

- das wäre aber in diesem Fall schlecht, weil dann bei jedem Aufruf von `erfasseText` ein neues Objekt auf dem Heap erzeugt werden würde, auf das `eingabe` zeigen würde
- weil `erfasseText` von jeder anderen Methode aus `MeineEingabe` benutzt wird, würde der Heap im Laufe der Zeit „zumüllen“
- warum kann `erfasseText` überhaupt auf das Attribut `eingabe` zugreifen?
 - es gibt ja keine Objekte der Klasse !
- was also bewirkt der Attributmodifizierer `static` genau?

- um einzusehen, wozu `static` gut ist, betrachten wir eine (noch zu verbessernde) einfache Klasse:

```
public class Preis
{
    private final double MEHRWERT_STEUER_SATZ = 0.19;
    private double bruttoPreis;

    public Preis(double derBruttoPreis)
    {
        if (derBruttoPreis >= 0.0)
            bruttoPreis = derBruttoPreis;
        else
            throw new IllegalArgumentException();
    }

    public double liefereBruttoPreis()
    {
        return bruttoPreis;
    }
}
```

```
public double liefereMehrwertSteuerAnteil()
{
    return MEHRWERT_STEUER_SATZ * bruttoPreis;
}

public double liefereNettoPreis()
{
    return bruttoPreis - MEHRWERT_STEUER_SATZ * bruttoPreis;
}
}
```

- der Attributmodifizierer `final` leistet dasselbe, wie der Variablenmodifizierer `final`
 - weil sich der Mehrwertsteuersatz bis zur nächsten Gesetzesänderung nicht ändert, sollte das entsprechende Attribut eine **Konstante** sein
 - genau das bewirkt `final`

- nach dem Bauplan der Klasse `Preis` können beliebig viele Objekte erstellt werden, die man sich z.B. so veranschaulichen kann:

MEHRWERT_STEUER_SATZ	bruttoPreis
0.19	78.13
MEHRWERT_STEUER_SATZ	bruttoPreis
0.19	9.75
MEHRWERT_STEUER_SATZ	bruttoPreis
0.19	345.04
MEHRWERT_STEUER_SATZ	bruttoPreis
0.19	9506.80

- der konstante Mehrwertsteuersatz ist in jedem Objekt gleich
 - eigentlich wäre es sinnvoller, ihn **nur einmal** zu **speichern**
 - genau das passiert, wenn ein Attribut als **static** deklariert wird
- der Attributmodifizierer **static** bewirkt, dass für das betroffene Attribut **nur ein Mal Speicherplatz** zur Verfügung gestellt wird
 - und zwar **sofort beim Laden** der Klasse, unabhängig davon, ob überhaupt schon ein Objekt erzeugt wurde

- alle später erzeugten Objekte **teilen sich diesen Speicherplatz**
- Achtung:** Änderungen des Wertes eines **static**-Attributes **wirken sich** daher simultan **auf alle Objekte der Klasse aus!**

- es ist also sinnvoll, die Deklaration des Attributs für den Mehrwertsteuersatz so zu ändern:

```
public class Preis
{
    private static final double MEHRWERT_STEUER_SATZ = 0.19;
    private double bruttoPreis;

    // Konstruktor und Methoden
}
```

- der Mehrwertsteuersatz hat dann folgende Eigenschaften:
 - er wird genau **einmal gespeichert**
 - er kann danach **nicht mehr geändert** werden
 - er ist auch **für viele andere Klassen** (zum Beispiel eines Warenwirtschaftsprogramms) **von Interesse**
- daher ist es **nicht unbedingt erforderlich**, das Attribut hinter einem API zu verbergen
 - es darf **ausnahmsweise** als **public** deklariert werden

```
public class Preis
{
    public static final double MEHRWERT_STEUER_SATZ = 0.19;
    private double bruttoPreis;

    // Konstruktor und Methoden
}
```

- dies ist **eine der wenigen Ausnahmen**, in denen ein Attribut **public** sein darf
 - in der Java-Klassenbibliothek gibt es eine Reihe wichtiger **Konstanten**, die **public static final** sind
- worin besteht der Vorteil?
 - da die Attribute unabhängig von Objekten existieren, kann ihr Wert benutzt werden, ohne zuvor ein Objekt erzeugen zu müssen

- wäre **MEHRWERT_STEUER_SATZ** **private**, müsste es eine Methode geben, die den Zugriff auf das Attribut ermöglicht
 - z.B. **liefernMehrwertSteuerSatz**
- um an den Wert zu kommen, müsste eine Methode einer anderen Klasse z.B. so vorgehen:

```
Preis dummyPreis = new Preis(0.00);
double mehrwertSteuer = dummyPreis.liefernMehrwertSteuerSatz();
```

- bei Attributen, die **public** und **static** sind, kann stattdessen **durch Voranstellen des Klassennamens** direkt auf das Attribut zugegriffen werden:

```
double mehrwertSteuer = Preis.MEHRWERT_STEUER_SATZ;
```

- von unseren aus objektorientierter Sicht typischen Klassen **AnnuitaetenDarlehen** und **Preis** können zahlreiche Objekte mit unterschiedlichem Zustand erzeugt werden
 - mit anderen Worten: der **Wertebereich** der beiden Referenzdatentypen **ist** sehr **groß**
- in manchen Situationen ist es aber nicht nötig, viele verschiedene Objekte zu erzeugen
 - weil es aus inhaltlichen Gründen nur wenige „Experten“ geben kann

- bei der Berliner Verkehrsgesellschaft (BVG) gibt es zum Beispiel bei den Standardtarifen nur fünf Gruppen:
 - AB, BC, ABC, Gesamtnetz, Kurzstrecke
 - in einem Programm würden fünf „Tarifexperten“ reichen, um Auskunft über die verschiedenen Ausprägungen der Tarife geben zu können (Normalpreis, Ermäßigungspreis, 4-Fahrten-Karte, Tageskarte, Kleingruppenkarte)
- weil es in unserem Sonnensystem nur acht Planeten gibt, müsste es in einem entsprechenden Programm auch nur acht „Planetenexperten“ geben

- für derartige Fälle gibt es in Java seit der J2SE 5.0 die Möglichkeit, Klassen zu deklarieren, zu denen es nur **endlich viele Objekte** gibt, die
 - bei der Deklaration **einzelne** anzugeben sind
 - **ihren Zustand** danach **nicht ändern können**
- einfache Klassen dieses Typs **enum** (**enumeration**) sind nach folgendem Muster zu deklarieren:

```
EnumModifizierer enum EnumName
{
    EnumKonstante1 // können fehlen
    ...
    EnumKonstanteM
    KlassenrumpfDeklaration1 // können fehlen
    ....
    KlassenrumpfDeklarationN
}
```

- statt des Schlüsselwortes `class` ist `enum` zu benutzen
- `enum`-Klassen erhalten einen Namen, dessen erster Buchstabe immer groß geschrieben wird
- es sind nur folgende `Enummodifizierer` zugelassen
 - `Annotations`
 - die `Zugriffsmodifizierer` `public`, `protected` und `private`
 - die `sonstigen Enummodifizierer` `static` und `strictfp`

- dabei ist Folgendes zu beachten:
 - die Zugriffsmodifizierer `protected` und `private` dürfen `nur` dann benutzt werden, `wenn` die `enum`-Klasse als `innere Klasse` deklariert wird
 - wird kein Zugriffsmodifizierer angegeben, kann die `enum`-Klasse nur innerhalb ihres Packages benutzt werden (Zustand „friendly“)
 - `static` darf `nur` dann benutzt werden, `wenn` die `enum`-Klasse als `innere Klasse` deklariert wird
 - und dann ist es überflüssig, weil alle inneren Klassen vom Typ `enum` automatisch `static` sind

- der Rumpf von `enum`-Klassen unterscheidet sich vom Rumpf typischer Klassen nur dadurch, dass zuerst `enum-Konstanten` anzugeben sind
- danach kann alles folgen, was bei Klassen zulässig ist (Attributdeklarationen, Konstruktord deklarationen, Methodendeklarationen sowie die Deklaration innerer Klassen, innerer Interfaces, statischer Initialisierungsblöcke und so genannter instance initializer)
 - wobei aber manche Einschränkungen zu beachten sind

- die `enum`-Konstanten sind `die einzigen Objekte`, die es von `der enum-Klasse` gibt
 - formal handelt es sich um Attribute, die `public`, `static` und `final` sind
 - weswegen es üblich ist, sie `vollständig groß` zu `schreiben`
 - ihr `Datentyp` ist die `enum`-Klasse, in deren Rumpf sie deklariert werden (Rekursion)
- im einfachsten Fall besteht eine `enum`-Klasse nur aus `enum-Konstanten`
 - die durch Kommata zu trennen sind

Beispiel:

```
public enum AmpelFarbe
{
    ROT, GRUEN, GELB
}
```

- **Achtung:** obwohl es sich bei den **enum**-Konstanten um Attribute des Referenzdatentyps **AmpelFarbe** handelt, die jeweils die Attributmodifizierer **public**, **static** und **final** haben, darf das in der Deklaration nicht angegeben werden
- zulässig wäre es, auch hinter der letzten **enum**-Konstanten ein Komma zu schreiben:

```
public enum AmpelFarbe
{
    ROT, GRUEN, GELB,
}
```

- die Objekte vom Typ **AmpelFarbe**, auf die die **enum**-Konstanten zeigen, werden **vom Compiler automatisch erzeugt**
 - es sind die einzigen Objekte vom Typ **AmpelFarbe**, die es geben kann
- die **enum**-Klasse **AmpelFarbe** kann nun zum Beispiel so genutzt werden:

```
public class AmpelFarbenMain
{
    public static void main(String[] args)
    {
        schreibeBelehrung(AmpelFarbe.ROT);
        schreibeBelehrung(AmpelFarbe.GRUEN);
        schreibeBelehrung(AmpelFarbe.GELB);
    }
}
```

```
private static void schreibeBelehrung(AmpelFarbe farbe)
{
    String info = "Bei ";
    if (farbe == AmpelFarbe.GRUEN)
        info += (AmpelFarbe.GRUEN + " kannst Du fahren.");
    else
        if (farbe == AmpelFarbe.ROT)
            info += (AmpelFarbe.ROT + " musst Du warten.");
        else
            info += (AmpelFarbe.GELB + " darfst Du nur fahren, " +
                "wenn Du nicht rechtzeitig bremsen kannst.");
    System.out.println(info);
}
```

auf dem Bildschirm wird ausgegeben:

```
Bei ROT musst Du warten.
Bei GRUEN kannst Du fahren.
Bei GELB darfst Du nur fahren, wenn Du nicht rechtzeitig bremsen
kannst.
```

- zu beachten ist, dass die **enum**-Konstanten so auf dem Bildschirm ausgegeben werden, wie sie deklariert wurden
- außerdem ist bemerkenswert, dass sich der Vergleich mit Hilfe des Operators **==** sinnvoll verhält
 - wie wir noch sehen werden, gilt das in der Regel für Objekte beliebiger Klassen nicht
 - weil dabei lediglich Adressen verglichen werden
- eine weitere Besonderheit von **enum**-Konstanten wird deutlich, wenn wir die Methode **schreibeBelehrung** so neu schreiben:

```
private static void schreibeBelehrung(AmpelFarbe farbe)
{
    String info = "Bei ";
    switch (farbe)
    {
        case ROT:
            info += (AmpelFarbe.ROT + " musst Du warten.");
            break;
        case GELB:
            info += (AmpelFarbe.GELB + " darfst Du nur fahren, " +
                    "wenn Du nicht rechtzeitig bremsen kannst.");
            break;
        case GRUEN:
            info += (AmpelFarbe.GRUEN + " kannst Du fahren.");
            break;
        default:
            break;
    }
    System.out.println(info);
}
```

➤ bei der Verwendung als **case**-Label in **switch**-Anweisungen können **enum**-Konstanten benutzt werden, ohne den Klassennamen voranzustellen

- sollen die Objekte von **enum**-Klassen, d.h. die **enum**-Konstanten, über mehr Fähigkeiten verfügen, muss die Deklaration der **enum**-Klasse im Rumpf weitere Dinge enthalten
 - in diesem Fall ist hinter der letzten **enum**-Konstanten ein **Semikolon** zu schreiben
- als Beispiel betrachten eine **enum**-Klasse, deren Objekte „Experten für Papierformate“ sind

- dabei beschränken wir uns auf die A-Reihe der 1922 festgelegten DIN-Norm 476
 - wir sehen zwei Attribute vor, einen Konstruktor und drei Methoden
 - dabei müssen wir beachten, dass
 - die **enum**-Konstanten mit Hilfe des Konstruktors initialisiert werden
 - Konstruktoren von **enum**-Klassen immer **private** sein müssen
 - damit keine anderen Objekte erzeugt werden können
- Hinweis:** weil sie immer **private** sind, ist es zulässig und üblich, den Modifizierer wegzulassen

```
public enum PapierFormat
{
    DIN A0(841, 1189), DIN A1(594, 841), DIN A2(420, 594),
    DIN A3(297, 420), DIN A4(210, 297), DIN A5(148, 210),
    DIN A6(105, 148), DIN A7(74, 105), DIN A8(52, 74),
    DIN A9(37, 52), DIN A10(26, 37);

    private int breite;
    private int hoehe;

    PapierFormat(int dieBreite, int dieHoehe)
    {
        breite = dieBreite;
        hoehe = dieHoehe;
    }

    public int liefereBreite()
    {
        return breite;
    }
}
```

```
public int liefereHoehe()
{
    return hoehe;
}

public int liefereFlaeche()
{
    return breite * hoehe;
}
```

- wir benutzen die **enum**-Klasse **PapierFormat** in einem Programm **PapierFormatMain**, das zum Beispiel Folgendes leistet:

Sie erhalten Informationen zu Papierformaten.

Bitte das Papierformat eingeben [Din A0 - Din A10]: Din A7

Das Format Din A7 wird als Achtelblatt bezeichnet.

Es hat folgende Groesse:

Breite :	74 mm
Hoehe :	105 mm
Flaeche :	7770 qmm

- für **PapierFormatMain** müssen wir folgende Probleme lösen:
 - wie können wir sicherstellen, dass die Benutzer **nur zulässige Papierformate eingeben** können (**Eingabeüberprüfung**)?

- wie können wir erreichen, dass für jedes Papierformat die **passende Bezeichnung** (Bogen, Blatt, Achtelblatt, etc.) **ausgegeben** wird?

- das erste Problem lösen wir **elegant** mit Hilfe so genannter **regulärer Ausdrücke** (regular expressions)
 - dabei handelt es sich um **Zeichenketten**, die – mit Hilfe einer speziellen Syntax – jeweils **eine gewisse Menge von Zeichenketten definieren**
 - die grundlegenden Ideen hierzu stammen vom Mathematiker **Stephen Kleene** (1909 – 1994)

- sie wurden u.a. von Ken Thompson zur Gestaltung des Werkzeugs **grep** benutzt (global regular expression print), das seit 1973 zum Betriebssystem Unix gehört
- insbesondere in der seit 1987 von Larry Wall entwickelten Skriptsprache Perl werden reguläre Ausdrücke intensiv genutzt (**Quasi-Standard**)
 - aber auch (über Bibliotheken) in anderen Programmiersprachen wie C, Python, Ruby und dem .Net-Framework
- in Java werden sie seit der J2SE 5.0 unterstützt
 - u.a. durch die Klasse **java.util.regex.Pattern**
- **Achtung:** die Syntax regulärer Ausdrücke ist in den verschiedenen Programmiersprachen ähnlich, aber nicht identisch

- für unsere Zwecke besonders geeignet ist folgende Methode der Klasse **String** :
`public boolean matches(String regulärerAusdruck)`
 - sie liefert **true**, wenn die Zeichenkette, für die sie aufgerufen wird, zu der Menge von Zeichenketten gehört, die durch **regulärerAusdruck** definiert ist
 - ansonsten liefert sie **false**
- wir benutzen sie, um eine Variante unserer Methode **erfasseText** zu schreiben, die wir sofort unserer Klasse **MeineEingabe** hinzufügen

```
public static String erfasseText(String eingabeAufforderung,
                                String eingabeFormat)
{
    String text = "";
    boolean eingabeOk = false;
    while (!eingabeOk)
    {
        text = erfasseText(eingabeAufforderung);
        eingabeOk = text.matches(eingabeFormat);
    }
    return text;
}
```

- das zweite Problem lösen wir **unelegant** mit Hilfe einer länglichen **switch**-Anweisung
- **PapierFormatMain** nimmt daher folgende Gestalt an :

```
public class PapierFormatMain
{
    public static void main(String[] args)
    {
        System.out.println("\nSie erhalten Informationen zu Papierformaten.\n");
        String eingabe = erfassePapierFormat();
        schreibeFormatInfo(eingabe);
    }

    private static String erfassePapierFormat()
    {
        String info = "Bitte das Papierformat eingeben [Din A0 - Din A10]: ";
        return MeineEingabe.erfasseText(info, "Din A{0-9}|10");
    }

    private static void schreibeFormatInfo(String papierFormat)
    {
        String ausgabe = "\n\nDas Format " + papierFormat;
        papierFormat = papierFormat.replace("in ", "IN");
        PapierFormat format = PapierFormat.valueOf(papierFormat);
        if (format == PapierFormat.DINA9 || format == PapierFormat.DINA10)
            ausgabe += " hat keine spezielle Bezeichnung.";
        else
            ausgabe += (" wird als " + liefereBezeichnung(format) + " bezeichnet.");
    }
}
```

```
ausgabe += "\n\nEs hat folgende Groesse:\n\n";
ausgabe += formatiereZeile("Breite", format.liefereBreite(), "mm");
ausgabe += formatiereZeile("Hoehe", format.liefereHoehe(), "mm");
ausgabe += formatiereZeile("Flaeche", format.liefereFlaeche(), "qmm");
System.out.println(ausgabe);

private static String liefereBezeichnung(PapierFormat format)
{
    String bezeichnung = "";
    switch (format)
    {
        case DINA0 :
            bezeichnung = "Doppelbogen";
            break;
        case DINA1 :
            bezeichnung = "Bogen";
            break;
        case DINA2 :
            bezeichnung = "Halbbogen";
            break;
        case DINA3 :
            bezeichnung = "Viertelbogen";
            break;
    }
}
```



```
case DINA4 :
    bezeichnung = "Blatt";
    break;
case DINA5 :
    bezeichnung = "Halbblatt";
    break;
case DINA6 :
    bezeichnung = "Viertelblatt";
    break;
case DINA7 :
    bezeichnung = "Achtelblatt";
    break;
case DINA8 :
    bezeichnung = "Sechzehntelblatt";
    break;
default :
    break;
}
return bezeichnung;
}

private static String formatiereZeile(String name, int wert, String einheit)
{
    return String.format("%-8s:%12d%4s\n", name, wert, einheit);
}
}
```

- besondere Beachtung verdienen folgende Anweisungen:

```
papierFormat = papierFormat.replace("in ", "IN");
PapierFormat format = PapierFormat.valueOf(papierFormat);
```

- die erste **verändert die Zeichenkette**:
aus **Din A4** wird zum Beispiel **DINA4**
- die zweite benutzt die statische Methode **valueOf**, die vom Compiler automatisch erzeugt wurde und **in jeder enum-Klasse automatisch vorhanden** ist
- der Output von **valueOf** ist die **enum-Konstante**, deren Namen als Input übergeben wurde

- nicht selten ist wenig eleganter Code wie die Methode **liefereBezeichnung** ein Indiz dafür, dass die Struktur der involvierten Klassen nicht gut ist
 - so ist es auch hier: die „Experten für Papierformate“ sollten selbst „wissen“, wie sie bezeichnet werden
- wir verbessern die Klasse **PapierFormat** entsprechend und nutzen dabei aus, dass wir deren Interna jederzeit ändern können, sofern das bereits vorhandene API davon nicht betroffen ist:
 - wir fügen **bezeichnung** als **weiteres Attribut** hinzu

- wir fügen einen **weiteren Konstruktor** hinzu und passen den Code des bereits vorhandenen Konstruktors der neuen Situation an
 - wir sorgen dafür, dass die **enum-Konstanten** mit dem jeweils passenden Konstruktor erzeugt werden
 - wir fügen **liefereBezeichnung** als **weitere Methode** hinzu
- danach sieht **PapierFormat** so aus:

```
public enum PapierFormat
{
    DINA0(841, 1189, "Doppelbogen"), DINA1(594, 841, "Bogen"),
    DINA2(420, 594, "Halbbogen"), DINA3(297, 420, "Viertelbogen"),
    DINA4(210, 297, "Blatt"), DINA5(148, 210, "Halbblatt"),
    DINA6(105, 148, "Viertelblatt"), DINA7(74, 105, "Achtelblatt"),
    DINA8(52, 74, "Sechzehntelblatt"), DINA9(37, 52, "DINA10(26, 37);

    private int breite;
    private int hoehe;
    private String bezeichnung;

    PapierFormat(int dieBreite, int dieHoehe)
    {
        breite = dieBreite;
        hoehe = dieHoehe;
        bezeichnung = "";
    }

    PapierFormat(int dieBreite, int dieHoehe, String dieBezeichnung)
    {
        breite = dieBreite;
        hoehe = dieHoehe;
        bezeichnung = dieBezeichnung;
    }
}
```

```
public int liefereBreite()
{
    return breite;
}

public int liefereHoehe()
{
    return hoehe;
}

public int liefereFlaeche()
{
    return breite * hoehe;
}

public String liefereBezeichnung()
{
    return bezeichnung;
}
}
```

- das Indiz für das schlechte Design, die unschöne Methode **liefereBezeichnung** der Klasse **PapierFormatMain**, wird dadurch überflüssig:

```
public class PapierFormatMain
{
    public static void main(String[] args)
    {
        System.out.println("\nSie erhalten Informationen zu Papierformaten.\n");
        String eingabe = erfassePapierFormat();
        schreibeFormatInfo(eingabe);
    }

    private static String erfassePapierFormat()
    {
        String info = "Bitte das Papierformat eingeben [Din A0 - Din A10]: ";
        return MeineEingabe.erfasseText(info, "Din A([0-9]|10)");
    }
}
```

```
private static void schreibeFormatInfo(String papierFormat)
{
    String ausgabe = "\n\nDas Format " + papierFormat;
    papierFormat = papierFormat.replace("in ", "IN");
    PapierFormat format = PapierFormat.valueOf(papierFormat);
    if (format == PapierFormat.DINA9 || format == PapierFormat.DINA10)
        ausgabe += " hat keine spezielle Bezeichnung.";
    else
        ausgabe += (" wird als " + format.liefereBezeichnung() + " bezeichnet.");
    ausgabe += "\n\nEs hat folgende Groesse:\n\n";
    ausgabe += formatiereZeile("Breite", format.liefereBreite(), "mm");
    ausgabe += formatiereZeile("Hoehe", format.liefereHoehe(), "mm");
    ausgabe += formatiereZeile("Flaeche", format.liefereFlaeche(), "qmm");
    System.out.println(ausgabe);
}

private static String formatiereZeile(String name, int wert, String einheit)
{
    return String.format("%-8s:%12d%4s\n", name, wert, einheit);
}
}
```

- auch wenn wir sie zum Teil erst später richtig verstehen, erwähnen wir der Vollständigkeit halber noch einige **Besonderheiten von enum-Klassen**:
 - die **enum**-Konstanten können mit **Annotations** versehen werden
 - jede einzelne von ihnen kann individuell mit einer **inneren Klasse** versehen werden, die nach speziellen Regeln zu gestalten ist
 - wenn Konstruktoren von **enum**-Klassen auf Attribute zugreifen, die **static** sind, müssen diese durch einen konstanten Ausdruck initialisiert worden sein

- jede **enum**-Klasse hat die Methode **values**, die **automatisch vom Compiler erzeugt** wird
 - sie liefert ein Array, das – der Reihe nach – alle **enum**-Konstanten enthält
- jede **enum**-Klasse **erbt** von der Klasse **java.lang.Enum**
 - dadurch hat sie weitere Methoden
 - und es werden die **Interfaces Comparable** und **Serializable** übernommen
- von **enum**-Klassen kann nicht geerbt werden
- **enum**-Klassen werden oft selbst als innere Klassen anderer Klassen deklariert

- abschließend thematisieren wir noch ein **Verhalten von Klassen**, das wir bisher als selbstverständlich angesehen haben
- dazu betrachten wir folgende Anweisungen:

```
Preis preis1 = new Preis(32.85);
System.out.printf("%.2f\n", preis1.liefereBruttoPreis()); // 32,85
Preis preis2 = new Preis(5.10);
System.out.printf("%.2f\n", preis2.liefereBruttoPreis()); // 5,10
```

- die Methode **liefereBruttoPreis** ist so deklariert:

```
public double liefereBruttoPreis()
{
    return bruttoPreis;
}
```

- ihr Code wird nur einmal (in der method area) gespeichert
- woher „weiß“ sie, dass sie beim Aufruf

```
preis1.liefereBruttoPreis()
```

den Wert **32.85** und beim Aufruf

```
preis2.liefereBruttoPreis()
```

den Wert **5.10** liefern muss?

- sie muss irgendwie unterscheiden können, dass sie einmal für das Objekt **preis1** aktiviert wurde und das andere Mal für das Objekt **preis2**

- in der Tat **hat** sie – wie **jeder Konstruktor und jede Methode**, die nicht **static** ist – **einen zusätzlichen** (ersten) **Parameter** namens **this**
 - dessen Datentyp ist die Klasse selbst
 - er wird **vom Compiler automatisch erzeugt**
 - beim Aufruf eines Konstruktors wird **this** mit einer Referenz auf das vom **new**-Operator erzeugte Objekt versehen
 - bei jedem Aufruf einer Methode erhält **this** eine Referenz auf das Objekt, für das die Methode aktiviert wurde

- über **this** greifen Konstrukturen oder Methoden dann jeweils auf das „richtige“ Objekt zu

- **liefereBruttoPreis** verhält sich also so, als wäre sie so deklariert worden:

```
public double liefereBruttoPreis(Preis this)    // nein !!!  
{  
    return this.bruttoPreis;  
}
```

- das ist aber kein korrektes Java
 - der Parameter **this** wird automatisch erzeugt
 - er darf nicht noch einmal deklariert werden!

- korrekt, aber umständlicher als unsere ursprüngliche Deklaration, wäre:

```
public double liefereBruttoPreis()  
{  
    return this.bruttoPreis;  
}
```