

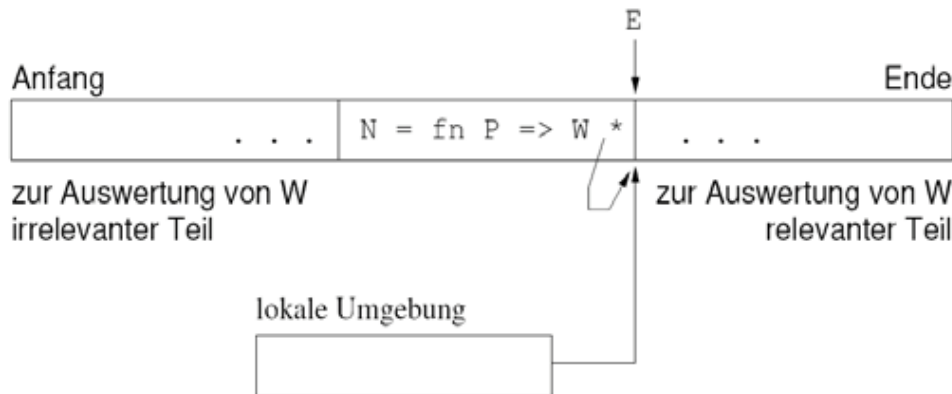
3. Speicherverwaltung

- 3.1 Umgebung und Speicherstrukturen
- 3.2 Speicherverwaltung in Java
- 3.3 Zusammenfassung

3. Speicherverwaltung

- 3.1 Umgebung und Speicherstrukturen
- 3.2 Speicherverwaltung in Java
- 3.3 Zusammenfassung

- Die Bindung von Variablennamen und ihren Werten muß verwaltet werden.
- Einfaches Modell aus Info I: Umgebung als Liste, lokale Umgebungen für Methoden-Aufrufe



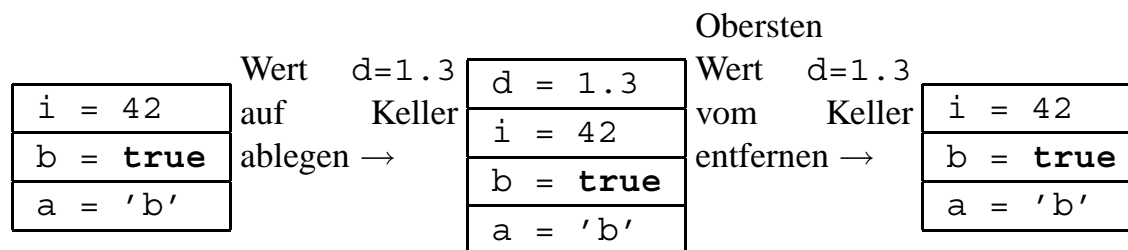
- In diesem einfachen Modell mußte nichts aus der Umgebung entfernt werden, lokale Deklarationen wurden als (geschachtelte) lokale Umgebungen dargestellt.

Abbildung von Schachtelungen in ein Umgebungsmodell

- Im imperativen Paradigma sind Blöcke und ihre Schachtelung das wichtigste Strukturierungselement. (Klassen, Methoden, Kontrollstrukturen bilden ja stets auch einen Block.)
- Ein Block in einem Block führt Namensbindungen ein, die zusätzlich zu den Bindungen des äußeren Blocks gelten.
- Nach Verlassen des inneren Blocks gelten wieder nur noch jene Bindungen, die im äußeren Block bereits vor Betreten des inneren Blockes galten.
- Die Modellierung der Umgebung sollte also dieser Anforderung Rechnung tragen: Die Bindungen, die zuletzt hinzukamen, werden als erste wieder entfernt.
- Prinzip: LIFO (last-in-first-out)

Der Keller (*stack*) verwirklicht als Datenstruktur das LIFO-Prinzip:

- Was man ablegt, legt man wie auf einem Stapel obendrauf.
- Entfernen kann man nur, was auf dem Stapel zuoberst liegt.
- Nach dem Entfernen des obersten Items (Eintrag, Wert) liegen genau die Items vor, die vor dem Ablegen des eben entfernten Items vorlagen.
- Man kann allerdings tiefer liegende Einträge ablesen (und Werte verändern).



Um Werte in einem Keller (Stapel) verwalten zu können, nimmt man an, dass die Werte alle gleichviel Speicherplatz benötigen. Dinge unterschiedlicher Größe kann man ja auch nicht gut stapeln.

Oft muß man Werte sehr unterschiedlicher Größe verwalten. Hierfür braucht man eine dynamische Speicherplatzverwaltung (*dynamic storage allocation*), durch die Werte unterschiedlicher Größe in einem gemeinsamen Speicherbereich abgelegt werden können.

Ein Pool verfügbaren Speicherplatzes für dynamischen Zugriff heißt Halde, engl. *heap*.

Achtung:

Mit *heap* bezeichnet man im Zusammenhang mit effizienten Algorithmen auch eine Prioritätsliste (*priority queue*). Bitte nicht verwechseln!

3. Speicherverwaltung

3.1 Umgebung und Speicherstrukturen

3.2 Speicherverwaltung in Java

3.3 Zusammenfassung

Wir wollen im folgenden ein wenig die interne Speicherverwaltung anschauen. Wir bleiben hier aber informell und vereinfachend und betrachten die Zusammenhänge nur, soweit wir sie benötigen, um Phänomene zu verstehen, die uns auf der Ebene der Programmierung beschäftigen können.

Ein tieferes Verständnis der hier behandelten Zusammenhänge kann in anderen Vorlesungen erworben werden.

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {  
2  int a = 5;  
3  int b = 0;  
4  for(int i = 0; i < a; i++)  
5  {  
6      int c = a-i;  
7      b += c;  
8  }  
9  boolean d = b == a*(a+1) / 2;  
10 }
```

Kellerzustand nach Zeile 2

a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 3

b = 0
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 4

i = 0
b = 0
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 6

c = 5
i = 0
b = 0
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 7

c = 5
i = 0
b = 5
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```

1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 4'

i = 1
b = 5
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```

1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 6'

c = 4
i = 1
b = 5
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 7'

c = 4
i = 1
b = 9
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 4''

i = 2
b = 9
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1 {
2   int a = 5;
3   int b = 0;
4   for(int i = 0; i < a; i++)
5   {
6     int c = a-i;
7     b += c;
8   }
9   boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 6''

c = 3
i = 2
b = 9
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1 {
2   int a = 5;
3   int b = 0;
4   for(int i = 0; i < a; i++)
5   {
6     int c = a-i;
7     b += c;
8   }
9   boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 7''

c = 3
i = 2
b = 12
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1 {
2   int a = 5;
3   int b = 0;
4   for(int i = 0; i < a; i++)
5   {
6     int c = a-i;
7     b += c;
8   }
9   boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 4'''

i = 3
b = 12
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1 {
2   int a = 5;
3   int b = 0;
4   for(int i = 0; i < a; i++)
5   {
6     int c = a-i;
7     b += c;
8   }
9   boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 6'''

c = 2
i = 3
b = 12
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```

1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 7'''

c = 2
i = 3
b = 14
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```

1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 4''''

i = 4
b = 14
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1 {
2   int a = 5;
3   int b = 0;
4   for(int i = 0; i < a; i++)
5   {
6     int c = a-i;
7     b += c;
8   }
9   boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 6''''

c = 1
i = 4
b = 14
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```
1 {
2   int a = 5;
3   int b = 0;
4   for(int i = 0; i < a; i++)
5   {
6     int c = a-i;
7     b += c;
8   }
9   boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 7''''

c = 1
i = 4
b = 15
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```

1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 4''''''

i = 5
b = 15
a = 5

Die lokalen Variablen in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

```

1  {
2  int a = 5;
3  int b = 0;
4  for(int i = 0; i < a; i++)
5  {
6      int c = a-i;
7      b += c;
8  }
9  boolean d = b == a*(a+1) / 2;
10 }
```

Kellerzustand nach Zeile 9

d = true
b = 15
a = 5

Die globalen Variablen sind (zunächst) in jedem Block sichtbar, müssen also auch im Keller gesondert behandelt werden.

Es gibt daher für die globalen Variablen einen eigenen Bereich, auf den von anderen Bereichen zugegriffen werden kann: den *Constant Pool*.

- Die lokalen Variablen in einem Block sind in einer anderen Methode nicht sichtbar, auch wenn diese andere Methode im gleichen Block *aufgerufen* wird.
- Deshalb werden die lokalen Variablen innerhalb des Kellers in sog. Frames angeordnet.
- Ein Frame wird bei einem Methodenaufruf erzeugt und beinhaltet
 - die lokalen Variablen einer Methode,
 - die übergebenen (aktuellen) Parameter,
 - den Rückgabewert (und evtl. Ausnahmen),
 - einen *Operand Stack* für die Methodenausführung,
 - einen Verweis auf den Constant Pool.

- Der Keller sieht für jeden Eintrag nur begrenzt viel Speicherplatz vor.
- Die Werte von primitiven Typen können direkt im Keller abgelegt werden.
- Andere Typen, die Objekte (wie z.B. Strings und Arrays), können sehr viel mehr Speicherplatz in Anspruch nehmen.
- Objekte werden daher in der Halde (Heap) gespeichert, im Keller wird (als Wert des entsprechenden Bezeichners) eine Referenz (die Speicheradresse im Heap) abgelegt.

```

char a = 'b';
String gruss1 = "Hi";
String gruss2 = "Hallo";
String[] gruesse = {gruss1, gruss2};
int[] zahlen = {1, 2, 3};
boolean b = true;
int i = 42;

```

Stack:

i = 42
b = true
zahlen = <adr4>
gruesse = <adr3>
gruss2 = <adr2>
gruss1 = <adr1>
a = 'b'

Heap:

```

<adr1>: "Hi" <adr2>: "Hallo" <adr3>
: { <adr1>, <adr2> } <adr4>: {1, 2, 3 }

```



```

1  public class Exchange
2  {
3      public static void swap(int i, int j)
4      {
5          int c = i;
6          i = j;
7          j = c;
8      }
9
10     public static void main(String[] args)
11     {
12         int x = 1;
13         int y = 2;
14         swap(x,y);
15     }
16 }

```

main-Frame nach Zeile 13

y = 2
x = 1
args = <adr1>

```

1  public class Exchange
2  {
3      public static void swap(int i, int j)
4      {
5          int c = i;
6          i = j;
7          j = c;
8      }
9
10     public static void main(String[] args)
11     {
12         int x = 1;
13         int y = 2;
14         swap(x,y);
15     }
16 }

```

main-Frame nach Zeile 13

y = 2
x = 1
args = <adr1>

swap-Frame nach Zeile 3

j = 2
i = 1

```
1 public class Exchange
2 {
3     public static void swap(int i, int j)
4     {
5         int c = i;
6         i = j;
7         j = c;
8     }
9
10    public static void main(String[] args)
11    {
12        int x = 1;
13        int y = 2;
14        swap(x, y);
15    }
16 }
```

main-Frame nach Zeile 13

y = 2
x = 1
args = <adr1>

swap-Frame nach Zeile 5

c = 1
j = 2
i = 1

```
1 public class Exchange
2 {
3     public static void swap(int i, int j)
4     {
5         int c = i;
6         i = j;
7         j = c;
8     }
9
10    public static void main(String[] args)
11    {
12        int x = 1;
13        int y = 2;
14        swap(x, y);
15    }
16 }
```

main-Frame nach Zeile 13

y = 2
x = 1
args = <adr1>

swap-Frame nach Zeile 6

c = 1
j = 2
i = 2

```

1  public class Exchange
2  {
3      public static void swap(int i, int j)
4      {
5          int c = i;
6          i = j;
7          j = c;
8      }
9
10     public static void main(String[] args)
11     {
12         int x = 1;
13         int y = 2;
14         swap(x,y);
15     }
16 }

```

main-Frame nach Zeile 13

y = 2
x = 1
args = <adr1>

swap-Frame nach Zeile 7

c = 1
j = 1
i = 2

```

1  public class Exchange
2  {
3      public static void swap(int i, int j)
4      {
5          int c = i;
6          i = j;
7          j = c;
8      }
9
10     public static void main(String[] args)
11     {
12         int x = 1;
13         int y = 2;
14         swap(x,y);
15     }
16 }

```

main-Frame nach Zeile 14

y = 2
x = 1
args = <adr1>

```

1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3      zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8      int[] werte = {0, 1, 2};
9      changeValues(werte, 1, 3);
10 }

```

main-Frame nach Zeile 6

args = <adr1>

Heap nach Zeile 6

<adr1>: { }

```

1  public static void changeValues(int[] zahlen, int index, int wert)
2  {
3      zahlen[index] = wert;
4  }
5
6  public static void main(String[] args)
7  {
8      int[] werte = {0, 1, 2};
9      changeValues(werte, 1, 3);
10 }

```

main-Frame nach Zeile 8

werte = <adr2>

args = <adr1>

Heap nach Zeile 8

<adr1>: { } <adr2>: { 0, 1, 2 }

```

1 public static void changeValues(int[] zahlen, int index, int wert)
2 {
3     zahlen[index] = wert;
4 }
5
6 public static void main(String[] args)
7 {
8     int[] werte = {0, 1, 2};
9     changeValues(werte, 1, 3);
10 }

```

main-Frame nach Zeile 8

werte = <adr2>
args = <adr1>

Heap nach Zeile 8

<adr1> : { } <adr2> : { 0, 1, 2 }

changeValues-Frame nach
Zeile 1

wert = 3
index = 1
zahlen = <adr2>

```

1 public static void changeValues(int[] zahlen, int index, int wert)
2 {
3     zahlen[index] = wert;
4 }
5
6 public static void main(String[] args)
7 {
8     int[] werte = {0, 1, 2};
9     changeValues(werte, 1, 3);
10 }

```

main-Frame nach Zeile 8

werte = <adr2>
args = <adr1>

Heap nach Zeile 3

<adr1> : { } <adr2> : { 0, 3, 2 }

changeValues-Frame nach
Zeile 3

wert = 3
index = 1
zahlen = <adr2>

```
1 public static void changeValues(int[] zahlen, int index, int wert)
2 {
3     zahlen[index] = wert;
4 }
5
6 public static void main(String[] args)
7 {
8     int[] werte = {0, 1, 2};
9     changeValues(werte, 1, 3);
10 }
```

main-Frame nach Zeile 9

werte = <adr2>

args = <adr1>

Heap nach Zeile 9

<adr1> : { } <adr2> : { 0, 3, 2 }

Keller: Aufräumen unnötig

Die Keller werden schon von ihrer Struktur her automatisch aufgeräumt, d.h. es gibt in Kellern keine Speicherplätze, die belegt sind, obwohl die Lebensdauer des entsprechenden Namens abgelaufen ist.

Halde: Aufräumen nötig

Für die Halde gilt das nicht: Hier wird im Laufe eines Programmes Speicherplatz zugewiesen und belegt, aber nicht automatisch freigegeben, falls die Lebensdauer eines Namens, der für den gespeicherten Wert steht, abgelaufen ist.

In vielen Sprachen muß der Programmierer dafür sorgen, dass Speicherplatz, der nicht mehr gebraucht wird, freigegeben wird (explizite Speicherplatzfreigabe – Gefahr des Speicherlecks).

In vielen modernen Sprachen (auch Java) gibt es eine automatische Speicherplatzfreigabe (Garbage Collection). Der Heap wird immer wieder durchsucht nach Adressen, auf die nicht mehr zugegriffen werden kann (da kein Name diese Adresse als Wert hat).

Problem hier: der Programmierer kann nicht oder nur sehr eingeschränkt kontrollieren, wann dieser Reinigungsprozess läuft. Das ist unter Umständen (z.B. in sekundengenauen, empfindlichen Echtzeitsystemen) nicht akzeptabel.

3. Speicherverwaltung

3.1 Umgebung und Speicherstrukturen

3.2 Speicherverwaltung in Java

3.3 Zusammenfassung

Sie kennen jetzt:

- Keller und Halde als Datenstrukturen,
- die Verwendung von Keller und Halde zur Speicherung aktueller Werte in Java,
- die Auswirkungen des call-by-value Prinzips bei primitiven Typen und bei Speicheradressen als Werten,
- das Prinzip der automatischen Speicherplatzbereinigung.