt.

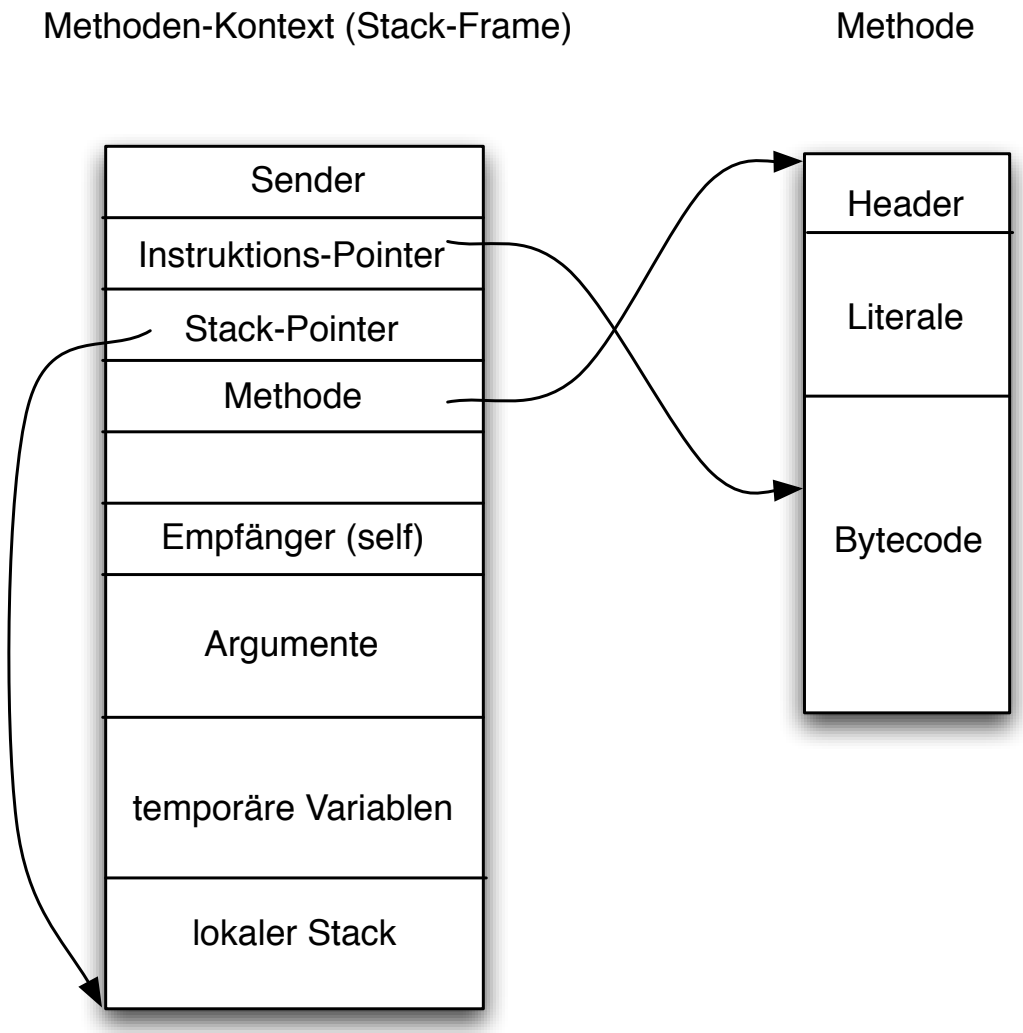


Klassenobjekte

Da Klassen ebenfalls Objekte sind und die virtuelle Maschine auf die Methoden zugreifen muss, werden die Methoden in einem „Dictionary“ gespeichert. Der Zugriff erfolgt mittels einer Hash-Funktion über den Selektor der Nachricht.



Stack-Frame


Methoden-Kontext (Stack-Frame)
Methode
Sender
Instruktions-Pointer
Stack-Pointer
Methode
Empfänger (self)
Argumente
temporäre Variablen
lokaler Stack
Header
Literale
Bytecode