TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 03c: Systematische Abrundung bisheriges Java: Methoden

Karsten Weihe

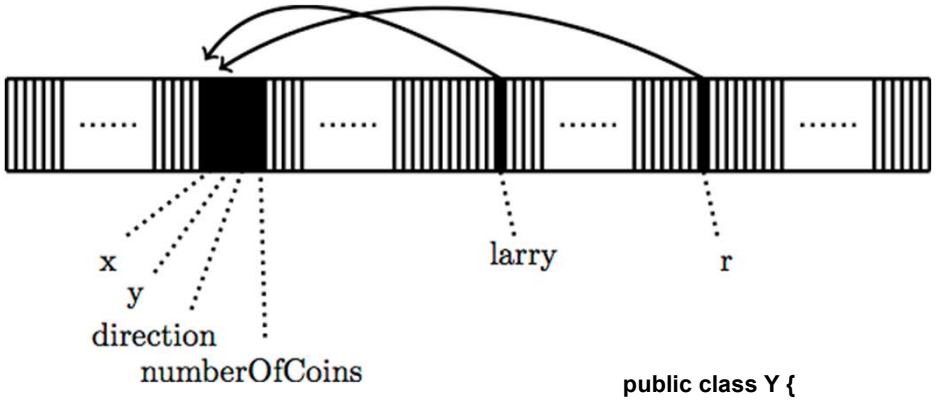
TECHNISCHE UNIVERSITÄT DARMSTADT

Referenzen und Objekte
bei
Methodenaufrufen

Mit diesen Begrifflichkeiten können wir dann leichter über das nun folgende grundlegende Thema reden.

Parameter von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT



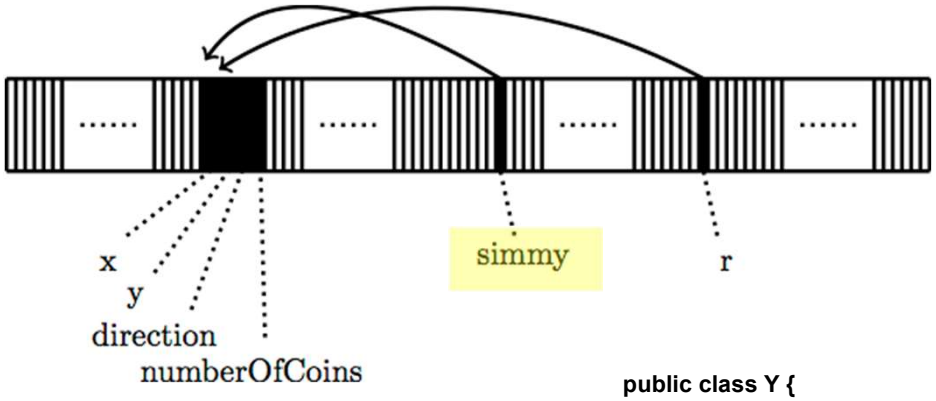

```
Y demo = new Y();
Robot larry = new Robot ( ..... );
demo.m ( larry );
```

```
public class Y {
    public void m ( Robot r ) {
        r.move();
    }
}
```

Wenn ein Parameter einer Methode von einem Referenztyp ist, dann verweist der formale Parameter innerhalb der Methode auf dasselbe Objekt wie der aktuale Parameter außerhalb der Methode. Beziehungsweise wenn der aktuale Parameter den symbolischen Wert null hat, dann hat auch der formale Parameter den Wert null. In der Initialisierung eines formalen Parameters durch den aktualen Parameter bei Aufruf der Methode wird bei Referenztypen also wieder nur die Adresse kopiert, nicht das Objekt.

Parameter von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT



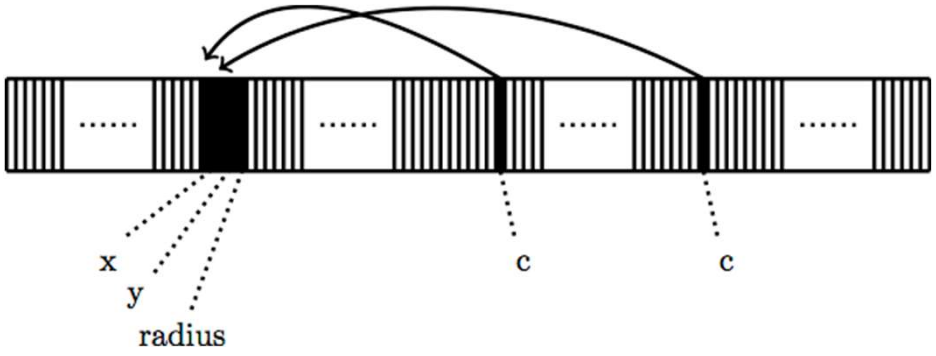

```
Y demo = new Y();
SymmTurner simmy = new SymmTurner ( ..... );
demo.m ( simmy );
```

```
public class Y {
    public void m ( Robot r ) {
        r.move();
    }
}
```

Wenn wir statt dessen ein Objekt einer abgeleiteten Klasse hineinstecken, ändert sich das Bild nicht prinzipiell.

Parameter von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT



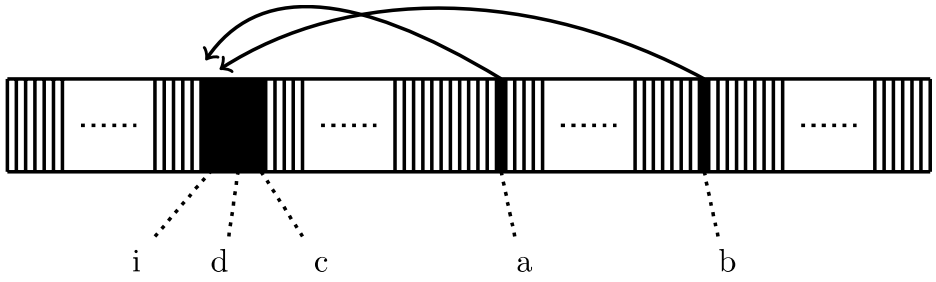

```
Y demo = new Y();
Circle c = new Circle ( ..... );
demo.m ( c );
```

```
public class Y {
    public void m ( Circle c ) {
        c.paint();
    }
}
```

Hier noch ein anderes Beispiel für dieselbe Situation, diesmal aus unserem geometrischen Fallbeispiel. In diesem Beispiel nun haben der formale und der aktuale Parameter der Methode m von Klasse Y denselben Namen – rein zur beispielhaften Illustration dieses nicht seltenen Falles.

Parameter von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT



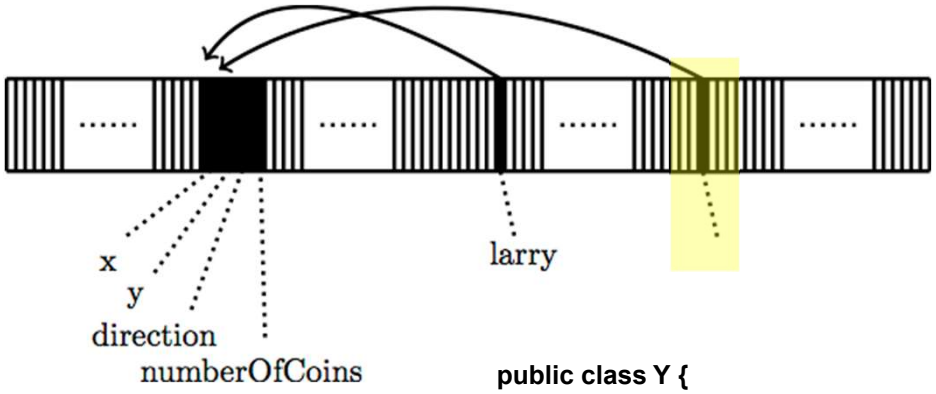

```
public class Y {
    public void m ( X a ) {
       a.d += a.i;
   }
}
```

```
Y demo = new Y();
X b = new X();
demo.m ( b );
```

Und noch einmal anhand unseres illustrativen Beispiels mit Klassen X und Y und Methode m.

Rückgaben von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT




```
Y demo = new Y();
Robot larry = demo.m ( 2, 3 );
larry.move();
```

```
public class Y {
    public Robot m ( int x, int y ) {
        return new Robot ( x, y, UP, 0 );
    }
}
```

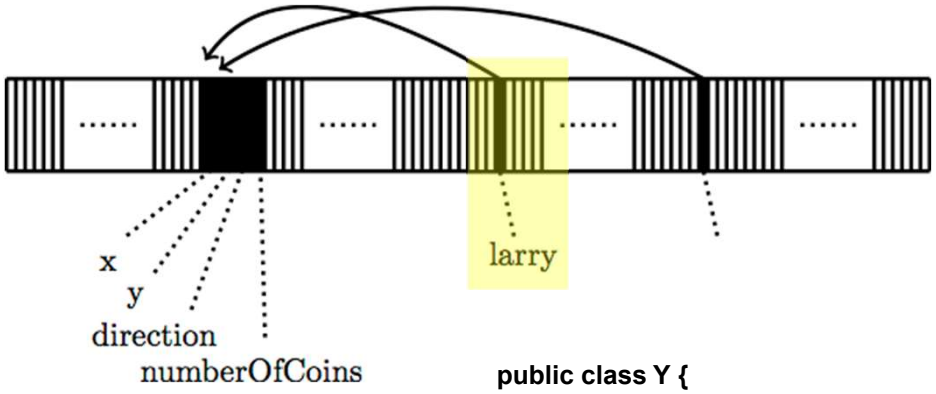
Eine return-Anweisung wird generell so umgesetzt, dass der Wert hinter dem Schlüsselwort return

an einer unbenannten Stelle im Computerspeicher hinterlegt wird, von der dieser Wert dann außerhalb der Methode gelesen und weiterverwendet werden kann. In der auf dieser Folie gezeigten Methode m wird die Adresse, die von Operator new zurückgeliefert wird, nicht in einer benannten Referenz zwischengespeichert, sondern gleich mittels return an diese unbenannte Stelle geschrieben.

***Nebenbemerkung*: Typischerweise wird der Rückgabewert in ein bestimmtes *Register* geschrieben.**

Rückgaben von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT




```
Y demo = new Y();
Robot larry = demo.m ( 2, 3 );
larry.move();
```

```
public class Y {
    public Robot m ( int x, int y ) {
        return new Robot ( x, y, UP, 0 );
    }
}
```

Von dieser festen Stelle wird sie dann außerhalb der Methode ausgelesen und durch die Zuweisung in die Referenz larry geschrieben. Die Referenz larry verweist also auf das Objekt, das in dem Aufruf von m rechts mit Operator new erzeugt worden ist.

TECHNISCHE UNIVERSITÄT DARMSTADT

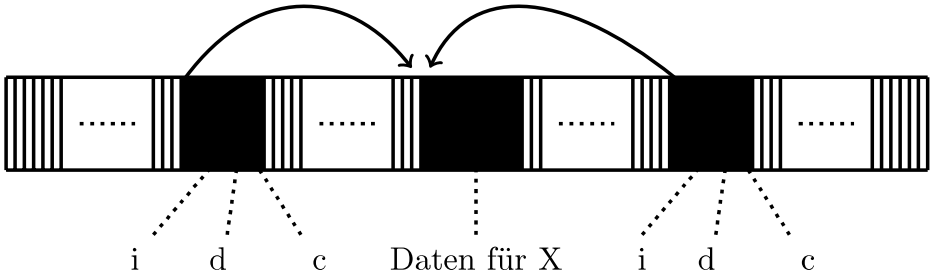
Verborgene Informationen

Wikipedia:
Tabelle virtueller Methoden / Virtual method table

***Erinnerung*: In 01f , Abschnitt „Allgemein: Methoden vererben / überschreiben“, sowie in Kapitel 01g, Abschnitt „PacmanRobot in Aktion“, hatten wir schon gesehen, dass es noch weitere Informationen in einem Objekt gibt, unter anderem die Methodentabelle. Das rekapitulieren wir hier noch einmal anhand weiterer Beispiele.**

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT




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

Vor unseren Augen verborgen, enthält jedes Objekt einer Klasse noch einen Verweis auf ein anonymes Objekt, das in jedem Programm für jede Klasse nur einmal eingerichtet wird.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Informationen
  - zur Klasse selbst
  - zu den Attributen der Klasse
  - zu den Methoden der Klasse
- Methodentabelle

Dieses anonyme Objekt enthält im Prinzip alle Informationen, die notwendig sind, um eine Klasse, ihre Attribute und ihre Methoden korrekt zu verwenden.

***Vorgriff*: Im Kapitel zu Polymorphie, Stichwort Reflection, werden wir sehen, wie man an diese Informationen herankommt und wofür das sinnvoll ist. Hier benötigen wir diese Informationen noch nicht.**

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Informationen
  - zur Klasse selbst
  - zu den Attributen der Klasse
  - zu den Methoden der Klasse
- Methodentabelle

Was wir hingegen dringend für das weitere Verständnis benötigen, ist die zweite Informationseinheit, die Methodentabelle. Die schauen wir uns auf den nächsten Folien genauer an.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public void m1() { ..... }
   public void m2() { ..... }
}

public class Y extends X {
   public void m2() { ..... }
   public void m3() { ..... }
}
```

Dazu sehen wir uns wieder zwei Klassen X und Y an, so dass Y von X abgeleitet ist. Was die Methoden genau machen, interessiert hier wieder nicht.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public void m1() { ..... }
  public void m2() { ..... }
}

public class Y extends X {
  public void m2() { ..... }
  public void m3() { ..... }
}
```

Die Methode m1 wird in X implementiert und an Y vererbt.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public void m1() { ..... }
  public void m2() { ..... }
}

public class Y extends X {
  public void m2() { ..... }
  public void m3() { ..... }
}
```

Die Methode m2 hingegen wird in Y überschrieben. Für m2 existieren also *zwei* Implementationen, eine in X und eine in Y.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

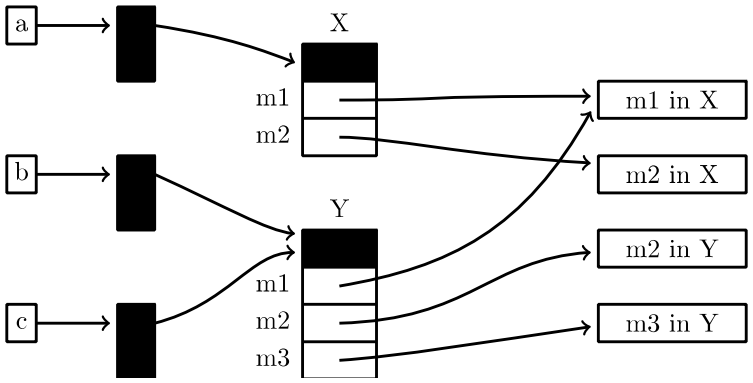
```
public class X {
   public void m1() { ..... }
   public void m2() { ..... }
}

public class Y extends X {
   public void m2() { ..... }
   public void m3() { ..... }
}
```

Die Methode m3 schließlich ist in X noch nicht implementiert, sondern kommt erst in der abgeleiteten Klasse Y hinzu.

Damit repräsentieren m1, m2 und m3 im Prinzip alle Möglichkeiten: Vererben von Methoden, Überschreiben von Methoden und zusätzliche Methoden.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

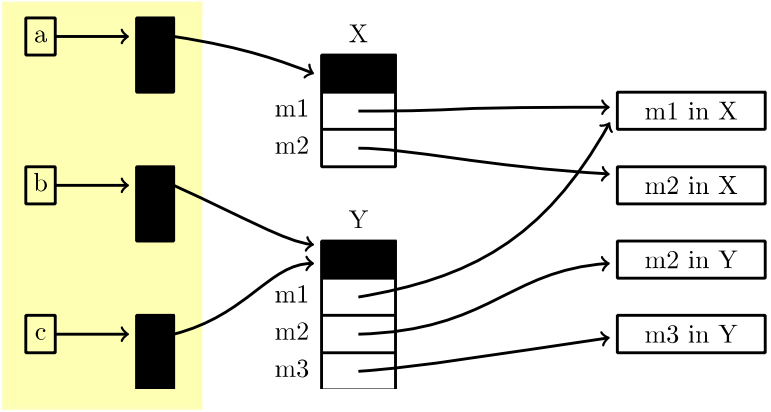



```
X a = new X();
Y b = new Y();
X c = new Y();
```

Beim Aufruf der Methoden kommt die Methodentabelle ins Spiel. So sieht das Schema bei unserem Beispiel mit Klasse X und Y und den drei Variablen a, b und c aus, die unten links definiert und initialisiert werden.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

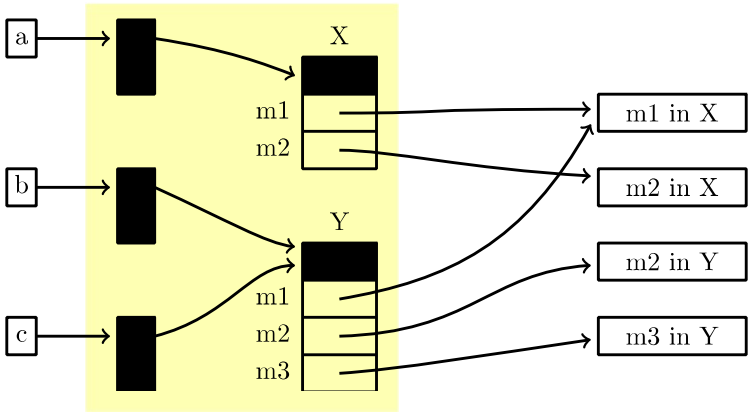



```
X a = new X();
Y b = new Y();
X c = new Y();
```

Jede der drei Variablen, a, b und c, verweist auf ein separates, mit new eingerichtetes Objekt.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

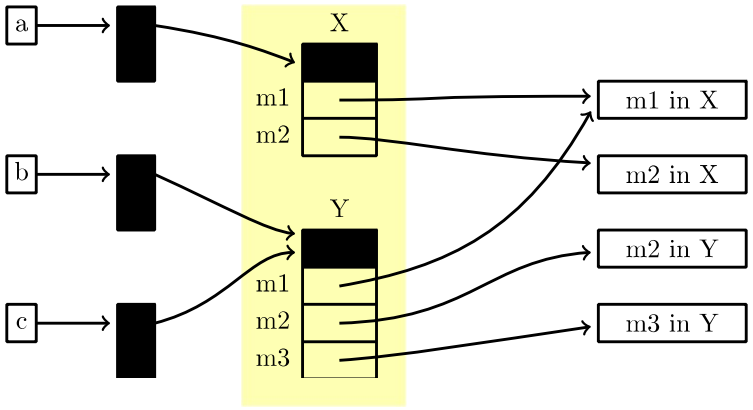



```
X a = new X();
Y b = new Y();
X c = new Y();
```

Bei der Einrichtung eines Objektes wird in dem anonymen Zusatzattribut gleich auch ein Verweis auf die Information zur Klasse des Objekts gespeichert.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT



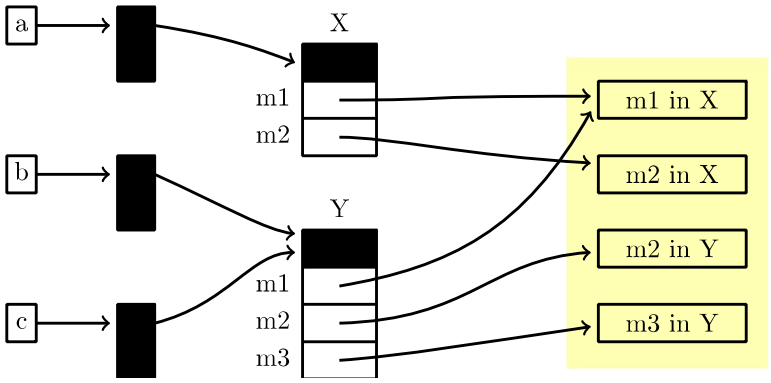

```
X a = new X();
Y b = new Y();
X c = new Y();
```

Die Methodentabelle hat einen festen, für alle Klassen identischen Offset in den Informationen zur Klasse. Alle Informationen zur Klasse, die nicht zur Methodentabelle gehören, sind im farblich unterlegten Teil des Bildes jeweils durch ein schwarzes Kästchen angedeutet.

***Nebenbemerkung*: Wie üblich, sind die technischen Details von dem, was im Hintergrund so passiert, leicht vereinfacht dargestellt, denn es kommt hier nur auf das grundlegende Verständnis an.**

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

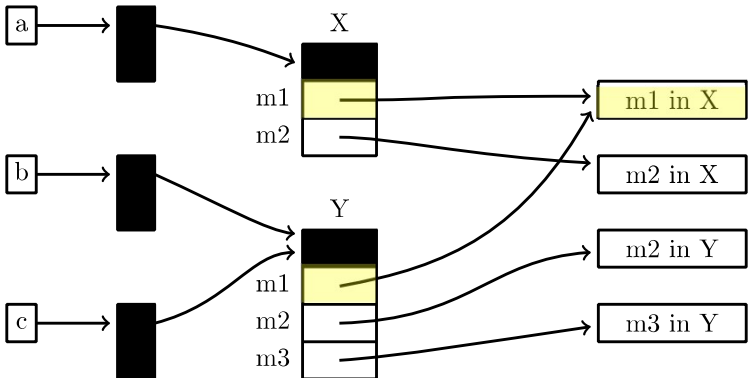



```
X a = new X();
Y b = new Y();
X c = new Y();
```

Diese Kästen sollen die Implementationen aller Methoden der Klassen X und Y schematisch darstellen. Die Methode m1 ist nur einmal implementiert, nämlich für X. Ebenso ist die Methode m3 nur für Y implementiert. Die Methode m2 hingegen ist einmal für X und noch einmal für Y implementiert.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

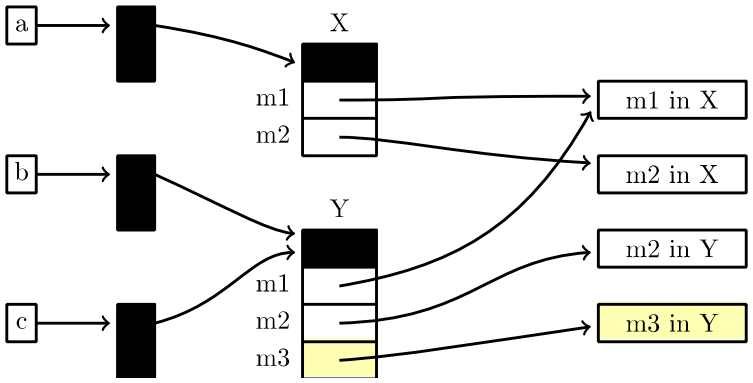



```
X a = new X();
Y b = new Y();
X c = new Y();
```

Da die Methode m1 von X an Y vererbt ist, enthalten beide Methodentabellen am Index für Methode m1 einen Verweis auf die Implementation von m1 in X. Dafür sorgt der Compiler beim Übersetzen der Klassen X beziehungsweise Y: Beim Übersetzen der Klasse Y kopiert er diese Adresse einfach aus der Methodentabelle von X in die Methodentabelle von Y.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT



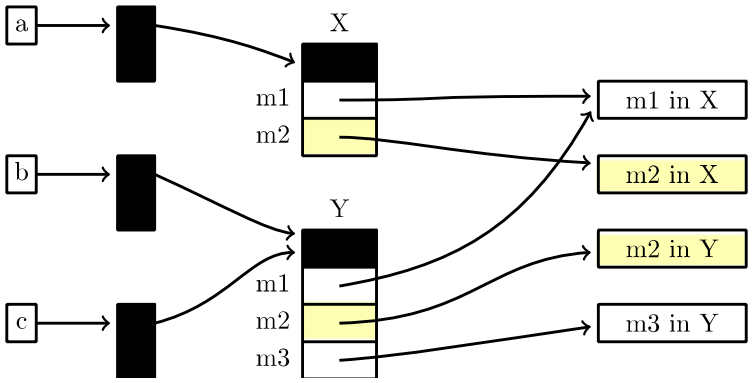

```
X a = new X();
Y b = new Y();
X c = new Y();
```

Die Methode m3 ist erst in Y hinzugekommen. Ihr Eintrag in der Methodentabelle wird hinten angehängt, damit die Offsets von m1 und m2 in beiden Methodentabellen identisch bleiben können.

***Nebenbemerkung*: Das ist übrigens einer der wesentlichen Gründe dafür, dass man sich bei der Entwicklung der Sprache Java dagegen entschieden hat, dass eine Klasse von mehreren anderen Klassen direkt erben kann, denn in einer von mehreren anderen Klassen abgeleiteten Klasse kann man die Einträge in der Methodentabelle unmöglich so reihen, dass jede Methode jeder der Basisklassen in der Methodentabelle der abgeleiteten Klasse denselben Offset hat wie in der Methodentabelle der Basisklasse. Wenn Sie das nicht glauben: Versuchen Sie es!**

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

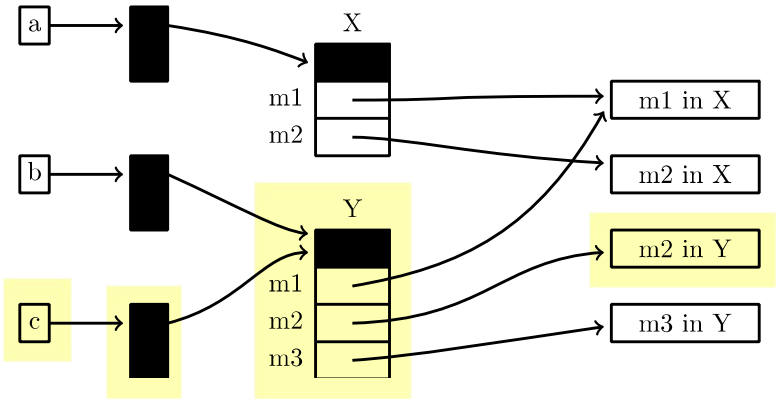



```
X a = new X();
Y b = new Y();
X c = new Y();
```

Die Methode m2 ist in X erstmals implementiert und in Y dann überschrieben worden. Die Methodentabelle für X enthält einen Verweis auf die Implementation von m2 in X, und die Methodentabelle für Y enthält am selben Index einen Verweis auf die Implementation von m2 in Y.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT




```
X a = new X();
Y b = new Y();
X c = new Y();
```

Der Compiler übersetzt den Aufruf einer Methode in eine ganze Kette von Aktionen, die diese Verweisstruktur durchläuft. Nehmen wir an, die Methode m2 wird mit der Variablen c aufgerufen. Dann wird zuerst auf die Adresse, die in c steht, zugegriffen, genauer gesagt auf das anonyme Attribut darin, das seinerseits auf die Informationen zur Klasse des Objekts verweist. Dieser zweite Verweis führt zur Klasseninformation für Y und damit auch zur Methodentabelle. Dort wird auf den Index der Methode m2 zugegriffen, der ja wie gesagt bei allen Klassen in der Hierarchie derselbe ist und daher vom Compiler beim Übersetzen des Methodenaufrufs als Konstante eingesetzt werden kann. Schlussendlich wird die Methode, auf die dieser Index verweist, aufgerufen und ausgeführt.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new X();

a.m1();

a.m2();

a.m3();
```

Wir gehen das Ganze nochmals in Java durch. Als erstes schauen wir uns Klasse X allein an.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new X();
```

```
a.m1();
```

```
a.m2();
```

```
a.m3();
```

Die Methoden m1 und m2 sind ganz normal in Klasse X implementiert und können daher wie immer aufgerufen werden.

Methode m3 hingegen ist erst in Klasse Y implementiert und existiert daher für X gar nicht. Aufruf von m3 mit a führt zu einer Fehlermeldung des Compilers und zum Abbruch des Übersetzungsvorgangs.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Y b = new Y();

b.m1();    //  Implementation in X

b.m2();    //  Implementation in Y

b.m3();    //  Implementation in Y
```

Als nächstes schauen wir uns Klasse Y für sich genommen an.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
Y b = new Y();

b.m1();  // Implementation in X
b.m2();  // Implementation in Y
b.m3();  // Implementation in Y
```

Für Y sind alle drei Methoden vorhanden, allerdings aus den drei unterschiedlichen Gründen, die wir schon diskutiert hatten: Die Methode m1 ist nur in Klasse X implementiert und daher an Y vererbt; die Methode m2 ist sowohl in X als auch in Y implementiert, die Implementation von Y wird aufgerufen; die Methode m3 ist *nur* in Y implementiert.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X c = new Y();

c.m1();    //  Implementation in X

c.m2();    //  Implementation in Y
```

Spannend wird es jetzt, wenn wir X und Y kombinieren. Wir haben ja schon mehrfach gesehen, dass eine Variable von der Basisklasse auf ein Objekt der abgeleiteten Klasse verweisen darf.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X c = new Y();

c.m1();   // Implementation in X

c.m2();   // Implementation in Y
```

Die Methode m1 hat nur eine Implementation, nämlich in Klasse X. Nur diese kann es sein, die hier aufgerufen wird.

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X c = new Y();

c.m1();   // Implementation in X
c.m2();   // Implementation in Y
```

Für m2 gibt es zwei Implementationen, eine in X und eine in Y. Aufgerufen wird hier die in Y, wie wir schon gesehen haben.

***Nebenbemerkung*: Man könnte sicher den Compiler so schreiben, dass er in der hier gezeigten einfachen Situation erschließen kann, dass c auf ein Objekt von Klasse Y verweist. Aber wie wir auf der nächsten Folie sehen werden, ist das völlig unmöglich, wenn die Situation nur ein wenig komplizierter ist.**

Verborgene Informationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a;
if ( moon.isWaning() )
   a = new X();
else
   a = new Y();
a.m2();
```

Dies ist ein einfaches Beispiel, in dem der Compiler keine Chance hat festzustellen, ob das Objekt, auf das a verweist, von Klasse X oder von Klasse Y ist, denn diese Information steht zur Kompilierzeit noch gar nicht zur Verfügung: Wann immer der Mond abnehmend ist, verweist a auf ein Objekt vom Typ X, sonst auf ein Objekt vom Typ Y.

TECHNISCHE UNIVERSITÄT DARMSTADT

Klassenmethoden

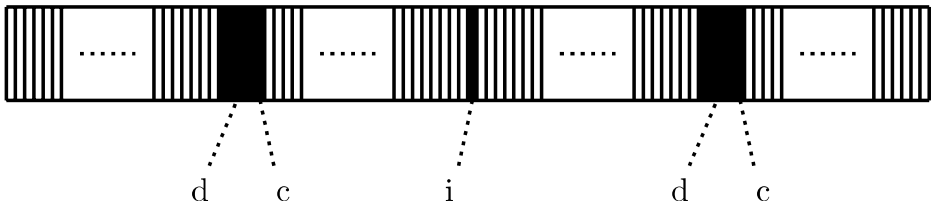
Oracle Java Tutorials:
Understanding Class Members
(Abschnitt Class Methods)

Alles bisher Gesagte gilt allerdings nicht für alle Methoden, sondern nur für *eine* Art von Methoden, genannt *Objekt*methoden. Jetzt sehen wir die zweite Art, *Klassen*methoden.

***Erinnerung*: Wir hatten in Kapitel 03b, Abschnitt zu Klassenattributen, schon zwischen Objekt- und Klassen*attributen* unterschieden und gesehen, dass letztere durch das Schlüsselwort static kenntlich gemacht werden. Die Zweiteilung bei Methoden ist ziemlich analog dazu und auch eng mit dieser Zweiteilung bei Attributen verzahnt, wie wir gleich sehen werden.**

Erinnerung: Klassenattribute

TECHNISCHE UNIVERSITÄT DARMSTADT




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

So sah es mit Klassenattributen aus: Durch das Schlüsselwort static wird i ein Klassenattribut, während d und c weiterhin Objektattribute sind.

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public int i;
    public void m1 ()  { ..... }
    public void m2 ()  {
        i = 1;
        m1 ();
    }
}
```

Alle Methoden, die wir bisher selbst implementiert haben, waren Objektmethoden. Auch diese beiden Methoden in diesem kleinen Beispiel sind zunächst einmal Objektmethoden.

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public int i;
    public void m1 ()  { ..... }
    public static void m2 ()  {
        i = 1;
        m1 ();
    }
}
```

Eine Klassenmethode erkennt man wie ein Klassenattribut am Schlüsselwort static. Die zweite Methode ist also nun eine Klassenmethode.

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public int i;
    public void m1 ()  { ..... }
    public static void m2 ()  {
        i = 1;
        m1 ();
    }
}
```

Der Unterschied lässt sich schon an diesem einfachen Beispiel gut demonstrieren: Was eben bei der Objektmethode noch erlaubt war, ist jetzt bei der Klassenmethode nicht mehr erlaubt.

Abstrakt formuliert ist die Regel ganz einfach: Eine Klassenmethode darf nicht auf das Objekt zugreifen, mit dem sie aufgerufen wurde.

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Klassenmethode darf / darf nicht:

- **Objektattribute lesen oder schreiben**
- **Objektmethoden aufrufen**
- **Klassenattribute lesen oder schreiben**
- **Klassenmethoden aufrufen**

Wir schauen uns jetzt genauer an, was die abstrakte Regel, dass eine Klassenmethode nicht auf ihr Objekt zugreifen darf, nun *konkret* bedeutet, indem wir im Einzelnen durchgehen, was erlaubt ist und was nicht.

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Klassenmethode darf / darf nicht:

- **Objektattribute lesen oder schreiben**
- **Objektmethoden aufrufen**
- **Klassenattribute lesen oder schreiben**
- **Klassenmethoden aufrufen**

Zugriff auf ein Objektattribut geht gar nicht, das haben wir schon im einleitenden Beispiel gesehen.

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Klassenmethode darf / darf nicht:

- **Objektattribute lesen oder schreiben**
- **Objektmethoden aufrufen**
- **Klassenattribute lesen oder schreiben**
- **Klassenmethoden aufrufen**

Ebenso geht nicht der Aufruf einer Objektmethode, also einer Methode ohne static. Das ist auch ganz logisch, denn diese Objektmethode könnte ja ihrerseits auf Objektattribute zugreifen, so dass die Klassenmethode indirekt über die Objektmethode dann doch auf das Objekt zugreifen würde.

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Klassenmethode darf / darf nicht:

- Objektattribute lesen oder schreiben
- Objektmethoden aufrufen
- Klassenattribute lesen oder schreiben
- Klassenmethoden aufrufen

Was problemlos geht, ist der Zugriff auf ein Klassenattribut, also ein mit static deklariertes Attribut, denn das ist ja unabhängig von jedem Objekt der Klasse.

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Klassenmethode darf / darf nicht:

- Objektattribute lesen oder schreiben
- Objektmethoden aufrufen
- Klassenattribute lesen oder schreiben
- Klassenmethoden aufrufen

Sowie der Aufruf einer Klassenmethode.

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X.m1 ();      // Objektmethode

X.m2 ();      // Klassenmethode
```

Warum diese Einschränkungen für Klassenmethoden? Welchen Sinn soll das Ganze haben?

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X.m1 ();      // Objektmethode

X.m2 ();      // Klassenmethode
```

Nun, es macht offensichtlich keinen Sinn, eine Objektmethode nur mit dem Klassennamen aufzurufen. Denn eine Objektmethode darf auf das Objekt zugreifen, aber dafür muss dann natürlich auch ein Objekt vorhanden sein. Daher ist das nicht bei der Objektmethode m1 erlaubt, der Versuch führt zu einem Fehler beim Übersetzen und zum Abbruch der Übersetzung.

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X.m1 ();    // Objektmethode

X.m2 ();    // Klassenmethode
```

Aber da eine Klassenmethode gar nicht erst auf das Objekt zugreifen darf, muss auch kein Objekt da sein. Die Methode darf einfach mit dem Klassennamen aufgerufen werden – in perfekter Analogie zu Klassenattributen.

Objekt-/Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
    public int i;
    public void m1 ()  { ..... }
    public static void m2 ( X a )  {
        a.i = 1;
        a.m1 ();
    }
}
```

Zur Klarstellung: *Dies hier* geht auch in Klassenmethoden. Warum auch nicht? Über a ist ja ein Objekt von Klasse X vorhanden. Es gibt keinen Grund, Zugriffe auf a wie hier gezeigt zu verbieten.

Klasse java.lang.Math

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Math {
    .........
    public static double sin ( double a )    { ......... }
    public static double cos ( double a )    { ......... }
    public static double tan ( double a )    { ......... }
    public static double asin ( double a )   { ......... }
    .........
}
```

Den Sinn von Klassen*konstanten* haben wir an den Konstanten PI und E in der Klasse java.lang.Math eingesehen.

Auch für den Sinn von Klassen*methoden* ist die Klasse Math ein gutes Beispiel. Eigentlich dient die Klasse Math nur als Sammlung einerseits für die beiden Konstanten, andererseits für eine Vielzahl mathematischer Methoden. Die berechnen aus ihren Parameterwerten ein Ergebnis und benötigen dafür keine weiteren Daten als nur die Parameter, also auch kein Objekt der Klasse Math.

Hier ein paar Beispiele von mathematischen Funktionen in Klasse Math. Methode asin ist der Arcussinus, die anderen sollten selbsterklärend sein.

Klasse java.lang.Math

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double x = 2 * Math.PI;

double y = Math.cos ( x );
```

So sieht etwa ein einfaches Anwendungsbeispiel aus: Es muss kein Objekt der Klasse Math erzeugt werden, sondern es reicht, den Klassennamen Math vor den Methodenaufruf zu stellen.

Statischer Import

TECHNISCHE UNIVERSITÄT DARMSTADT

```
import static java.lang.Math.sin;

y = sin ( Math.PI * x );
```

Beim Thema Klassen*attribute* hatten wir schon statischen Import gesehen. Das funktioniert auch bei Klassen*methoden*. In diesem Beispiel wird nur die Klassenmethode sin statisch importiert, nicht aber die Klassenkonstante PI. Daher kann sin ohne Math angesprochen werden, PI aber nicht.

Methoden von Enum

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public enum Weekday {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
  THURSDAY, FRIDAY, SATURDAY
}



for ( Weekday day : Weekday.values() )
   System.out.println ( day.name() );
```

Enum-Typen erben von Klasse java.lang.Enum ein paar nützliche Klassen- und Objektmethoden, hier nur jeweils ein Beispiel.

Methoden von Enum

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public enum Weekday {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
  THURSDAY, FRIDAY, SATURDAY
}
```

```
for ( Weekday day : Weekday.values() )
   System.out.println ( day.name() );
```

Diese Klassenmethode liefert ein Array zurück, in dem die in der Enumeration definierten Objekte die Komponenten sind. Das Objekt mit Namen SUNDAY steht an Index 0, MONDAY an Index 1 und so weiter.

***Erinnerung*: In Kapitel 01d und auch später haben wir schon diese Kurzform der for-Schleife gesehen, mit der man die Komponenten eines Arrays in der Reihenfolge aufsteigender Indizes durchlaufen kann.**

Methoden von Enum

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public enum Weekday {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
  THURSDAY, FRIDAY, SATURDAY
}



for ( Weekday day : Weekday.values() )
   System.out.println ( day.name() );
```

Diese Objektmethode liefert den Namen des Objektes, so wie er in der Definition des Enum-Typs steht, als String zurück, der dann beispielsweise wie hier mit System.out.println auf dem Bildschirm ausgegeben werden kann. In den einzelnen Durchläufen der hier gezeigten for-Schleife werden also nacheinander die Strings "SUNDAY" bis "SATURDAY" zurückgeliefert.

TECHNISCHE UNIVERSITÄT DARMSTADT

Variable Parameterzahl

Ein letzter Punkt bei Parameterlisten. Es ist in engen Grenzen möglich, die Anzahl der aktualen Parameter einer Methode variabel zu halten, und zwar am Ende der Parameterliste.

Variable Parameterzahl

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public static void m ( char c, double... args ) {
      System.out.println ( c );
      for ( double x : args )
        System.out.println ( x );
   }
}
```

Dazu wieder ein illustratives Beispiel.

Variable Parameterzahl

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public static void m ( char c, double... args ) {
      System.out.println ( c );
      for ( double x : args )
        System.out.println ( x );
   }
}
```

Konkret dürfen beim letzten formalen Parameter drei Punkte an den Typ des Parameters angehängt werden.

Variable Parameterzahl

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public static void m ( char c, double... args ) {
      System.out.println ( c );
      for ( double x : args )
         System.out.println ( x );
   }
}
```

Wie man hier sieht, macht der Compiler aus dem letzten formalen Parameter ein Array des eigentlichen Datentyps, in diesem konkreten Beispiel also ein Array von double.

Variable Parameterzahl

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double [ ] a = new double [ 123 ];
X.m ( ´e´, a );

X.m ( ´a´ );
X.m ( ´b´, 1.41 );
X.m ( ´c´, 1.41, 2.71 );
X.m ( ´d´, 1.41, 2.71, 3.14 );
```

Warum nun eine zweite Möglichkeit, ein Array zu übergeben, zum Beispiel ein Array von double wie in diesem Beispiel?

Variable Parameterzahl

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double [ ] a = new double [ 123 ];
X.m ( ´e´, a );

X.m ( ´a´ );
X.m ( ´b´, 1.41 );
X.m ( ´c´, 1.41, 2.71 );
X.m ( ´d´, 1.41, 2.71, 3.14 );
```

Als aktualer Parameter kann durchaus ein Array von double so wie hier übergeben werden.

Variable Parameterzahl

TECHNISCHE UNIVERSITÄT DARMSTADT

```
double [ ] a = new double [ 123 ];
X.m ( ´e´, a );

X.m ( ´a´ );
X.m ( ´b´, 1.41 );
X.m ( ´c´, 1.41, 2.71 );
X.m ( ´d´, 1.41, 2.71, 3.14 );
```

Aber im Unterschied zu einem formalen Parameter vom Arraytyp kann anstelle eines Arrays eine beliebige Anzahl von double-Werten angegeben werden. Aus diesen double-Werten macht der Compiler ein Array von double. Die Komponenten des Arrays sind genau diese Werte in aufsteigender Reihenfolge der Indizes.

Variable Parameterzahl

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double max ( double... a ) {
  double result = Double.NEGATIVE_INFINITY;
  for ( int i = 0; i < a.length; i++ )
     if ( result < a[i] )
        result = a[i];
  return result;
}


X.max ( y1, y2, y3, y4, y5, y6 );
```

Zum Abschluss des Abschnitts zu variabler Parameterzahl noch ein konkretes Anwendungsbeispiel.

Variable Parameterzahl

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double max ( double... a ) {
   double result = Double.NEGATIVE_INFINITY;
   for ( int i = 0; i < a.length; i++ )
      if ( result < a[i] )
         result = a[i];
   return result;
}
```

```
X.max ( y1, y2, y3, y4, y5, y6 );
```

Eine Klasse X habe eine Klassenmethode max mit einer variablen Anzahl von Parametern vom Typ double, so dass diese Methode beispielsweise wie unten mit sechs double-Werten als aktualen Parametern aufgerufen werden kann, aber natürlich genauso auch mit einer anderen Zahl von double-Werten.

Variable Parameterzahl

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static double max ( double... a ) {
  double result = Double.NEGATIVE_INFINITY;
  for ( int i = 0; i < a.length; i++ )
     if ( result < a[i] )
        result = a[i];
  return result;
}


X.max ( y1, y2, y3, y4, y5, y6 );
```

Dieser Code realisiert die Berechnung des Maximums aller Werte in einem Array, dessen Komponenten von einem arithmetischen Datentyp sind. In result steht nach jedem Durchlauf der Schleife das Maximum aller Werte in a, die bis zu diesem Zeitpunkt gelesen worden sind. Zum Beispiel nach 5 Durchläufen steht in result das Maximum aller Werte, die in a an den Indizes 0 bis 4 zu finden sind. Nach Beendigung der Schleife steht daher in result das Maximum aller Werte in a an den Indizes 0 bis a.length-1, und das sind natürlich gerade *alle* Werte in a.

Falls keine aktualen Parameter übergeben werden, wird ein leeres Array erzeugt. In diesem Fall wird – konform zu den Gesetzen der Mathematik – das neutrale Element der Maximumsbildung zurückgeliefert, also minus unendlich.

TECHNISCHE UNIVERSITÄT DARMSTADT

Struktur von Methoden

Oracle Java Tutorials: Defining Methods

Als nächstes schauen wir uns Methoden noch einmal allgemeiner an.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

- Head (Kopf): alles vor den eigentlichen Anweisungen
- Body (Rumpf): die Anweisungen in geschweiften Klammern

Generell unterscheiden wir zwischen dem Kopf und dem Rumpf einer Methode. Der Rumpf ist auf dieser Abstraktionsebene noch simpel strukturiert: einfach eine Sequenz von Anweisungen, eingerahmt durch geschweifte Klammern.

TECHNISCHE UNIVERSITÄT DARMSTADT

Kopf einer Methode

Den Rumpf schauen wir uns später in diesem Kapitel genauer an, wenn wir zu Anweisungen und Ausdrücken kommen. Hier kümmern wir uns erst einmal um den Kopf.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void main ( String [ ] args )
```

Als Beispiel nehmen wir den Kopf einer Methode, die von vielen Java-Systemen als Einstiegspunkt genommen wird, um ein Java-Programm zu starten. Der vorgegebene Kopf einer Einstiegsmethode muss exakt eingehalten werden.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void main ( String [ ] args )
```

Jede Methode hat natürlich einen Namen. Das ist ein Identifier.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void main ( String [ ] args )
```

Der Rückgabetyp beziehungsweise das Schlüsselwort void steht immer unmittelbar vor dem Namen der Methode, getrennt durch mindestens ein Whitespace.

***Erinnerung*: Whitespaces hatten wir im gleichnamigen Abschnitt in Kapitel 01a eingeführt.**

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void main ( String [ ] args )
```

Vor dem Rückgabetyp beziehungsweise vor void stehen die *Modifier*. Grob gesprochen ändern sie nichts an der Methode selbst, sondern daran, wie man die Methode verwenden kann.

Diese beiden Modifier haben wir schon im Kapitel zu Referenztypen gesehen: public besagt, dass diese Methode main frei außerhalb der Klasse, zu der sie gehört, in einer beliebigen anderen Klasse aufgerufen werden kann, im Unterschied zu private und protected; und wie schon bekannt, besagt static, dass main eine Klassenmethode ist.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void main ( String [ ] args )
```

Die Parameterliste folgt immer unmittelbar nach dem Namen der Methode. Ein Whitespace muss zwischen dem Namen der Methode und ihrer Parameterliste nicht stehen, da die öffnende Klammer unzweideutig den Namen von der Parameterliste abgrenzt.

Wir haben schon des Öfteren gesehen, dass eine Parameterliste eine Aufzählung von Parametern in runden Klammern ist, die auch leer sein kann, also leeres Klammerpaar. Wenn die Parameterliste *nicht* leer ist, sondern sogar mehr als einen Parameter hat, dann haben wir gesehen, dass die einzelnen Parameter durch Kommas voneinander getrennt sind.

Jeder einzelne Parameter ist spezifiziert durch seinen Typ und darauf folgend seinen Namen, getrennt durch mindestens ein Whitespace. Die Methode main, so wie sie von vielen Programmen als Einstiegspunkt verwendet wird, hat nur einen Parameter.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public static void main ( String [ ] args ) {
    for ( String str : args )
      System.out.println ( str );
  }
}
```

java X a bc 1d2 → a bc 1d2

Es fehlt noch die Erklärung, was Java-Systeme beim Aufruf von main eigentlich als Parameter übergeben. Einen Array von String, soweit klar, aber was sind die einzelnen Strings darin? Das schauen wir uns an diesem simplen Beispiel an.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public static void main ( String [ ] args ) {
    for ( String str : args )
      System.out.println ( str );
  }
}
```

java X a bc 1d2 → a bc 1d2

***Zur Erinnerung*: Mit dieser abgekürzten Form einer for-Schleife kann man bequem die Komponenten eines Arrays nach aufsteigenden Indizes durchlaufen. In Klammern muss zuerst der Komponententyp kommen, dann der Name, den wir den einzelnen Komponenten in den einzelnen Durchläufen geben wollen, nach dem Doppelpunkt dann der Name des Arrays.**

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public static void main ( String [ ] args ) {
    for ( String str : args )
      System.out.println ( str );
  }
}
```

java X a bc 1d2 → a bc 1d2

Die Methode println von Printstream und das Klassenattribut out von Klasse System hatten wir schon im Kapitel 03b zu Referenztypen kennen gelernt, Stichwort Attribute von Klassentypen.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public static void main ( String [ ] args ) {
    for ( String str : args )
      System.out.println ( str );
  }
}
```

java X a bc 1d2 → a bc 1d2

Java-Systeme, die main als Einstiegspunkt nehmen, bieten eine Möglichkeit an, dem Aufruf Parameter mitzugeben. Man nennt sie häufig Kommandozeilenparameter, weil sie zum Beispiel beim Standard-Interpreter namens java beim Aufruf auf der Kommandozeile angegeben werden.

Wie wir sehen, hat das zur Konsequenz, dass ein Array von String eingerichtet wird, dessen Länge gleich der Anzahl der Kommandozeilenparameter ist. Die einzelnen Kommandozeilenparameter sind die Komponenten des Arrays, und zwar in derselben Reihenfolge, in der sie auf der Kommandozeile aufgeführt sind.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorgriff:

```
public int m()
    throws XException, YException
```

In der beispielhaften Methode main fehlte noch ein Bestandteil, den wir jetzt in diesem künstlichen Beispiel der Vollständigkeit halber nachtragen, obwohl wir ihn erst später, im Kapitel 05 zur Fehlerbehandlung, verstehen werden.

Unmittelbar nach der Parameterliste, also unmittelbar vor der öffnenden geschweiften Klammer für den Methodenrumpf, kann optional eine throws-Klausel so wie hier stehen. Die throws-Klausel in diesem Beispiel besagt, dass die Methode m potentiell eine Exception wirft, und zwar entweder eine von einem Typ namens XException oder eine von einem Typ namens YException.

Was das genau bedeutet, betrachten wir, wie gesagt, später. Hier halten wir nur fest, dass nach der Parameterliste optional das Schlüsselwort throws stehen kann, gefolgt von einem oder mehreren Typen von Exceptions, separiert durch Kommas.

Die Klasse main als Einstiegspunkt darf *keine* throws-Klausel haben.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

- Name
- Ungeordnete Menge der Modifier
- Rückgabetyp
- Parameterliste: geordnete Sequenz der *Typen* der Parameter
- Ungeordnete Menge der Exceptions

Noch einmal die Bestandteile des Kopfes einer Methode zusammengefasst.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

- Name
- Ungeordnete Menge der Modifier
- Rückgabetyp
- Parameterliste: geordnete Sequenz der *Typen* der Parameter
- Ungeordnete Menge der Exceptions

Zu beachten ist, dass nur die *Typen* der einzelnen Parameter und ihre *Reihenfolge* wichtig sind. Dem Nutzer einer Methode kann es egal sein, mit welchen Namen die Parameter innerhalb der Methode angesprochen werden, daher sind die Namen der Parameter *kein* signifikanter Teil eines Methodenkopfes.

Natürlich sind gut gewählte Namen der Parameter dennoch hilfreich für das Verständnis des Nutzers.

Kopf einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

- Name
- Ungeordnete Menge der Modifier
- Rückgabetyp
- Parameterliste: geordnete Sequenz der *Typen* der Parameter
- Ungeordnete Menge der Exceptions

Bei den Parametern einer Methode ist die Reihenfolge eine wichtige Information für den Nutzer, denn er muss diese Reihenfolge ja beim Aufruf der Methode einhalten.

Bei den Modifiern hingegen kommt es nicht darauf an, in welcher Reihenfolge sie im Kopf der Methode angegeben worden sind; man könnte sie dort beliebig umordnen, und es würde sich für den Nutzer der Methode nichts ändern.

Dies gilt auch für Exceptions.

TECHNISCHE UNIVERSITÄT DARMSTADT

Javadoc für Methodenköpfe

***Erinnerung*: In Kapitel 01a, Abschnitt zu Kommentaren, haben wir schon javadoc-Kommentare kennen gelernt in Form von zwei Klauseln, mit denen man typischerweise die Quelldateien als Ganzes kommentiert: @author und @version.**

Hier schauen wir uns jetzt zwei weitere Klauseln an, die zur Kommentierung von Methodenköpfen gedacht sind, ...

Javadoc für Methodenköpfe

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/**
* @param   x   the dividend
* @param   y   the divisor, must not be zero
* @return x integer-divided by y
*/
public int quotient ( int x, int y ) {
   return x / y;
}
```

... und zwar die Klauseln @param und @return. Da die Typnamen der Parameter und der Rückgabe in Java zwingend im Methodenkopf anzugeben sind, werden sie in der Regel – und so auch hier – im javadoc-Kommentar nicht noch einmal aufgeführt.

Javadoc für Methodenköpfe

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/**
* @param   x   the dividend
* @param   y   the divisor, must not be zero
* @return x integer-divided by y
*/
public int quotient ( int x, int y ) {
   return x / y;
}
```

Vorbedingungen, die die Parameter zu erfüllen haben, schreiben wir ebenfalls in die entsprechende @param-Klausel.

***Vorgriff*: Im Kapitel 05 werden wir noch eine weitere javadoc-Klausel für Methodenköpfe sehen: @throws.**

TECHNISCHE UNIVERSITÄT DARMSTADT

Signatur
und
Überschreiben / Überladen
von
Methoden

An die Besprechung von Methodenköpfen lässt sich unmittelbar das Thema Überschreiben beziehungsweise Überladen von Methoden anschließen. Zur Systematisierung führen wir noch einen grundlegenden Begriff ein, die *Signatur* einer Methode.

Wir halten vorab schon einmal zusammenfassend fest: Eine Klasse kann keine zwei Methoden mit derselben Signatur haben, denn beim Über*schreiben* geht die überschriebene Methode verloren, und beim Über*laden* müssen sich die Signaturen unterscheiden.

Signatur einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

- Name
- Parameterliste: geordnete Sequenz der *Typen* der Parameter

Die Verwendung des Begriffs Signatur ist nicht einheitlich. Als Autorität für Begriffsbildungen bei Java wird allgemein Oracle Docs verwendet, so auch hier. Auf Basis des bisher Gesagten lässt sich die Signatur leicht definieren: Diese beiden Bestandteile des Methodenkopfes bilden zusammen die Signatur einer Java-Methode.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public void m () { ..... }
}

public class Y extends X {
   public void m () { ..... }
}
```

***Erinnerung*: Eine Methode, die in der Basisklasse definiert ist, kann in einer direkt oder indirekt abgeleiteten Klasse überschrieben werden.**

Die Implementation in der Basisklasse ist die *überschriebene* Methode und die Implementation in der abgeleiteten Klasse die *überschreibende* Methode. Wir schauen uns im Folgenden die genauen Modalitäten an, und welche Rolle die Signatur dabei spielt.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

- Die Signatur muss bei überschriebener und überschreibender Methode identisch sein
- Die anderen Bestandteile des Methodenkopfes können – in Grenzen – variieren
  - ➢ Nächste Folie

Die Rolle, die die Signatur beim Überschreiben spielt, ist ganz einfach: Die Signatur ist das, was bei beiden Implementationen derselben Methode absolut identisch sein muss. Die anderen Bestandteile des Methodenkopfes müssen nur „beinahe“ identisch sein, kleine Abweichungen sind möglich, aber nur bestimmte.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

Jeweils nur in eine Richtung variabel:

- **private → ε → protected → public**
- **Falls Rückgabe von einem Referenztyp: durch Subtyp ersetzbar**
  - **Klassen, Interfaces, Arrays**
- **Exceptionklassen durch abgeleitete Exceptionklassen ersetzbar**

Wir gehen die verschiedenen weiteren Bestandteile des Methodenkopfes im Einzelnen durch.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

Jeweils nur in eine Richtung variabel:

- private → ε → protected → public
- Falls Rückgabe von einem Referenztyp: durch Subtyp ersetzbar
  - Klassen, Interfaces, Arrays
- Exceptionklassen durch abgeleitete Exceptionklassen ersetzbar

Erinnerung: In Kapitel 01f, Abschnitt „Zugriffsrechte und Packages“, haben wir gesehen, dass auf Attribute und Methoden bei private nur aus der Klasse selbst heraus zugegriffen werden darf; bei Auslassung des Modifiers, also implizitem Zugriffsrecht, nur aus demselben Package; bei protected aus Package und abgeleiteten Klassen; und bei public von überall her.

Die Zugriffsrechte dürfen in einer abgeleiteten Klasse erweitert sein gegenüber der Basisklasse. Das heißt konkret, ist die Methode in der Basisklasse private, dann darf sie in der abgeleiteten Klasse private, implizit, protected oder public sein; ist sie in der Basisklasse implizit, dann darf sie in der abgeleiteten Klasse implizit, protected oder public sein; bei protected ist protected oder public erlaubt und bei public nur public. Andere Diskrepanzen zwischen überschriebener und überschreibender Methode darf es bei den Zugriffsrechten nicht geben, sonst bricht der Compiler mit einer Fehlermeldung ab.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

Jeweils nur in eine Richtung variabel:

- **private → ε → protected → public**
- **Falls Rückgabe von einem Referenztyp: durch Subtyp ersetzbar**
  - **Klassen, Interfaces, Arrays**
- **Exceptionklassen durch abgeleitete Exceptionklassen ersetzbar**

Bei primitiven Datentypen muss der Rückgabetyp bei überschriebener und überschreibender Methode identisch sein. Bei Referenztypen ist das anders.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

Jeweils nur in eine Richtung variabel:

- **private → ε → protected → public**
- **Falls Rückgabe von einem Referenztyp: durch Subtyp ersetzbar**
  - **Klassen, Interfaces, Arrays**
- **Exceptionklassen durch abgeleitete Exceptionklassen ersetzbar**

Eine Klasse darf durch jede von ihr direkt oder indirekt abgeleitete Klasse ersetzt werden.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

Jeweils nur in eine Richtung variabel:

- **private → ε → protected → public**
- **Falls Rückgabe von einem Referenztyp: durch Subtyp ersetzbar**
  - **Klassen, Interfaces, Arrays**
- **Exceptionklassen durch abgeleitete Exceptionklassen ersetzbar**

Analog darf ein Interface durch jedes direkt oder indirekt erweiternde Interface und durch jede direkt oder indirekt implementierende Klasse ersetzt werden.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

Jeweils nur in eine Richtung variabel:

- **private → ε → protected → public**
- **Falls Rückgabe von einem Referenztyp: durch Subtyp ersetzbar**
  - **Klassen, Interfaces, Arrays**
- **Exceptionklassen durch abgeleitete Exceptionklassen ersetzbar**

Auch der Typ Array von einem Referenztyp kann durch Array von Subtyp ersetzt werden.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

Jeweils nur in eine Richtung variabel:

- private → ε → protected → public
- Falls Rückgabe von einem Referenztyp: durch Subtyp ersetzbar
  - Klassen, Interfaces, Arrays
- Exceptionklassen durch abgeleitete Exceptionklassen ersetzbar

Vorgriff: Wir haben es noch nicht gesagt, aber Exceptions sind Objekte von bestimmten Klassen. Wird eine Methode mit Exceptions in einer abgeleiteten Klasse überschrieben, dann dürfen die geworfenen Exceptionklassen durch deren direkte oder indirekte Ableitungen ersetzt werden.

Die ersten beiden Punkte auf der Folie schauen wir uns noch kurz an einem Beispiel an. Den dritten Punkt, Exceptions, greifen wir dann im Kapitel 05 nochmals auf.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A {

   ..........

}

public class B extends A {

   .........

}
```

```
public class X {
   private A m () {
      ..........
   }
}
public class Y extends X {
   public B m () {
      .........
   }
}
```

Die beiden Klassen A und B links sind nur Spielmaterial, es geht um die beiden Klassen X und Y rechts.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A {

   ..........

}

public class B extends A {

   .........

}
```

```
public class X {
   private A m () {
      ..........
   }
}

public class Y extends X {
   public B m () {
      .........
   }
}
```

Hier der erste Punkt: Ist die überschriebene Methode private, darf die überschreibende Methode private, implizit, protected oder public sein. Hier ist sie beispielhaft public.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class A {

   ..........

}

public class B extends A {

   .........

}
```

```
public class X {
   private A m () {
      ..........
   }
}

public class Y extends X {
   public B m () {
      .........
   }
}
```

Nun der zweite Punkt: Der Rückgabetyp in der überschreibenden Methode darf ein Subtyp des Rückgabetyps der überschriebenen Methode sein.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  A m () {
    .........
  }
}
public class Y extends X {
  public B m () {
    .........
  }
}
```

Im selben Package:

```
X c = new Y();

A d = c.m();
```

Als nächstes ein Anwendungsbeispiel mit einer leicht variierten Definition der Klassen X und Y. Die rechte Seite der letzten Folie ist auf dieser Folie noch einmal links mit einem kleinen Unterschied zu sehen: In Klasse X ist die Methode m nicht mehr private, sondern implizit.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  A m () {
    .....
  }
}
public class Y extends X {
  public B m () {
    .....
  }
}
```

Im selben Package:

```
X c = new Y();

A d = c.m();
```

Die Regeln, wie die überschreibende von der überschriebenen Methode abweichen darf, sind perfekt so gestaltet, dass in solchen Situationen nichts schiefgeht: Wenn wie hier die Implementation der Methode m von Klasse Y aufgerufen und somit ein Objekt von B – oder von einem Subtyp von B – zurückgeliefert wird, dann ist das kein Problem, denn B und alle Subtypen von B sind ja auch Subtypen von A. Und da auch die Zugriffsrechte in der abgeleiteten Klasse gegenüber der Basisklasse nur erweitert, aber nicht eingeengt werden können, ist immer gewährleistet, dass die Methode m von Y überall da aufgerufen werden darf, wo sie mit X aufgerufen werden darf – so wie in diesem Beispiel.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  protected A m () {
    ..........
  }
}
public class Y extends X {
  public B m () {
    .........
  }
}
```

```
public class Z extends Y {
  private A c;
  public void n ( X d ) {
    c = d.m();
}
```

Jetzt noch ein leicht anders gelagertes Anwendungsbeispiel, diesmal mit protected in Klasse X.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  protected A m () {
     ..........
  }
}
public class Y extends X {
  public B m () {
     .........
  }
}
```

```
public class Z extends Y {
  private A c;
  public void n ( X d ) {
     c = d.m();
}
```

Auch in diesem Beispiel hat die Methode m in der abgeleiteten Klasse Y breitere Zugriffsrechte als in der Basisklasse: public statt protected.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  protected A m () {
     ..........
  }
}
public class Y extends X {
  public B m () {
     .........
  }
}
```

```
public class Z extends Y {
  private A c;
  public void n ( X d ) {
     c = d.m();
}
```

Da Z indirekt – nämlich über Y – von X abgeleitet ist, kann die protected-Methode m in der Methode n von Z aufgerufen werden.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  protected A m () {
    ..........
  }
}
public class Y extends X {
  public B m () {
    .........
  }
}
```

```
public class Z extends Y {
  private A c;
  public void n ( X d ) {
    c = d.m();
}
```

Und da die Zugriffsrechte mit public die Zugriffsrechte mit protected umfassen, ist es kein Problem, dass die Methode m in Y nicht protected, sondern public ist: Wo immer eine protected-Methode verwendet werden kann, kann sie natürlich auch als public-Methode verwendet werden; analog private und implizit.

Überschreiben einer Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  protected A m () {
    ..........
  }
}
public class Y extends X {
  public B m () {
    .........
  }
}
```

```
public class Z extends Y {
  private A c;
  public void n ( X d ) {
    c = d.m();
}
```

Dass hier auch wie im ersten Anwendungsfall zusätzlich der Rückgabetyp variiert wird, macht natürlich keinen Unterschied.

Vererbung Klassenmethoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Klassenmethoden

- **werden an Subtypen vererbt,**
- **im Subtyp kann auch eine Klassenmethode mit derselben Signatur definiert werden,**
- **aber die Auswahl der Methodenimplementation richtet sich nach dem statischen Typ,**
- **nicht nach dem dynamischen Typ wie bei Objektmethoden.**

Zum Ende des Themas Überschreiben noch ein Blick auf Klassenmethoden. Klassenmethoden können ja ohne ein Objekt aufgerufen werden. In diesem Fall gibt es keinen dynamischen, nur einen statischen Typ. Daher wird hier die Implementation der Methode nicht durch den dynamischen, sondern durch den statischen Typ bestimmt.
***Erinnerung*: In Kapitel 03b, Abschnitt „Begriffsbildung: Subtypen und statischer / dynamischer Typ“, hatten wir uns das für Objektmethoden angesehen.**

Damit verlassen wir das Thema Über*schreiben* von Methoden ...

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.print ( -123 );
System.out.print ( 3.14 );
System.out.print ( ´!´ );
System.out.print ( “Hello World“ );
System.out.println ( -123 );
System.out.println ( 3.14 );
System.out.println ( ´!´ );
System.out.println ( “Hello World“ );
```

... und kommen zum Über*laden* von Methoden.

Wir haben schon Beispiele für das Überladen von Methoden gesehen, zum Beispiel die zweite Methode move bei FastRobot in Kapitel 01f. Überladen bedeutet, dass zwei oder mehr Methoden in einer Klasse denselben Namen haben, so wie hier einerseits mehrere Methoden mit Namen print, andererseits mehrere Methoden mit Namen println.

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Alle Methoden einer Klasse müssen unterschiedliche Signatur haben

- **Bei Überladung müssen die Sequenzen der Parametertypen unterschiedlich sein**

Wie müssen sich zwei Methoden derselben Klasse unterscheiden?

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Alle Methoden einer Klasse müssen unterschiedliche Signatur haben

- **Bei Überladung müssen die Sequenzen der Parametertypen unterschiedlich sein**

Mit dem Begriff der Signatur lässt sich auch diese Frage sehr kompakt beantworten: Sie müssen unterschiedliche Signaturen haben.

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

Alle Methoden einer Klasse müssen unterschiedliche Signatur haben

- **Bei Überladung müssen die Sequenzen der Parametertypen unterschiedlich sein**

Überladung heißt ja, dass der Methodenname identisch ist. Was dann noch zur Unterscheidung bei der Signatur übrig bleibt, sind die Parametertypen.

Dabei ist es egal, ob beide Methoden erst in dieser Klasse definiert worden sind oder ob eine davon von einer Basisklasse oder einem Interface ererbt wurde.

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public void m () { ..... }
   public void m ( int i ) { ..... }
   public void m ( double d ) { ..... }
   public void m ( int i, double d ) { ..... }
}
```

Ein illustratives Beispiel: Die Klasse X kann alle diese Methoden m zugleich besitzen, da alle vier Parametertypsequenzen paarweise unterschiedlich sind.

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public void m () { ..... }
  public void m ( int i ) { ..... }
  public void m ( double d ) { ..... }
  public void m ( int i, double d ) { ..... }
}
```

Die Teile des Methodenkopfes, die *nicht* zur Signatur gehören, können, müssen aber nicht gleich sein. Das betrifft auch die hier nicht gezeigten optionalen Teile eines Methodenkopfes wie static und Exceptions (siehe Kapitel 05 für Exceptions).

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public void m ( int i ) { ..... }
   public int m ( int i ) { ..... }
   public static void m ( int i ) { ..... }
   private void m ( int i ) { ..... }
}
```

In dieser Variation des Beispiels hingegen dürfen keine zwei dieser vier Methoden zugleich in X sein, denn die Signatur ist bei allen vier Methoden identisch.

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public void m ( int i, double d ) {
    .........
  }
  public void m ( double d, int i ) {
    .........
  }
}
```

```
X a = new X();
a.m ( 123, 2.71 );
a.m ( 3.14, 321 );
a.m ( 123, 321 );
```

Eine Problematik muss noch angesprochen werden, wieder anhand eines illustrativen Beispiels.

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public void m ( int i, double d ) {
    .........
  }
  public void m ( double d, int i ) {
    .........
  }
}
```

```
X a = new X();
a.m ( 123, 2.71 );
a.m ( 3.14, 321 );
a.m ( 123, 321 );
```

Die beiden Parametertyplisten sind unterschiedlich, also ist diese Überladung absolut korrekt, *hier* gibt es noch kein Problem.

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public void m ( int i, double d ) {
    .....
  }
  public void m ( double d, int i ) {
    .....
  }
}
```

```
X a = new X();
a.m ( 123, 2.71 );
a.m ( 3.14, 321 );
a.m ( 123, 321 );
```

Diese beiden Aufrufe von m sind daher auch absolut ok: Es werden die beiden Methoden m jeweils einmal aufgerufen.

Überladen von Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public void m ( int i, double d ) {
    .....
  }
  public void m ( double d, int i ) {
    .....
  }
}
```

```
X a = new X();
a.m ( 123, 2.71 );
a.m ( 3.14, 321 );
a.m ( 123, 321 );
```

Bei diesem Aufruf hingegen ist nicht klar, welche der beiden Methoden m von X gemeint ist, denn beide sind gleichermaßen möglich. Daher führt diese Zeile zu einer Fehlermeldung des Compilers nebst Abbruch der Übersetzung.

TECHNISCHE UNIVERSITÄT DARMSTADT

Konstruktoren
und
Static Initializer

Konstruktoren haben wir schon öfters gesehen, nun noch einmal ein paar systematische Anmerkungen. Static Initializer sind in gewisser Weise analog zu Konstruktoren, nur nicht für ein einzelnes Objekt, sondern für die Klasse als Ganzes.

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kein Rückgabetyp
  - Auch nicht void
- Name des Konstruktors ist gleich Name der Klasse
- Ansonsten im Prinzip identisch zu void-Methoden

***Erinnerung*: Wir haben schon gesehen, dass Konstruktoren eine besondere Art von Methoden sind, die nur bei Anwendung des Operators new aufgerufen werden. Der Name eines Konstruktors ist immer der Name der Klasse, andere Methoden dürfen nicht wie die Klasse heißen. Und der Rückgabetyp beziehungsweise void wird ganz ausgelassen.**

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public final int DAY_OF_WEEK;
   .........
}
```

Wir schauen uns ein Beispiel für Konstruktoren an, das einen neuen Punkt einführt: Objektkonstanten müssen nicht unbedingt gleich in ihrer Definition initialisiert werden, wie wir das in Kapitel 03b, Abschnitt „Referenzen und Objekte“, gesehen hatten. sondern das ist auch im Konstruktor möglich. Die Objektkonstante in diesem Beispiel heißt DAY_OF_WEEK, also Wochentag.

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public X ( ) {
  DAY_OF_WEEK =
      Calendar.getInstance().get
            ( Calendar.DAY_OF_WEEK );
}

public X ( int selectedDayOfWeek ) {
  DAY_OF_WEEK = selectedDayOfWeek;
}
```

Diese Klasse X soll zwei Konstruktoren haben, die Sie auf dieser Folie sehen.

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public X ( ) {
  DAY_OF_WEEK =
      Calendar.getInstance().get
            ( Calendar.DAY_OF_WEEK );
}

public X ( int selectedDayOfWeek ) {
  DAY_OF_WEEK = selectedDayOfWeek;
}
```

Die Regel ist: Eine Objektkonstante wird entweder in ihrer Definition initialisiert oder ihr wird in jedem Konstruktor der Klasse genau einmal ein Wert zugewiesen. Letzteres ist hier der Fall.

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public X ( ) {
  DAY_OF_WEEK =
      Calendar.getInstance().get
            ( Calendar.DAY_OF_WEEK );
}

public X ( int selectedDayOfWeek ) {
  DAY_OF_WEEK = selectedDayOfWeek;
}
```

Im zweiten Konstruktor wird die Objektkonstante einfach auf den Wert des Parameters des Konstruktors gesetzt.

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public X ( ) {
  DAY_OF_WEEK =
     Calendar.getInstance().get
          ( Calendar.DAY_OF_WEEK );
}

public X ( int selectedDayOfWeek ) {
  DAY_OF_WEEK = selectedDayOfWeek;
}
```

Die Klassenmethode getInstance von java.util.Calendar liefert ein Calendar-Objekt zurück, das mit den kalendarischen Daten und der Uhrzeit des Moments initialisiert wird, in dem getInstance aufgerufen wird. Die Klasse Calendar hat eine Objektmethode get sowie eine ganze Reihe von Klassenkonstanten, unter anderem für die Verwendung als Parameter von get. Die Klassenkonstante DAY_OF_WEEK von Calendar sorgt dafür, dass get einen der int-Werte Calendar.SUNDAY ==1 bis Calendar.SATURDAY == 7 zurückliefert, nämlich den Wochentag, an dem getInstance laut Systemzeit aufgerufen wurde.

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public X () { ..... }
   public X ( int i ) { ..... }
   public X ( double d ) { ..... }
   public X ( int i, double d ) { ..... }
}
```

Die Regel zur Überladung von Konstruktoren ist dieselbe wie bei „normalen“ Methoden: Die Parametertypenlisten müssen unterschiedlich sein. Ob Bestandteile des Methodenkopfes, die nicht zur Signatur gehören, identisch so wie hier oder unterschiedlich sind, ist auch bei Konstruktoren egal.

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

Falls kein Konstruktor für eine Klasse definiert:

- **Konstruktor mit leerer Parameterliste und mit leerem Body wird vom Compiler eingefügt.**
  - **Default Constructor**
- **Die unmittelbare Basisklasse muss ebenfalls einen Konstruktor mit leerer Parameterliste haben.**
  - **Wird vom Default Constructor aufgerufen.**

Wir haben gesehen, dass man auch Klassen *ohne* Konstruktoren definieren kann. Und auch ohne Konstruktor kann man Objekte von solchen Klassen einrichten.

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

Falls kein Konstruktor für eine Klasse definiert:

- **Konstruktor mit leerer Parameterliste und mit leerem Body wird vom Compiler eingefügt.**
  - **Default Constructor**
- **Die unmittelbare Basisklasse muss ebenfalls einen Konstruktor mit leerer Parameterliste haben.**
  - **Wird vom Default Constructor aufgerufen.**

Das liegt daran, dass in diesem Fall automatisch ein leerer Konstruktor eingerichtet wird, der beim Einrichten eines Objekts durch Angabe eines leeren Klammerpaares nach Operator new und Klassennamen aufzurufen ist. Dieser Konstruktor macht aufgrund des leeren Body eben nichts.

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

Falls kein Konstruktor für eine Klasse definiert:

- **Konstruktor mit leerer Parameterliste und mit leerem Body wird vom Compiler eingefügt.**
  - ➢ **Default Constructor**
- **Die unmittelbare Basisklasse muss ebenfalls einen Konstruktor mit leerer Parameterliste haben.**
  - ➢ **Wird vom Default Constructor aufgerufen.**

Der Fachbegriff dazu ist *Default Constructor*. Allerdings gibt es zwei leicht verschiedene Definitionen dieses Begriffs, und die Formulierung im Oracle-Tutorial (Abschnitt „Providing Constructors for Your Classes“) ist auch nicht ganz eindeutig: Entweder zählt man als Default Constructor nur den vom Compiler automatisch hinzugefügten im Falle, dass man selbst keinen Konstruktor für die Klasse definiert hat; oder man zählt auch selbstdefinierte hinzu, die eine leere Parameterliste haben.

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

Falls kein Konstruktor für eine Klasse definiert:

- **Konstruktor mit leerer Parameterliste und mit leerem Body wird vom Compiler eingefügt.**
  - **Default Constructor**
- **Die unmittelbare Basisklasse muss ebenfalls einen Konstruktor mit leerer Parameterliste haben.**
  - **Wird vom Default Constructor aufgerufen.**

***Erinnerung*: Bei FopBot, konkret bei der Einführung der Roboterklasse SymmTurner, haben wir gesehen, dass ein Konstruktor einer Klasse immer auch einen Konstruktor der unmittelbaren Basisklasse mit Schlüsselwort super aufrufen muss.**

Der Default Constructor ruft den Konstruktor mit leerer Parameterliste der unmittelbaren Basisklasse auf. Dieser muss dann natürlich auch existieren, entweder implizit oder explizit definiert.

Umgekehrt gesprochen: Hat eine Klasse *keinen* Konstruktor mit leerer Parameterliste, dann hat er logisch zwingend mindestens einen explizit definierten Konstruktor, und jede davon abgeleitete Klasse muss ihrerseits mindestens einen explizit definierten Konstruktor haben, der einen Konstruktor der Basisklasse explizit mit super aufruft.

Konstruktor

TECHNISCHE UNIVERSITÄT DARMSTADT

Konstruktoren werden *nicht* vererbt!

Warum nicht:

- **Ein von Klasse X nach Klasse Y vererbter Konstruktor würde nur X initialisieren**
- **Aber auch Y muss initialisiert werden!**

→ Schwerwiegende Fehlerquelle

Zu beachten ist, dass Konstruktoren einer Basisklasse nicht in den davon abgeleiteten Klassen zur Verfügung stehen. Den Grund lesen Sie auf dieser Folie: In der Regel muss man davon ausgehen, dass für die abgeleitete Klasse eigene Initialisierungsschritte nötig sind, die aber im Konstruktor der Basisklasse fehlen. Fehlende Initialisierung ist eine Quelle für schwierig zu findende Fehler.

***Erinnerung an FopBot*: Den Fall, dass die abgeleitete Klasse zusätzlich zur Basisklasse initialisiert werden muss, hatten wir erstmals bei SlowMotionRobot gesehen (Attribut delay).**

Static Initializer

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public static final int DAY_OF_WEEK;
  static {
    DAY_OF_WEEK =
        Calendar.getInstance().get ( Calendar.DAY_OF_WEEK );
  }
  ..........
}
```

Noch eine besondere Methode, die *Static Initializer* genannt wird. Der Methodenkopf besteht nur aus dem Wort static und sonst nichts. Sie wird zu Beginn der Abarbeitung des Java-Programms ausgeführt. Der Hauptsinn ist, Klassenkonstanten zu initialisieren in Fällen, in denen es nicht möglich oder sinnvoll ist, sie schon in der Definition zu initialisieren.

Oben sehen Sie eine Klassenkonstante DAY_OF_WEEK, die nicht schon in der Definition initialisiert ist.

Static Initializer

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public static final int DAY_OF_WEEK;
  static {
    DAY_OF_WEEK =
      Calendar.getInstance().get ( Calendar.DAY_OF_WEEK );
  }
  .....
}
```

Da der Static Initializer zu Beginn der Ausführung des Programms ausgeführt wird, wird DAY_OF_WEEK in diesem Beispiel also auf den Wochentag gesetzt, an dem das Programm startet. Wenn die Ausführung des Programms sich über mehrere Tage erstreckt, ist es also wichtig zu beachten, dass DAY_OF_WEEK nicht den aktuellen Tag bezeichnet, sondern eben den bei Prozessstart – und zwar den Tag gemäß Systemzeit.

TECHNISCHE UNIVERSITÄT DARMSTADT

Finale Methoden

Oracle Java Tutorials:
Writing Final Classes and Methods

Erinnerung: Abschnitt zu finalen Klassen in Kapitel 03b. Finale Methoden werden genauso schnell gehen.

Finale Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public final void m () { ..... }
}

public class Y extends X {
   public void m () { ..... }
   public void m ( int n ) { ..... }
}
```

Ist wie hier nicht die ganze Klasse X final deklariert, sondern nur einzelne Methoden von X, dann darf von dieser Klasse zwar abgeleitet werden, ...

Finale Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public final void m () { ..... }
}

public class Y extends X {
   public void m () { ..... }
   public void m ( int n ) { ..... }
}
```

... aber diese Methoden dürfen in den abgeleiteten Klassen nicht überschrieben werden, das geht nicht durch den Compiler.

Finale Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public final void m () { ..... }
}

public class Y extends X {
   public void m () { ..... }
   public void m ( int n ) { ..... }
}
```

Die zweite Methode m in Y ist trotz der Namensgleichheit natürlich nicht berührt.

Finale Methoden

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public final void m () { ..... }
}

public class Y extends X {
   public void m () { ..... }
   public void m ( int n ) { ..... }
}
```

Generell wird empfohlen, Methoden m einer Klasse X, die im Konstruktor von X aufgerufen werden, final zu deklarieren. Denn wenn man ein Objekt einer direkt oder indirekt abgeleiteten Klasse Y erzeugt, wird ja neben dem Konstruktor von Y immer auch der Konstruktor von X aufgerufen. Der würde dann wie immer die Implementation von m für Y aufrufen, nicht die für X, weil Y der dynamische Typ ist. Wenn der Entwickler von Y nicht beachtet (oder gar nicht erst sehen kann, weil er nur den Java Byte Code von X hat), wie sich m im Konstruktor von X genau auswirkt, ist die Gefahr groß, dass die Implementation von m für Y nicht mehr zum Konstruktor von X passt. Solche Laufzeitfehler sind aller Erfahrung nach schwer zu finden.

TECHNISCHE UNIVERSITÄT DARMSTADT

Anweisungen: Variablen- und Konstantendefinitionen

Wie angekündigt, kommen wir nun zum Methodenrumpf. Der Rumpf einer Methode besteht, wie gesagt, aus einer Sequenz von Anweisungen in einem geschweiften Klammerpaar. Wir werden die verschiedenen Arten von Anweisungen nacheinander durchgehen und beginnen mit Variablendefinitionen.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m;
int n = 1;
final int r = 2;
String str1;
String str2 = new String ( “Hello World“ );
double [ ] a;
double [ ] b = new double [10];
```

Hier die Definition einer Variablen namens m vom Typ int.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m;
int n = 1;
final int r = 2;
String str1;
String str2 = new String ( “Hello World“ );
double [ ] a;
double [ ] b = new double [10];
```

Und einer Variablen, diesmal gleich mit Initialisierung. Die Variable n speichert den Wert 1.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m;
int n = 1;
final int r = 2;
String str1;
String str2 = new String ( “Hello World“ );
double [ ] a;
double [ ] b = new double [10];
```

Mit Schlüsselwort final wird aus der Variablen eine Konstante, das heißt, nach der Initialisierung darf der Wert von r nicht mehr verändert werden. Der Versuch, den Wert von r später durch eine Zuweisung zu ändern, resultiert in einem Fehler beim Übersetzen. Das werden wir uns gleich beim Thema Zuweisungen noch einmal anschauen.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m;
int n = 1;
final int r = 2;
String str1;
String str2 = new String ( “Hello World“ );
double [ ] a;
double [ ] b = new double [10];
```

Die Definition einer Variable von einer Klasse sieht exakt genauso aus, hier eine Variable str1 der vordefinierten Klasse String, die uns schon im gleichnamigen Abschnitt in Kapitel 03b begegnet ist.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m;
int n = 1;
final int r = 2;
String str1;
String str2 = new String ( “Hello World“ );
double [ ] a;
double [ ] b = new double [10];
```

Wie wir wissen, sieht die Initialisierung bei einer Klasse allerdings anders aus aufgrund der Trennung zwischen Referenz und Objekt.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m;
int n = 1;
final int r = 2;
String str1;
String str2 = new String ( “Hello World“ );
double [ ] a;
double [ ] b = new double [10];
```

Definition einer Variable a vom Typ „Array von double“.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m;
int n = 1;
final int r = 2;
String str1;
String str2 = new String ( “Hello World“ );
double [ ] a;
double [ ] b = new double [10];
```

Definition einer Arrayvariable b, die gleich auch initialisiert wird, nämlich als Verweis auf ein Array mit zehn Komponenten. Das wurde in Kapitel 01d und 03b genauer erläutert.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m;
int n = 1;
final int r = 2;
String str1;
String str2 = new String ( “Hello World“ );
double [ ] a;
double [ ] b = new double [10];
```

Das sind natürlich nur Beispiele, diese sollten aber repräsentativ für alle auftretenden Fälle sein.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int m = 1, n, k = 2;
```

Mehrere Definitionen mit und ohne Initialisierungen lassen sich mit Kommas zu *einer* Definition zusammenfassen. Alle drei Variablen – m, n und k – sind vom Typ int.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Ohne *explizite* Initialisierung:

- (Lokale) Variable: undefinierter Wert
- Konstante: nicht möglich
- Attribute und Arraykomponenten: implizit auf *Nullwert* gesetzt

Welchen Wert hat eine Variable nun, wenn sie *nicht* sofort bei der Definition initialisiert wird?

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Ohne *explizite* Initialisierung:

- **(Lokale) Variable: undefinierter Wert**
- **Konstante: nicht möglich**
- **Attribute und Arraykomponenten: implizit auf *Nullwert* gesetzt**

In normalen Variablen, man nennt sie auch *lokale* Variable, kann irgendein beliebiger Wert stehen. Wir kommen gleich darauf zurück.

Wichtig ist, dass auf eine Variable nicht lesend zugegriffen werden darf, solange sie nicht initialisiert ist. Der Versuch führt zu einer Fehlermeldung des Compilers nebst Abbruch der Übersetzung.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Ohne *explizite* Initialisierung:

- **(Lokale) Variable: undefinierter Wert**
- **Konstante: nicht möglich**
- **Attribute und Arraykomponenten: implizit auf *Nullwert* gesetzt**

Konstanten, also mit Schlüsselwort final definiert, muss man auf jeden Fall bei der Definition initialisieren. Denn später kann man den Wert ja nicht mehr setzen.

Wie wir eben im Abschnitt zu Konstruktoren und Static Initializer gesehen haben, kann die Initialisierung eines konstanten Attributs in den Konstruktor verschoben werden, aber das ändert nichts an der Situation, denn auf den Wert in der Konstante kann so oder so erst nach der Initialisierung zugegriffen werden.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Ohne *explizite* Initialisierung:

- **(Lokale) Variable: undefinierter Wert**
- **Konstante: nicht möglich**
- **Attribute und Arraykomponenten: implizit auf *Nullwert* gesetzt**

Wenn man ein Objekt einer Klasse einrichtet, aber nicht gleich dabei initialisiert, dann ist dieses Attribut automatisch mit dem *Nullwert* seines Datentyps initialisiert. Und genauso sind bei Einrichtung eines Arrayobjekts alle Komponenten automatisch mit dem Nullwert des Komponententyps initialisiert.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Nullwert:

- **Zahlentypen: 0**
- **char: ´\u0000´**
- **boolean: false**
- **Referenztypen: null**

Was ist nun dieser ominöse Nullwert?

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Nullwert:

- **Zahlentypen: 0**
- **char: ´\u0000´**
- **boolean: false**
- **Referenztypen: null**

Bei jedem primitiven Zahlentyp, egal ob ganzzahlig oder gebrochenzahlig, ist der Nullwert einfach der Zahlenwert 0. Also: Bei jedem Array eines primitiven Zahlentyps darf man davon ausgehen, dass die einzelnen Komponenten den Wert 0 haben, und bei jedem Attribut einer Klasse, das von einem primitiven Zahlentyp ist, ebenso.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Nullwert:

- **Zahlentypen: 0**
- **char: ´\u0000´**
- **boolean: false**
- **Referenztypen: null**

Der Datentyp char für Schriftzeichen ist ja im Prinzip ebenfalls ein Zahlentyp, nur dass der Zahlenwert eine spezielle Bedeutung hat, nämlich ein bestimmtes Schriftzeichen eindeutig zu identifizieren. Auch hier ist der Nullwert die Zahl 0. Die Zahl 0 steht für kein Schriftzeichen, sondern ist genau für solche Fälle reserviert, das passt also.

***Erinnerung*: Für die hier verwendete Schreibweise der 0 als char schauen Sie sich nochmals Kapitel 01a, Abschnitt „Allgemein: Primitive Datentypen“, an.**

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Nullwert:

- **Zahlentypen: 0**
- **char: ´\u0000´**
- **boolean: false**
- **Referenztypen: null**

Datentyp boolean hat ja nur zwei Werte, true und false. Der Nullwert ist der Wert false.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Nullwert:

- **Zahlentypen: 0**
- **char: ´\u0000´**
- **boolean: false**
- **Referenztypen: null**

Wie Sie mehrfach gesehen haben, gibt es für Klassen, Interfaces und Arrays einen symbolischen Nullwert, der durch das Literal null ausgedrückt wird. Eine ordentlich initialisierte Variable oder Konstante von einem Referenztyp verweist entweder auf ein Objekt, oder es hat den Wert null, eine dritte Möglichkeit gibt es nicht.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n;

if ( moon.isWaning() ) {
   n = 1;
}
else {
   n = 2;
}
```

Noch einmal zurück zum undefinierten Wert von nichtinitialisierten lokalen Variablen. Bevor man eine solche Variable benutzt, muss man ihr unbedingt einen Wert zuweisen, sonst bricht der Compiler die Übersetzung mit einer Fehlermeldung ab.

Noch besser wäre natürlich eine Initialisierung gleich bei der Definition. Aber das geht nicht immer, denn manchmal kennt man den Wert, den die Variable bekommen soll, nicht vorher. Hier ein fiktives Beispiel, aber absolut repräsentatives Beispiel.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n;

if ( moon.isWaning() ) {
   n = 1;
}
else {
   n = 2;
}
```

Der Wert der nichtinitialisierten Variable n hängt von der Mondphase ab, die erst im Programmlauf abgefragt wird.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n;

if ( moon.isWaning() ) {
  n = 1;
}
else {
   n = 2;
}
```

Bei abnehmendem Mond soll n den Wert 1 haben.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n;

if ( moon.isWaning() ) {
   n = 1;
}
else {
   n = 2;
}
```

Und ansonsten den Wert 2.

Variablendefinitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int n;

if ( moon.isWaning() ) {
   n = 1;
}
else {
   n = 2;
}
```

Wichtig ist erstens, dass der Wert von n, wie gesagt, nicht vor dieser if-Abfrage gelesen werden darf, da dann noch ein Zufallswert in n steht. Zweitens ist wichtig, dass n tatsächlich in jedem Fall auf einen Wert gesetzt wird, was durch den else-Teil gewährleistet ist.

***Nebenbemerkung*: Natürlich könnten wir die Variable n bei ihrer Definition pro forma initialisieren, zum Beispiel mit 1 oder 2 oder auch mit 0. Aber dies könnte den Leser potentiell in die Irre führen, denn der Leser wird sich sicher überlegen, warum die Variable mit diesem und keinem anderen Wert initialisiert wurde. Man müsste also noch als Kommentar dazuschreiben, dass die Initialisierung nutzlos und wirkungslos ist.**

TECHNISCHE UNIVERSITÄT DARMSTADT

Anweisungen: Schlüsselwörter this und super

Diese Schlüsselwörter sind innerhalb von Methoden wichtig, um Entitäten, die dasselbe bedeuten, auch identisch benennen, aber dennoch unterscheiden zu können.

Schlüsselwort this

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StudentData {
    ..........
    private int studentID;
    ..........
    public void setStudentID ( int studentID ) {
        this.studentID = studentID;
    }
    ..........
}
```

Zuerst Schlüsselwort this: Dieses Beispiel zeigt einen häufigen Fall.

Schlüsselwort this

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StudentData {
    .........
    private int studentID;
    .........
    public void setStudentID ( int studentID ) {
        this.studentID = studentID;
    }
    .........
}
```

Ein Parameter einer Methode soll genauso heißen wie ein Attribut, weil die beiden eben letztendlich dasselbe sind. Natürlich könnte man einen der beiden Identifier immer ein klein bisschen variieren, so dass beide Namen unterschiedlich sind. Aber das wäre schon misslich, immer neue, leicht variierte Namen erfinden zu müssen, um Namenskonflikte zu vermeiden.

Schlüsselwort this

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class StudentData {
    .........
    private int studentID;
    .........
    public void setStudentID ( int studentID ) {
        this.studentID = studentID;
    }
    .........
}
```

Mit this und einem Punkt vor dem Namen wird in der Methode das Attribut angesprochen, ohne this vorneweg wird der Parameter angesprochen.

Schlüsselwort this

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public X ( int n ) { ..... }
  public X () {
    this ( 123 );
  }
}
```

So, wie auf dieser Folie gezeigt, kann man auch in einem Konstruktor einer Klasse einen anderen Konstruktor derselben Klasse aufrufen: einfach Schlüsselwort this gefolgt von den Parametern des anderen Konstruktors. Das ist beispielsweise sinnvoll, wenn wie hier zwei Konstruktoren eigentlich dasselbe tun sollen, aber der eine Konstruktor hat einen Parameter weniger als der andere, weil er den Parameterwert selbst festlegt und nicht durch den Nutzer festlegen lässt.

Schlüsselwort super

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public int n;
   public void m1() {
      .........
   }
}
```

```
public class Y extends X {
   public double n;
   public void m1() { ......... }
   public void m2() {
      n = 3.14;
      super.n = 123;
      m1();
      super.m1();
   }
}
```

Um das Schlüsselwert super zu demonstrieren, brauchen wir zwei Klassen, wobei die eine von der anderen abgeleitet ist, wieder einmal ein rein illustratives Beispiel mit X und Y.

Schlüsselwort super

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public int n;
  public void m1() {
    .....
  }
}
```

```
public class Y extends X {
  public double n;
  public void m1() { ..... }
  public void m2() {
    n = 3.14;
    super.n = 123;
    m1();
    super.m1();
  }
}
```

Wie man sieht, kann es in der abgeleiteten Klasse ein Attribut desselben Namens wie in der Basisklasse geben. Trotz der Namensgleichheit sind das zwei verschiedene Attribute der Klasse Y, die nichts miteinander zu tun haben. Ob beide Attribute vom selben Typ oder wie hier von unterschiedlichen Typen sind, ist egal.

Mit super und wieder einem Punkt wird in der abgeleiteten Klasse das Attribut aus der Basisklasse angesprochen, ohne super das Attribut aus der abgeleiteten Klasse. Voraussetzung dafür, dass das Attribut der Basisklasse angesprochen werden darf, ist natürlich, dass dieses Attribut ein entsprechendes Zugriffsrecht hat, also public oder protected, und wenn Y im selben Package wie X ist, geht auch implizit.

Attribute desselben Namens in Basisklasse und abgeleiteter Klasse sind also möglich, werden aber nicht empfohlen (siehe z.B. Java Oracle Docs, Abschnitt Hiding Fields), da dies verwirrend sein kann.

Schlüsselwort super

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public int n;
  public void m1() {
    .....
  }
}
```

```
public class Y extends X {
  public double n;
  public void m1() { ..... }
  public void m2() {
    n = 3.14;
    super.n = 123;
    m1();
    super.m1();
  }
}
```

Der wahrscheinlich deutlich häufiger auftretende Fall ist der, dass nicht ein Attribut, sondern eine Methode aus der Basisklasse in der abgeleiteten Klasse überschrieben wird, so wie hier die Methode m1.

Auch hier lässt sich die Methode aus der Basisklasse mit super und Punkt ansprechen; ohne super wird die überschreibende Methode, also die Implementation in der abgeleiteten Klasse angesprochen.

Schlüsselwort super

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   public X ( int n ) {
      .........
   }
}
```

```
public class Y extends X {
   private int n;
   public Y () {
      super ( 123 );
      n = 1;
   }
}
```

Wenn die Basisklasse einen oder mehrere Konstruktoren hat, dann muss jeder Konstruktor der abgeleiteten Klasse einen Konstruktor der Basisklasse aufrufen, und das muss die erste Anweisung im Konstruktor der abgeleiteten Klasse sein.

Schlüsselwort super

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  public X ( int n ) {
     .........
  }
}
```

```
public class Y extends X {
  private int n;
  public Y () {
    super ( 123 );
    n = 1;
  }
}
```

In diesem Fall hat die Basisklasse einen Konstruktor mit einem formalen Parameter vom Typ int, und der Konstruktor der abgeleiteten Klasse ruft ihn mit einem int-Wert als aktualem Parameter auf. Dazu schreibt man das Schlüsselwort super, einzig gefolgt von der Parameterliste.

TECHNISCHE UNIVERSITÄT DARMSTADT

Ausdrücke: Rechtsausdrücke (rvalues) und Linksausdrücke (lvalues)

Wir führen noch eine sehr nützliche Begriffsbildung ein, Rechtsausdrücke und Linksausdrücke. Die englischen Fachbegriffe sind etwas ungenau, rvalue und lvalue.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- 23 + (12.34 – ´c´) / (short)2
- ! a && (b || ! (x == 0))
- str.charAtPosition(1)
- null

➔ Haben Typ und Wert

Zuerst ein paar illustrative Beispiele zu *Rechts*ausdrücken.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- 23 + (12.34 – ´c´) / (short)2
- ! a && (b || ! (x == 0))
- str.charAtPosition(1)
- null

➔ Haben Typ und Wert

Ein illustratives, aber nicht besonders sinnreiches Beispiel für arithmetische Ausdrücke, in dem verschiedene Operatoren, ein paar implizite Konversionen und auch eine explizite Konversion vorkommen. Der Typ dieses Ausdrucks ist double.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- 23 + (12.34 – ´c´) / (short)2
- ! a && (b || ! (x == 0))
- str.charAtPosition(1)
- null

➔ Haben Typ und Wert

Ein boolescher Ausdruck, also ein logischer Ausdruck oder anders gesagt ein Ausdruck vom Typ boolean. Die Variablen a und b sind also vom Typ boolean, und die Logikoperatoren sind anwendbar. Der Typ des Gesamtausdrucks ist ebenfalls boolean.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- 23 + (12.34 – ´c´) / (short)2
- ! a && (b || ! (x == 0))
- str.charAtPosition(1)
- null

➔ Haben Typ und Wert

Bedingungen wie hier dieser Test auf Gleichheit liefern boolean zurück, dürfen also so wie hier Bestandteil eines booleschen Ausdrucks sein.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- 23 + (12.34 – ´c´) / (short)2
- ! a && (b || ! (x == 0))
- str.charAtPosition(1)
- null

➔ Haben Typ und Wert

Der Aufruf einer Methode, die nicht void ist, ist ein Rechtsaudruck. Der Typ des Ausdrucks ist der Rückgabetyp der Methode, in diesem Fall char, und der Wert des Ausdrucks ist der Rückgabewert des Ausdrucks.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- 23 + (12.34 – ´c´) / (short)2
- ! a && (b || ! (x == 0))
- str.charAtPosition(1)
- null

➔ Haben Typ und Wert

Auch der symbolische Wert null ist ein Beispiel für Rechtsausdrücke.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- 23 + (12.34 – ´c´) / (short)2
- ! a && (b || ! (x == 0))
- str.charAtPosition(1)
- null

➔ Haben Typ und Wert

Das alles sind Beispiele für Rechtsausdrücke. Jeder Rechtsausdruck hat einen Wert, und der ist von einem bestimmten Typ.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

Beispiele:

- 23 + (12.34 – ´c´) / (short)2
- ! a && (b || ! (x == 0))
- str.charAtPosition(1)
- null

➔ Haben Typ und Wert

Erinnerung: Pro forma hat auch null in Java einen Typ, und der hat den Namen NullType. Daher passt auch null in die Systematik.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

`int n = 2 * 3 + 1;` `n = 3 * 4 + 2;`	Rechte Seite einer Initialisierung oder Zuweisung
`a.meth ( 4 * 5 + 3 );`	Aktualer Parameter
`return 5 * 6 + 4;`	Rückgabewert
`n = b [ 7 * 8 + 5 ];`	Arrayindex (int)

Wie der Name schon nahelegt, sind Rechtsausdrücke genau die Ausdrücke, die auf der rechten Seite stehen können ...

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

int n = 2 * 3 + 1;	Rechte Seite einer Initialisierung oder Zuweisung
n = 3 * 4 + 2;	
a.meth (4 * 5 + 3);	Aktualer Parameter
return 5 * 6 + 4;	Rückgabewert
n = b [7 * 8 + 5];	Arrayindex (int)

... bei der Initialisierung einer Variablen oder Konstanten ...

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

`int n = 2 * 3 + 1;`	Rechte Seite einer Initialisierung oder Zuweisung
`n = 3 * 4 + 2;`	
`a.meth ( 4 * 5 + 3 );`	Aktualer Parameter
`return 5 * 6 + 4;`	Rückgabewert
`n = b [ 7 * 8 + 5 ];`	Arrayindex (int)

... beziehungsweise später, nach der Definition, bei der Zuweisung eines Wertes an eine Variable.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

int n = 2 * 3 + 1;	Rechte Seite einer Initialisierung oder Zuweisung
n = 3 * 4 + 2;	
a.meth (4 * 5 + 3);	Aktualer Parameter
return 5 * 6 + 4;	Rückgabewert
n = b [7 * 8 + 5];	Arrayindex (int)

Aktuale Parameter dürfen ebenfalls beliebige Rechtsausdrücke sein, sofern der Typ passt.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

`int n = 2 * 3 + 1;`	Rechte Seite einer Initialisierung oder Zuweisung
`n = 3 * 4 + 2;`	
`a.meth ( 4 * 5 + 3 );`	Aktualer Parameter
`return 5 * 6 + 4;`	Rückgabewert
`n = b [ 7 * 8 + 5 ];`	Arrayindex (int)

Weiter sind Rechtsausdrücke auch genau das, was hinter einem return stehen darf, wobei auch hier der Typ natürlich passen muss.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

int n = 2 * 3 + 1;	Rechte Seite einer Initialisierung oder Zuweisung
n = 3 * 4 + 2;	
a.meth (4 * 5 + 3);	Aktualer Parameter
return 5 * 6 + 4;	Rückgabewert
n = b [7 * 8 + 5];	Arrayindex (int)

Schlussendlich sind Rechtsausdrücke auch das, was bei Arrays in eckigen Klammern stehen darf – sowohl bei der Einrichtung des Arrayobjektes mit Operator new zur Angabe der Größe des Arrays als auch wie hier zur Angabe eines Arrayindex.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

int n = 2 * 3 + 1;	Rechte Seite einer Initialisierung oder Zuweisung
n = 3 * 4 + 2;	
a.meth (4 * 5 + 3);	Aktualer Parameter
return 5 * 6 + 4;	Rückgabewert
n = b [7 * 8 + 5];	Arrayindex (int)

Allerdings mit einer Einschränkung: Bei Arrays in eckigen Klammern muss der Typ des Rechtsausdrucks int sein oder implizit in int konvertierbar sein.

Rechtsausdrücke (rvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursives Zusammensetzen:

```
( n + 2 ) < 5 == b
&&
( ( m != n ? 3 * x : y / 4 ) >= 5 * z + 3 )
```

Rechtsausdrücke lassen sich rekursiv zu immer komplexeren Rechtsausdrücken zusammensetzen, wie man schon an diesem einfachen Beispiel sieht. Beachten Sie, dass alle drei Zeilen zusammen *einen einzelnen*, schon reichlich komplexen Ausdruck bilden!

Linksausdrücke (lvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

int n = 2 * 3 + 1; n = 3 * 4 + 2;	Linke Seite einer Initialisierung oder Zuweisung
b [4 * 5 + 2]	Arraykomponente
a.i	Attribut

➔ Verweisen auf Speicherstellen

Analog zu Rechtsausdrücken sind Linksausdrücke das, was auf der linken Seite stehen darf ...

Linksausdrücke (lvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

int n = 2 * 3 + 1;	Linke Seite einer Initialisierung oder Zuweisung
n = 3 * 4 + 2;	
b [4 * 5 + 2]	Arraykomponente
a.i	Attribut

➔ Verweisen auf Speicherstellen

... bei einer Initialisierung ...

Linksausdrücke (lvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

int n = 2 * 3 + 1;	Linke Seite einer Initialisierung oder Zuweisung
n = 3 * 4 + 2;	
b [4 * 5 + 2]	Arraykomponente
a.i	Attribut

➔ Verweisen auf Speicherstellen

... beziehungsweise bei einer Zuweisung.

Linksausdrücke (lvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

int n = 2 * 3 + 1; n = 3 * 4 + 2;	Linke Seite einer Initialisierung oder Zuweisung
b [4 * 5 + 2]	Arraykomponente
a.i	Attribut

➔ Verweisen auf Speicherstellen

Das muss keine einzelne Variable sein, sondern kann beispielsweise auch eine Komponente eines Arrays sein ...

Linksausdrücke (lvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

int n = 2 * 3 + 1;	Linke Seite einer Initialisierung oder Zuweisung
n = 3 * 4 + 2;	
b [4 * 5 + 2]	Arraykomponente
a.i	Attribut

➔ Verweisen auf Speicherstellen

... oder auch ein einzelnes Attribut eines Objekts von einer Klasse.

Linksausdrücke (lvalues)

TECHNISCHE UNIVERSITÄT DARMSTADT

int n = 2 * 3 + 1; n = 3 * 4 + 2;	Linke Seite einer Initialisierung oder Zuweisung
b [4 * 5 + 2]	Arraykomponente
a.i	Attribut

➔ Verweisen auf Speicherstellen

Allen diesen Beispielen gemeinsam ist, dass sie jeweils auf eine Speicherstelle verweisen, in der der zugewiesene Wert gespeichert werden soll. Wir schließen die Betrachtung von Rechts- und Linksausdrücken ab mit der Einsicht, dass jeder Lvalue ein Rvalue ist, aber nicht umgekehrt.

TECHNISCHE UNIVERSITÄT DARMSTADT

Ausdrücke haben Typ und Wert und optional Seiteneffekte

Jeder Ausdruck hat einen Typ und einen Wert. Der Wert eines Ausdrucks ist der Wert, der von ihm zurückgeliefert wird. Der Typ des Ausdrucks ist der Typ dieses zurückgelieferten Wertes. Daneben haben manche Ausdrücke auch Seiteneffekte.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Inkrement / Dekrement:

```
int n = 12;
System.out.println ( ++n );        // 13
System.out.println ( n−− );        // 13
System.out.println ( n );          // 12
System.out.println ( −−n );        // 11
System.out.println ( n++ );        // 11
System.out.println ( n );          // 12
```

Als erstes betrachten wir die arithmetischen Operatoren: ++ und −−. Diese beiden Operatoren sind auf arithmetische Datentypen anwendbar, aber nur auf Linksausdrücke, denn sie verändern jeweils den Wert des Ausdrucks, auf den sie angewendet werden.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Inkrement / Dekrement:

```
int n = 12;
System.out.println ( ++n );          //  13
System.out.println ( n−− );          //  13
System.out.println ( n );            //  12
System.out.println ( −−n );          //  11
System.out.println ( n++ );          //  11
System.out.println ( n );            //  12
```

Der Operator ++ erhöht den Wert des Ausdrucks, auf den er angewandt wird, um 1.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Inkrement / Dekrement:

```
int n = 12;
System.out.println ( ++n );          //  13
System.out.println ( n−− );          //  13
System.out.println ( n );            //  12
System.out.println ( −−n );          //  11
System.out.println ( n++ );          //  11
System.out.println ( n );            //  12
```

Der Operator – – vermindert den Wert von n tatsächlich um 1. Dennoch ist der auf dem Bildschirm ausgegebene Wert unverändert der Wert *vor* der Dekrementoperation.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Inkrement / Dekrement:

```
int n = 12;
System.out.println ( ++n );          //  13
System.out.println ( n−− );          //  13
System.out.println ( n );            //  12
System.out.println ( −−n );          //  11
System.out.println ( n++ );          //  11
System.out.println ( n );            //  12
```

Die Auflösung des Rätsels ergibt sich, wenn man sich danach nochmals den Wert von n anschaut. Jetzt ist tatsächlich der Dekrement vollzogen. Der Punkt ist Folgender:

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Inkrement / Dekrement:

```
int n = 12;
System.out.println ( ++n );      // 13
System.out.println ( n−− );      // 13
System.out.println ( n );        // 12
System.out.println ( −−n );      // 11
System.out.println ( n++ );      // 11
System.out.println ( n );        // 12
```

Wird der Inkrement- oder Dekrementoperator als *Präfix*operator angewandt, dann wird der *neue* Wert von n als Wert des Ausdrucks zurückgeliefert. Bei Verwendung als *Postfix*operator hingegen wird der *alte* Wert von n zurückgeliefert. Man kann sich das als Eselsbrücke so merken: erst erhöhen/vermindern, dann Wert von n beziehungsweise erst Wert von n, dann erhöhen/vermindern.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zuweisungsbasierte Operatoren:

```
a = b += c = d *= e = f;
```

```
System.out.println ( x /= y );
```

Diese Art von Operator mit Seiteneffekten lässt sich nach allem, was Sie bisher darüber wissen, schnell abhandeln. Jeder zuweisungsbasierte Operator hat einen Seiteneffekt: Die Speicherstelle, die der Linksausdruck auf der linken Seite des Operators anspricht, wird so überschrieben, wie Sie es schon kennen gelernt haben.

Ein zuweisungsbasierter Operator hat aber auch noch einen Wert, das ist der neue Wert auf der linken Seite. Dieser Wert wird zurückgeliefert und kann dann weiterverwendet werden. Der Typ des gesamten Ausdrucks ist der Typ des Linksausdrucks auf der linken Seite der Zuweisung.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zuweisungsbasierte Operatoren:

```
a = b += c = d *= e = f;
```

```
System.out.println ( x /= y );
```

Die erste Zuweisung überschreibt e mit dem Wert von f und liefert diesen Wert zurück. Der Typ des zurückgelieferten Wertes ist der Typ von e.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zuweisungsbasierte Operatoren:

```
a = b += c = d *= e = f;
```

```
System.out.println ( x /= y );
```

In d steht nach Abarbeitung dieses Operators also das Produkt aus dem Wert von f und dem vorherigen Wert von d.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zuweisungsbasierte Operatoren:

```
a = b += c = d *= e = f;

System.out.println ( x /= y );
```

Mit derselben Wert wird dann der vorherige Wert von c überschrieben.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zuweisungsbasierte Operatoren:

```
a = b += c = d *= e = f;
```

```
System.out.println ( x /= y );
```

In b steht nun die Summe aus b und dem Produkt der ursprünglichen Werte von d und f.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zuweisungsbasierte Operatoren:

```
a = b += c = d *= e = f;
```

```
System.out.println ( x /= y );
```

Und dieser Wert wird dann schlussendlich in a gespeichert.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zuweisungsbasierte Operatoren:

```
a = b += c = d *= e = f;
```

```
System.out.println ( x /= y );
```

Durch das Semikolon hinter einer Zuweisung beziehungsweise hinter einer solchen Kette von Zuweisungen wird aus diesem Ausdruck eine Anweisung. Die Rückgabe der letzten ausgeführten Zuweisung wird nicht mehr weiterverwendet.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

Zuweisungsbasierte Operatoren:

```
a = b += c = d *= e = f;
```

```
System.out.println ( x /= y );
```

Dieser **Ausdruck hat als Seiteneffekt, dass der Wert in x durch den Quotienten aus x und y überschrieben wird. Der von diesem Ausdruck zurückgelieferte Wert ist genau dieser Quotient, und dieser Quotient wird dann als aktualer Parameterwert an die Methode println übergeben.**

Achtung**: Dieses Beispiel dient nur zur Illustration, denn generell sollte man solche Konstruktionen mit Seiteneffekten aus Gründen der Verständlichkeit besser vermeiden.**

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

```
X a = new Y ();

int [ ] b = new int [ 123 ];
```

Das Schlüsselwort new ist ebenfalls ein Operator. Wie vielfach in dieser Vorlesung gesehen, liefert Operator new eine Referenz zurück, also letztendlich eine Speicheradresse. Der Seiteneffekt ist, dass ein neues Objekt vom Laufzeitsystem eingerichtet ist. Und die Adresse dieses neuen Objektes ist der Rückgabewert von new.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int m1 ( int n ) {
  System.out.println ( n );
  return 2 * n;
}

public void m2 ( int n ) {
  System.out.println ( n );
}
```

Methoden können, müssen aber nicht unbedingt Seiteneffekte haben.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int m1 ( int n ) {
  System.out.println ( n );
  return 2 * n;
}

public void m2 ( int n ) {
  System.out.println ( n );
}
```

Beide Methoden haben einen Seiteneffekt: Ausgabe des Wertes des Parameters auf dem Bildschirm nebst Zeilenumbruch.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int m1 ( int n ) {
  System.out.println ( n );
  return 2 * n;
}

public void m2 ( int n ) {
  System.out.println ( n );
}
```

Die erste Methode hat sozusagen einen Haupteffekt, nämlich einen int-Wert zurückzuliefern. Eine void-Methode hingegen hat nur Seiteneffekte. Anders gesagt: Eine void-Methode, die keine Seiteneffekte hat, hat gar keine Effekte.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  private int n;
  public int getN () {
    return n;
  }
  public void setN ( int n ) {
    this.n = n;
  }
}
```

Als Seiteneffekte zählen auch schreibende Zugriffe auf Attribute, so wie in diesem kleinen Beispiel.

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  private int n;
  public int getN () {
    return n;
  }
  public void setN ( int n ) {
    this.n = n;
  }
}
```

Diese **Methode greift nur lesend zu, hat also keinen Seiteneffekt.**

Seiteneffekte

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
  private int n;
  public int getN () {
    return n;
  }
  public void setN ( int n ) {
    this.n = n;
  }
}
```

Diese **Methode hat hingegen einen Seiteneffekt, nämlich Überschreiben des Attributs n für das Objekt, mit dem die Methode aufgerufen wird.**

TECHNISCHE UNIVERSITÄT DARMSTADT

Verzweigungen: die switch-Anweisung

In Kapitel 01a hatten wir in Abschnitt „Anweisungen abarbeiten“ die if-Verzweigung mit und ohne else sowie drei Schleifen: die while-Schleife, die do-while-Schleife und die for-Schleife. Eine weitere Art von Verzweigung, die switch-Anweisung, hatten wir ausgelassen, weil sie etwas komplizierter ist, und holen ihre Einführung jetzt nach.

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( month ) {
  case Calendar.JANUARY:
      System.out.println ( “It\´s ski season!“ );
      break;
  case Calendar.FEBRUARY:
      System.out.println ( “Winter depression...“ );
      break;
  ..........
}
```

Die switch-Anweisung sehen wir uns anhand von ein paar Beispielen an.

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( month ) {
   case Calendar.JANUARY:
        System.out.println ( “It\´s ski season!“ );
        break;
   case Calendar.FEBRUARY:
        System.out.println ( “Winter depression...“ );
        break;
   ..........
}
```

Eine switch-Anweisung wird mit dem Schlüsselwort switch eingeleitet. Dahinter kommt in runden Klammern ein Ausdruck. Dieser Ausdruck muss von einem der ganzzahligen Typen byte, char, short oder int oder von einem Enum-Typ oder von Klasse String sein. Unter dieser Voraussetzung kann er ein beliebiger Rechtsausdruck sein.

***Vorgriff*: Auch Variable der Klassen Byte, Character, Short und Integer können dank Unboxing verwendet werden, siehe Kapitel 06, Abschnitt zu Wrapper-Klassen.**

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( month ) {
  case Calendar.JANUARY:
       System.out.println ( “It\´s ski season!“ );
       break;
  case Calendar.FEBRUARY:
       System.out.println ( “Winter depression...“ );
       break;
    .....
}
```

Eine beliebige Anzahl von Fällen kann abgefragt werden, jeweils eingeleitet mit dem Schlüsselwort case. Danach kommt ein Wert, dessen Typ gleich dem Typ des Ausdrucks hinter switch oder implizit darin konvertierbar sein muss.

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( month ) {
  case Calendar.JANUARY:
      System.out.println ( “It\´s ski season!“ );
      break;
  case Calendar.FEBRUARY:
      System.out.println ( “Winter depression...“ );
      break;
  ..........
}
```

Danach kommen dann beliebig viele Anweisungen; geschweifte Klammern darum herum sind nicht nötig.

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( month ) {
  case Calendar.JANUARY:
      System.out.println ( “It\´s ski season!“ );
      break;
  case Calendar.FEBRUARY:
      System.out.println ( “Winter depression...“ );
      break;
  ……
}
```

In den allermeisten Fällen wird man am Ende eine break-Anweisung setzen. Eine break-Anweisung besteht einfach aus dem Schlüsselwort break und sonst nichts.

Ohne diese break-Anweisung am Ende eines Falles geht es dann mit den Anweisungen für den nächsten Fall weiter; mit der break-Anweisung geht es mit den Anweisungen hinter der switch-Anweisung weiter, so wie man es beispielsweise auch von Schleifen kennt: Wenn die Schleife beendet ist, geht es mit den Anweisungen weiter, die als nächstes nach dem Schleifenrumpf kommen.

***Achtung*: Erfahrungsgemäß vergisst man das break leicht, darauf muss man also besonders achten.**

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( month ) {
  case Calendar.JANUARY:
      System.out.println ( “It\´s ski season!“ );
      break;
  case Calendar.FEBRUARY:
      System.out.println ( “Winter depression...“ );
      break;
  ..........
}
```

Wie schon gesagt, können noch beliebig viele weitere Fälle in analoger Form hinzugefügt werden.

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( n ) {
  case m: .....
    .....
}
```

Wichtig ist: Der Wert hinter case muss zur Kompilierzeit schon feststehen. Ein Literal oder eine Konstante darf hier stehen oder auch ein arithmetischer Ausdruck, der nur Literale und Konstanten enthält – aber keine Variablen. Ist etwa m eine Variable, dann ist dieser Fall nicht erlaubt.

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( ..... ) {
  case m < 2: .....
    .....
}
```

Hinter case darf auch wirklich nur ein *Wert* stehen, kein boolescher Ausdruck wie bei der if-Verzweigung. Der boolesche Ausdruck, der in einem case in Java ausgewertet wird, ist der Test auf Gleichheit zwischen einerseits dem ganzzahligen Wert in runden Klammern nach dem Schlüsselwort switch und andererseits dem Wert hinter case.

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( n ) {
  case 1:
       m = n * 2;
       break;
  case 2:
       m = n / 3;
       break;
  default:
       m = 5;
}
```

Es fehlt noch ein Bestandteil von switch-Anweisungen. Den sehen wir uns in diesem kleinen Beispiel an.

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( n ) {
  case 1:
       m = n * 2;
       break;
  case 2:
       m = n / 3;
       break;
  default:
       m = 5;
}
```

Wenn vorhanden, wird der Default-Fall genau dann ausgeführt, wenn keiner der vorhergehenden Fälle zutrifft. Ist der Default-Fall nicht vorhanden, dann wird die ganze switch-Anweisung übersprungen, falls keiner der case-Fälle zutrifft.

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( month ) {
  case Calendar.JANUARY:
      .........;
      break;
  case Calendar.FEBRUARY:
  case Calendar.MARCH:
      ..........;
      break;
  ..........
}
```

Ein weiteres illustratives Beispiel, nun mit einem Enum-Typ. Dies ist auch ein Beispiel dafür, dass break manchmal doch nicht sinnvoll ist, zum Beispiel wenn in mehreren Fällen dasselbe ausgeführt wird. In diesem Beispiel wird im Fall Februar und im Fall März jeweils dieselbe Folge von Anweisungen ausgeführt. Wenn man will, dass Februar und März identisch behandelt werden, dann ist der Quelltext auf dieser Folie genau richtig.

Verzweigungen: switch

TECHNISCHE UNIVERSITÄT DARMSTADT

```
switch ( str ) {
  case “January“:
      .....
      break;
  case “February“:
      .....
      break;
  default:
      .....
}
```

Wie angekündigt nun noch ein Beispiel mit Strings. Nach dem bisher Gesagten sollte dieses Beispiel selbsterklärend sein.

Wichtig ist auch hier wieder, dass der Ausdruck hinter einem case keine Variablen enthalten darf, nur Literale und Konstanten beziehungsweise Ausdrücke, die nur Literale und Konstanten enthalten. In diesem Beispiel ist jeder Ausdruck hinter einem case ein Literal.