

4.2.1. Was sind Klassen?

- Eine Klasse ist ein Datentyp.
- Ein Datentyp ist generell charakterisiert durch
 - ◇ die abstrakten Zustände, die ein Objekt dieses Typs annehmen kann, und
 - ◇ die Zugriffsmöglichkeiten, mit denen der Zustand eines Objekts gelesen und/oder verändert werden kann.

Beispiel 1: Datentyp `int`

- Die abstrakten Zustände, die ein Objekt vom Datentyp `int` annehmen kann, sind alle ganzen Zahlen zwischen der kleinsten und der größten mit `int` (32 Bit) darstellbaren Zahl.

- Abstrakte vs. konkrete Zustände:

Die *konkreten* Zustände sind die verschiedenen Bitmuster aus 32 Bit.

- Die Zugriffsmöglichkeiten zum Verändern des Zustands einer `int`-Variable sind Zuweisungen mit `=`, `+=`, `*=` usw. sowie `++` und `--`:

```
int i = 1;
```

- Zum Lesen des Zustands eines `int`-Objekts `i` schreibt man den Namen des Objekts einfach hin:

```
int j = i + 1; // Zustand von 'i' gelesen
```

Beispiel 2: Klasse `StringBuffer`

- Die abstrakten Zustände sind alle möglichen Zeichenketten aus Unicode–Zeichen einschließlich der "leeren" Zeichenkette ohne Zeichen.
- Die konkreten Zustände im Innern eines `StringBuffer`–Objekts konstituieren sich im wesentlichen in
 - ◊ den konkreten Bitmustern der einzelnen Unicode–Zeichen und
 - ◊ der Art und Weise, wie Zeichenketten intern organisiert sind
- Verändernde Zugriffsmöglichkeiten:
 - ◊ Zum Beispiel die zehn Methoden mit Namen `append`.
 - ◊ Analog dazu gibt es z.B. auch neun verschiedene Methoden `StringBuffer.insert`, mit denen neuer Text irgendwo mitten in der Zeichenkette eingefügt werden kann.

Lesende Zugriffsmöglichkeiten auf StringBuffer

- Insbesondere (aber nicht nur) mit Methode `StringBuffer.charAt` kann man lesend auf ein `StringBuffer`-Objekt zugreifen.
- *Verwendung*: Der Ausdruck `str.charAt(i)`
 - ◊ hat `char` als Rückgabebetyp,
 - ◊ und der Rückgabewert ist das Zeichen mit Index `i` in der momentan im `StringBuffer`-Objekt `str` gehaltenen Zeichenkette.
- *Index*: Wie bei Arrays, d.h. das erste Zeichen hat Index 0 usw.
- Eine völlig identische Methode `charAt` ist übrigens auch für Klasse `String` definiert.

```
String str = new String("Hello");  
System.out.print(str.charAt(4)); // -> "o"
```

Variablen und Konstanten in Klassen

```
public class MeineKlasse
{
    int n1;
    final int n2 = 1;
}
```

Erläuterungen:

- Der Unterschied zwischen Variablen und Konstanten (mit Schlüsselwort `final`) wurde bereits behandelt.
- Bei Komponenten von Klassen gibt es exakt dieselbe Unterscheidung mit exakt denselben Konsequenzen:
 - ◊ Eine konstante Komponente muss sofort initialisiert werden.
 - ◊ Der Wert einer konstanten Komponente darf nach der Initialisierung nicht mehr geändert werden.

4.2.2. Klassen- vs. Objektvariable Klassen- vs. Objektkonstanten

- Wie bei Methoden gibt es auch bei den Datenkomponenten einer Klasse die Unterscheidung zwischen
 - ◊ Klassen- und Objektvariable bzw.
 - ◊ Klassen- und Objektkonstanten (also mit Schlüsselwort `final`).
- Syntaktische Unterscheidung:

Analog zu Methoden durch Schlüsselwort `static` bei Klassenvariablen bzw. -konstanten vor der Angabe des Datentyps.

Semantischer Unterschied

- Eine Klassenvariable/-konstante ist ein einzelnes Objekt.
- Eine Objektvariable/-konstante gibt es einmal pro Objekt der Klasse.
- Die Bestandteile von Objekten sind also genauer gesagt Objektvariable.
- Analog zu Klassenmethoden kann man auf Klassenvariablen auch ohne konkretes Objekt der Klasse zugreifen.

Beispiel

```
public class MeineKlasse
{
    public int n1;           // Objektvariable
    public static int n2;   // Klassenvariable
}
```

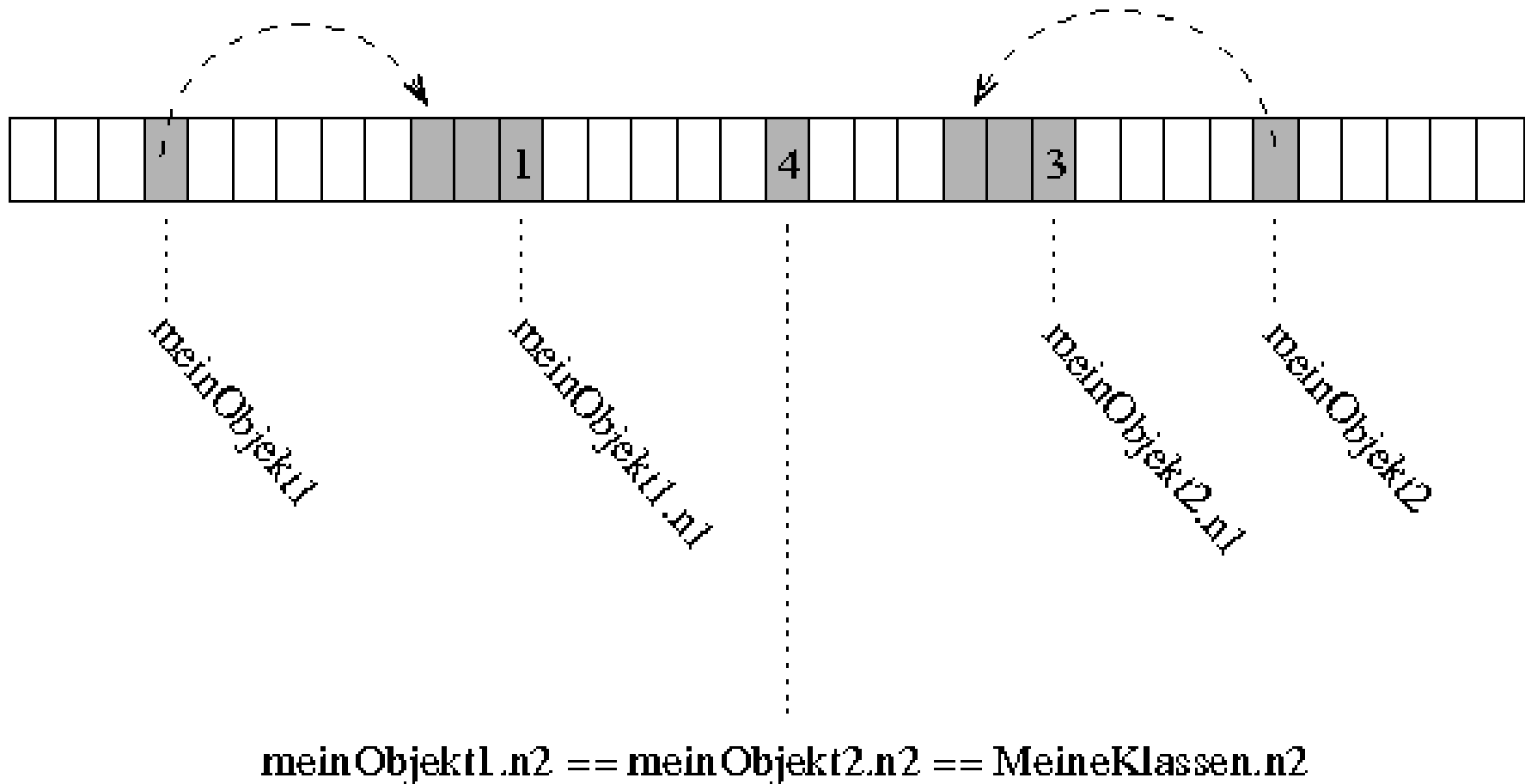
...

```
MeineKlasse meinObjekt1 = new MeineKlasse();
MeineKlasse meinObjekt2 = new MeineKlasse();
```

```
meinObjekt1.n1 = 1;
meinObjekt1.n2 = 2;
meinObjekt2.n1 = 3;
meinObjekt2.n2 = 4;
```

```
System.out.println ( meinObjekt2.n1 );           // -> 3
System.out.println ( meinObjekt2.n2 );           // -> 4
System.out.println ( meinObjekt1.n1 );           // -> 1
System.out.println ( meinObjekt1.n2 );           // -> 4 (!)
System.out.println ( MeineKlasse.n2 );           // -> 4 (!)
```


Veranschaulichung



Erläuterungen

- `meinObjekt1.n1` und `meinObjekt2.n1` sind **zwei separate `int`-Objekte**, die Bestandteile der Objekte `meinObjekt1` bzw. `meinObjekt2` sind.
- `MeineKlasse.n2` ist im Gegensatz dazu nur **ein einzelnes, isoliertes `int`-Objekt**, das ein einziges Mal irgendwo im Speicher angelegt wird und für alle Objekte des Typs `MeineKlasse` gilt.
 - `meinObjekt1.n2` und `meinObjekt2.n2` bezeichnen dasselbe `int`-Objekt.
 - daher ist eine Änderung für alle `MeineKlasse`-Objekte gültig
- Die letzte Zeile auf der vorherigen Folie zeigt, wie man ohne ein Objekt von `MeineKlasse` auf dieses einzelne Objekt `n2` zugreifen kann.
 - Vgl. Benutzung von Klassenmethoden ohne Objekt

Realisierung

- Beim Übersetzen interpretiert der Compiler die beiden Ausdrücke

`meinObjekt1.n1` und `meinObjekt1.n2`

völlig verschieden.

- **In beiden Fällen** konstruiert der Compiler Java Byte Code, der die Adresse des Objektes

`meinObjekt1.n1` bzw. `meinObjekt1.n2`

berechnet.

- **Bei `meinObjekt1.n1`** wird die Adresse berechnet, indem die Position von `n1` in `MeineKlasse` auf den Wert von `meinObjekt1` aufaddiert wird.
- **Bei `meinObjekt1.n2`** wird eine globale Adresse direkt eingesetzt.
 - Der Compiler hat sich natürlich irgendwo intern die Adresse von `MeineKlasse.n2` gemerkt.

Klassen-Konstanten

- sind unveränderbar wie alle Konstanten (daher `final`)
- sind identisch für alle Objekte der Klasse (daher `static`)

```
public class Nonsens
{
    public          int a = 1;           // Ok
    public final   int b = 1;           // Ok
    public         int c;               // Ok
    public final   int d;               // Fehler!

    public static  int e = 1;           // Ok
    public static  final int f = 1;     // Ok
    public static  int g;               // Ok
    public static  final int h;         // Fehler!
}
```

Klassen-Konstanten in der Standardbibliothek

- Die Kreiszahl $\pi = 3.14159$ ist selbstverständlich reellwertig und konstant (also `final double`):

```
java.lang.Math.PI
```

- Die wichtigsten Farben sind als Konstanten vom Typ `java.awt.Color` mit den entsprechenden RGB–Werten schon in der Klasse `java.awt.Color` definiert (vgl. diverse Übungsaufgaben):

- ◊ `java.awt.Color.red`
- ◊ `java.awt.Color.yellow`
- ◊ `java.awt.Color.green`
- ◊ **USW.**

Implementation der Farb-Objekte

```
public class Color
{
    static final Color red      = new Color(255,  0,  0);
    static final Color yellow  = new Color(255, 255, 0);
    static final Color green   = new Color(  0, 255, 0);
    ...
}
```

-
- `red` ist logisch gesehen konstant → `final`
 - `red` ist immer und überall gleich → `static`

Weiteres Beispiel

System.out.print(ln):

- `java.lang.System.out`: Eine Klassenvariable der Klasse `java.lang.System`.
→ Daher verwendbar in der Form `System.out`.

- **Erinnerung:**

`java.lang.*` wird immer automatisch importiert, deswegen kann man für `java.lang.System.out.print()` auch `System.out.print()` verwenden

- Der Typ von `java.lang.System.out` ist `java.io.PrintStream`.

Typ von `System.out`

`java.io`

- Sammlung von Standard-Klassen für Eingaben von Tastatur und Files und für Ausgaben auf `xterm`-Fenster und Files.
- I/O = Input/Output = Ein-/Ausgabe.

`java.io.PrintStream`

- spezielle Klasse zur Datenausgabe.
- bietet unter anderem Methoden namens `print` und `println` zur Ausgabe.
- Diese Methoden sind für alle eingebauten Typen und einige gängige Klassentypen wie `String` als Parametertypen überladen

.

Zugriff auf Objektmethoden mit `this`

- Das Schlüsselwort `this` ist in einer Objektmethode einer Klasse ein **Verweis auf das Objekt, mit dem diese Methode aufgerufen wurde**.
 - Mit `this` kann insbesondere auf eine Klassen- oder Objektvariable/-konstante zugegriffen werden (`this.n`).
- Da dies ein extrem häufiger Fall ist, darf man `this` dann auch weglassen (d.h. `this.n==n`).
- *Aber:*
 - ◇ Der Name einer Klassen- oder Objektvariable bzw. -konstante kann innerhalb der Methode für ein gänzlich anderes Objekt noch einmal vergeben werden.
 - ◇ Die Deklaration eines solchen Objekts "überdeckt" die Klassen- bzw. Objektvariable/-konstante.
 - Letztere kann von da an *nur* noch mit Hilfe von `this` angesprochen werden.

Beispiel

```
public class MeineKlasse2 {
```

```
    int n = 1;
```

```
    public void nocheinemethode( int n ) {
```

```
        n = 27;
```

```
        System.out.print( n );
```

```
        System.out.print( " " );
```

```
        System.out.print( this.n );
```

```
    }
```

```
}
```









```
...
```

n bezieht sich auf die Variable im momentanen Scope

this.n bezieht sich auf die Variable im momentanen Scope

```
MeineKlasse2 meinObjekt2 = new MeineKlasse2();  
meinObjekt2.nocheinemethode( 12 ); // -> "27 1"
```

Beispiel

```
public class MeineKlasse {  
    public int n = 2;      
    public void meineMethode1 () {  
        int n = 7;   
        System.out.print ( n );  
        System.out.print ( " " );  
        System.out.println ( this.n );  
    }  
    public void meineMethode2 () {  
        (this.n)++;   
        System.out.println ( n );   
        System.out.print ( " " );  
        System.out.println ( this.n );   
    }  
}  
  
...  
MeineKlasse meinObjekt = new MeineKlasse ();  
meinObjekt.meineMethode1 (); // -> "7 2"  
meinObjekt.meineMethode2 (); // -> "3 3"  
meinObjekt.meineMethode2 (); // -> "4 4"  
meinObjekt.meineMethode1 (); // -> "7 4"
```

Verwendungszweck

- Man braucht `this` natürlich nicht wirklich, da man den Variablen verschiedene Namen geben könnte
- Aber manchmal erleichtert es die Programmierarbeit und verbessert die Lesbarkeit des Programms.
- *Konkret*: Manchmal ist es einfach unnatürlich, verschiedenen Variablen, die im Konflikt zueinander stehen, unterschiedliche Namen zu geben.
- *Beispiel*:
 - ◊ Eine Kreisklasse, in der Kreise durch Mittelpunkt und Radius gegeben sind.
 - ◊ Die entsprechenden Datenkomponenten sollten dann sinnvollerweise auch `x`, `y` und `radius` heißen.
 - ◊ In einer Methode `setzeKreis` können die Parameter aber ebenfalls natürlicherweise `x`, `y` und `radius` heißen.

Beispiel: Klasse für Kreis

```
public class Kreis {  
    private double x;  
    private double y;  
    private double radius;  
  
    public Kreis ( double x,  
                  double y,  
                  double radius ) {  
  
        this.x      = x;  
        this.y      = y;  
        this.radius = radius;  
    }  
  
    public Kreis ( Kreis k ) {  
        x      = k.x;  
        y      = k.y;  
        radius = k.radius;  
    }  
}
```

Konstruktor mit Variablen
zur Initialisierung

Überlagerter Konstruktor
mit einem anderen Kreis-
Objekt zur Initialisierung
(→ *Copy-Konstruktor*)

Klassenmethoden und `this`

- *Erinnerung*: Zum Aufruf einer Klassenmethode bedarf es keines Objekts der Klasse.
- Objektvariable und -konstante gehören aber per Definition zu konkreten Objekten.
- Daher wäre es semantischer Unsinn, wenn eine Klassenmethode
 - ◊ mit `this` auf dieses nicht unbedingt existierende Objekt selbst oder
 - ◊ mit oder ohne `this` auf die Objektvariablen und -konstanten dieses Objekts zugreifen oder
 - ◊ eine andere Objektmethode mit diesem Objekt aufrufen dürfte.

Beispiel

```
public class MeineFehlerhafteKlasse
{
    public      int n1;
    public static int n2;

    public static void meineFehlerhafteKlassenMethode1 ()
    {
        System.out.println ( n1 );
    }

    public static void meineFehlerhafteKlassenMethode2 ()
    {
        System.out.println ( this.n1 );
    }
}
```

- beide Aufrufe sind **falsch**, da eine Klassenmethode nicht auf einem Objekt arbeitet
 - und daher die Komponente `n1` keinen Wert hat!

Beispiel

```
public class MeineFehlerhafteKlasse
{
    public          int n1;
    public static  int n2;

    public static void meineFehlerhafteKlassenMethode1 ()
    {
        System.out.println (n2);
    }

    public static void meineFehlerhafteKlassenMethode2 ()
    {
        System.out.println (this.n2);
    }
}
```

- der erste Aufruf ist **korrekt**, da `n2` eine Klassenvariable ist!
 - Klassenvariable sind identisch für alle Objekte einer Klasse
- der zweite Aufruf ist **falsch**, da für eine Klassenmethode kein Objekt definiert ist
 - und daher auch `this` keinen Wert hat!

Klassenmethoden und Objektvariablen

- Klassenmethoden können zwar **nicht direkt** bzw. mittels `this` auf ein Objekt der eigenen Klasse zugreifen
- Andererseits spricht aber nichts dagegen (und ist auch absolut korrekt), wenn eine Klassenmethode auf die Objektvariablen, -konstanten und -methoden **eines benannten Objekts** derselben Klasse zugreift

Beispiel

```
public class MeineKorrekteKlasse {  
    public          int n1;  
    public static  int n2;  
  
    public void meineObjektMethode () {  
        System.out.println ( n1 );  
        System.out.println ( n2 );  
    }  
}
```

Klarerweise erlaubt,
da Objektmethode

```
public static void meineKorrekteKlassenMethode  
    ( MeineKorrekteKlasse weiteresObjekt ) {  
    System.out.println ( n2 );  
    System.out.println ( weiteresObjekt.n1 );  
    System.out.println ( weiteresObjekt.n2 );  
    weiteresObjekt.meineObjektMethode ();  
    }  
}
```

erlaubt, da Klassen-Methoden auf Klassen-Variablen zugreifen dürfen (ein Zugriff auf `n1` wäre hier nicht erlaubt).

alle drei auch in der Klassenmethode erlaubt, da `weiteresObjekt` ein Zeiger auf ein bereits definiertes Objekt ist.

4.2.3. Konstruktoren

Wir haben Konstruktoren bereits kennen gelernt

- Zweck:
 - Konstruktoren dienen dazu, um anzugeben, wie ein Objekt initialisiert werden soll
- syntaktisch werden sie wie Objektmethoden definiert, allerdings
 - haben sie keinen Rückgabotyp
 - müssen gleich heißen wie die Klasse
- Eine Klasse kann beliebig viele Konstruktoren haben
 - bei keinem Konstruktor wird ein Default-Konstruktor angelegt
 - bei einem Konstruktor muß dieser verwendet werden
 - es können aber auch mehrere Konstruktoren definiert werden (Überlagerung)

Erzwungener Aufruf von Konstruktoren

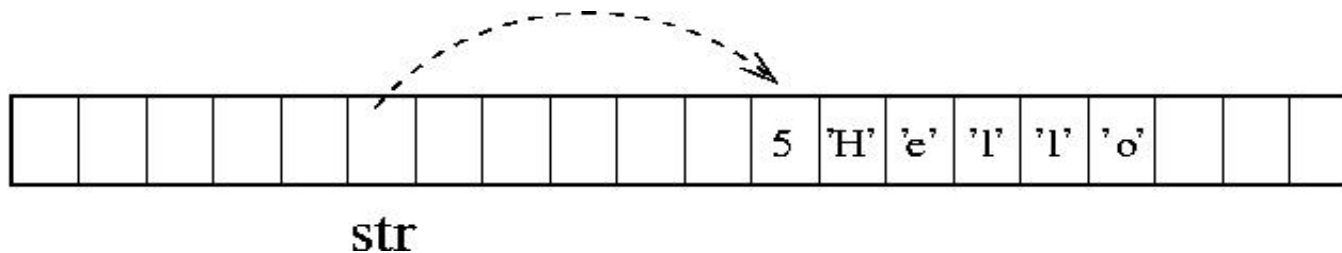
- Wenn eine Klasse einen oder mehrere Konstruktoren hat, dann *muss* einer davon bei der Erzeugung eines Objekts dieser Klasse mit `new` aufgerufen werden.
 - Ansonsten gibt es einen Fehler beim Kompilieren.
- Bei der Implementation einer Klasse kann man mit diesem Mechanismus also **erzwingen**, dass die Initialisierung eines Objekts niemals vergessen werden kann:
 - ◊ Man gibt der Klasse eben einen oder mehrere Konstruktoren, die ein mit `new` neu eingerichtetes Objekt der Klasse **adäquat initialisieren**.
 - ◊ Solange man bei der Einrichtung eines Objekts der Klasse keinen dieser Konstruktoren benutzt, liefert der Compiler eine Fehlermeldung.

Adäquate Initialisierung

- Ist "adäquate Initialisierung" wirklich so wichtig, dass man in Java (und anderen Programmiersprachen) feste Regeln einführen muss?

Antwort durch einfaches Beispiel:

- *Erinnerung:* Es gibt verschiedene Möglichkeiten, die Klasse `java.lang.String` intern zu realisieren.
- Betrachten wir zum Beispiel die erste Variante:



- Bei der Initialisierung des String-Objekts mit "Hello" muss unbedingt gewährleistet sein, dass die Anzahl der Zeichen als 5 initialisiert wird.
→ der Konstruktor kann das automatisch übernehmen, sodaß der Programmierer gar nicht wissen muß, wie Strings gespeichert werden

Partielle Initialisierung der Komponenten

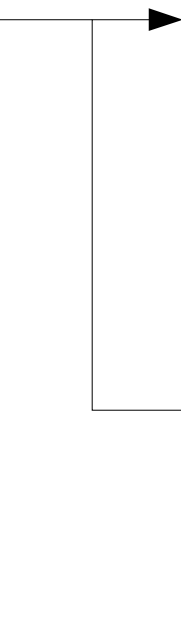
- *Erinnerung*: Wird eine Variable bei der Deklaration nicht sofort initialisiert, dann wird ihr Inhalt automatisch auf einen datentypspezifischen Null-Wert gesetzt.
- Das Gleiche gilt natürlich auch für die Datenkomponenten eines Objekts, das mit `new` eingerichtet wurde.
- Erst nach dieser Initialisierung auf Null-Werte wird der Konstruktor aufgerufen.
- *Konsequenz*: Falls der Konstruktor nur einen Teil der Datenkomponenten explizit initialisiert, sind die anderen Datenkomponenten nicht uninitialisiert.
→ sondern mit Standard-Werten initialisiert
- Insbesondere initialisiert der Default-Konstruktor alle Komponenten mit Standardwerten

Verschachtelte Konstruktoren

- Man kann einen Konstruktor einer Klasse auch in einem anderen Konstruktor aufrufen.
- *Syntax*: `this` steht vor der Parameterliste.
- Dieser Aufruf eines Konstruktors in einem zweiten Konstruktor muss die allererste Anweisung im zweiten Konstruktor sein!
 - Sonst Fehlermeldung vom Compiler!
- Es gibt insgesamt nur **drei Möglichkeiten** überhaupt, wie man **einen Konstruktor aufrufen** kann:
 - ◇ Bei der Einrichtung eines Objektes **mit `new`**.
 - ◇ In der allerersten Zeile eines anderen Konstruktors derselben Klasse.
 - **mit `this`** (Was wir gerade betrachten)
 - ◇ In der allerersten Zeile einer abgeleiteten Klasse.
 - **mit `super`**. Kommt später

Beispiel

```
class Bla {  
    private int i;  
    private double d;  
    private char c;  
  
    public Bla ( int i, double d, char c ) {  
        this.i = i;  
        this.d = d;  
        this.c = c;  
    }  
  
    public Bla ( int i ) {  
        this ( i, 3.14, 'a' );  
    }  
  
    public Bla ( int i, char c ) {  
        this ( i, 3.14, c );  
    }  
}
```



Konstruktor 1: initialisiert alle drei Komponenten

Konstruktor 2: initialisiert die Integer, verwendet fixe Werte für die anderen Komponenten

Konstruktor 3: initialisiert die Integer und Char-Komponenten, verwendet einen fixen Wert für d