

3.2.1. Variable, Konstante, Literale

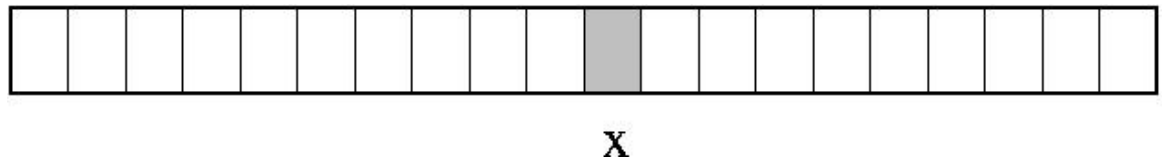
Beachte: Der Begriff *Variable* bedeutet in Mathematik und Programmierung etwas fundamental Verschiedenes!

- *Mathematik:* Eine Variable ist ein Platzhalter für die Elemente einer Menge.

Beispiel: Für alle reellen Zahlen x (kurz: für $x \in \mathbb{R}$) gilt $x^2 \geq 0$.

- *Programmierung:* Eine Variable ist ein zusätzlicher, symbolischer Name für eine feste Speicheradresse.

Beispiel: $x = x+1$



liest den Inhalt der Speicherzelle mit Namen "x" in das Rechenwerk, addiert 1 dazu und schreibt das Ergebnis wieder in dieselbe Speicherzelle zurück.

Konstante

- Eine **Variable** ist wie auf der letzten Folie beschrieben ein symbolischer Name für eine Speicheradresse.

Beispiel: Eine Zeichenvariable namens "var", die ein Zeichen speichern soll, kann man einrichten und sofort mit dem Zeichen "a" initialisieren durch

```
char var = 'a' ;
```

- Eine **Konstante** ist im Prinzip dasselbe, nur:
 - ◊ Eine Konstante *muss* **sofort initialisiert** werden.
 - ◊ Danach darf der Wert der Konstante **nicht mehr geändert** werden.
- Zur Einrichtung einer Konstante statt Variable schreibt man das Schlüsselwort `final` vorweg:

```
final char var = 'a' ;
```

Sinn von Konstanten

Mit der Einrichtung von "var" als einer Konstanten anstelle einer Variablen verbaut man sich ja Möglichkeiten.

Sinn:

- Oft ist ein Objekt von seiner inneren Logik her wirklich konstant.

Beispiele:

```
final float pi = 3.14159;  
final char waehrung = '$';
```

- Mit "final" kann man verhindern, dass der Wert irgendwo im Source File aus Versehen überschrieben wird (Fehler beim Kompilieren).
- Man muß sich deren Wert während der Programmierung nicht merken
- Konsistenz wird sichergestellt
- Konstanten müssen bei einer Portierung des Programms aber nur einmal geändert werden (z.B. Änderung der Währung)

Literale

Erinnerung: Zeichenketten der folgenden Formen

- ◇ 69534
- ◇ 3.14159
- ◇ 'a'
- ◇ "Hello World"

sind *keine* Konstanten, sondern *Literale*.

- *Also:*

- ◇ Eine *Konstante* ist ein *Objekt im Hauptspeicher*, deren Wert nicht geändert werden kann.
- ◇ Ein *Literal* ist ein explizit *ins Source File hineingeschriebener* (und damit natürlich ebenfalls unveränderlicher) *Wert*.

- *Beispiel:*

```
final char var = 'a';
```


 Konstante Literal

3.2.2 Datentypen in Java

Grundsätzlich gibt es zwei Arten von Datentypen in Java:

Eingebaute Typen:

- grundlegende Typen
- In diesen Datentypen finden alle konkreten, tatsächlichen Datenmanipulationen statt.

Klassen (Bausteintypen):

- Wiederverwendbare Bausteine zur Entwicklung größerer Programme mehr durch "Zusammenstecken" als durch Neuprogrammierung von Grund auf.
- Einkapselung der technischen Details in Bausteine erlaubt Programmierung auf einer abstrakteren, problemorientierteren Ebene.

Eingebaute Datentypen

Typ	Länge	Wertebereich	Standardwert
boolean	1	true, false	false
char	2	Alle Unicode-Zeichen	\u0000
byte	1	$-2^7 \dots 2^7 - 1$	0
short	2	$-2^{15} \dots 2^{15} - 1$	0
int	4	$-2^{31} \dots 2^{31} - 1$	0
long	8	$-2^{63} \dots 2^{63} - 1$	0
float	4	$\pm 3.40282347 * 10^{38}$	0
double	8	$\pm 1.79769313486231570 * 10^{308}$	0

Standardwert / Nullwert

(s.a. Skriptum p.314)

- Wie auf der vorigen Folie zu sehen, gibt es zu jedem eingebauten Datentyp einen Standardwert
- Wenn ein Objekt des Datentyps bei seiner Einrichtung nicht explizit initialisiert wird, dann wird das Objekt mit dem zugehörigen Standardwert initialisiert.

```
int i;  
int i = 0;
```

Beide Möglichkeiten sind äquivalent



- Standardwerte für eingebaute Typen:
 - ◇ **Numerische Typen:** Wert 0.
 - ◇ **Zeichentyp** `char`: Das "Nichtzeichen" mit Unicode-Wert 0.
 - ◇ **Logiktyp** `boolean`: Wert "false".
- Da die Werte alle 0 sind (false wird hier auch als 0 angesehen), nennt man diese Standardwerte auch **Nullwerte**.

Der Zeichentyp char

- Dient zur Abspeicherung eines einzelnen Zeichens
 - die meisten Programmiersprachen verwenden ASCII-Codes
 - in Java: Unicode-Zeichen, daher 2 Bytes!
- `char`-Literale werden zwischen einfache Hochkommas gesetzt.
 - Beispiele: `'A'`, `'a'`, `'+'`, `'\n'`, etc.
- Beliebige Unicode-Escape-Sequenzen können in der Form `\uxxxx` angegeben werden
 - wobei `xxxx` eine Folge von bis zu 4 hexadezimalen Ziffern ist.
 - `\u000a` für die Zeilenschaltung
 - `\u0020` für das Leerzeichen

Spezielle Zeichencodes

Zeichen	Bedeutung
<code>\b</code>	Rückschritt (Backspace)
<code>\t</code>	Horizontaler Tabulator
<code>\n</code>	Zeilenschaltung (Newline)
<code>\f</code>	Seitenumbruch (Formfeed)
<code>\r</code>	Wagenrücklauf (Carriage return)
<code>\"</code>	Doppeltes Anführungszeichen
<code>\'</code>	Einfaches Anführungszeichen
<code>\\</code>	Backslash

double and float

- Der Datentyp `float` (=floating–point numbers) war in anderen Programmiersprachen (z.B. C) gedacht als *der* Datentyp für reelle Zahlen schlechthin.
 - in Java: `float` = 32 bits (23 Mantisse + 8 Exponent + 1 Vorzeichen)
- Der Datentyp `double` (=double precision)
 - ◇ verwendet mehr Bits als `float` zum Abspeichern reeller Zahlen
 - in Java: `double` = 64 bits (53 Mantisse + 10 Exponent + 1 Vorzeichen)
 - ◇ und war nur für Berechnungen mit besonders kniffligen numerischen Fehlerproblemen vorgesehen.

Heute ist Speicherplatz kein Problem mehr.

→ Der Datentyp mit dem weitaus weniger intuitiven Namen `double` hat sich inzwischen als *der* Standardtyp für reelle Zahlen etabliert.

Relationale Operatoren

- Relationale Operatoren retournieren einen `boolean` Wert (i.e., `true` oder `false`)
- Sie arbeiten auf allen numerischen Datentypen (auch gemischten)

Operator	Bezeichnung	Bedeutung
<code>==</code>	Gleich	<code>a == b</code> ergibt <code>true</code> , wenn <code>a</code> gleich <code>b</code> ist.
<code>!=</code>	Ungleich	<code>a != b</code> ergibt <code>true</code> , wenn <code>a</code> ungleich <code>b</code> ist.
<code><</code>	Kleiner	<code>a < b</code> ergibt <code>true</code> , wenn <code>a</code> kleiner <code>b</code> ist.
<code><=</code>	Kleiner gleich	<code>a <= b</code> ergibt <code>true</code> , wenn <code>a</code> kleiner oder gleich <code>b</code> ist.
<code>></code>	Größer	<code>a > b</code> ergibt <code>true</code> , wenn <code>a</code> größer <code>b</code> ist.
<code>>=</code>	Größer gleich	<code>a >= b</code> ergibt <code>true</code> , wenn <code>a</code> größer oder gleich <code>b</code> ist.

Arithmetische Operatoren

- Arithmetische Operatoren erwarten numerische Operanden und liefern einen numerischen Rückgabewert.
 - Haben die Operanden unterschiedliche Typen, beispielsweise `int` und `float`, so entspricht der Ergebnistyp des Teilausdrucks dem größeren der beiden Operanden.
 - Zuvor wird der kleinere der beiden Operanden mit Hilfe einer erweiternden Konvertierung in den Typ des größeren konvertiert.
- Sonderfälle:
 - Zeichen vom Typ `char` können addiert bzw. subtrahiert werden (entspricht der Addition/Subtraktion der Zeichen-Codes)
 - Strings können addiert werden (entspricht der Konkatination)
 - Generell können Operatoren für Objekte definiert bzw. überlagert werden (dazu mehr später)

Arithmetische Operatoren

Operator	Bezeichnung	Bedeutung
+	Positives Vorzeichen	+n ist gleichbedeutend mit n
-	Negatives Vorzeichen	-n kehrt das Vorzeichen von n um
+	Summe	a + b ergibt die Summe von a und b
-	Differenz	a - b ergibt die Differenz von a und b
*	Produkt	a * b ergibt das Produkt aus a und b
/	Quotient	a / b ergibt den Quotienten von a und b
%	Restwert	a % b ergibt den Rest der ganzzahligen Division von a durch b . In Java läßt sich dieser Operator auch auf Fließkommazahlen anwenden.
++	Präinkrement	++a ergibt a+1 und erhöht a um 1
++	Postinkrement	a++ ergibt a und erhöht a um 1
--	Prädecrement	--a ergibt a-1 und verringert a um 1
--	Postdecrement	a-- ergibt a und verringert a um 1

Logische Operatoren

Operator	Bezeichnung	Bedeutung
!	Logisches NICHT	!a ergibt <code>false</code> , wenn a wahr ist, und <code>true</code> , wenn a falsch ist.
&&	UND mit Short-Circuit-Evaluation	a && b ergibt <code>true</code> , wenn sowohl a als auch b wahr sind. Ist a bereits falsch, so wird <code>false</code> zurückgegeben und b nicht mehr ausgewertet.
	ODER mit Short-Circuit-Evaluation	a b ergibt <code>true</code> , wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Ist bereits a wahr, so wird <code>true</code> zurückgegeben und b nicht mehr ausgewertet.
&	UND ohne Short-Circuit-Evaluation	a & b ergibt <code>true</code> , wenn sowohl a als auch b wahr sind. Beide Teilausdrücke werden ausgewertet.
	ODER ohne Short-Circuit-Evaluation	a b ergibt <code>true</code> , wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Beide Teilausdrücke werden ausgewertet.
^	Exklusiv-ODER	a ^ b ergibt <code>true</code> , wenn beide Ausdrücke einen unterschiedlichen Wahrheitswert haben.

Quelle: <http://www.javabuch.de>

Short Circuit Operatoren

- In einigen Fällen ist es oft schon klar, was das Ergebnis des Ausdrucks ist, ohne daß der gesamte Ausdruck ausgewertet werden muß:
 - Beispiel:
 - $a \vee b$:
 - Wenn a bereits true ist, dann muß man b nicht mehr betrachten, da der Ausdruck $a \vee b$ auf jeden Fall wahr sein wird.
- Das kann zur Verbesserung der Laufzeit ausgenützt werden
 - in vielen Fällen wird das aber nicht gewünscht!
 - z.B. wenn die beiden logischen Ausdrücke Seiteneffekte haben, die für die weitere Durchführung des Programms wichtig sind (unschöne Programmierung!)
 - daher gibt es auch “normale” Operatoren

Zuweisungsoperatoren

- Zuweisung eines Wertes: $=$
 - *Beispiel:* $a = b$
 - a erhält den Wert von b zugewiesen.
 - **Rückgabewert:** der zugewiesene Wert
- Veränderung eines Werts: $op=$
 - Wobei op für einen anderen zweistelligen logischen oder arithmetischen Operator steht
 - **Rückgabewert:** der zugewiesene Wert
 - *Beispiele:*
 - $a += b$: a erhält den Wert von $a + b$
 - $a \&= b$: a erhält den Wert von $a \& b$
- Anmerkung:
 - In allen Beispielen kann b selbstverständlich ein beliebiger Ausdruck sein!
 - *Beispiel:* $a -= b.length * 2$
 - vermindert a um die doppelte Länge des Arrays b

Konversion zwischen eingebauten Typen

- Objekte unterschiedlicher numerischer eingebauter Typen können in Zuweisungen, Vergleichen und arithmetischen Operationen direkt miteinander kombiniert werden.

Beispiele:

```
int i = 1;
double d = 3.14;

double x = i;
double y = i + d;

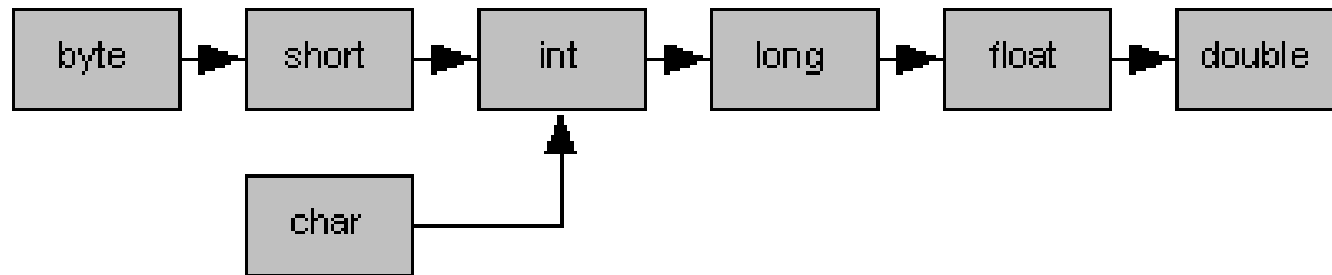
if ( i < d )
```

- Der eingebaute Typ `char` für Zeichen ist hier mit den Unicodes als Zahlenwerten mit im Spiel:

```
char c = 'a';
int i = c + 1;
```

Automatisches Type-Cast

Automatisch konvertiert werden:



- alle aus dieser unmittelbaren Beziehung sich ergebenden transitiven Konversionen (z.B. `char` → `double`)

Anmerkung zur Konversion `int` → `float`:

- Die Konversion `int` → `float` ist nicht 100% sicher.
- Bei sehr großen `int`-Zahlen (bzw. `long`-Zahlen) ändert sich der Wert ein wenig bei der Konversion nach `float`.
 - z.B. weil `int` 4 Bytes = 32 bits zur Repräsentation haben, während `floats` nur eine Mantissen-Länge von 23 bits haben.
- Trotz dieser kleinen Unsicherheit wird diese Konversion als sicher erachtet.

Explizites Type-Cast

- Wenn bei einer solchen Zuweisung der Typ auf der linken Seite nicht jeden Wert darstellen kann, den der Typ auf der rechten Seite darstellen kann, ist man zur eigenen Sicherheit gezwungen, explizit hinzuschreiben, dass man die Konversion wirklich will:

```
int i = 'a';  
char c1 = (char) i;  
char c2 = (char) (i+1);
```

→ Sonst Fehlermeldung beim Kompilieren.

- *Generelle Syntax* (Altlast aus C, nicht aus C++):
 - ◊ Der Zieltyp muss in Klammern vor den zu konvertierenden Ausdruck in Klammern hingeschrieben werden.
 - ◊ Der zu konvertierende Ausdruck muss ebenfalls in Klammern gesetzt werden, falls er nicht einfach aus einem einzelnen Literal oder Identifier besteht.

Datentyp von Literalen

- **Zeichenliterale** sind vom Datentyp "char".
 - **Ganzzahlige Literale** sind vom Typ "int".
 - **Reelle Literale** sind vom Typ "double".
- Die folgende Zeile ergibt widersinnigerweise eine Fehlermeldung beim Kompilieren:

```
float f = 3.14;
```

- Man muss daher schreiben:

```
float f = (float) 3.14;
```

Beispiel: Konversion auf Großbuchstaben

```
c = (char) (c - 'a' + 'A');
```

Erläuterungen:

- von `c` wird der Code des Kleinbuchstaben `'a'` abgezogen und der Code von `'A'` addiert \rightarrow Code von `'C'`
- Arithmetische Operationen sind für `char` eigentlich gar nicht definiert.
- Im obigen Ausdruck werden die drei `char`-Werte (eine Variable, zwei Literale) daher implizit zu `int` konvertiert.
- Das Ergebnis solcher arithmetischen Operationen auf `int`-Werten ist generell wieder vom Typ `int`.
- Um das Ergebnis in einer `char`-Variablen abzuspeichern, muss es daher explizit nach `char` konvertiert werden.
- Das ist schlussendlich die Erklärung, warum nicht einfach dastehen darf:

```
c -= 'a' + 'A';
```

3.2.3 Klassen

- Prozedurale Programmierung
 - im Zentrum stehen Unterprogramme (Prozeduren)
- Objekt-Orientierte Programmierung
 - im Zentrum stehen Datenstrukturen
- Prinzip der Abstraktion
 - Es wird nicht erlaubt, auf die Datenstrukturen selbst zuzugreifen, sondern nur mittels vordefinierter Methoden
 - dadurch wird gewährleistet, daß nicht unvorhergesehene Dinge passieren können
 - man kann z.B. die Anzahl der Beeper oder die Position eines Roboters nicht direkt verändern, sondern nur mit entsprechenden Methoden `pickBeeper` oder `move`
- Man muß lernen, in Objekten und Methoden zu denken!
 - Kapitel 4 der Vorlesung beschäftigt sich damit

Klassen

- Klassen sind abstrakte Einheiten, die das Verhalten von Gruppen von Objekten definieren
 - Beispiele:
 - `Robot` ist eine KarelJ Klasse für Roboter
 - `Applet` ist eine Klasse für Internet-Applikationen
- Klassen bestehen aus
 - Daten, die den Zustand eines Objekts beschreiben
 - Anzahl der Beeper, Position eines Roboters
 - Zustand des momentanen Zeichenfelds eines Applets
 - Methoden, die es erlauben, den Zustand zu ändern bzw. ein vom Zustand abhängiges Verhalten an den Tag zu legen
 - `move()`, `pickBeeper()`, etc.
 - `paint()`

Klassen und Objekte

- Objekte bzw. Instanzen sind konkrete Ausprägungen einer abstrakten Klasse
 - `karel` ist ein `Robot`
 - `myApplet` ist ein `Applet`
- Definition von Klassen und Objekten:
 - Klassen werden mittels `class` definiert und zur Compile-Zeit übersetzt
 - Objekte werden mittels `new` definiert und zur Laufzeit angelegt

Vererbung

- Klassen kann man definieren, indem man existierende Klassen erweitert

→ Erkennbar an der "extends"-Klausel:

```
public class DemoApplet extends Applet
                        ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
```

- dabei werden die Methoden der existierenden Klasse (Super-Class) an die neue Klasse (Sub-Class) weitervererbt
 - können aber überschrieben bzw. ergänzt werden
- das erspart sehr viel Arbeit, da viele existierende Bausteine wiederverwertet werden können
 - Objekt-Orientierte Programmierung unterstützt Wiederverwertbarkeit von Programmen durch
 - Abstraktion von Datenstrukturen
 - Bereitstellung von klar definierten Zugangsmethoden

Vordefinierte Java-Klassen

Klassen sind komplexe “Bausteine”, die aus Basis-Elementen zusammengesetzt werden.

Willkürlich herausgegriffene Beispiele für schon vorgefertigte Bausteine in Java:

- `String`: Verwaltung von beliebig langen, aber unveränderlichen Zeichenketten.
- `StringBuffer`: Wie `String`, aber Zeichenkette im `StringBuffer`-Objekt ist auch veränderbar (z.B. mit `append`).
- `Math`: Bereitstellung diverser mathematischer Funktionen und Konstanten.
- `Applet`: “Urstamm“ aller Java-Applets (vergleichbar zu `UrRobot`)

Mehr Beispiele: Klassen für Windows

- `Window`: "Urstamm" aller durch Java-Programme geöffneten Bildschirmfenster.
- `Frame`: Eine Erweiterung von `Window`, die die zusätzliche Funktionalität eines Top-Level-Windows bereitstellt
- "Urstämme" für diverse Ingredienzen von Fenstern, z.B.
 - ◊ `Choice`: Auswahlmenü,
 - ◊ `Dialog`: Dialogbox,
 - ◊ `Button`: Knopf zum Draufklicken mit dem Mauszeiger,
 - ◊ `Image`: Graphik-/Bildelement,
 - ◊ `TextArea`: Fläche zur interaktiven Textbearbeitung,
 - ◊ `Scrollbar`: Balken zum Verschieben ("Scrollen") des Fensterinhalts.

Dokumentation

- Voraussetzung für die Wiederverwendbarkeit von Software-Modulen ist natürlich eine entsprechende Dokumentation
 - das Lesen des Source-Codes oder darin enthaltenen Kommentare ist i.A. nicht notwendig, wenn man ein Klasse nur benutzen oder erweitern möchte!
- Dokumentation des **API (Application Programming Interface)** für Java Klassen:
 - <http://java.sun.com/j2se/1.4.2/docs/api/>
 - enthält genaue Beschreibungen aller im Basis-Java-Paket vordefinierten Klassen und ihrer Methoden

String und StringBuffer

- Beispiel Klasse `String`:
 - <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html>
 - eine große Anzahl vordefinierter Methoden erlaubt die Manipulation von String-Objekten
- Beispiel Klasse `StringBuffer`:
 - <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/StringBuffer.html>
 - `StringBuffer` Objekte sind wie Strings, können jedoch verändert werden (z.B. `append`).

```
StringBuffer s = new StringBuffer("Eins");  
s.append(" Zwei");  
// s enthält nun den String "Eins Zwei"
```

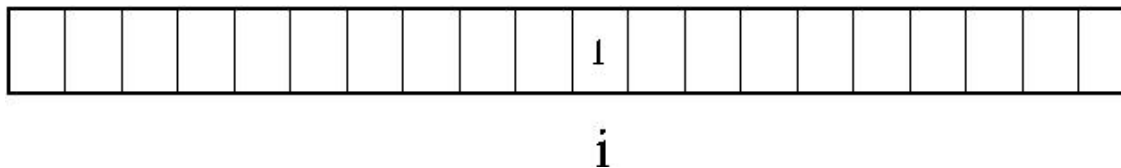
3.2.4 Objekte und Referenzen

- Mit dem Namen eines Objekts von einem eingebauten Typ spricht man das Objekt unmittelbar an
 - d.h. die Variable speichert direkt die zugrundeliegende Repräsentation des Objekts
- Bei Bausteintypen ist der Name des Objekts nur als eine Referenz (d.h. Verweis) auf ein anonymes Objekt vom Bausteintyp zu verstehen.
 - d.h. die Variable speichert eine Adresse, die angibt, wo die komplexe Datenstruktur zu finden ist.
- Gründe:
 - Objekte haben keine fixe Größe (können sehr groß sein), die Zeiger haben die Größe einer Speicher-Adresse.
 - bei Übergabe eines Objekts an eine Methode ist es daher viel effizienter, nur die Adresse zu übergeben, anstatt den gesamten Inhalt zu kopieren!

Veranschaulichung

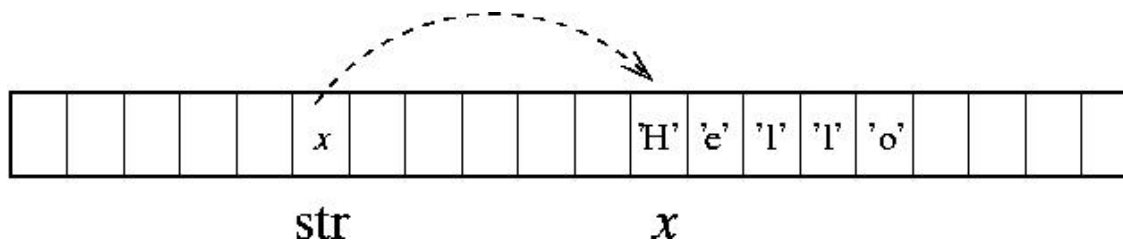
- `int i = 1;`

Den Bezeichner `i` kann man sich vorstellen als einen symbolischen Namen für die Speicheradresse, in der der Wert der Variable abgespeichert ist.



- `String str = new String ("Hello");`

Der Bezeichner `str` ist eher ein symbolischer Name für eine Speicheradresse, deren Inhalt die *Anfangsadresse* `x` der gespeicherten Zeichenkette ist.



Initialisierung

- Als Konsequenz wird bei der Deklaration eines Objektes eines **Bausteintyps** nur der Speicherplatz für die Referenz reserviert.
 - Das Objekt selbst wird durch `new` generiert
 - Und erst dann der zugehörige Speicherplatz reserviert.
- Bei **eingebauten Typen** wird das Objekt auf jeden Fall initialisiert!
 - Wenn nicht anders angegeben, dann mit dem Standardwert!

Beispiel

- `int i;`

Objekt `i` ist schon fertig konstruiert und mit dem Standard-Wert `0` initialisiert.

- `int i = 1;`

Objekt `"i"` ist schon fertig konstruiert und mit Wert `1` initialisiert.

- `String str;`

Nur die *Referenz* `str` auf eine Zeichenkette ist eingerichtet worden, aber damit ist noch keine Zeichenkette eingerichtet worden.

- `String str = new String ("Hello");`

Korrekt: Referenz `"str"` ist eingerichtet und mit einem String-Literal des Inhalts `"Hello"` initialisiert

- **Alle Objekte müssen mit `new` angelegt werden!**

Initialisierung von Strings

- Für Strings ist aber auch folgendes korrekt:

```
String str = "Hello";
```

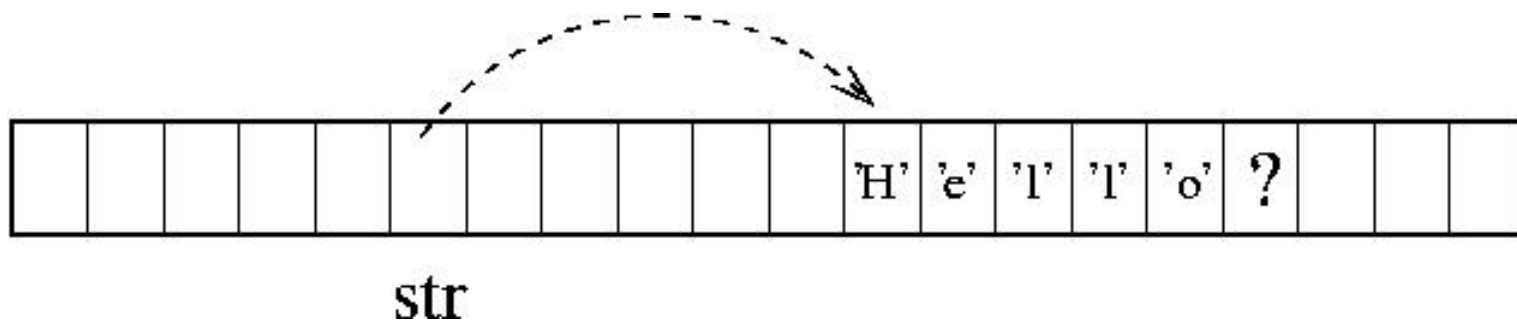
- *Hintergrund:*

- `String` ist eine der wichtigsten Klassen überhaupt in Java.
- Aus diesem Grund hat man sich entschieden, einige der wichtigsten Operationen auf Strings
 - ◊ nicht nur mit der üblichen Syntax für solche Operationen zu realisieren,
 - ◊ sondern auch noch einmal mit einer wesentlich einfacheren und bequemeren Syntax.
- Die Anweisung ist einfach eine Abkürzung für

```
String str = new String("Hello");
```

Speicherung von Strings

- Die Zeichen in "Hello" sind ja im Speicher einfach nur beliebige Bitmuster
- Auch die Bitmuster im Maschinenwort unmittelbar nach dem Ende der Zeichenkette könnten prinzipiell als Zeichen interpretierbar sein.
- Woher "weiß" ein String-Objekt `str` eigentlich, wo genau "seine" Zeichenkette aufhört?

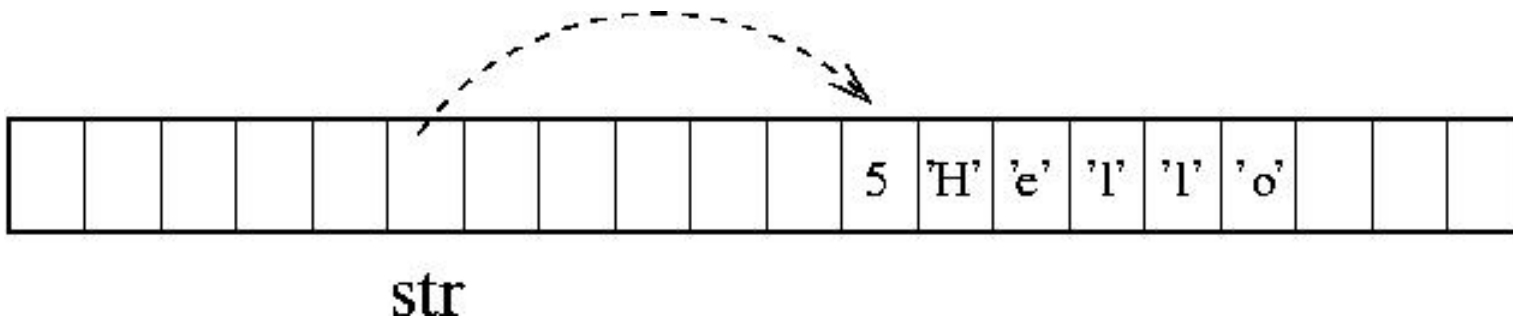


Speicherung von Strings

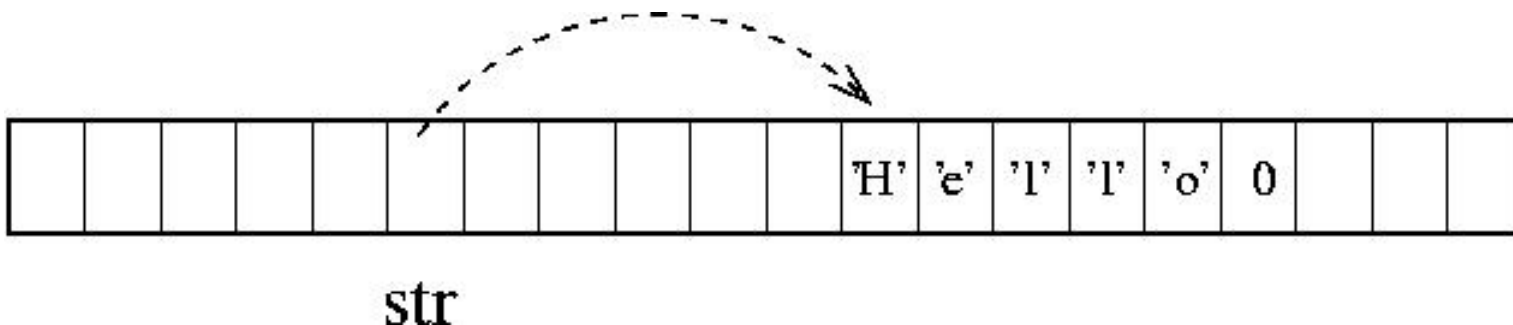
Mindestens zwei grundlegend verschiedene Möglichkeiten.

Nämlich:

- Vorweg wird die Anzahl der Zeichen gespeichert.



- An jede Zeichenkette wird als Begrenzungsanzeiger ein fest gewähltes "unmögliches" Zeichen angehängt (z.B. Unicode-Wert 0).



Umsetzung in Programmiersprachen

- In C/C++ ist Begrenzung durch ASCII/Unicode–Wert 0 als Strategie festgelegt und muss beim Programmieren beachtet werden!
 - Wenn dort ein String nicht mit `\0` beendet wird, gibt's ein Problem.
- In abstrakteren Sprachen wie Java sind das alles intern verwaltete technische Details hinter der Fassade von `String`, die der Programmierer gar nicht zu sehen bekommt.

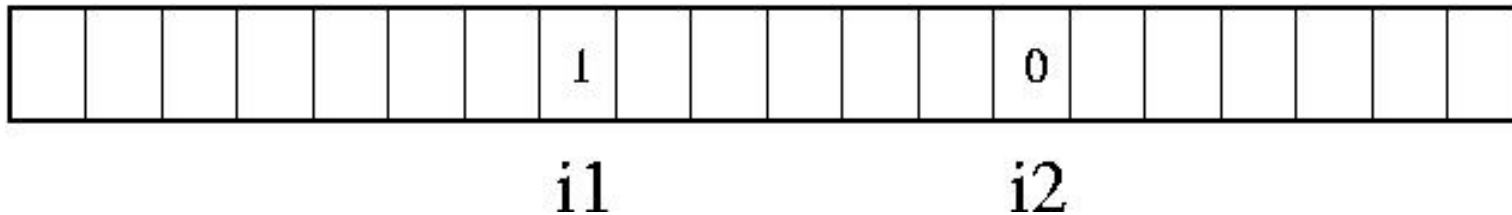
Standard-Wert von Referenzen

```
int    i1 = 1;
int    i2;
String str1 = new String ("Hello");
String str2;
```

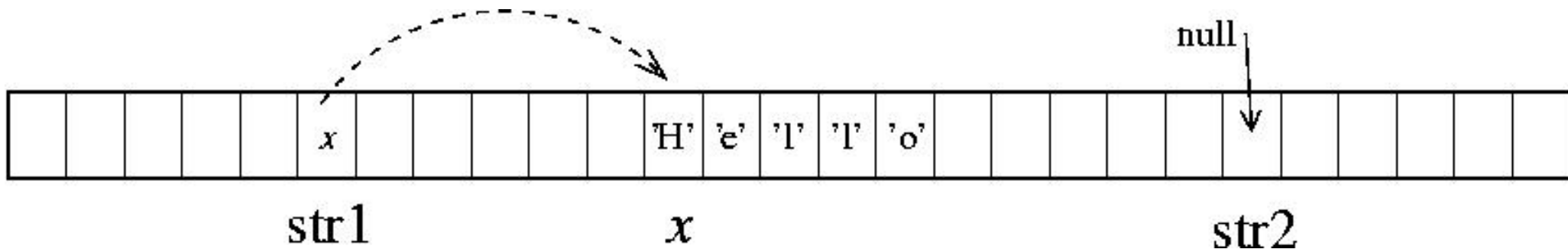
- Es ist klar, dass das Objekt `i1` den Wert `1` enthält und `str1` auf ein (anonymes) Objekt mit Inhalt "Hello" verweist.
- Wir wissen auch schon, daß `i2` den Standard-Wert `0` hat.
- Klassen werden mit einem symbolischen Referenzwert mit Namen `null` initialisiert
 - ◊ keine wirkliche Referenz auf irgendein Objekt,
 - ◊ sondern ein "unmöglicher" Wert,
 - ◊ der nur anzeigt, dass die Variable momentan auf kein Objekt verweist.

Standard-Wert von Referenzen

```
int    i1 = 1;  
int    i2;
```



```
String str1 = new String ("Hello");  
String str2;
```



Nicht initialisierte Referenzen

- Es gibt grundsätzlich kein uninitialisiertes Objekt in Java.
 - Eine wichtige Quelle für undefiniertes Programmverhalten ist eliminiert.
 - Im Gegensatz zu anderen Programmiersprachen wie C und C++.
- *Allerdings:*
 - ◇ Wenn eine Klassenvariable den Wert `null` hat
 - ◇ und man trotzdem auf das dahinterstehende Objekt,
 - ◇ bzw. auf das eben **nicht** dahinterstehende Objekt zugreift,
 - ◇ dann stürzt das Programm ab:

```
StringBuffer str; // == null
str.append ( "bla" ); // Absturz!
```


Bug oder Feature?

- Dieser Absturz ist kein undefiniertes Programmverhalten
- sondern es ist "garantiert", dass das Programm sofort abstürzt.
 - Damit ist immerhin garantiert, dass das Programm keinen weiteren Schaden anrichtet.

Vorgreifende Bemerkungen:

- Durch geeignete Java-Konstrukte kann man einen solchen Programmabsturz auch abfangen und behandeln.
 - Stichwort *Exceptions*
- Die Initialisierung von Objekten einer selbstgebastelten Klasse kann man auch selbst programmieren.
 - Stichwort *Konstruktoren*

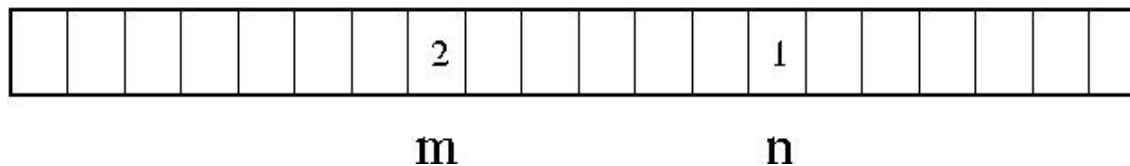
Zuweisung von Referenzen

Beachte: Zuweisung bei Bausteintypen bedeutet, dass nun zwei Referenzen auf dasselbe anonyme Objekt verweisen!

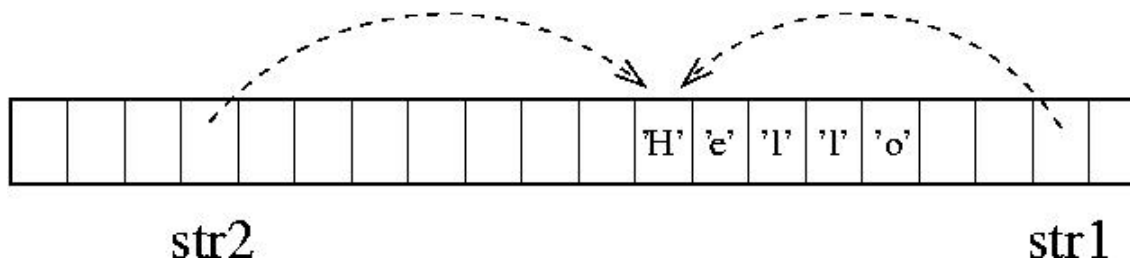
- Eine Methode einer Klasse wird zwar mit dem Namen einer Variablen dieser Klasse aufgerufen.
- Aber sie macht eigentlich gar nichts mit dieser Variable.
- Statt dessen macht sie etwas mit dem *Objekt*, auf das diese Variable verweist.
- Wenn zwei Variable einer Klasse auf dasselbe Objekt verweisen, ist es also logisch, dass der Effekt eines Methodenaufrufs (z.B. `append`) mit einer Variablen (`str2`) zugleich über die andere Variable (`str1`) sichtbar wird.

Beispiel

```
int n = 1;
int m = n;
m++;
System.out.print (m);      // Ausgabe: "2"
System.out.print (n);      // Ausgabe: "1"
```



```
StringBuffer str1 = new StringBuffer ( "Hello" );
StringBuffer str2 = str1;
str2.append ( ", World" );
System.out.print ( str2 );      // Ausgabe: "Hello, World"
System.out.print ( str1 );      // Ausgabe: "Hello, World"
```



Gleichheit von Referenzen

- Test mit `==` auf Gleichheit bedeutet bei Klassentypen
 - ◊ nicht Test auf Wertgleichheit wie bei eingebauten Typen,
 - ◊ sondern Test auf Objekt**identität!**
- Jede vorgefertigte Klasse in der Java–Standardbibliothek besitzt für den Test auf Wertgleichheit eine Methode namens `equals` mit
 - ◊ einem Argument, dem zu vergleichenden Objekt, und
 - ◊ Rückgabetyt `boolean`

→ **Also** `true/false`.

Beispiel

```
String str1 = new String ( "Hello" );  
String str2 = str1;  
String str3 = new String ( "Hello" );
```

```
if ( str1 == str2 )  
    System.out.println ( "Ja" );  
else  
    System.out.println ( "Nein" );  
// Ausgabe: Ja
```

```
if ( str1.equals(str2) )  
    System.out.println ( "Ja" );  
else  
    System.out.println ( "Nein" );  
// Ausgabe: Ja
```

```
if ( str1 == str3 )  
    System.out.println ( "Ja" );  
else  
    System.out.println ( "Nein" );  
// Ausgabe: Nein!!!
```

```
if ( str1.equals(str3) )  
    System.out.println ( "Ja" );  
else  
    System.out.println ( "Nein" );  
// Ausgabe: Ja
```

Übergabe von Referenzen an Methoden

Argumente von Methoden haben unterschiedliche Bedeutung für eingebaute Typen und Bausteintypen.

Simplex Beispiel:

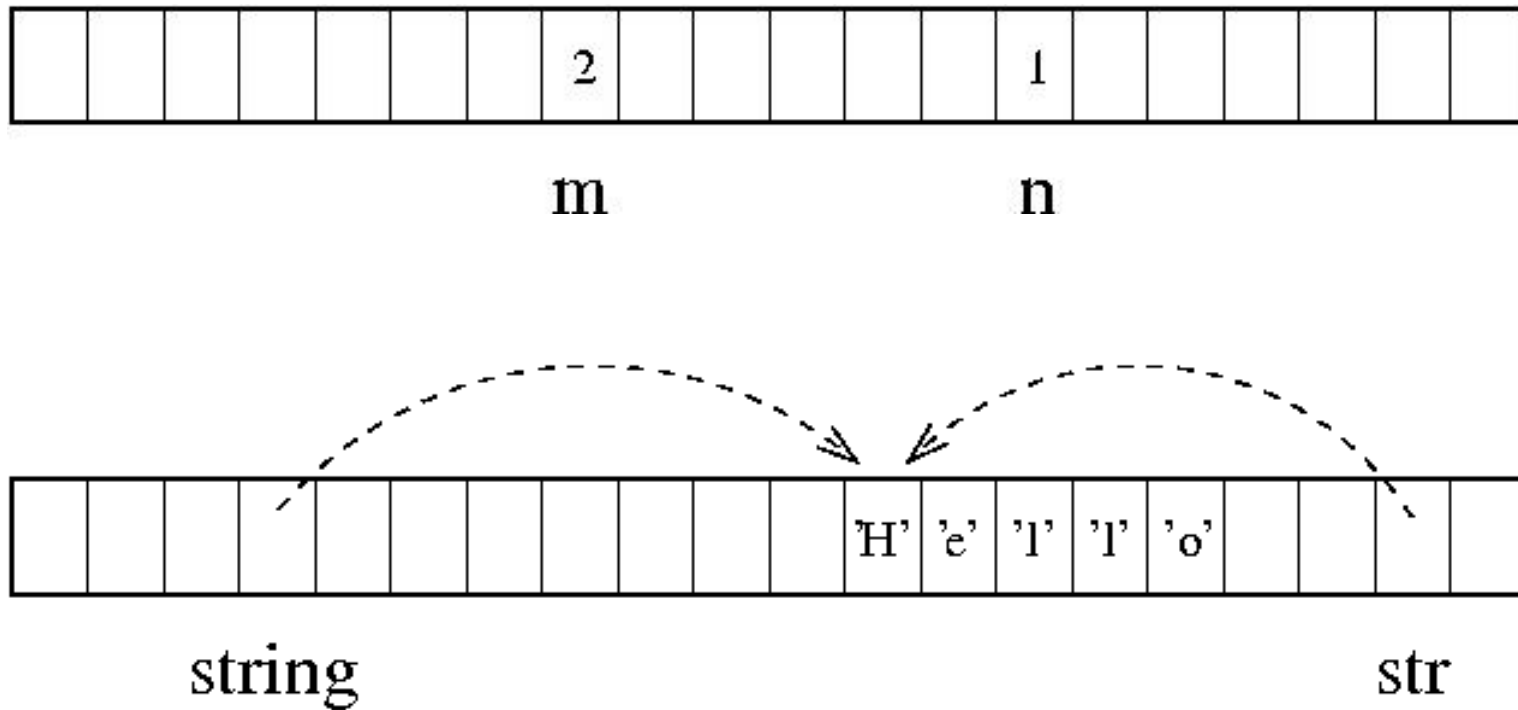
```
void f ( int m, StringBuffer string )
{
    m++;
    string.append ( ", World" );
}
...

int n = 1;
StringBuffer str = new StringBuffer ( "Hello" );

f ( n, str );

System.out.print (n);           // -> "1"
System.out.print (str);        // -> "Hello, World"
```

Veranschaulichung



Referenzen als Rückgabewerte

Rückgabewerte von Methoden haben unterschiedliche Bedeutung für eingebaute Typen und Bausteintypen.

Betrachte als Beispiel dazu folgende zwei Methoden:

```
int f1 ( int n )  
{  
    return n;  
}
```

```
StringBuffer f2 ( StringBuffer string )  
{  
    return string;  
}
```


Beispiel (Fortsetzung)

```
int          m1      = 1;
StringBuffer str1 = new StringBuffer ( "Hello" );

int          m2      = f1 ( m1 );
StringBuffer str2 = f2 ( str1 );

m1++;
str1.append ( ", World" );

System.out.print ( m1 );      // -> "2"
System.out.print ( m2 );      // -> "1"

System.out.print ( str1 );    // -> "Hello, World"
System.out.print ( str2 );    // -> "Hello, World"
```

Wrapper-Klassen

- Zu jedem eingebauten Typ gibt es eine spezifische Klasse ("Wrappertyp").
- Ein Wert des eingebauten Typs kann in ein Objekt des zugehörigen Wrappertyps "eingepackt" (engl. "to wrap") und "herumtransportiert" werden.
- Der Wrappertyp bietet eine Methode, mit dem der momentane Wert des eingepackten Wertes abgefragt werden kann.

Beispiel:

```
Integer x = new Integer (1);  
int n = x.intValue ();  
  
System.out.print (n); // -> "1"
```



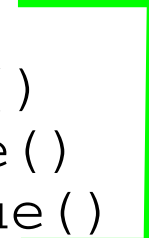

Wrapper Klassen

Wrapper-Klasse	Primitiver Typ
Byte	byte
Short	short
Integer	int
Long	long
Double	double
Float	float
Boolean	boolean
Character	char
Void	void

Anmerkung: Eigentlich lautet der volle Name der Klassen `java.lang.Byte`, `java.lang.Short`, etc. Gleiches gilt auch für `String` und `StringBuffer`

Methoden der Wrapper-Klassen

- Wrapper-Klassen bieten auch viele Methoden zur Konversion zwischen verschiedenen Typen

<code>public boolean</code>	<code>booleanValue()</code>	 nur für Boolean definiert
<code>public char</code>	<code>charValue()</code>	 nur für Character definiert
<code>public int</code>	<code>intValue()</code>	 für alle numerischen Wrapper-Klassen definiert
<code>public long</code>	<code>longValue()</code>	
<code>public float</code>	<code>floatValue()</code>	
<code>public double</code>	<code>doubleValue()</code>	
<code>public String</code>	<code>toString()</code>	 für alle Wrapper-Klassen

- Weiters kann man Objekte jeder Wrapper-Klasse nicht nur mit dem korrespondierenden primitiven Typ initialisieren
 - sondern auch mit einem String
- Weiter Methoden findet man wiederum in der Dokumentation!

Beispiel

```
String str1 = new String ( "7654" ); // (a)
Integer x   = new Integer ( str1 );
String str2 = x.toString();         // (b)
double y    = x.doubleValue();      // (c)
```

Erläuterung:

Unter anderem bietet die Klasse `Integer` auch Möglichkeiten, ganze Zahlen

- (a) aus Zeichenketten, die Dezimalzahlen darstellen, zu konstruieren,
- (b) in ebensolche Zeichenketten zu transformieren und
- (c) in andere numerische Datentypen (z.B. `double`) zu konvertieren

Methoden zur Zeichenmanipulation

(s.a. Skriptum 292-294)

```
char c = 'A';  
char d;  
if (Character.isLowerCase(c))  
    d = Character.toUpperCase(c);  
else  
    d = c;
```

- Falls momentan ein Kleinbuchstabe in `c` gespeichert ist, soll in `d` der entsprechende Großbuchstabe gespeichert werden, sonst das Zeichen von `c` selbst.
- Die technischen Details sind hinter den einfachen, intuitiven **Schnittstellen** `Character.isLowerCase` bzw. `Character.toUpperCase` **versteckt**.

Methoden zur Zeichenmanipulation

- Die interne technische Realisierung könnte zum Beispiel so aussehen:

- ◊ `Character.isLowerCase`:

Größenvergleich der Unicode-Nummern.

→ Kann man auch direkt in Java schreiben:

```
if ( c >= 'a' && c <= 'z' )
```

- ◊ `Character.toUpperCase`:

Ziehe die Unicode-Nummer von 'a' ab und addiere die Unicode-Nummer von 'A'.

- ◊ Kann man fast direkt in Java schreiben (haben wir schon besprochen):

```
d = (char) (c - 'a' + 'A')
```

3.2.5. Zusammengesetzte Objekte

- Objekte von **eingebauten Typen** sind aus Sicht eines Java-Programmierers *atomar*.
- *Das heißt:* Eine etwaige weitere interne Struktur und Zerlegbarkeit eines solchen Objekts auf Maschinenebene wird im Java-Quelltext nicht sichtbar.
- *Ausnahme:* Es gibt Operatoren in Java zur logischen Verknüpfung von Zahlen "Bit-für-Bit".
→ Betrachten wir in dieser Veranstaltung nicht (für Interessierte: Stichwort *Bitlogik*).
- Objekte von Klassen sind hingegen im allgemeinen aus mehreren Variablen von eingebauten Typen und/oder Klassen zusammengesetzt.

Beispiel

```
public class MeineKlasse
{
    public int i;
    public double d;
    public char c;
}
...
```

Jedes Objekt der Klasse

MeineKlasse ist aus

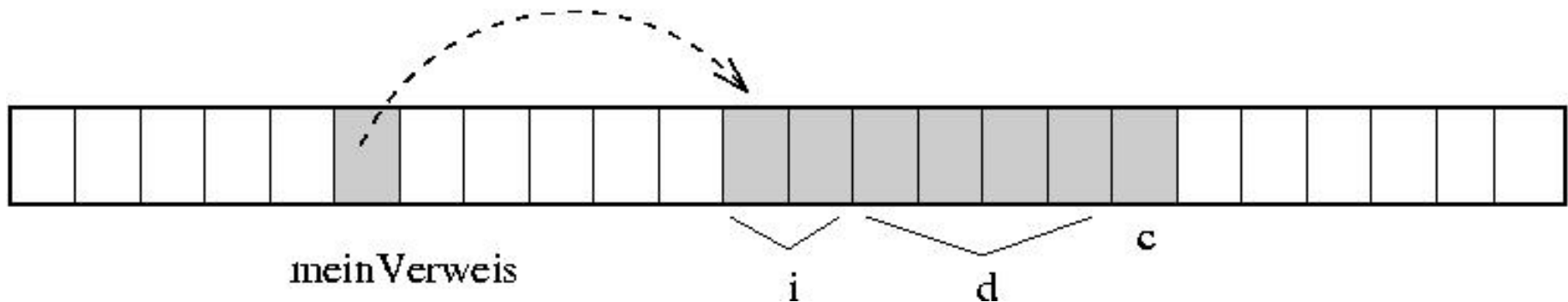
- einem `int`-Objekt
 - einem `double`-Objekt und
 - einem `char`-Objekt
- zusammengesetzt.

```
MeineKlasse meinVerweis = new MeineKlasse();
```

Erste, einführende Erläuterung:

- Mit obigem Code ist eine Klasse namens `MeineKlasse` definiert und sogleich eine Variable `meinVerweis` von `MeineKlasse` mit dahinterstehendem Objekt angelegt worden.
- Die genauen Details der Syntax (insbesondere der Sinn von `public`) werden erst später in der Vorlesung behandelt

Veranschaulichung



Erläuterungen:


- *Erinnerung*: Variablen von Klassen sind nur Verweise auf die eigentlichen (anonymen) Klassenobjekte.
- Die drei Objekte von eingebauten Typen, die im Objekt hinter `meinVerweis` zusammengefasst sind, werden mit
 - `meinVerweis.i`
 - `meinVerweis.d` und
 - `meinVerweis.c`angesprochen.

Klassen mit Klassenkomponenten

Klassen können anderen Klassen als Komponenten enthalten.

```
public class ErsteKlasse
{
    public int i;
    public double d;
    public char c;
}

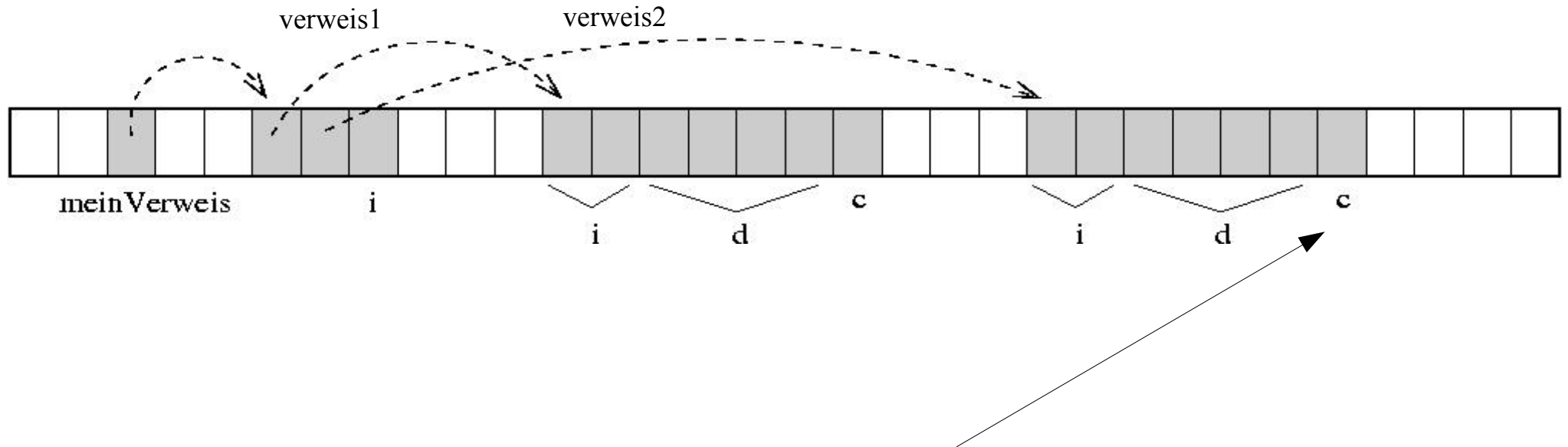
public class ZweiteKlasse
{
    public ErsteKlasse verweis1;
    public ErsteKlasse verweis2;
    public int i;
}
```



...

```
ZweiteKlasse meinVerweis = new ZweiteKlasse();
meinVerweis.verweis1 = new ErsteKlasse();
meinVerweis.verweis2 = new ErsteKlasse();
```

Veranschaulichung



→ Zum Beispiel greift `meinVerweis.verweis2.c` in dieser Schemazeichnung auf die äußerst rechte graue Speicherzelle zu.

Konstruktoren

(s. Skriptum Folie 441 ff.)

- Ein *Konstruktor* einer Klasse ist eine bestimmte Art von Methode.
- Syntaktische Eigenarten:
 - ◊ Der Name der Methode muss zugleich der Name der Klasse selbst sein.
 - ◊ Bei einem Konstruktor wird kein Rückgabetyt angegeben (auch nicht "void").
 - Eine Methode, die genau gleich wie ihre Klasse heißt, ist durch diese Namensgleichheit automatisch ein Konstruktor, und es darf daher kein Rückgabetyt angegeben werden.
 - Konstruktoren sind die einzigen Methoden in Java, bei denen kein Rückgabetyt angegeben wird (wieder aus C++ übernommen).
- Eine Klasse darf mehrere Konstruktoren haben.
 - Verschiedene Konstruktoren müssen verschiedene Parameterlisten haben (das ist auch bei normalen Methoden möglich, kommt später)

Verwendung von Konstruktoren

- Ein Konstruktor einer Klasse wird normalerweise in einer einzigen spezifischen Situation aufgerufen: bei der Erzeugung eines Objekts dieser Klasse mit `new`.
- Der Klassenname hinter `new` ist also genauer gesagt der (identische) Name des Konstruktors bei seinem Aufruf:

```
String str = new String ( "Hello" );  
                  ^ ^ ^ ^ ^ ^
```

- Damit wird auch klar, was die Klammern hinter dem Klassennamen in einem `new`-Ausdruck sollen:
 - ◊ Das ist einfach die Parameterliste für den Aufruf des Konstruktors.
 - ◊ Ein leeres Klammerpaar bedeutet dann, dass ein Konstruktor mit leerer Parameterliste aufgerufen wird.

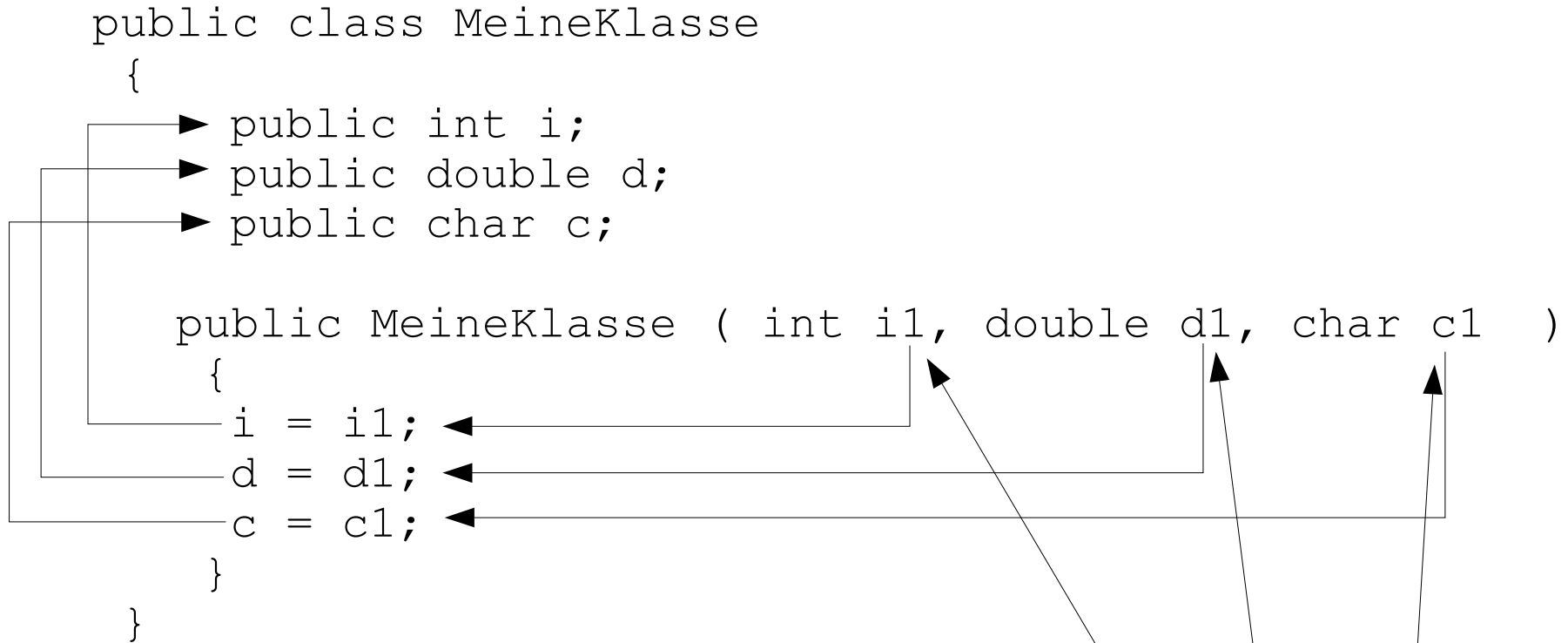
Beispiele für Konstruktoren

- `String str1 = new String ("Hallo");`
`StringBuffer str2 = new StringBuffer ("Hallo");`
- `Color c = new Color (1, 1, 0);`
→ Farbobjekt "c" hat RGB-Wert (1,1,0) (also reines Gelb).
- `Integer i = new Integer (1);`
→ ein Wrapper-Objekt für eine Integer-Zahl wird mit 1 initialisiert
- `String str1 = new String ();`
`StringBuffer str2 = new StringBuffer ();`
→ Jeweils Konstruktor mit leerer Parameterliste.
(Semantik: Die Zeichenkette ist leer.)

Definition eines Konstruktors

- Für zusammengesetzte Objekte werden Konstruktoren verwendet, um das Objekt zu initialisieren.
 - d.h. es werden Werte berechnet, mit denen jede Variable des zusammengesetzten Objekts initialisiert wird
 - im einfachsten Fall gibt es ein Argument für jeden Wert des zusammengesetzten Objekts
 - die Initialisierung besteht dann darin, daß jedem Wert des Objektes einer der an den Konstruktor übergebenen Werte zugeordnet wird
 - Komplexere Konstruktoren sind natürlich möglich
- Wird kein Konstruktor angegeben, so wird automatisch ein leerer Konstruktor angelegt
 - initialisiert das Objekt mit Standard-Werten
- Mehr über Konstruktoren folgt später

Beispiel



- **Aufruf des Konstruktors:**

```
MeineKlasse x = new MeineKlasse(5,3.14,'a');  
// ein Objekt der Klasse MeineKlasse wird  
// angelegt und mit den Werten i = 5, d = 3.14  
// und c = 'a' initialisiert  
// werden dann mit x.i, x.d, x.c angesprochen
```

3.2.6. Arrays

- Eine spezielle Familie von Datentypen.
- Im Gegensatz zu den bisherigen eingebauten Typen sind Objekte von Arrays zusammengesetzt.
- *Hauptunterschied* zu zusammengesetzten Klassenobjekten:
 - ◇ Die **in einem Klassenobjekt** zusammengefassten Objekte können von **unterschiedlichen Typen** sein und werden mit **symbolischen Namen** (d.h. Identifiern) angesprochen.
 - ◇ Die **in einem Arrayobjekt** zusammengefassten Objekte müssen **alle von demselben Typ** sein (dem *Elementtyp* des Arrayobjekts) und werden mit **ganzzahligen Indizes** angesprochen.

Eingebaut, aber Referenz

In gewisser Weise sind die Arraytypen ein "Zwitter" zwischen eingebauten Typen und Klassen.

Das heißt:

- Eigentlich ist ein Arrayobjekt **ein eingebauter Typ** wie in anderen Programmiersprachen auch.
- *Aber*: Der Name eines Arrays ist wie bei Bausteintypen **nur eine Referenz**.

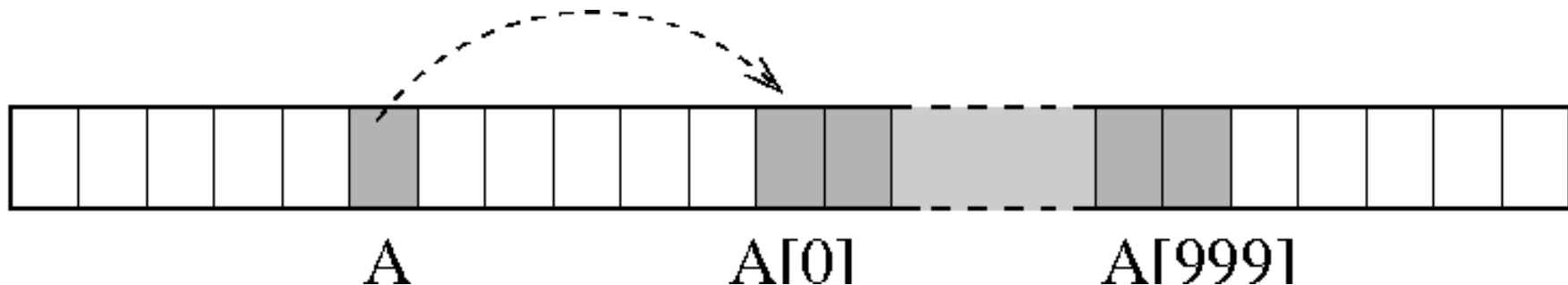
Konsequenz: Dieselben wie bei Klassen.

Beispiel:

```
int[] A = new int [1000];  
// Arrayobjekt erzeugt wie bei Klassenobjekten  
sort (A);  
// 'A' wird wie Klassenobjekt als Referenz uebergeben  
// und kann daher in einer Methode veraendert  
// (z.B. sortiert) werden.
```

Details

- Zu jedem beliebigen eingebauten und Klassentyp kann man Arrays mit diesem Typ als Elementtyp einrichten.
- Der Indexbereich eines Arrays mit n Elementen ist immer das Intervall $[0, 1, 2, \dots, n - 1]$.
- Syntax zum Ansprechen eines einzelnen Elements eines Arrayobjekts:
 $A[i]$
→ Dieser Ausdruck spricht das Element mit Index i (also das $(i+1)$ -te!) im Arrayobjekt namens A an.



Einfügen neuer Elemente

- Es können grundsätzlich keine neuen Elemente in ein Arrayobjekt eingefügt oder Elemente aus einem Arrayobjekt entfernt werden.
 - Der **Indexbereich** eines Arrayobjekts ist **unveränderlich**.

Vorgreifende Bemerkung:

Es gibt in Java zusätzlich noch eine Klasse namens `java.util.Vector`, die

- dieselbe Funktionalität wie Arrays bietet,
- aber zusätzlich noch das Einfügen und Löschen und einige weitere bequeme Zusatzfunktionalität bietet.

Hauptnachteil:

- Höhere Laufzeit für die einzelnen Komponentenzugriffe.

3.2.7 Scope und Lebenszeit

- *Erinnerung*: Klammern dürfen immer nur strikt paarweise auftreten.
- *Grundregel*: Eine Variable (bzw. Konstante) darf nur innerhalb von geschweiften Klammern ”{ . . . }“ deklariert werden.
- Der *Scope* einer Variable ist in der Regel der Bereich von ihrer Deklaration bis zur schließenden Klammer des ersten *umschließenden Blocks* { . . . }.
- *Wichtigste Ausnahmen*: Der Scope
 - ◇ einer Variable deklariert im Kopf einer `for`-Schleife oder
 - ◇ einem Parameter einer Methodeendet mit dem Ende der `for`-Schleife bzw. Methode.

Code Beispiel

```
public void F ( int a )
{
    int b = 1;
    if ( a == b )
    {
        for ( int i=0; i<10; i++ )
        {
            int c = 2;
        } // <- Scope-Ende von c und i
        int d = 3;
        {
            int e = 4;
        } // <- Scope-Ende von e
        int f = 5;
        {
            int g = 4;
        } // <- Scope-Ende von g
    } // <- Scope-Ende von d und f
} // <- Scope-Ende von a und b
```


Lebenszeit

- Eine Variable existiert, solange die Abarbeitung des Programms in ihrem Scope ist.
- Eine Komponente eines Array- oder Klassenobjektes existiert, solange das Gesamtobjekt existiert.

Achtung:

- Wenn der **Scope** einer Variablen **verlassen und wieder betreten** wird, wird die Variable
 - ◇ nicht nur **wieder eingerichtet**,
 - ◇ sondern auch **neu initialisiert**.
- Der **alte Wert**, den die Variable beim Verlassen des Scopes hatte, ist **verloren**.
- Das Objekt, auf das eine Variable eines Klassentyps verweist, kann durchaus länger als die Variable selbst leben.

Beispiel

```
int a = 1;
for ( int i=0; i<10; i++ )
{
    System.out.println(a);
    a++;
}
System.out.println(a);
```

a wird angelegt und mit 1 initialisiert.

a wird bei jedem Schleifendurchlauf ausgegeben und um 1 erhöht

a wird ausgegeben (a = 11)

```
for ( int i=0; i<10; i++ )
{
    int a = 1;
    System.out.println(a);
    a++;
}
System.out.println(a);
```

a wird bei jedem Schleifendurchlauf neu angelegt und mit 1 initialisiert.

Der Wert von a wird ausgegeben (jedes Mal 1).

a wird um eins erhöht. Das hat allerdings keinen Effekt.

a ist außerhalb des Scopes → Compiler-Fehler

Beispiel mit Referenzen

```
public class MeineKlasse
{
    public int i;
    public double d;
    public char c;
}
```

```
for ( int i=0; i<10; i++ )
{
    MeineKlasse a = new MeineKlasse();
    a.i++;
    System.out.println(a.i);
}
System.out.println(a.i);
```

Bei **jedem Schleifendurchlauf** wird ein Verweis **a** auf ein Objekt von Typ **MeineKlasse** angelegt.
Die Komponenten **a.i**, **a.d** und **a.c** werden **jedes Mal** mit den Standard-Werten initialisiert.

a ist außerhalb des Scopes
→ Compiler-Fehler

Beispiel mit Methoden

```
public void f ()  
{  
    MeineKlasse meinVerweis = new MeineKlasse();  
    // meinVerweis.i == 0  
    System.out.print (meinVerweis.i);  
    meinVerweis.i++;  
    System.out.print (meinVerweis.i);  
}
```

...

f ();

f ();

Ausgabe: 01

Ausgabe:
ebenfalls 01
(nicht 12 !!)

Beispiel für Rückgabe eines Objekts

```
public String englischerNikolaus ()
{
    String str = new String ( "Santa Claus" );
    return str;
}

...

String nicksName = englischerNikolaus();
System.out.println ( nicksName );
                    // ^^^^^^^^^^ "Santa Claus"
```

Erläuterungen:

- Die Zeichenkette `Santa Claus` ist zwar über die Variable `str` erzeugt worden,
- und die Variable `str` beendet ihre Existenz mit dem Ende der Abarbeitung der Methode `englischerNikolaus`,
- aber die Zeichenkette existiert darüber hinaus.

3.2.8 Garbage Collection

Erinnerung:

- Variablen von Klassen
 - ◊ bezeichnen nicht die Objekte selbst,
 - ◊ sondern nur *Referenzen* auf die eigentlichen Objekte,
 - ◊ und die eigentlichen Objekte müssen mit `new` erst noch explizit angelegt werden.
- Bei der Abarbeitung eines Programms arbeitet im Hintergrund immer ein Laufzeitsystem mit.

Weitere Aufgabe des Laufzeitsystems:

- Verwaltung eines "Pools" von Speicherplatz.
- Jedes `new` ist eine Anfrage an diese Poolverwaltung.

Abarbeitung von `new`

- Ein Ausdruck "`new X . . .`" liefert als Rückgabe einen Verweis auf ein Objekt der Klasse `X` zurück.
- Falls die Poolverwaltung momentan ausreichend Speicherplatz zur Verfügung hat,
 - ◊ wird wie gewünscht ein neues Objekt erzeugt,
 - ◊ seine Adresse wird als Wert des `new`-Ausdrucks zurück geliefert
 - ◊ und kann daher mit Operator `=` einer Variablen der zugehörigen Klasse zugewiesen werden.
- Falls der Speicherplatz hingegen **nicht** ausreicht, tritt ein Fehlermechanismus in Aktion.
- Nach momentanem Stand der Vorlesung bedeutet das: unvermeidbarer Programmabsturz.
- *Später* in Vorlesung und Übungen: Vermeidung des Programmabsturzes durch *Exceptions*.

Frei gewordener Speicher

```
String str1 = new String ( "Hallo" );  
String str2 = str1;  
  
...  
str1 = new String ( "Holla" );  
str2 = str1;
```

- Am Schluss gibt es keinen Verweis auf die zuerst erzeugte Zeichenkette "Hallo" mehr.
- Beide Variable, die zuerst auf diese Zeichenkette verwiesen haben, sind ja später auf die Adresse einer anderen Zeichenkette umgesetzt worden.
- Der für "Hallo" reservierte Speicherplatz ist nun vom Programm aus nicht einmal mehr erreichbar und daher völlig nutzlos.
- Er steht dem Laufzeitsystem aber nicht für die Bedienung weiterer Anfragen mit `new` zur Verfügung.

Krasses Beispiel

```
String str;  
while ( true )  
    str = new String ("Hallo");
```

Erläuterung:

- Mit `while` wird bekanntlich eine Schleife eingeleitet, die solange durchlaufen wird, bis die logische Bedingung in Klammern falsch (`==false`) wird.
- Das Literal `true` wird natürlich niemals falsch.
→ Endlosschleife.
- Es wird also endlos neuer Speicherplatz eingerichtet.

Frage:

Ist Programmabsturz damit nicht vorprogrammiert?

Mögliche Gegenstrategie

- Neben `new`-Anweisungen gibt es noch ein weiteres Konstrukt, um Speicherplatz wieder an die Poolverwaltung zurückzugeben.
- Zum Beispiel `delete` in C++.

Beispielhafter C++-Code:

```
char* str;           // Referenzvariable ("Pointer")
str = new char[100]; // Speicherplatz fuer 100 Zeichen
...
delete [] str;      // Wieder freigegeben
```

- Ähnliche Konstrukte gibt es in Pascal, C, Ada...
- Aber zum Beispiel nicht in Java!

Probleme

- In komplexeren Programmen
 - ◊ können ein `new` und das zugehörige `delete` potentiell sehr weit auseinanderliegen,
- *Praktisch unvermeidliches Resultat*: Schwer zu findende Programmierfehler mit beliebig üblen Konsequenzen.
 - ◊ Das `delete` wird oft **vergessen**.
 - Wenn oft genug Speicherplatz angefordert, aber nicht zurückgegeben wird, geht irgendwann gar nichts mehr.
 - ◊ Ein Stück Speicherplatz, das mit `delete` wieder freigegeben (und vielleicht schon weiterverwendet!) wurde, wird **aus Versehen weiter benutzt** oder ein weiteres Mal mit `delete` freigegeben.
 - Programmabsturz wäre nicht das Schlimmste, was passieren könnte...

Andere Strategie: Garbage Collection

- Das Laufzeitsystem startet hin und wieder im Hintergrund einen zusätzlichen **Prozess**, der
 - ◊ alle momentan reservierten **Speicherbereiche absucht**, ob sie vom Programm über Referenzen überhaupt noch erreichbar sind
 - ◊ jedes als nicht mehr erreichbar klassifizierte Stück Speicherplatz an die Poolverwaltung **zurückgibt**.
- Stichwort in der Literatur: *Garbage Collection*.
- *Ergebnis*: Das Problem ist fast gelöst.
 - Warum nur fast: Man kann natürlich immer noch zuviel Speicherplatz anlegen, ohne dass ein einziges Byte davon unerreichbar wird. (Beispiel folgt später bei Listen)

Wie lange existiert ein mit `new` erzeugtes Objekt?

- Das Objekt existiert mindestens noch solange, wie es eine "Kette" von Verweisen gibt, über die man das Objekt vom Programm aus ansprechen kann.
- Wenn die letzte solche Kette "abreißt", existiert das Objekt zunächst einmal weiter.
- Erst wenn der *Garbage Collector* das nächste Mal aktiv wird, vernichtet er das Objekt.
- Das passiert zu einem Zeitpunkt den der Java-Programmierer weder vorhersehen noch beeinflussen kann.