LMU LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT INSTITUTE FOR INFORMATICS
DATABASE SYSTEMS GROUP

# Kapitel 11 (eigentlich 10)

# Speicherverwaltung

Skript zur Vorlesung
Einführung in die Programmierung
im Wintersemester 2012/13
Ludwig-Maximilians-Universität München


# Übersicht

# Übersicht

# 11. Speicherverwaltung



- In diesem Kapitel wollen wir kurz die interne Speicherverwaltung von Java (genauer gesagt der JVM) betrachten.
- 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.
- Wir werden feststellen, dass diese Aspekte leider tatsächlich einige Effekte haben, die wir bei bei der Programmentwicklung berücksichtigt müssen
- Ein tieferes Verständnis der hier behandelten Zusammenhänge kann in anderen Vorlesungen erworben werden.

# 11.1 Umgebungsmodell



- Wir haben gesehen, dass Programme in einer imperativen Sprache wie Java durch Anweisungen charakterisiert sind, die Zustandsübergänge definieren.
- Ein Zustand ist dabei durch die Bindungen (Substitution) von den aktuellen Variablen an Werte definiert (dadurch bekommen insbesondere Ausdrücke in Anweisungen eine bestimmte Bedeutung)
- Diese Menge von Bindungen muss in einer geeigneten Datenstruktur gespeichert werden.
- Anforderungen an diese Datenstruktur werden durch ein *Umgebungsmodell* beschrieben.

# 11.1 Umgebungsmodell



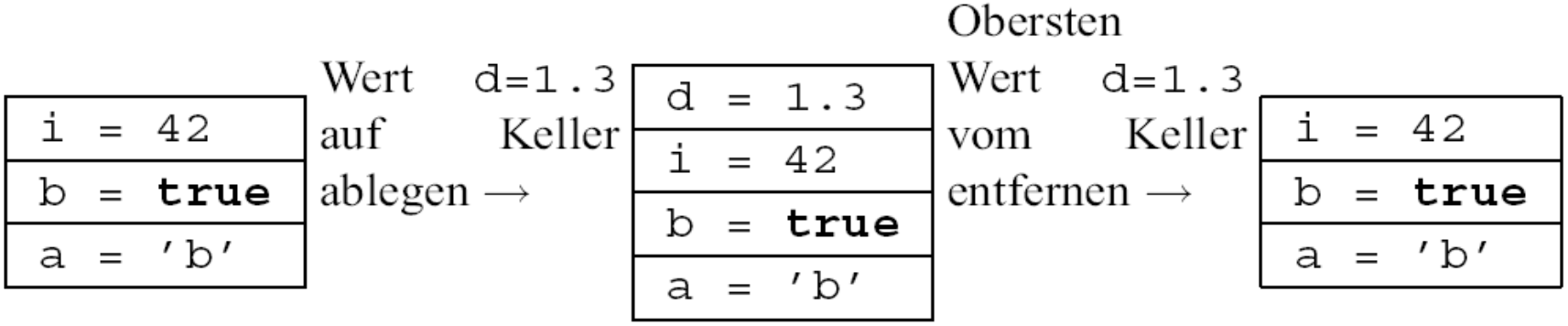
- Was sollte das Umgebungsmodell leisten, d.h. was muss diese Datenstruktur berücksichtigen?
  - Im imperativen Paradigma sind Blöcke und ihre Schachtelung das wichtigste Strukturierungselement. (Klassen, Methoden, Kontrollstrukturen bilden ja stets auch einen Block.)
  - Ein Block in einem Block führt 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 diese Anforderung erfüllen:
    1. Neue Bindungen werden oft eingefügt und alte Bindungen werden oft wieder gelöscht, d.h. die Datenstruktur muss dynamisch sein
    2. Die Bindungen, die zuletzt hinzukamen, werden als erste wieder entfernt. Dieses Zugriffs-Prinzip nennt man *LIFO* (*last-in-first-out*)
- Eine entsprechende Datenstruktur heißt *Keller* (auch *Stapel*, *Stack*), Spezifika lernen wir im Kapitel über Datenstrukturen kennen

# 11.1 Umgebungsmodell



Wichtig ist momentan nur, dass Keller als Datenstruktur das LIFO-Prinzip verwirklichen und dynamisch sind:

- 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 Eintrags liegen genau die Einträge vor, die vor dem Ablegen des eben entfernten Eintrags vorlagen.
- Im Kontext der imperativen Speicherverwaltung verwendet man meist eine Spezialform des Kellers, bei der man tiefer liegende Einträge ablesen (und verändern aber nicht löschen) kann.



# 11.1 Umgebungsmodell



- Um Werte in einem Keller verwalten zu können, nimmt man an, dass die Werte alle gleichviel Speicherplatz benötigen (Dinge unterschiedlicher Größe kann man ja auch nicht gut stapeln); dies ist vereinfachend für primitive Datentypen ausreichend
- Oft muss man Werte sehr unterschiedlicher Größe verwalten (insbesondere bei Objekttypen)
- Hierfür braucht man eine dynamische Speicherplatzverwaltung (*dynamic storage allocation*), durch die Werte unterschiedlicher Größe in einem gemeinsamen Speicherbereich abgelegt werden können
- Ein solcher Pool verfügbaren Speicherplatzes für dynamischen Zugriff heißt *Halde*, (*Heap*)
  **Achtung**:
  Mit „Heap“ bezeichnet man im Zusammenhang mit effizienten Algorithmen auch eine Prioritätsliste (*priority queue*). Bitte nicht verwechseln!
- Daten werden also im Keller und auf der Halde verwaltet

# Übersicht

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

**Kellerzustand nach Zeile 2**

```
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  }
```

a = 5

# 11.2 Speicherverwaltung

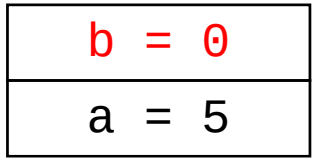
## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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**



# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

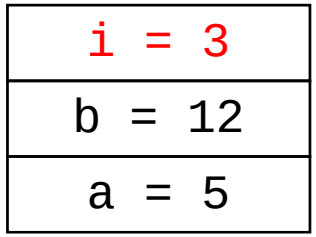
## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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'''**



# 11.2 Speicherverwaltung

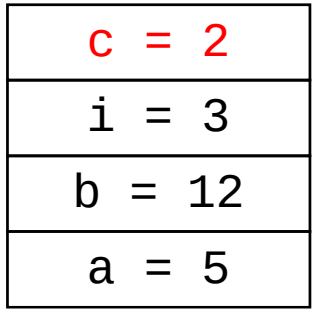
## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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'''**



# 11.2 Speicherverwaltung

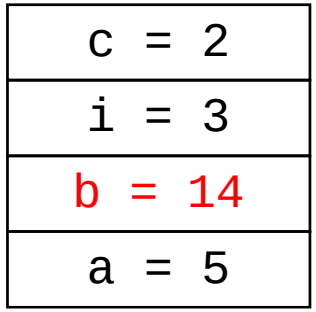
## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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'''**



# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

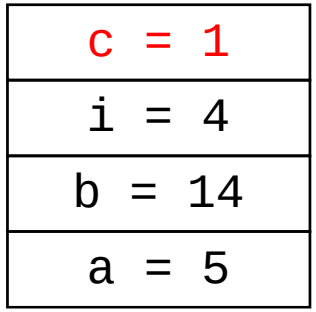
## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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''''**



# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ in einem Block oder in ineinander geschachtelten Blöcken werden in Java in einem Keller gespeichert.

**Kellerzustand nach Zeile 4'''''**

```
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  }
```

| i = 5 |
|---|
| b = 15 |
| a = 5 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 8**

| b = 15 |
|---|
| a = 5 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die lokalen Variablen mit primitivem Typ 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 |

# 11.2 Speicherverwaltung

## 11.2.1 Verwaltung primitiver Typen



- Die globalen Variablen (mit primitivem Typ) 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 alle Informationen, die für die Abarbeitung der Methode notwendig sind, u.a.
  - die lokalen Variablen einer Methode,
  - die übergebenen (aktuellen) Parameter,
  - ...

# 11.2 Speicherverwaltung

## 11.2.2 Verwaltung von Referenztypen



Keller vs. Heap:

- 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.
- Strings, Arrays und Objekte werden bei der Speicherverwaltung daher gleich behandelt:
  - sie werden in der Halde (Heap) gespeichert
  - im Keller wird (als Wert des entsprechenden Bezeichners; „auf dem Zettel“) eine *Referenz* (die Speicheradresse im Heap) abgelegt (also ein Array von Objekten ist intern ein Array von Referenzen)
- Strings, Arrays und Objekte heißen daher auch *Referenztypen*

- Beispiel: Keller vs. Heap

```
char a = 'b';
String gruss1 = "Hi";
String gruss2 = "Hello";
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 }

# 11.2 Speicherverwaltung

## 11.2.2 Verwaltung von Referenztypen



- Während primitive Typen lediglich deklariert werden, reicht dies bei Referenztypen nicht aus, sie müssen mit Hilfe des **new**-Operators oder - im Falle von Arrays und Strings - durch Zuweisung von Literalen zusätzlich noch explizit erzeugt werden.
- Was passiert eigentlich, wenn eine Variable angelegt (vereinbart) aber nicht initialisiert wird?
- Bei primitiven Typen hatten wir die Intuition eines leeren Zettels: es wird eine Speicherzelle angelegt mit leerem Inhalt, d.h. es wird der "leere Wert" (ω) auf den Keller gelegt
- Bei Referenztypen passiert im Prinzip das Gleiche: im Keller steht *keine* Speicheradresse, ein sog. *null*-Pointer, der in Java durch die Zeichenkette **`null`** representiert wird
- **`null`** wird oft auch als Literal bezeichnet (und kann als solches z.B. in Ausdrücken verwendet werden): **`if`**`(a[0] != `**`null`**`)` …

# Übersicht

# 11.3 Besonderheiten bei Referenztypen



- Das Verständnis für Referenztypen (Strings, Arrays, Objekte) ist entscheidend für die Programmierung in Java
- Referenztypen können prinzipiell genauso benutzt werden wie primitive Typen, da sie jedoch lediglich einen Verweis darstellen, ist die Semantik einiger Operatoren anders als bei primitiven Typen!!!
- Ein Beispiel hatten wir schon kennengelernt: Array-Konstanten

```
final char[] ABC = {'a', 'b', 'c'};
final char[] DE = {'d', 'e'};
ABC = DE; // ungueltige Anweisung: Compilerfehler
```

  deren einzelne Array-Komponenten aber veränderbar sind. *Warum*?
- Drei weitere, sehr wichtige Beispiele, die wir im folgenden genauer betrachten:
  - *Gleichheit* von Objekten
  - *Kopieren* von Objekten
  - *Call-by-reference Effekt* bei Methodenaufruf mit Referenztypen

# 11.3 Besonderheiten bei Referenztypen

## 11.3.1 Gleichheit



- Zur Erinerung: Was bedeutet *Gleichheit* von Objekten?
- Intuitiv: Ihre Attribute haben dieselben Werte, d.h. sie haben *denselben Zustand*.

```
Einfamilienhaus villaKahn;
villaKahn = new Einfamilienhaus();
Einfamilienhaus hausBender;
hausBender = new Einfamilienhaus();

boolean vergleich = villaKahn == hausBender; // (*)

villaKahn = hausBender;

vergleich = villaKahn == hausBender; // (**)
```

- ***Was ist der Wert der Variablen `vergleich` an der Stelle (*)?***
  - Der Wert der Variablen `vergleich` an der Stelle (*) ist **`false`**!

# 11.3 Besonderheiten bei Referenztypen

## 11.3.1 Gleichheit



- Wie wir bereits gesehen haben: Es gibt zwei "Arten" von Gleichheit von Objekten bzw. Objektvariablen (Variablen mit Objekttyp):
- *Gleichheit*: Der Zustand der entsprechenden Objekte beider Objektvariablen ist gleich.
- *Identität*: Beide Objektvariablen verweisen auf die gleiche Speicheradresse.
- Der Operator == prüft den zweiten Fall (Identität). Er kann also nicht dazu benutzt werden, abzufragen, ob die Objekte zweier Objektvariablen gleich bzgl. ihres Zustands sind.
- Dies wird oft übersehen und ist daher eine häufige Fehlerquelle!

# 11.3 Besonderheiten bei Referenztypen

## 11.3.1 Gleichheit



- Um die Gleichheit zweier Objekte zu testen stellt die Klasse `Object` die Methode `boolean equals(Object obj)` zur Verfügung.
- Die Implementierung in der Klasse `Object` setzt zunächst die Identität um, d.h. testet auf die Gleichheit der Referenzen
- Da jede Klasse implizit Unterklasse von `Object` ist, steht diese Methode für alle Objekte in Java zur Verfügung
- Um die Gleichheit zu testen, muss diese Methode entsprechend überschrieben werden

# 11.3 Besonderheiten bei Referenztypen

## 11.3.1 Gleichheit



- Die Methode `equals` in der Klasse `Object` ist so spezifiziert, dass sie eine Äquivalenz-Relation auf nicht-**`null`** Objektreferenzen mit folgenden Eigenschaften implementiert:
  - Reflexivität: Für jede nicht-**`null`** Referenz `x` gilt: `x.equals(x)`.
  - Symmetrie: Für nicht-**`null`** Referenzen `x` und `y` ist `x.equals(y)` **`true`** g.d.w. `y.equals(x)` **`true`** ist.
  - Transitivität: Für nicht-**`null`** Referenzen `x`, `y` und `z` gilt: wenn `x.equals(y)` und `y.equals(z)` **`true`** sind, dann ist auch `x.equals(z)` **`true`**.
  - Konsistenz: Für nicht-**`null`** Referenzen `x` und `y` sind mehrfache Aufrufe von `x.equals(y)` entweder immer **`true`** oder immer **`false`**, solange keine Information in den Objekten `x` oder `y` verändert wurde, die von der Methode `equals` überprüft wird.
  - Außerdem gilt für eine nicht-**`null`** Referenz `x`: `x.equals(null)` ergibt **`false`**.

# 11.3 Besonderheiten bei Referenztypen

## 11.3.1 Gleichheit



- Diese Vorschrift *soll* eingehalten werden, wenn man in einer Klasse die Methode `equals` überschreibt. Das ist aber eine *semantische* Vorschrift, die nicht vom Compiler überprüft, sondern nur vom Programmierer bewiesen werden kann.
- Die Implementierung der Methode `equals` in der Klasse `Object` (nämlich durch den Operator ==) selbst ist die schärfste Möglichkeit, diese Vorschrift umzusetzen.
- Beispiel:

  Die Klasse `String` überschreibt `equals` so, dass für einen String `s` und ein Objekt `o` gilt: `s.equals(o)` g.d.w.:
  - `o` ist nicht **`null`**
  - `o` ist vom Typ `String` und
  - `o` repräsentiert genau die gleiche Zeichenkette wie `s` (d.h. `s` und `o` haben gleiche Länge und an jeder Stelle steht der gleiche Character.

# 11.3 Besonderheiten bei Referenztypen

## 11.3.2 Kopieren



- Eine Zuweisung mit einem Referenztyp erzeugt eine *Kopie der Referenz* (und *kein* neues Objekt), man spricht auch von *Aliasing*
- Beispiel: Klasse `Auto` (Kapitel 6.3, Folie 48) mit Standardkonstruktor `Auto()`

```
1       Auto golf1 = new Auto();
2       Auto golf2 = golf1;
3       golf1.setErstzulassung(2003);
4       golf2.setErstzulassung(2007);
```

- Was ist passiert?
  - Nach der Zuweisung (Zeile 2) verweisen beide Variablen `golf1` und `golf2` auf dasselbe `Auto`-Objekt (nur der Zettel mit dem Verweis wurde kopiert, aber nicht das Objekt selber)
  - Diese Kopie wird auch *flache* Kopie (*shallow copy*) genannt
  - Problem: es ist nicht sichergestellt, dass die Kopie unabhängig vom ursprünglichen Objekt ist!

# 11.3 Besonderheiten bei Referenztypen

## 11.3.2 Kopieren



- Zur Erinnerung: `clone` ist eine Methode, die von `Object` implizit auf alle Klassen vererbt wird und eine *Kopie* des aktuellen Objekts erzeugt.
- Die ursprüngliche Fassung von `clone` erzeugt ebenfalls eine *flache* Kopie
- Die Java-API stellt das *Marker-Interface*[1] `Cloneable` zur Verfügung, das für die Methode `clone` (die jede Klasse von der impliziten Vaterklasse `Object` erbt) eine spezielle Eigenschaft spezifiziert.
- Implementiert eine Klasse das Interface `Cloneable`, so garantiert der Implementierer, dass die Methode `clone` eine sog. *tiefe* Kopie (*deep copy*) erzeugt: Für alle Attribute mit Objekttypen müssen Kopien der entsprechenden Objekte angelegt werden.
- Die Methode `clone` muss dazu entsprechend überschrieben werden.
- Achtung: Beim Erstellen einer tiefen Kopie muss man darauf achten, dass die Objekte eines Attributs mit Objekttyp selbst wieder Attribute mit Objekttypen haben können.

[1] Was ein Marker-Interface ist, sehen wir auf der nächsten Folie

# 11.3 Besonderheiten bei Referenztypen

## 11.3.2 Kopieren



- Interfaces, die weder Methoden noch Konstanten definieren (also einen leeren Rumpf haben), werden *Marker-Interfaces* genannt.
- Marker-Interfaces sind dazu gedacht, gewisse (teilweise abstrakte) Eigenschaften von Objekten sicher zu stellen, die typischerweise im Kommentar des Interfaces spezifiziert sind.
- Implementiert eine Klasse ein Marker-Interface, sollte sich der Implementierer an diese Spezifikationen halten.
- Dies wird aber nirgends automatisch (z.B. vom Compiler) abgeprüft, d.h. es liegt alleine in der Verantwortung des Programierers, diese Eigenschaften zu garantieren.

# 11.3 Besonderheiten bei Referenztypen

## 11.3.2 Kopieren



Overview Package **Class** Use Tree Deprecated Index Help

*Java™ Platform Standard Ed. 6*

PREV CLASS NEXT CLASS | FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

**java.lang**

### Interface Cloneable

**All Known Subinterfaces:**

AclEntry, Attribute, AttributedCharacterIterator, Attributes, CertPathBuilderResult, CertPathParameters, CertPathValidatorResult, CertSelector, CertStoreParameters, CharacterIterator, CRLSelector, Descriptor, GSSCredential, Name

```
public interface Cloneable
```

A class implements the `Cloneable` interface to indicate to the `Object.clone()` method that it is legal for that method to make a field-for-field copy of instances of that class.

Invoking Object's clone method on an instance that does not implement the `Cloneable` interface results in the exception `CloneNotSupportedException` being thrown.

By convention, classes that implement this interface should override `Object.clone` (which is protected) with a public method. See `Object.clone()` for details on overriding this method.

Note that this interface does *not* contain the `clone` method. Therefore, it is not possible to clone an object merely by virtue of the fact that it implements this interface. Even if the clone method is invoked reflectively, there is no guarantee that it will succeed.

**Since:**

JDK1.0

**See Also:**

`CloneNotSupportedException`, `Object.clone()`

# 11.3 Besonderheiten bei Referenztypen

## 11.3.2 Kopieren



- Beispiel

```
public class DeepCopy implements Cloneable
{
    private int zahl;
    private int[] zahlen;

    public DeepCopy(int zahl, int[] zahlen)
    {
        this.zahl = zahl;
        this.zahlen = zahlen;
    }

    public Object clone()
    {
        int neueZahl = this.zahl;
        int[] neueZahlen = new int[this.zahlen.length];
        for(int i=0; i<this.zahlen.length; i++)
        {
            neueZahlen[i] = this.zahlen[i];
        }
        DeepCopy kopie = new DeepCopy(neueZahl, neueZahlen);
        return kopie;
    }
}
```

# 11.3 Besonderheiten bei Referenztypen

## 11.3.3 Call-by-reference Effekt



- Für Objekttypen wird in den Variablen (auf dem Keller) nur die Referenz gespeichert. Dies führt zu einem call-by-reference Effekt!!!
- Zur Erinnerung: in folgendem Beispiel hatte aufgrund von call-by-value der Aufruf von `swap` keinen Einfluss auf `x` und `y` in `main`

```
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  }
```

# 11.3 Besonderheiten bei Referenztypen

## 11.3.3 Call-by-reference Effekt



- Bei Objekttypen gibt es dagegen einen unerwarteten Effekt:

```
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  }
```

**Heap nach Zeile 6**

<adr1> : { }

* `args = <adr1>`

* **`main`-Frame nach Zeile 6**

# 11.3 Besonderheiten bei Referenztypen

## 11.3.3 Call-by-reference Effekt



- Bei Objekttypen gibt es dagegen einen unerwarteten Effekt:

```
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  }
```

**Heap nach Zeile 8**

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

*

| werte = <adr2> |
|---|
| args = <adr1> |

***main-Frame nach Zeile 8**

# 11.3 Besonderheiten bei Referenztypen

## 11.3.3 Call-by-reference Effekt



- Bei Objekttypen gibt es dagegen einen unerwarteten Effekt:

```
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  }
```

**Heap nach Zeile 8**

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

| | |
|---|---|
| ** | wert = 3 |
| | index = 1 |
| | zahlen = <adr2> |
| * | werte = <adr2> |
| | args = <adr1> |

**changeValues-Frame nach Zeile 1

*main-Frame nach Zeile 8

# 11.3 Besonderheiten bei Referenztypen

## 11.3.3 Call-by-reference Effekt



- Bei Objekttypen gibt es dagegen einen unerwarteten Effekt:

```
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  }
```

**Heap nach Zeile 8**

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

| | |
|---|---|
| ** | wert = 3 |
| | index = 1 |
| | zahlen = <adr2> |
| * | werte = <adr2> |
| | args = <adr1> |

**changeValues-Frame nach Zeile 3

*main-Frame nach Zeile 8

# 11.3 Besonderheiten bei Referenztypen

## 11.3.3 Call-by-reference Effekt



- Bei Objekttypen gibt es dagegen einen unerwarteten Effekt:

```
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  }
```

**Heap nach Zeile 9**

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

*

| werte = <adr2> |
|---|
| args = <adr1> |

***main-Frame nach Zeile 9**

# Übersicht

# 11.4 Zusammenfassung



- Anders als in C und C++, wo der *-Operator zur Dereferenzierung eines Zeigers nötig ist, erfolgt in Java der *Zugriff* auf Referenztypen in der gleichen Weise wie der auf primitive Typen.
- Einen expliziten Dereferenzierungsoperator gibt es in Java nicht.
- Anders als in C/C++ verfügt Java auch über ein automatisches Speichermanagement:
  - 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.
  - Für die Halde gilt das nicht: Hier wird im Laufe eines Programmes Speicherplatz zugewiesen und belegt, aber nicht automatisch frei-gegeben, falls die Lebensdauer eines Namens, der für den gespeichertenWert steht, abgelaufen ist.

# 11.4 Zusammenfassung



- In vielen Sprachen muss der Programmierer dafür sorgen, dass Speicherplatz auf der Halde, 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.