

252-0027

Einführung in die Programmierung

2.0 Einfache Java Programme

Thomas R. Gross

**Department Informatik
ETH Zürich**

Übersicht

- 2.0 Einfache Java Programme
- **2.1 Methoden**
 - Struktur
- **2.2 Typen und Variable**
 - 2.2.1 Einführung
 - 2.2.2 Basistypen: Einfache (eingebaute) Typen
 - 2.2.3 Deklaration von Variablen

2.1 Methoden

- Methode: Sequenz von Anweisungen mit einem Namen (dem der Methode)
- Methoden *strukturieren* die Anweisungen
 - Anstatt alle Anweisungen in einer Methode (main) unterzubringen
- Methoden erlauben es, *Wiederholungen* zu vermeiden
 - Mehrfache Ausführung, aber nur einmal im Programm(text)
- Eine (neue) Methode stellt eine neue Anweisung zur Verfügung

- **Beispiel: main im Program HelloWorld**
 - Methode enthält Anweisungen
 - Bei Aufruf der Methode werden die Anweisungen ausgeführt
 - Methode main wird automatisch aufgerufen (wenn Programm ausgeführt wird)

- **Eine Klasse kann mehrere Methoden enthalten**
 - Jede Methode definiert eine neue Anweisung
 - Aufruf nicht automatisch sondern explizit
 - Kann (auch) Methoden aufrufen

Programmieren

Programm Entwicklung

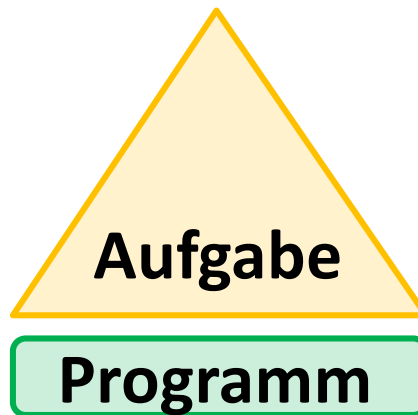
Methoden definieren

Wie zerlegen wir Aufgabe in Teilaufgaben die durch eine Methode gelöst werden

- Wie fügen wir die Teilergebnisse (von Methoden geliefert) zur Lösung der Aufgabe zusammen

(Zu) Einfaches Beispiel

- Aufgabe: Programm das Text ausgibt

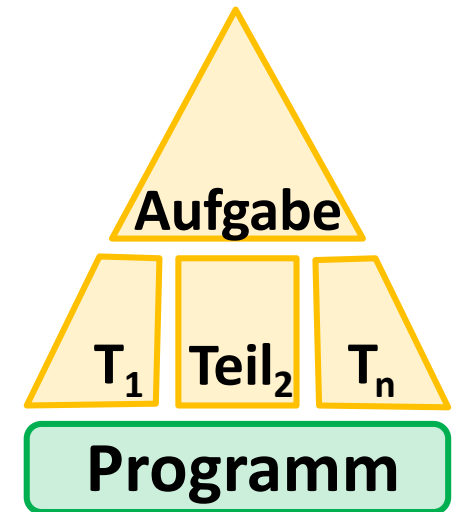


- Gesucht:

```
-----  
  
Warnung: sichern Sie die Daten  
  
-----  
  
Lange Erkl  rung  
  
-----  
  
Warnung: sichern Sie die Daten  
  
-----
```

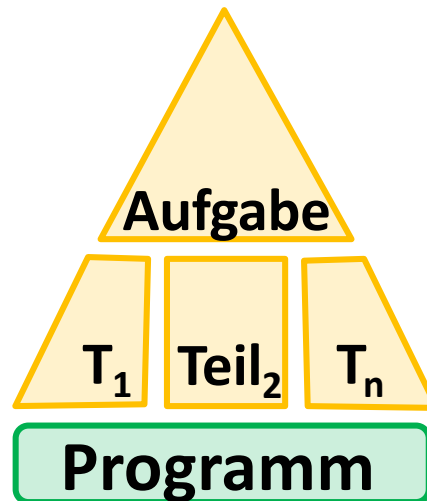
Zerlegen in Teilaufgaben

- **... soweit vereinfachen dass ein Schritt reicht**
 - Vereinfachen: zerlegen in Teilaufgaben T_1, T_2, \dots, T_n
 - Keine fixe Regeln (die immer zum Erfolg führen)
 - Statt dessen: Heuristiken
 - Hinweise, Ratschläge, («hints»)
- **Fall 1: ein Schritt/eine Java Anweisung genügt**
 - Um Teilaufgabe zu implementieren



(Zu) Einfaches Beispiel

- Aufgabe: Programm das Text ausgibt



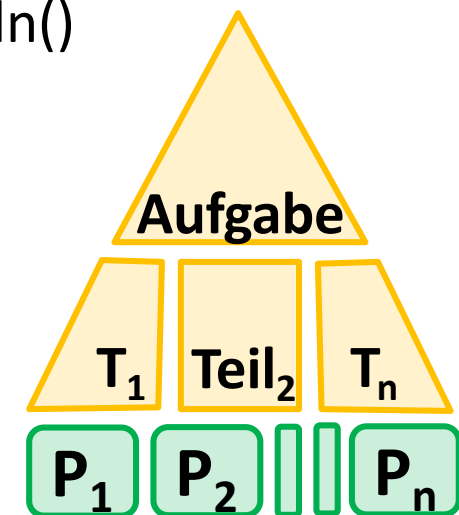
- Zerlegung

- Teilaufgabe T_i : i-te Zeile in einer `println()` Anweisung ausgeben
- Programm(e) P_i für Teilaufgaben hintereinander ausführen
- (Teil)Ergebnis(se) zu Lösung zusammenfügen

```
-----  
Warnung: sichern Sie die Daten  
-----  
  
Lange Erkl  erung  
  
-----  
  
Warnung: sichern Sie die Daten  
-----
```


Zerlegen in Teilaufgaben

- **Fall 2: Operationen hintereinander ausführen (verketteten)**
 - Resultat weiterverarbeiten –
`"hello".substring(3).toUpperCase();`
 - Hier gibt es kein «Resultat» -- der Effekt der `println()` Anweisung ist die Ausgabe auf der Konsole
 - Bald sehen wir anderen Möglichkeiten ...



(Zu) Einfaches Beispiel

```
public class PrintExample1 {  
    public static void main(String[] args) {  
        System.out.println("\n-----\n");  
        System.out.println("Warnung: sichern Sie die Daten\n");  
        System.out.println("\n-----\n");  
        System.out.println("Lange Erklaerung");  
        System.out.println("\n-----\n");  
        System.out.println("Warnung: sichern Sie die Daten\n");  
        System.out.println("\n-----\n");  
    }    // main  
}
```

(Zu) Einfaches Beispiel

T₁: System.out.println("\n-----\n");
T₂: System.out.println("Warnung: sichern Sie die Daten\n");
T₃: System.out.println("\n-----\n");
T₄: System.out.println("Lange Erklaerung");
T₅: System.out.println("\n-----\n");
T₆: System.out.println("Warnung: sichern Sie die Daten\n");
T₇: System.out.println("\n-----\n");

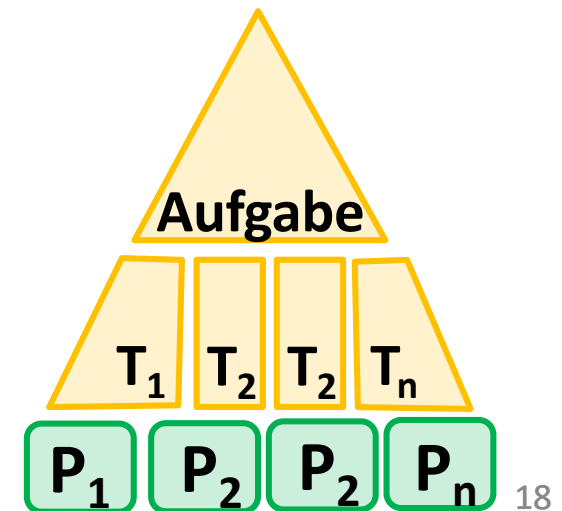
(Zu) Einfaches Beispiel

- Aufgabe: Programm das Text ausgibt
- Beobachtung: Redundanz
 - Redundanz: Wiederholungen im Programm
- Gesucht: Zerlegung, die die Struktur der Aufgabe widerspiegelt

```
-----  
Warnung: sichern Sie die Daten  
-----  
Lange Erkl aerung  
-----  
Warnung: sichern Sie die Daten  
-----
```

Zerlegen in Teilaufgaben

- ... so dass Teilaufgaben T_i wiederverwendet werden können
 - Genauer: die Anweisungen für T_i können wiederverwendet werden
- Anweisungen für T_i : Methode



(Zu) Einfaches Beispiel

- Aufgabe: Programm das Text ausgibt

```
-----  
Warnung: sichern Sie die Daten  
-----
```

- Zerlegung

- Teilaufgabe T_1 : drucke Warnung
- Teilaufgabe T_2 : drucke Erklärung

```
Lange Erkläerung
```

- Programm

- $T_1 ; T_2 ; T_1$

```
-----  
Warnung: sichern Sie die Daten  
-----
```

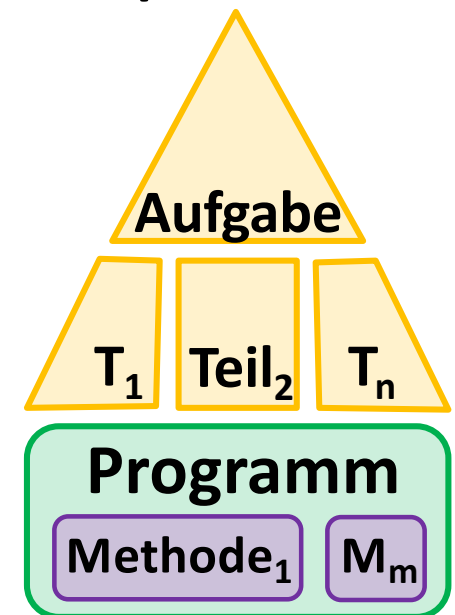
Erstellen von Programmen

Fall 1: ein Schritt/eine Java Anweisung genügt

Fall 2: Operationen hintereinander schalten (verketteten)

Fall 3:

Fall 4: Eine (selbstdefinierte) Methode genügt





(Zu) Einfaches Beispiel

```
public class PrintExample2 {  
  
    // main fehlt noch  
  
    public static void printWarning() {  
        System.out.println("\n-----\n");  
        System.out.println("Warnung: sichern Sie die Daten\n");  
        System.out.println("\n-----\n");  
    }  
  
    public static void printErklaerung() {  
        System.out.println("Lange Erklaerung");  
    }  
}
```


Programm erstellen ...

- In Teilaufgaben zerlegen
- Für jede Teilaufgabe eine Methode
 - Oder auch nur eine Anweisung ...
- **(Fall 2) Operationen (Methoden) verketten**
 - Verketten: nacheinander ausführen («hintereinander schalten»)
 - Ausführen: aufrufen
- Teilaufgaben zerlegen → Methoden schreiben → verketten

Aufruf einer Methode

- Es gibt zwei Wege eine Methode aufzurufen
 - Mit explizitem Objekt: `Objekt.methodName();` 
 - Beispiel: `System.out.println("Text"); "Hello".toUpperCase();`
 - **Ohne Objekt** 
 - Geht nur für Methoden mit besonderen Eigenschaften
 - **besondere Eigenschaften** : **static** Keyword
- **static methods** (Methoden mit Keyword **static**) werden ohne Objekt aufgerufen
 - Der **Methodenname genügt**
 - Beispiel: `printWarning();`

```

public class PrintExample2 {
    public static void main(String[] args) {
        printWarning();
        printErklaerung();
        printWarning();
    } // main

    public static void printWarning() {
        System.out.println("\n-----\n");
        System.out.println("Warnung: sichern Sie die Daten\n");
        System.out.println("\n-----\n");
    }

    public static void printErklaerung() {
        System.out.println("Lange Erklaerung");
    }

}

```

Aufruf einer Methode (mit oder ohne Objekt)

- Wenn die Anweisung `name()` (für die Methode `name`)

```
...;      // stmtN-1   irgendeine Anweisung  
name();   // stmtN    Aufruf, z.B. in main  
...;      // stmtN+1  naechste Anweisung
```

ausgeführt wird, dann wird die Methode `name` aufgerufen («invoked», «called»)

- Damit beginnt die Ausführung der Methode `name`.
 - Es gibt auch andere Wege, eine Ausführung zu starten, aber diese interessieren uns (noch) nicht.
- Wenn `name` fertig ist, geht es mit `stmtN+1` weiter

Ausführen einer Methode

- **Methode `name` wird aufgerufen (d.h. `name()`)**
 - 1. Anweisung von `name` ausgeführt
 - Gibt es weitere Anweisungen?
 - Nein: Ende der Ausführung von `name`
 - Ja:
 - Nächste Anweisung ausgeführt, weiter wie oben
- **Wir bezeichnen die Abfolge der *Ausführung* von Anweisungen als **Kontrollfluss** («control flow»)**
 - geradliniger Kontrollfluss: die *ausgeführten* Anweisungen folgen im Programm aufeinander
 - In Java: Anweisungsreihenfolge ist explizit

Kontrollfluss bei Methodenaufruf

- **Wenn eine Methode aufgerufen wird, dann**
 - «springt» die Ausführung zur Methode und führt die Anweisungen der Methode aus *und danach*
 - «springt» die Ausführung wieder zu dem Ort zurück von dem der Aufruf erfolgte
 - Und es geht weiter mit der nächsten Anweisung
- **Anordnung der Methoden im Programm(text) ohne Bedeutung**

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    }    // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    }    // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    }    // message2  
}
```

Methoden die Methoden aufrufen

Hier fangen wir an

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    } // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    } // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    } // message2  
}
```


Methoden die Methoden aufrufen

1. Anweisung

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    } // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    } // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    } // message2  
}
```

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    }    // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    }    // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    }    // message2  
}
```

Hier geht es
weiter

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    }    // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    }    // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    }    // message2  
}
```

1. und letzte
Anweisung

Methoden die Methoden aufrufen

Hier geht es weiter

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
        // main  
  
        public static void message1() {  
            System.out.println("Nachricht 1: Fertig");  
        } // message1  
  
        public static void message2() {  
            System.out.println("Die 2. Nachricht:");  
            message1();  
            System.out.println("Ende von Nachricht 2");  
        } // message2  
    }  
}
```

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    }    // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    }    // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    }    // message2  
}
```

Hier geht es
weiter

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    }    // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    }    // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    }    // message2  
}
```

1. Anweisung

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    }    // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    }    // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    }    // message2  
}
```

2. Anweisung

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    } // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    } // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    } // message2  
}
```

Hier geht es
weiter

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    }    // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    }    // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    }    // message2  
}
```

1. und letzte
Anweisung

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    }    // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    }    // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    }    // message2  
}
```

Hier geht es
weiter

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    }    // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    }    // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    }    // message2  
}
```

Letzte
Anweisung

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    } // main  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    } // message1  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    } // message2  
}
```

Hier geht es
weiter

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    } // main  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    } // message1  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    } // message2  
}
```

Letzte
Anweisung

Methoden die Methoden aufrufen

```
public class MethodsExample {  
    public static void main(String[] args) {  
        message1();  
        message2();  
        System.out.println("Ende von \"main\" ");  
    }    // main  
  
    public static void message1() {  
        System.out.println("Nachricht 1: Fertig");  
    }    // message1  
  
    public static void message2() {  
        System.out.println("Die 2. Nachricht:");  
        message1();  
        System.out.println("Ende von Nachricht 2");  
    }    // message2  
}
```

Ende main

Output

- Nachricht 1: Fertig
- Die 2. Nachricht:
- Nachricht 1: Fertig
- Ende von Nachricht 2
- Ende von "main"

Kontrollfluss

```
public class MethodsExample {
```

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

```
        message1();
```

```
        message2();
```

```
        System.out.println("Done");
```

```
    }
```

```
    ...
```

```
}
```

```
public static void message1() {  
    System.out.println("Nachricht 1: Fertig");  
}
```

```
public static void message2() {  
    System.out.println(" Die 2. Nachricht:");  
    message1();  
    System.out.println("Ende von Nachricht 2");  
}
```

```
public static void message1() {  
    System.out.println("Nachricht 1: Fertig");  
}
```


Einfaches Beispiele zur Illustration ...

Wir verwenden jetzt `println` weil es einfach ist. Spätere Java Programme benutzen eine andere Schnittstelle für Benutzer und weitere Anweisungen ...

Methoden Definition (1. Approximation)

- Zuerst definieren wir nur einfache Methoden

```
public static void name () {  
    statement(s);    // Rumpf  
}
```

- Die Methode *name* kann überall aufgerufen werden (`public static`) und gibt keinen Wert (`void`) zurück
 - Wir können uns vorstellen dass an der Stelle des Aufrufs der Rumpf (Body) der Methode ausgeführt (eingesetzt) wird.

Definition von static Methode(n)

methoddefinition \Leftarrow

```
public static void main( String[] args ) {  
    statementsequence  
}
```

othermethoddef

othermethoddef \Leftarrow { public static void *name* () {
 statementsequence

```
}
```

Definition von static Methode(n)

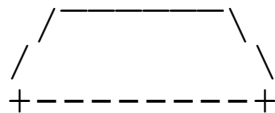
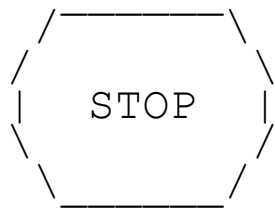
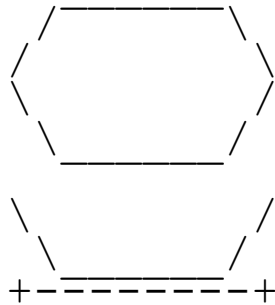
methoddefinition \Leftarrow *othermethoddef*

```
public static void main( String[] args ) {  
    statementsequence  
}
```

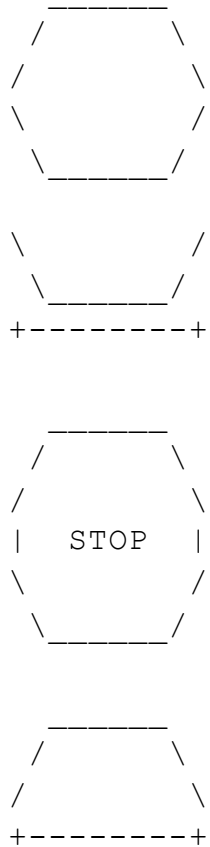
othermethoddef \Leftarrow { public static void *name* () {
 statementsequence
}

Beispiel mit static methods

Schreiben Sie ein Programm um diese Figuren zu drucken



Entwicklungsschritte



Version 1: (ohne Ausnutzen der Struktur)

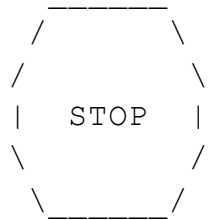
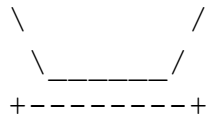
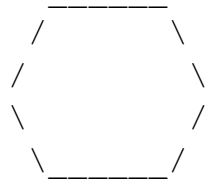
- Erstellen Sie ein Programm mit leerer `main` Methode.
- Kopieren Sie den erwünschten Output in `main` und schreiben für jede Zeile eine entsprechende `System.out.println` Anweisung.
- Führen Sie die Methode aus um den Output mit der gewünschten Figur zu vergleichen

Program Version 1

```
public class Figures1 {  
    public static void main(String[] args){  
        System.out.println("  _____");  
        System.out.println(" /           \\");  
        System.out.println("/           \\");  
        System.out.println("\\           /");  
        System.out.println(" \\_____ /");  
        System.out.println();  
        System.out.println("\\           /");  
        System.out.println(" \\_____ /");  
        System.out.println("+-----+");  
        System.out.println();  
        System.out.println("  _____");  
        System.out.println(" /           \\");  
    }  
}
```

```
        System.out.println("/           \\");  
        System.out.println("|   STOP   |");  
        System.out.println("\\           /");  
        System.out.println(" \\_____ /");  
        System.out.println();  
        System.out.println("  _____");  
        System.out.println(" /           \\");  
        System.out.println("/           \\");  
        System.out.println("+-----+");  
    } // main  
} // Figures1
```

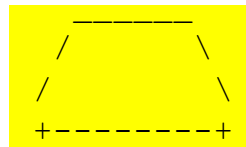
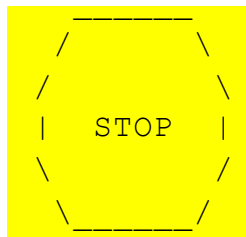
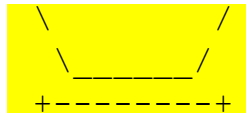
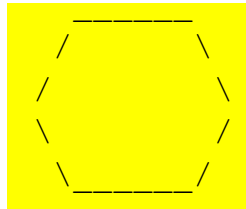
Entwicklungsschritte



Version 2: (mit Ausnutzen der Struktur, mit Redundanz)

- Identifizieren Sie (eventuell vorhandene) Structure(n).
- Unterteilen Sie die `main` Methode basierend auf Ihrer Strukturierung.
- Führen Sie die Methode aus um den Output mit der gewünschten Figur zu vergleichen

Struktur des Output



Strukturen in dieser Figur

- oben: Sechseck (Hexagon) (oder Ball ...)
- darunter: «Wanne» (oder Suppentasse ...)
- drittens «STOP Schild» Figur
- viertens «Trapez» (oder Hut Figur ...)

Struktur → Methoden :

- hexagon
- wanne
- stopSign
- hut

Program Version 2

```
public class Figures2 {  
    public static void main(String[] args) {  
        hexagon();  
        wanne();  
        stopSign();  
        hut();  
    }    //main  
}
```

...

Program Version 2, Fortsetzung

```
...
public static void hexagon() {
    System.out.println("          ");
    System.out.println(" /          \\");
    System.out.println("/          \\");
    System.out.println("\\          /");
    System.out.println(" \\          /");
    System.out.println();
}

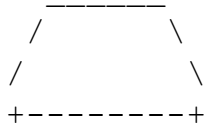
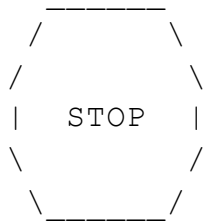
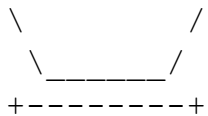
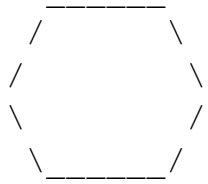
public static void wanne() {
    System.out.println("\\          /");
    System.out.println(" \\          /");
    System.out.println("+-----+");
    System.out.println();
}
...
```

Program Version 2, Fortsetzung

```
...
public static void stopSign() {
    System.out.println("      _____");
    System.out.println(" /           \\");
    System.out.println("/           \\");
    System.out.println("|   STOP   |");
    System.out.println("\\           /");
    System.out.println(" \\_____ /");
    System.out.println();
}

public static void hut() {
    System.out.println("      _____");
    System.out.println(" /           \\");
    System.out.println("/           \\");
    System.out.println("+-----+");
}
} //Figures2
```

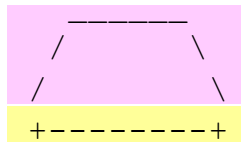
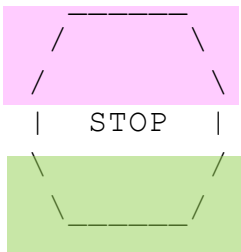
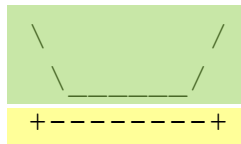
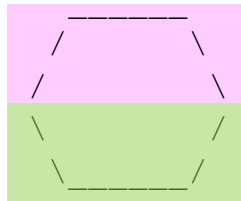
Entwicklungsschritte



Version 3 (mit Ausnutzen der Struktur, ohne Redundanz)

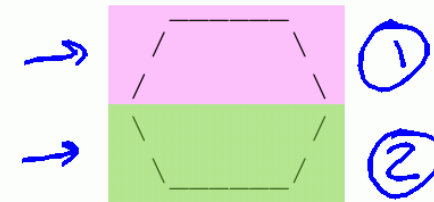
- Identifizieren Sie (eventuell vorhandene) Structure(n) und Redundanz
- Erstellen Sie Methoden um (soweit möglich) Redundanz zu vermeiden
- Kommentieren Sie den Code
- Führen Sie die Methode aus

Redundanz im Output

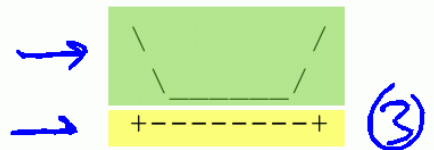


- Hexagon oberer Teil: im Stop Schild und Hut wiederverwendet
 - Hexagon unterer Teil: in Wanne und Stop Schild
 - Trennlinie: in Wanne und Hut
-
- Diese Redundanz kann durch diese Methoden ausgenutzt (d.h. eliminiert) werden:
 - `hexagonTop`
 - `hexagonBottom`
 - `line`

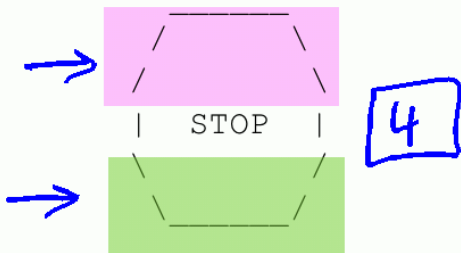
Redundanz im Output



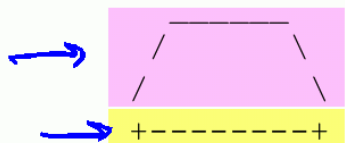
- Hexagon oberer Teil: im Stop Schild und Hut wiederverwendet



- Hexagon unterer Teil: in Wanne und Stop Schild
- Trennlinie: in Wanne und Hut



- Diese Redundanz kann durch diese Methoden ausgenutzt (d.h. eliminiert) werden:



- `hexagonTop`
- `hexagonBottom`
- `line`

Program Version 3

```
// Ihr Name, 252-0027, Herbst 2020
// Prints several figures, with methods for structure and redundancy.
public class Figures3 {
    public static void main(String[] args) {
        hexagon();
        wanne();
        stopSign();
        hut();
    }

    // Draws the top half of an an hexagon.
    public static void hexagonTop() {
        System.out.println("      _____");
        System.out.println(" /                \\");
        System.out.println("/                \\");
    }
    ...
}
```


Program Version 3, Fortsetzung

...

```
// Draws the bottom half of an hexagon.  
public static void hexagonBottom() {  
    System.out.println("\\" + " /");  
    System.out.println("  \\" + " /");  
}
```

```
// Draws a complete hexagon.  
public static void hexagon() {  
    hexagonTop();  
    hexagonBottom();  
    System.out.println();  
}
```

...

Program Version 3, Fortsetzung

```
...  
// Draws a tub («Wanne») figure.  
public static void wanne() {  
    hexagonBottom();  
    line();  
    System.out.println();  
}  
  
// Draws a stop sign figure.  
public static void stopSign() {  
    hexagonTop();  
    System.out.println("|  STOP  |");  
    hexagonBottom();  
    System.out.println();  
}
```

Program Version 3, Fortsetzung

...

```
// Draws a figure that looks sort of like a hat («Hut»).
```

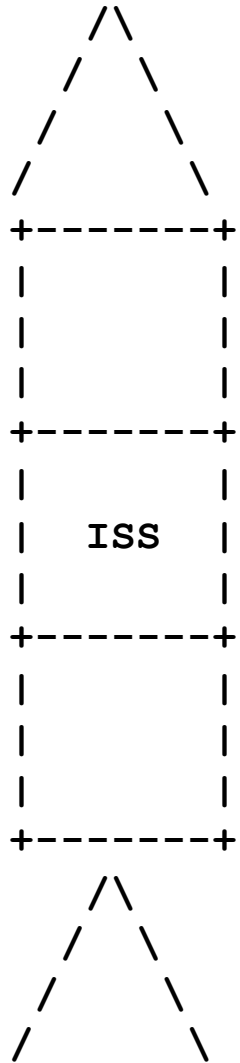
```
public static void hut() {  
    hexagonTop();  
    line();  
}
```

```
// Draws a line of dashes.
```

```
public static void line() {  
    System.out.println("+-----+");  
}
```

```
} //Figures3
```

Schreiben Sie ein Programm das
diese Rackete ausgibt:



Methoden (Übung)

Übersicht

- 2.0 Einfache Java Programme
- 2.1 Methoden
 - Struktur
- **2.2 Typen und Variable**
 - 2.2.1 Einführung
 - 2.2.2 Basistypen: Einfache (eingebaute) Typen
 - 2.2.3 Deklaration von Variablen

2.2 Typen und Variable

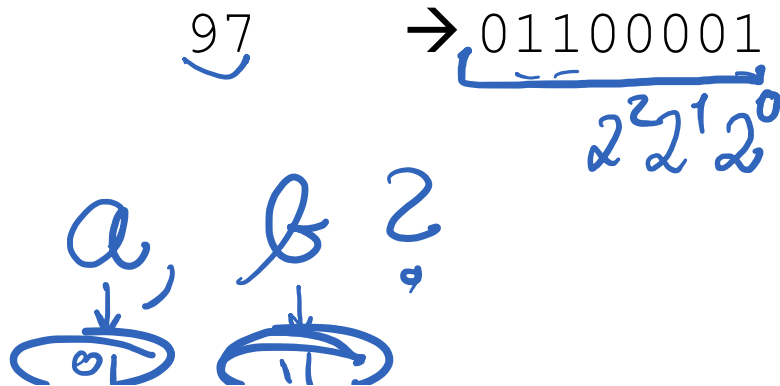
2.2.1 Einführung

Typen

- **Typen («types») beschreiben Eigenschaften von Daten**
- **Ein Typ beschreibt eine Menge (oder Kategorie) von Daten Werten.**
 - Bestimmt (beschränkt) die Operationen, die mit diesen Daten gemacht werden können
 - Viele Programmiersprachen erfordern die Angabe (Spezifikation) von Typen
 - Typen Beispiele: ganze Zahlen, reelle Zahlen, Strings
- **Typen sind Teil der Dokumentation (was verarbeitet diese Methode?)**

Typen

- Die Programmiersprache legt fest, wie ein Typ implementiert ist
 - «Implementiert» – Darstellung der Werte und Definition der Operationen
- Die interne Darstellung eines Types beruht auf Kombinationen von 1s und 0s



$$2^0 + 2^5 + 2^6$$
$$1 + 32 + 64 = 97$$

Typen

- Die Programmiersprache legt fest, wie ein Typ implementiert ist
- Die interne Darstellung eines Types beruht auf Kombinationen von 1s und 0s

97 → 01100001

"ab" → 01100001 01100010

ASCII Tabelle		
97	01100001	a
98	01100010	b

Typen

- Die Programmiersprache legt fest, wie ein Typ implementiert ist
- Die interne Darstellung eines Types beruht auf Kombinationen von 1s und 0s

97 → 01100001

"ab" → 01100001 01100010

¿ Was bedeutet 0110 0001 ?

Handwritten red annotations:
A red arrow points from the underlined binary code "0110 0001" to a red circle containing the letter "a".
Another red arrow points from the same underlined binary code to a red circle containing the number "97".
A red question mark is written to the right of these circles.

ASCII Tabelle		
97	01100001	a
98	01100010	b

Basistypen in Java

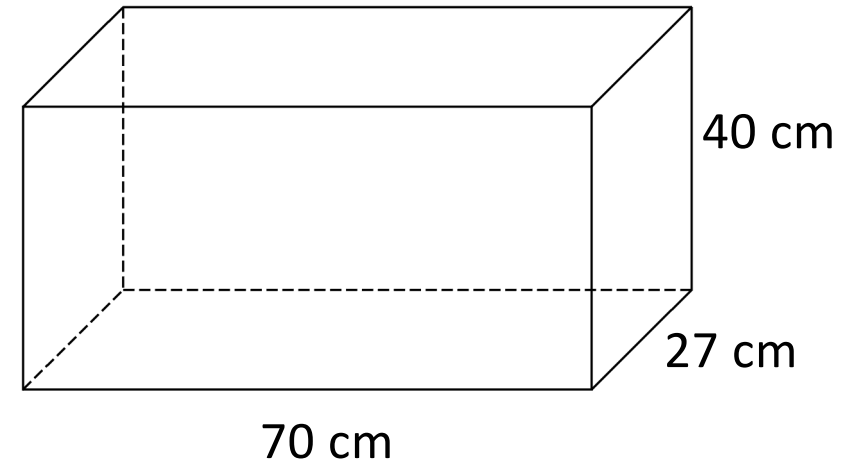
- Es gibt 8 eingebaute Typen («primitive types») für Zahlen, Buchstaben, etc.

<u>Name</u>	<u>Beschreibung</u>	<u>Beispiele</u>
int	ganze Zahlen	-2147483648, -3, 0, 42, 2147483647
long	grosse ganze Zahlen	-3, 0, 42, 9223372036854775807
double	reelle Zahlen	3.1, -0.25, 9.4e3
char	(einzelne) Buchstaben	'a', 'X', '?', '\n'
boolean	logische Werte	true, false

Oberfläche eines Quaders

- **Gegeben: Quader**

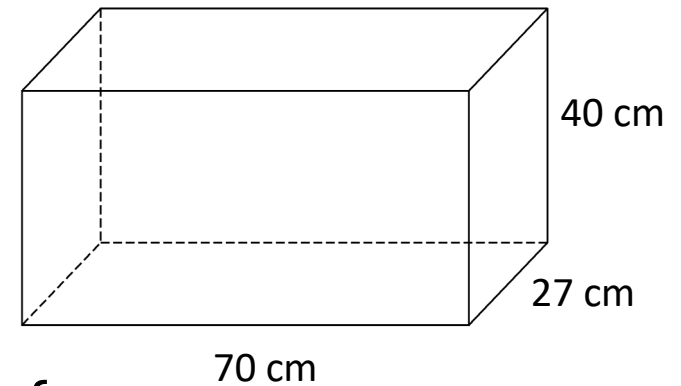
- Aufgabe: Java Programm zur Berechnung der Oberfläche



- **Oflaeche = $2 * (70 * 27 + 27 * 40 + 70 * 40) \text{ cm}^2$**

Programm ...

```
public class Quader {  
    // Berechnen wir die Oberflaeche ...  
    public static void main(String[] args) {  
        System.out.print("Die Oberflaeche ist ");  
        System.out.print(2 * (70 * 27 + 27 * 40 + 70 * 40));  
        System.out.println(" cm2");  
    } // Ende von main  
}
```

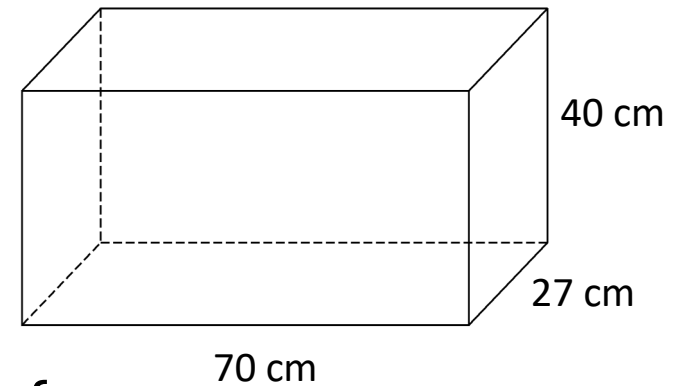


Output:

Die Oberflaeche ist 11540 cm2

Programm ...

```
public class Quader {  
    // Berechnen wir die Oberflaeche ...  
    public static void main(String[] args) {  
        System.out.print("Die Oberflaeche ist ");  
        System.out.print(2 * (70 * 27 + 27 * 40 + 70 * 40));  
        System.out.println(" cm2");  
    } // Ende von main  
}
```



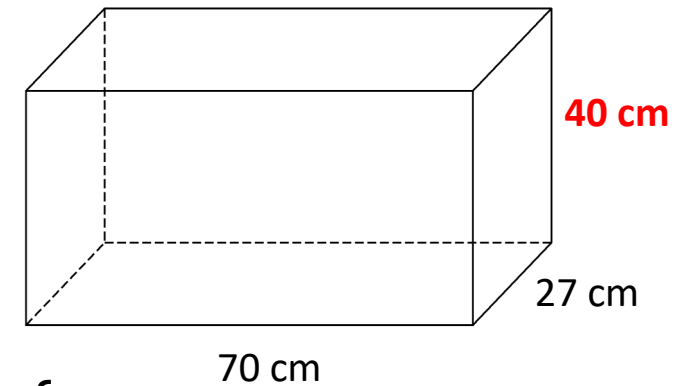
Aufmerksamer
Programmierer

Output:

Die Oberflaeche ist 11540 cm2

Programm ...

```
public class Quader {  
    // Berechnen wir die Oberflaeche ...  
    public static void main(String[] args) {  
        System.out.print("Die Oberflaeche ist ");  
        System.out.print(2 * (70 * 27 + 27 * 50 + 70 * 40));  
        System.out.println(" cm2");  
    } // Ende von main  
}
```



Unaufmerksamer
Programmierer

Output:

Die Oberflaeche ist 12080 cm2

Variable

- Wir führen einen Namen ein mit dem wir uns auf einen Wert (z.B. 40) beziehen können
- Erforderlich für Variable: Namen *und* auf was für Werte sich die Variable beziehen kann
 - Hier wären `int` Werte sinnvoll
 - Wir brauchen eine Variable für `int` Werte

Variable Deklaration: *type name*

- Beispiel: `int laenge;`
- `int` – Art (Typ) der Werte für diese Variable
 - Gleich mehr über Typen
- `laenge` – Name der Variable
 - Frei wählbar, mit Einschränkungen
 - Keine Java Keywords, muss mit Buchstabe anfangen, ...
 - Gross- und Kleinbuchstaben sind unterschiedlich
- Deklaration: Erklärung, Bekanntmachung

Programm mit Variablen

```
public class Quader {  
    // Berechnen wir die Oberflaeche ...  
    public static void main(String[] args) {  
        int laenge;  
        int hoehe;  
        int tiefe;  
        laenge = 70;  
        hoehe = 40;  
        tiefe = 27;  
        System.out.print("Die Oberflaeche ist ");  
        System.out.print(2*(laenge*tiefe + tiefe*hoehe + laenge*hoehe));  
        System.out.println(" cm2");  
    } // Ende von main  
}
```

Output:

Die Oberflaeche ist 11540 cm2

Variable Deklaration und Definition:

type name = value

- Java: können Deklaration mit Zuweisung verbinden
- Value ist ein (passender) Wert ...
 - «passend» -- d.h. vom Typ type

Programm mit Variablen

```
public class Quader {  
    // Berechnen wir die Oberflaeche ...  
    public static void main(String[] args) {  
        int laenge = 70;  
        int hoehe = 40;  
        int tiefe = 27;  
        System.out.print("Die Oberflaeche ist ");  
        System.out.print(2*(laenge*tiefe + tiefe*hoehe + laenge*hoehe));  
        System.out.println(" cm2");  
    } // Ende von main  
}
```

Output:

Die Oberflaeche ist 11540 cm2

Variable Deklaration

- **Deklaration legt Art/Typ der Werte fest – kann jeden Wert dieses Types darstellen**
- **Macht den Namen bekannt**
 - Name wird in ein Verzeichnis eingetragen
 - Variable wird nicht initialisiert wenn keine Definition
- **Gute Namen sind wichtig – besonders wenn Variable wiederholt verwendet wird**
 - ... und (manchmal) schwer zu finden
 - Kurzer Name wenn Variable nicht wichtig/langlebig ist
 - Daher oft in Beispielen (hier)

Variable Deklaration und Definition

- Das Zeichen = («Gleichheitszeichen» – aber **irreführender** Name) bedeutet **Zuweisung**



```
int laenge = 70;
```

- Der int Variable **laenge** wird der Wert **70** zugewiesen
- Wenn wir die Variable verwenden (z.B., um eine Operation auszuführen) so wird die Operation mit dem Wert (den die Variable gespeichert hat) **ausgeführt**

Wer definiert Typen?

- **Verlangen alle Programmiersprachen die Spezifikation von Typen?**
 - Nein. (Mit manchmal überraschenden Folgen)
 - Java verlangt eine Spezifikation des Typs
- **Welche Typen kann ein Java Programm verwenden?**
 - Typen die in der Sprache definiert sind: Basistypen («primitive types», integrierte Typen) – Beispiel: `int` und `long` für ganze Zahlen
 - Typen aus Bibliotheken, die immer verfügbar sind (z.B. `String`)
 - Benutzer-definierte Typen

Wer definiert Typen?

- **Verlangen alle Programmiersprachen die Spezifikation von Typen?**
 - Nein. (Mit manchmal überraschenden Folgen)
 - Java verlangt nicht immer eine Spezifikation des Typs
 - *Manchmal* kann der Compiler den Typ herausfinden
- **Welche Typen kann ein Java Programm verwenden?**
 - Typen die in der Sprache definiert sind: Basistypen («primitive types», integrierte Typen) – Beispiel: `int` und `long` für ganze Zahlen
 - Typen aus Bibliotheken, die immer verfügbar sind (z.B. `String`)
 - Benutzer-definierte Typen

Warum Typen

- **Typen beschreiben Eigenschaften von Daten**
 - Wertebereich
 - Operationen
 - Darstellung (welche Folge von 0 und 1 für einen Wert gewählt wird)
- **Richtige Wahl des Typs nötig**
 - Je nach Plattform werden unterschiedliche (Hardware) Operationen ausgeführt
 - Addition ganzer Zahlen -- Addition von reellen (Gleitkomma) Zahlen
 - Je nach Typ werden Sonderfälle behandelt
 - Resultat kann nicht dargestellt werden: Rundung? ∞ ? Andere Optionen ??

Warum Typen

- **Programme manipulieren Symbole**
 - Beispiel EBNF: Symbole aus Zeichen aus einer Menge («Alphabet»)
 - Bestimmt ob legal oder nicht – aber nur bzgl. der Struktur
 - Programme übersetzen Sprachen, spielen Schach, berechnen die Steuerschuld
 - Symbole werden verknüpft, verglichen, geändert, gelesen,
- **Symbole werden durch Folgen von 0 und 1 dargestellt**
 - Typ entscheidet die Interpretation dieser Folgen

Warum Typen

- **Typen verhindern Fehler**
 - Kann nicht die AHV Nummer zum Gehalt addieren
 - Kann nicht Volumen und Fläche addieren
- **Typen erlauben Optimierungen (der Berechnung, der Darstellung)**
- **Alle Darstellungen sind endlich!**
- **Richtige Wahl des Typs nötig**
 - Java kann dann entsprechend Platz reservieren so dass Ergebnisse von Operationen darstellbar sind (int oder long?)

Übersicht

- 2.0 Einfache Java Programme
- 2.1 Methoden
 - Struktur
- **2.2 Typen und Variable**
 - 2.2.1 Einführung
 - 2.2.2 Basistypen: Einfache (eingebaute) Typen
Operationen (mit Werten desselben und verschiedener Typen)
 - 2.2.3 Deklaration von Variablen

2.2.2 Primitive Types

- Deklaration und Definition

`int x = value;`

- Genauer:

- *value* wird durch einen Ausdruck («expression») bestimmt
- Der Ausdruck wird ausgewertet («evaluated»)

- Was sind die Regeln für Ausdrücke?

- Zuerst für `int` (und `long`)

Ausdrücke («Expressions»)

- Ausdruck («expression») für einen Typ: Ein Wert *oder* Operanden und Operator(en) die einen Wert berechnen

- Beispiele für int:
4 1 + 7
2 * 4 * 3
7 + (2 + 6) * 4

- Der einfachste Ausdruck ist ein *Literal* («*literal value*»)
 - Ein Wert der direkt im Programm erscheint (z.B. 4)
- Komplexe Ausdrücke können Teilausdrücke enthalten (später mehr)
 - (Teil)Ausdruck kann Operand sein

Arithmetische Operatoren

- **Operator: Verknüpft Werte oder Ausdrücke.**

- + Addition
- Subtraktion (oder Negation)
- * Multiplikation
- / Division
- % Modulus (Rest)

- **Während der Ausführung eines Programms werden seine Ausdrücke *ausgewertet* («*evaluated*»)**

- 1 + 1 ergibt 2
- `System.out.println(3 * 4);` ergibt (druckt) 12
 - Wie würden wir den Text `3 * 4` drucken?

EBNF Beschreibung Ausdruck (Expression)

number \Leftarrow *integer* | *integer* . { *digit* } | *sign* . *digit* { *digit* }

op \Leftarrow + | - | * | / | %

atom \Leftarrow *number* | *identifier*

term \Leftarrow (*expr*) | *atom*

expr \Leftarrow *term* { *op term* }

1. Nicht die vollständige Beschreibung für Java Ausdrücke
2. Beschreibt nur die Syntax (die Form)

Arithmetische Operatoren

- **Operator: Verknüpft Werte oder Ausdrücke.**

- + Addition

- / Division

-

- **Werte haben einen [festgelegten] Typ**

- Evaluation eines Ausdrucks ergibt Wert eines Typs

- EBNF beschreibt nur Form (int + int *ergibt* ? **int** , int + long *ergibt* ? **??**)

- Operator \otimes : Typ_A \otimes Typ_A *ergibt* Typ_A (für arithmetische Operatoren)
 Typ_A \otimes Typ_B *ergibt* ??? (hängt von \otimes , Typ_A, Typ_B ab – später)