252-0027
Einführung in die Programmierung

2. Erste Java-Programme

Manuela Fischer, Malte Schwerhoff

Departement Informatik
ETH Zürich

2. Erste Java-Programme

2.1 Einführung: Vom Quelltext zur Ausgabe

2.2 Methoden I: Programmstrukturierung

2.3 Einfache Berechnungen

2.4 Input und Random

2.5 Logische Aussagen über Programm(segment)e

Java-Programme erstellen und ausführen

Zwei Möglichkeiten:

A) Ganzes Programm

- Für jede Datei …
  - Erstellen
  - Ausführen
  - Modifizieren

Folgt jetzt

B) Einzelne Anweisungen

- Für jede Anweisung …
  - Lesen (Read)
  - Evaluieren (Ausführen)
  - Ausgeben (Print)
- REPL
  - Read-Evaluate-Print Loop

Java: Vom Quellcode zu Ausgabe

Java-Programm (Quellcode)

```
Java-Anweisung
Java-Anweisung
Java-Anweisung
Java-Anweisung
```

Editor

Übersetzt und prüft

Compiler

Java-Programm (Prozessor-Instruktionen)

```
Check
Befehl
Befehl
Befehl
Check
Check
Befehl
```

Führt aus und prüft

Prozessor + Laufzeit-system

Ausgabe (Grafik)

Ausgabe (Text)

```
Die Note ist: 6.0
```

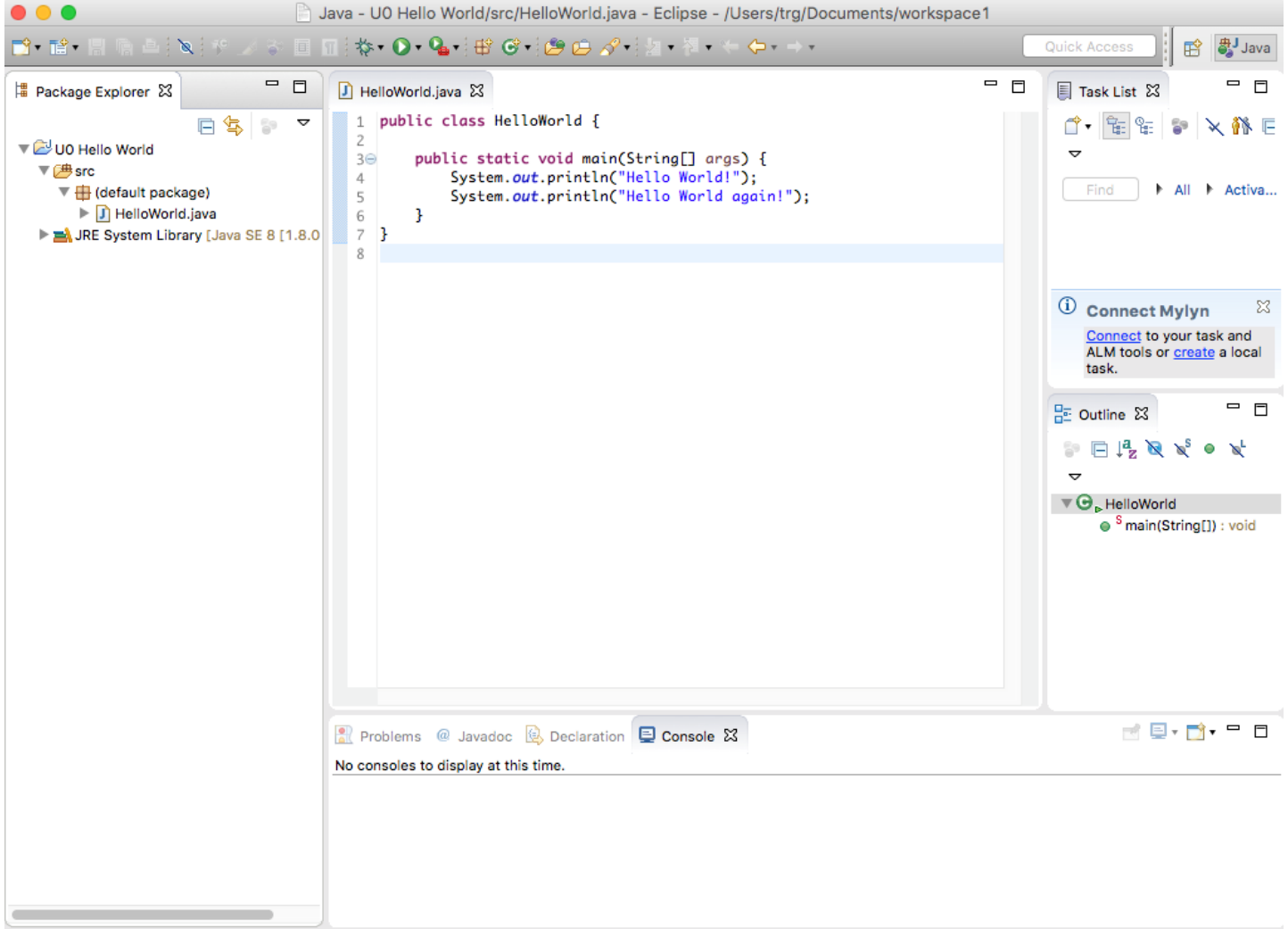
Ausgabe (...)

```
0929095 8395798
9216320 0082134
```

Java-Programme Erstellen und Ausführen

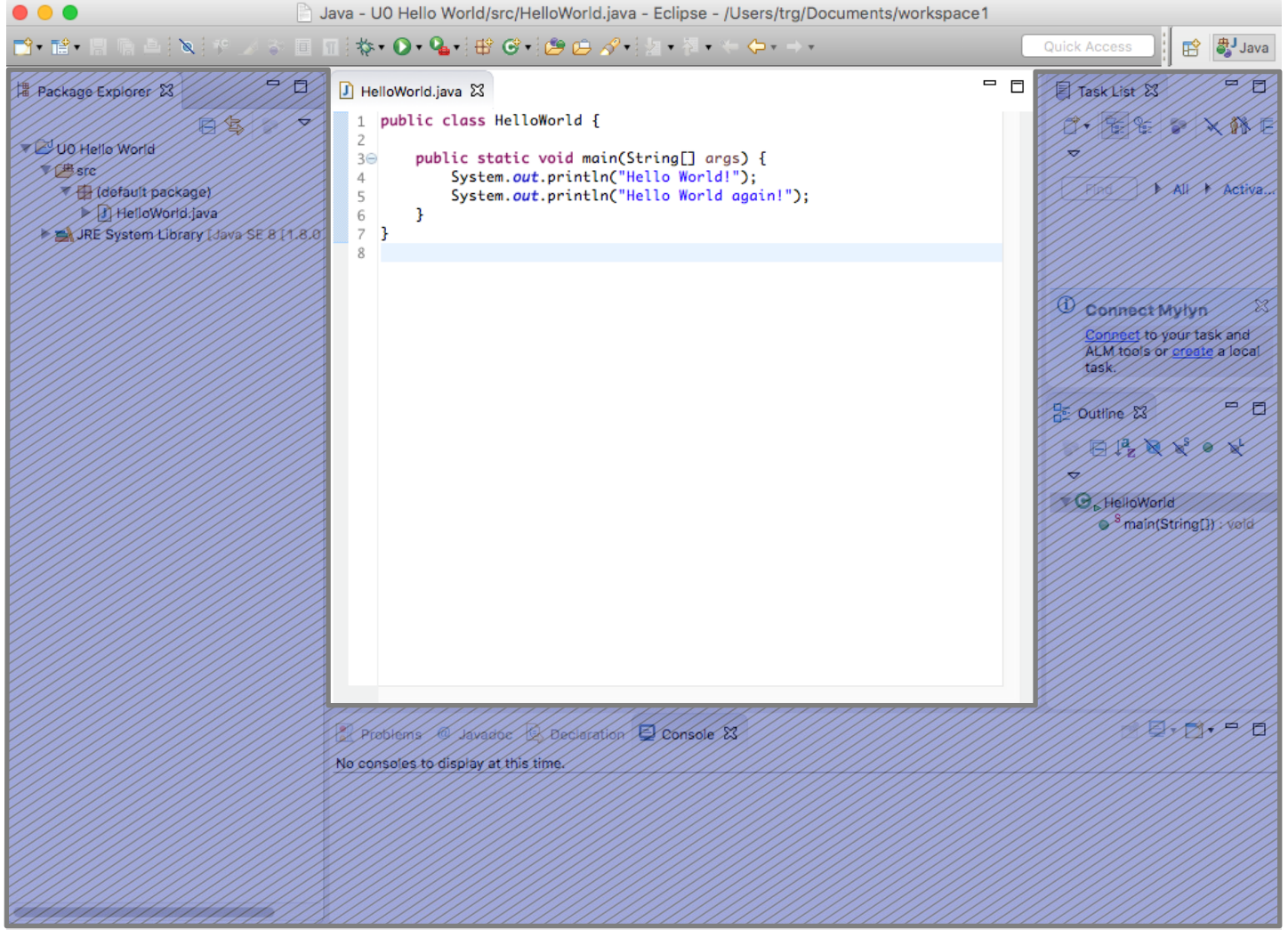
- Ganzes Programm: Für jede Datei …
  - Erstellen — mit einem Editor
  - Ausführen — Compiler übersetzt, Prozessor führt aus
  - Modifizieren — nach Analyse der erhaltenen Ausgabe

- **Entwicklungsumgebungen** («**integrated development environments**» = **IDE**) wie Eclipse
  - Enthalten notwendige Komponenten und Funktionalität
  - Sind Industriestandard, aber nicht zwingend notwendig


Java - U0 Hello World/src/HelloWorld.java - Eclipse - /Users/trg/Documents/workspace1
Quick Access
Java
Package Explorer
U0 Hello World
src
(default package)
HelloWorld.java
JRE System Library [Java SE 8 [1.8.0
HelloWorld.java
1 public class HelloWorld {
2
3 public static void main(String[] args) {
4 System.out.println("Hello World!");
5 System.out.println("Hello World again!");
6 }
7 }
8
Task List
Find
All
Activa...
Connect Mylyn
Connect to your task and ALM tools or create a local task.
Outline
HelloWorld
main(String[]) : void
Problems
Javadoc
Declaration
Console
No consoles to display at this time.

Java und Eclipse

- Viele Aspekte
- Zuerst: Fokus auf das wichtigste
  - Sie kennen Java/Eclipse schon: super
    - Aber denken Sie an die, die noch nicht so weit sind
  - Schrittweise erklären wir mehr Konzepte
- Was uns (jetzt) nicht interessiert: abdecken
  - Können diese Teile nicht ignorieren
  - Müssen diese Teile (fürs Erste) akzeptieren


Java - U0 Hello World/src/HelloWorld.java - Eclipse - /Users/trg/Documents/workspace1
Quick Access
Java
Package Explorer
U0 Hello World
src
(default package)
HelloWorld.java
JRE System Library [Java SE 8 [1.8.0
HelloWorld.java
1 public class HelloWorld {
2
3 public static void main(String[] args) {
4 System.out.println("Hello World!");
5 System.out.println("Hello World again!");
6 }
7 }
8
Task List
Find
All
Activa...
Connect Mylyn
Connect to your task and ALM tools or create a local task.
Outline
HelloWorld
main(String[]) : void
Problems
Javadoc
Declaration
Console
No consoles to display at this time.

Unser erstes Java-Programm

HelloWorld.java

```
1  public class HelloWorld {
2
3      public static void main(String[] args) {
4          System.out.println("Hello World!");
5          System.out.println("Hello World again!");
6      }
7  }
```

- Viele Aspekte nicht zwingend notwendig fürs Erste Verständnis
  - Ziele: Programme 1. lesen, 2. verstehen, 3. erstellen/modifizieren
- Wir können leider nicht alle davon vollständig ignorieren …

Unser erstes Java-Programm

HelloWorld.java

```
1  public class HelloWorld {
2
3      public static void main(String[] args) {
4          System.out.println("Hello World!");
5          System.out.println("Hello World again!");
6      }
7  }
```

Notwendige Struktur für ein eigenständiges Java-Programm

Von uns implementierte Funktionalität

Unser erstes Java-Programm

HelloWorld.java

```
1  public class HelloWorld {
2
3      public static void main(String[] args) {
4          System.out.println("Hello World!");
5          System.out.println("Hello World again!");
6      }
7  }
```

Wichtig (für den Anfang)

1. Programmname gleich Dateiname
2. Eine Klasse (`class`) pro Programm/Datei

Unser erstes Java-Programm

HelloWorld.java

```
1  public class HelloWorld {
2
3      public static void main(String[] args) {
4          System.out.println("Hello World!");
5          System.out.println("Hello World again!");
6      }
7  }
```

- `main`: Java-Methode (Java-Code den wir ausführen können)
- Methode enthält mehrere Anweisungen
  - Hier: `println("…")` — gibt Text zwischen Anführungszeichen aus

EBNF

- Hält die Syntaxregeln von Java-Programmen fest
- Beispiel: Bezeichner (Namen) in Programmen
- **Bezeichner** («**identifier**») müssen Anforderungen erfüllen, z.B.
  - Bezeichner muss mindestens ein Zeichen lang sein
  - … muss mit einem (Unicode-)Buchstaben anfangen
  - … kann Buchstaben oder Ziffern (0-9) enthalten

Wer es genau wissen möchte: Java Language Specs §3.8

EBNF-Regeln für Java (vereinfacht)

EBNF-Beschreibung für einzelne Bezeichner (<identifier>):

<lowercaseletter>	⟵	a | b | c | ... |z
<uppercaseletter>	⟵	A | B | C | ... | Z
<letter>	⟵	<lowercaseletter> | <uppercaseletter>
<digit>	⟵	0 | 1 | 2 | ... | 9
<identifier>	⟵	<letter> { <letter> | <digit> }

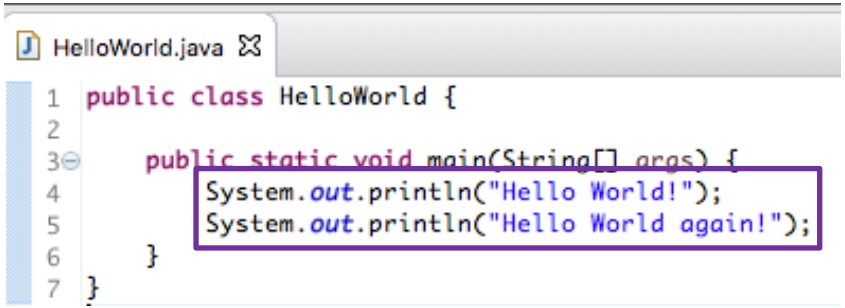
EBNF-Regeln für Java (vereinfacht)

EBNF-Beschreibung für ganze Programme (<javaprogram>):

```
<javaprogram>   ⟵   public class <identifier> {
                        <methoddefinition>
                    }
<methoddefinition>  ⟵
     public static void main ( String [] <identifier> ) {
         <statementsequence>
     }
```

Aufreihung von Anweisungen: <statementsequence>

- Methode enthält eine **Aufreihung** («**sequence**») von **Anweisungen** («**statements**»)
- Mögliche Anweisungen
  - Ein-/Ausgabe
  - Verzweigungen
  - Schleifen
  - Methoden-/Funktionsaufrufe
  - …



Aufreihung von Anweisungen: *statementsequence*

- Mögliche Anweisungen
  - Methoden-/Funktionsaufrufe
  - …
- Wichtig für viele Programmiersprachen, inklusive Java

1. Aufruf (global/lokal bekannter) Methoden
   Format: `methodenName();`
2. Methodenaufruf für ein *Objekt* (Beispiel folgt)
   Format: `objekt.methodenName();`

Typisch für Java

Beispiel

Mit dem Objekt `System.out` können wir mit der `println`-Methode Text ausgeben

```
System.out.println(…);
```

Mehr über Text in ein paar Minuten — es gibt viele Methoden für Textbearbeitung

```
"Hello".toUpperCase();
```

Objekt

Methode

Einschub: Objekte

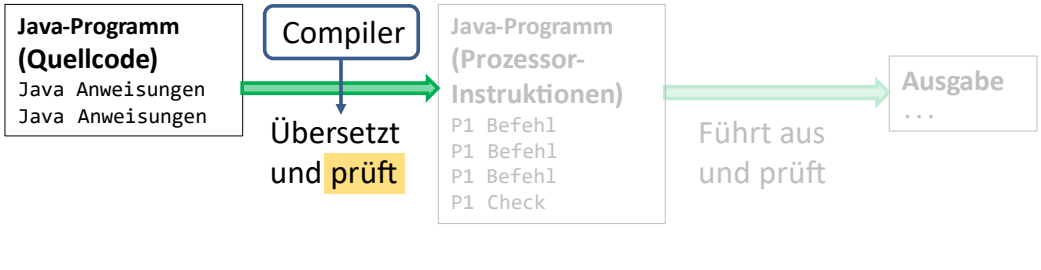
- Zentrales Konzept der Programmierung – zunächst (stark) vereinfacht, später mehr Details
- Objekt (vereinfacht): Kann von Programm erstellt, gelesen und verändert werden
  - Hat einen Zustand ➜ Ansammlung an Werten
  - Hat ein Verhalten ➜ Operationen (für oder mit diesem Objekt)
  - Hat eine Identität ➜ Objekte können unterschieden werden
- Intuitives Beispiel:
  - $5 \in \mathbb{N}$ —Wert (eine Fünf ist und bleibt eine Fünf)
  - Tabelle (mit Werten aus $\mathbb{N}$) — Objekt, kann z.B. sortiert werden

Ausführen (automatisch übersetzt – sonst geht Ausführung nicht)

Java-Programme und Compilerfehler

HelloWorld.java

```
1  public class HelloWorld {
2
3      public static void main(String[] args) {
4          System.out.println("Hello World!");
5          System.out.println("Hello World again!");
6      }
7  }
```



- Wir können Programmteile ignorieren, aber nicht weglassen
- Sonst können wir das Java-Programm nicht übersetzen
  - **Übersetzen** (**kompilieren**, «**compile**») = in ausführbare Form bringen
  - Es gibt **Fehlermeldungen** wenn wir Teile weglassen/falsch schreiben

HelloWorld.java

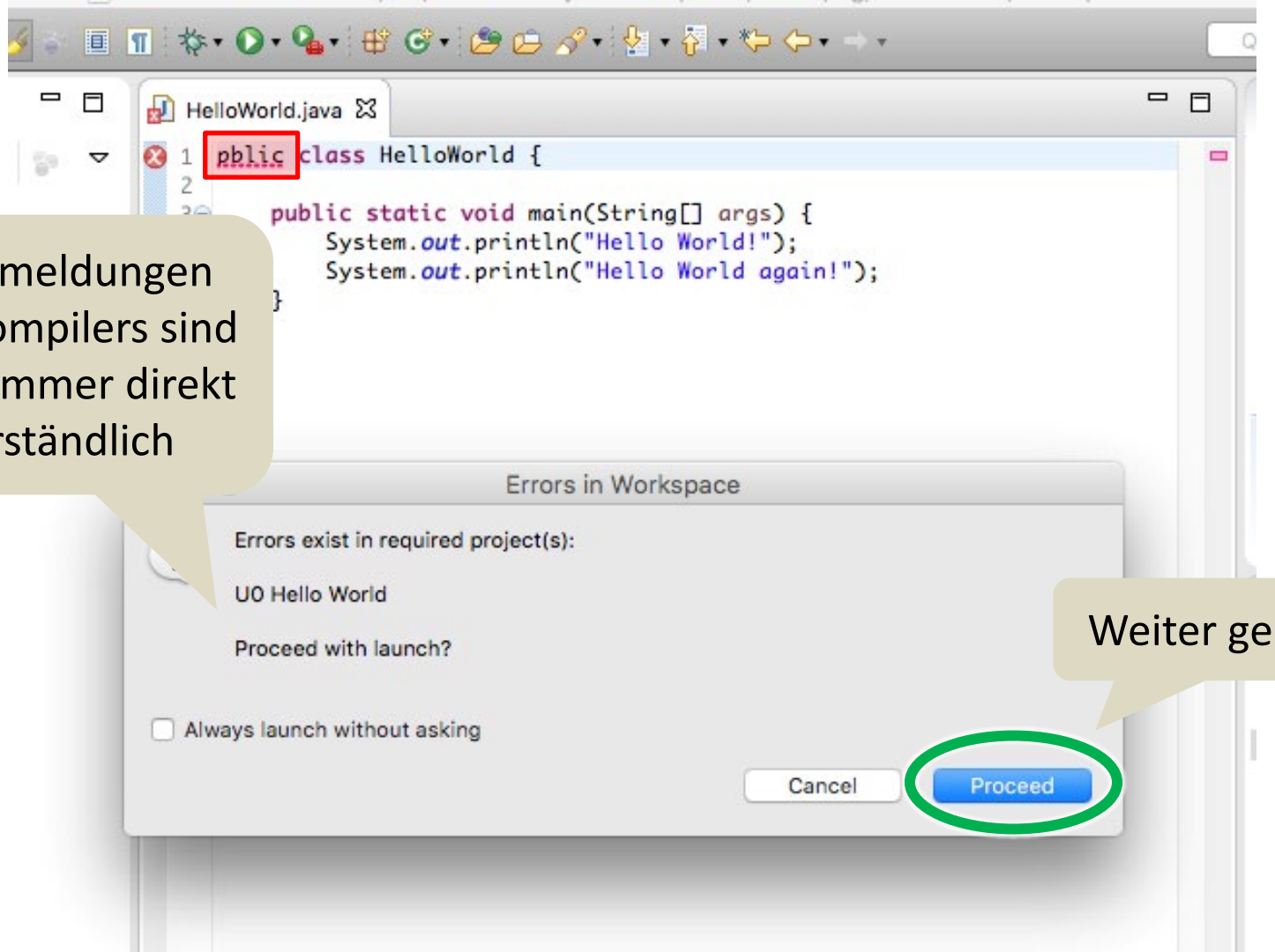
```
1  public class HelloWorld {
2
3      static void main(String[] args) {
4          System.out.println("Hello World!");
5          System.out.println("Hello World again!");
6      }
7  }
8
```

```
Error: Main method not found in class HelloWorld, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

HelloWorld.java

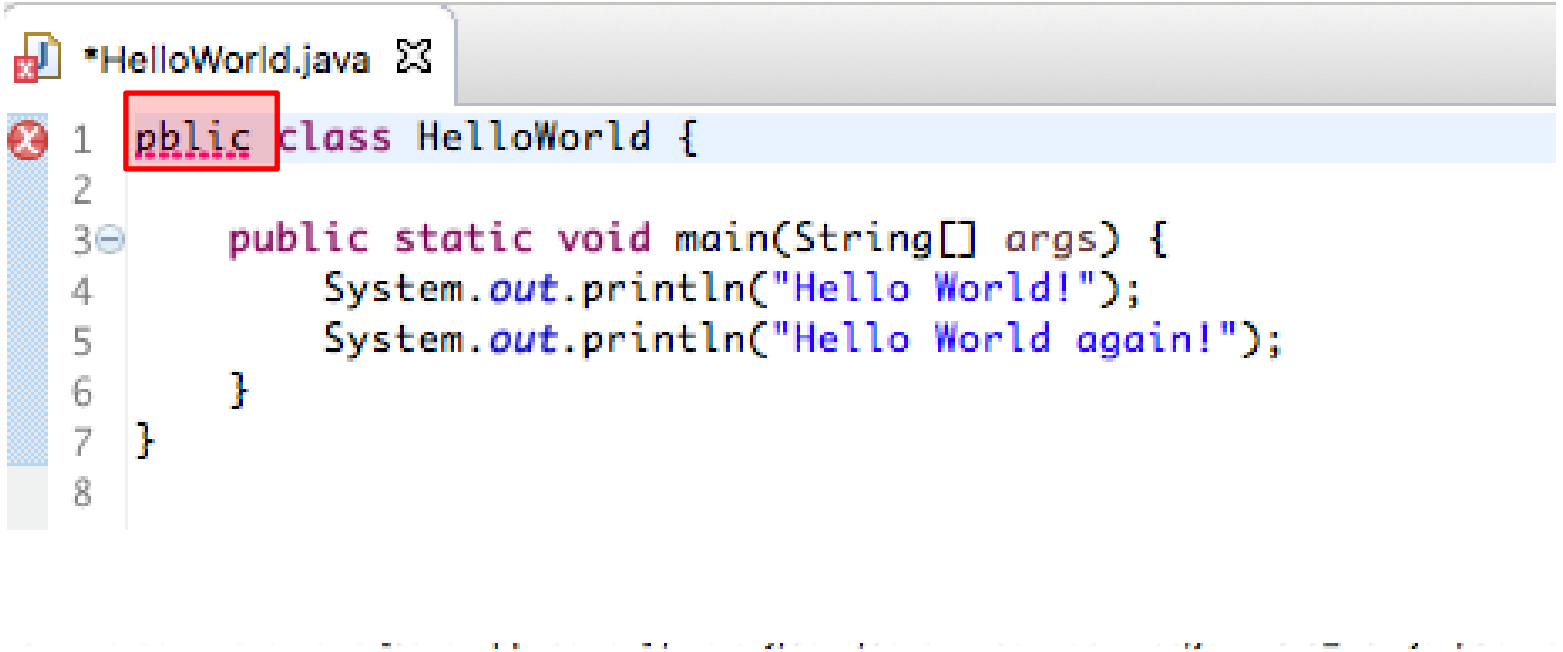
```
1
2  public class HelloWorld {
3
4      public static void main( ) {
5          System.out.println("Hello World!");
6          System.out.println("Hello World again!");
7      }
8
9  }
10
```

```
Error: Main method not found in class HelloWorld, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```



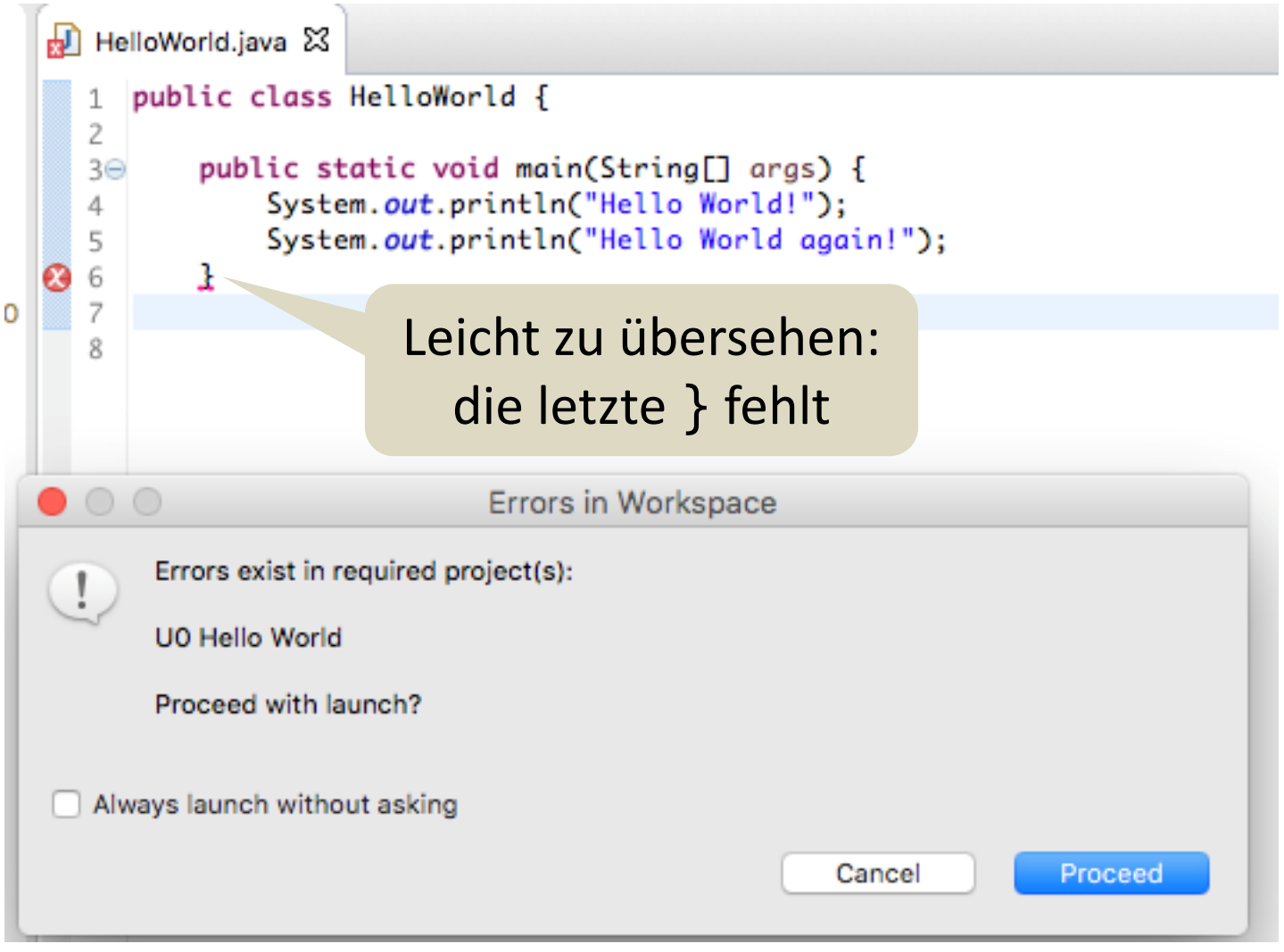
Rückmeldungen des Compilers sind nicht immer direkt verständlich

Weiter geht's …

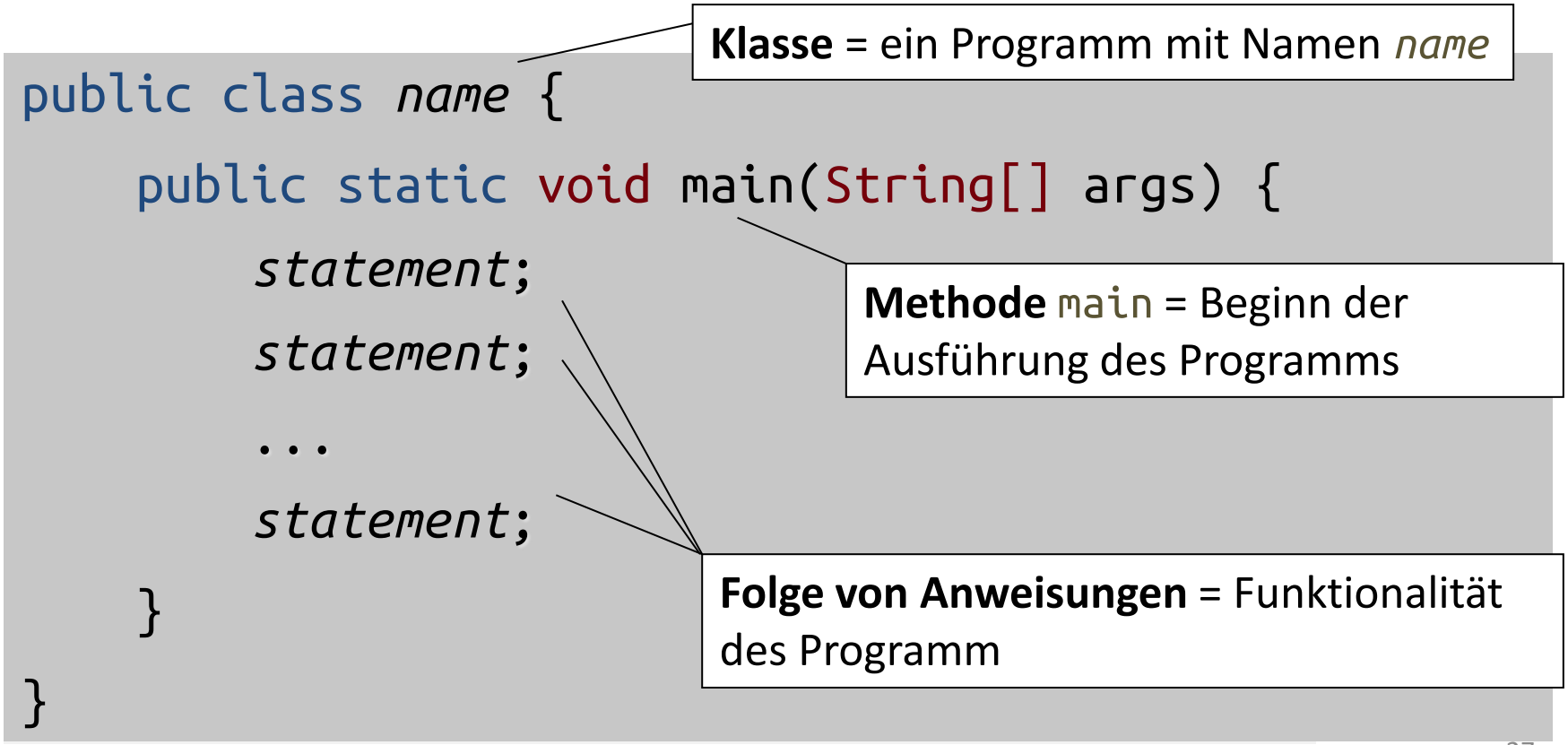



```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:

        at HelloWorld.main(HelloWorld.java:3)
```


HelloWorld.java
1 public class HelloWorld {
2
3 public static void main(String[] args) {
4 System.out.println("Hello World!");
5 System.out.println("Hello World again!");
6 }
7
8
Leicht zu übersehen:
die letzte } fehlt
Errors in Workspace
Errors exist in required project(s):
UO Hello World
Proceed with launch?
Always launch without asking
Cancel
Proceed

Zusammenfassung



Zusammenfassung

- Jedes (ausführbare) Java-Programm besteht aus
  1. Einer Datei mit Namen «*ProgrammName*.java»
  2. die eine Klasse Namens «*ProgrammName*» hat
  3. die eine Methode Namens «`main`» enthält
     - die eine Reihe von Anweisungen enthält
     - die automatisch ausgeführt wird wenn wir auf  klicken

- Später: Programme mit mehreren Dateien und Klassen

Namen: Java-Programme und -Klassen

Jedes *Programm* braucht einen Namen, z.B.

```
public class HelloWorld {…}
```

- Regel: Dateiname gleich Programmname

```
HelloWorld.java
```

- Konventionen: Jedes Einzelwort startet mit einem Grossbuchstaben («upper camel case»)

Namen: Methoden

Jede *Methode* braucht einen Namen, z.B.

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

- Konvention:
  - Fängt mit einem Kleinbuchstaben an
  - Weitere Einzelworte starten mit einem Grossbuchstaben («lower camel case»).
- Weiteres Beispiel: `toUpperCase`

Weitere Regeln und Konventionen folgen später.

Namen: Reservierte Bezeichner

- **Schlüsselwort** («**keyword**»): Ein Bezeichner, der reserviert ist, weil er für die Sprache eine besondere Bedeutung hat. In Java:

abstract	default	if	private	this	false
assert	do	implements	protected	throw	true
boolean	double	import	**public**	throws	null
break	else	instanceof	return	transient	const
byte	enum	int	short	try	goto
case	extends	interface	**static**	var	
catch	final	long	strictfp	**void**	
char	finally	native	super	volatile	
class	float	new	switch	while	
continue	for	package	synchronized		

Kommentare: Syntax

Kommentare («**comments**») sind Notizen im Programmtext, die

- Menschen beim Verstehen des Programmes helfen (sollen)
- Keinen Einfluss auf die Programmausführung haben

- 2 Varianten

1. `// Text bis zum Ende der Zeile (d.h. einzeilig)`

2. ```
   /* Text bis zum schliessenden
      Stern-Schrägstrich-Paar
      (d.h. mehrzeilig)
   */
   ```
   ```

Kommentare: Beispiel

```java
/*
 * Das vermutlich einfachste Java-Programm der Welt.
 * Author: ChatBot3000
 */
public class HelloWorld {

  // Hinweis: Parameter args aktuell nicht genutzt
  public static void main(String[] args) {
    System.out.println("Moin!"); // TODO: "Hoi" besser?
  } // main() endet hier
}
```

Zeichenketten

- **Zeichenketten** («**strings**»)
  - Folge von **Zeichen** («**characters**»)
  - Eingeschlossen in "-Anführungszeichen («**double quotes**»)
- Drei Beispiele: `"Hallo Welt"`, `"hallo welt"`, `"3+2"`
- Einschränkungen
  1. Nur eine Zeile lang:
     ```
     "Dies hier ist kein
                          Java-String, da mehrzeilig"
     ```
  2. Darf Zeichen " nicht enthalten: `"So geht "es" nicht"`

Zeichenketten: Sonderzeichen

Es gibt *Ersatzdarstellungen* («*escape sequences*») mit denen ein Sonderzeichen ausgedrückt werden kann

- Fangen alle mit `\` (Rückwärtsschrägstrich, «backslash») an, z.B.

  `\n` Neue Zeile («new line character»)

  `\"` Anführungszeichen, «double quotes», «quotation mark»

  `\\` Rückwärtsschrägstrich, «backslash»

  `\t` Tabulator, «tab»

- Vorangestelltes `\` heisst *Maskierungszeichen* «*escape character*»

Sonderzeichen: Beispiele

- Code:
```
System.out.println(
    "\\Hallo\nWie\tgeht's \"uns\"?\\\\");
```

- Ausgabe:
```
\Hallo
Wie    geht's "uns"?\\
```

Sonderzeichen: Fragen

1. Was wird jeweils ausgegeben?

```
System.out.println("a\tb\\\tc");
System.out.println("'");
System.out.println("\"\"\"");
System.out.println("C:\neuer-S\tiefel");
```

2. Welche Zeichenkette führt zu folgender Ausgabe?

```
/ \ /// \\\
```

Bemerkung: `println` vs. `print`

- `println(str)`: Gibt effektiv `str` gefolgt von `"\n"` aus
  - D.h. nach der Ausgabe von `str` erfolgt ein Zeilenumbruch
- `print(str)`: Gibt nur `str` aus
  - Erlaubt es, eine zusammenhängende Ausgabe (eine Zeile) über mehre `print`s zu verteilen
  - Künstliches Beispiel:

Java

```
System.out.print("Oberfläche = ");
System.out.print("700");
System.out.println(" cm2");
```

Ausgabe

```
Oberfläche = 700 cm2
```

  - Einsatz später sinnvoller, wenn wir Ausdrücke (und Schleifen) kennen

Java-Programme Erstellen und Ausführen

Zwei Möglichkeiten:

A) Ganzes Programm

- Für jede Datei …
  - Erstellen
  - Ausführen
  - Modifizieren

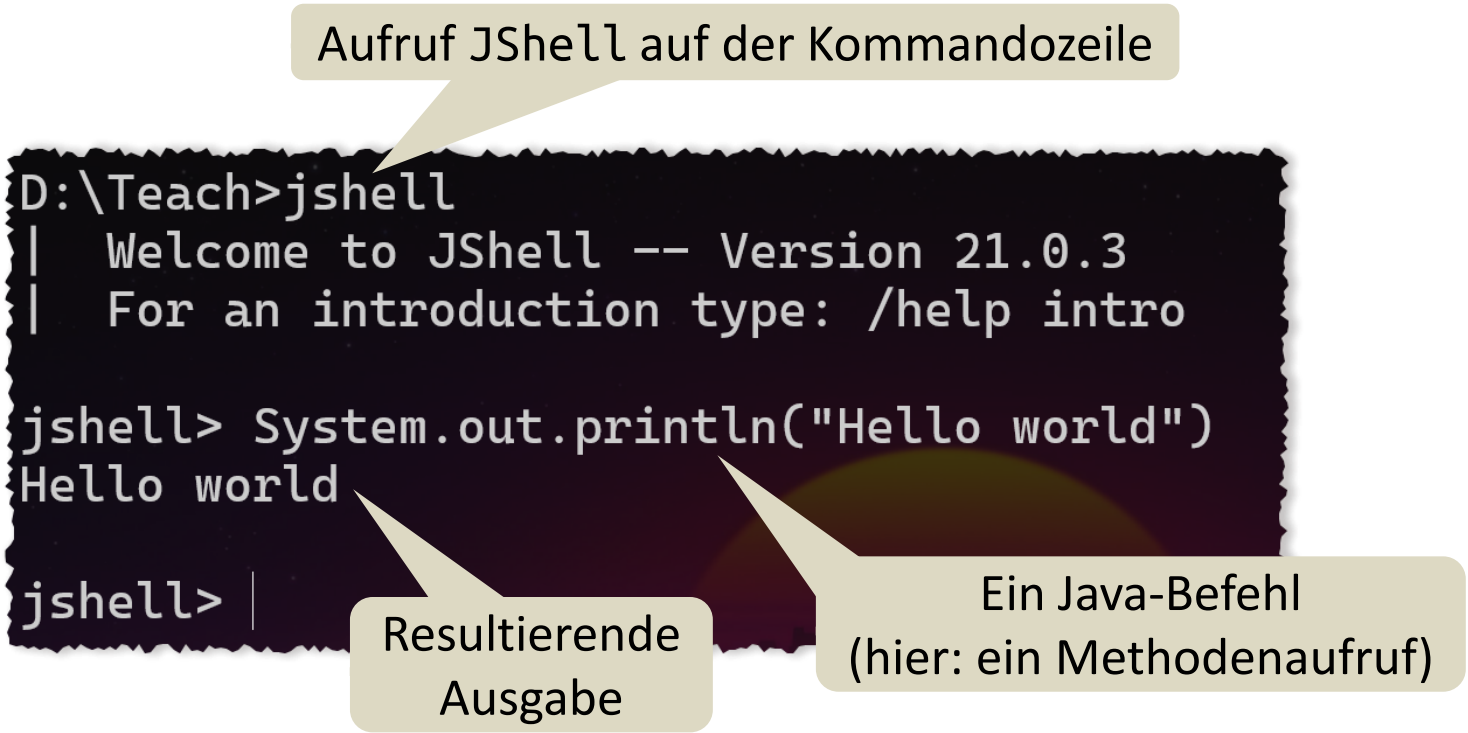
Bisher

Jetzt kurz

B) Einzelne Anweisungen

- Für jede Anweisung …
  - Lesen (Read)
  - Evaluieren (Ausführen)
  - Ausgeben (Print)
- REPL
  - Read-Evaluate-Print Loop

JShell REPL: Start



JShell REPL: Methoden

Eingabe einer Methode
- Methode jetzt global bekannt
- Kein `public static` notwendig
- Praktisch für kleine Programme

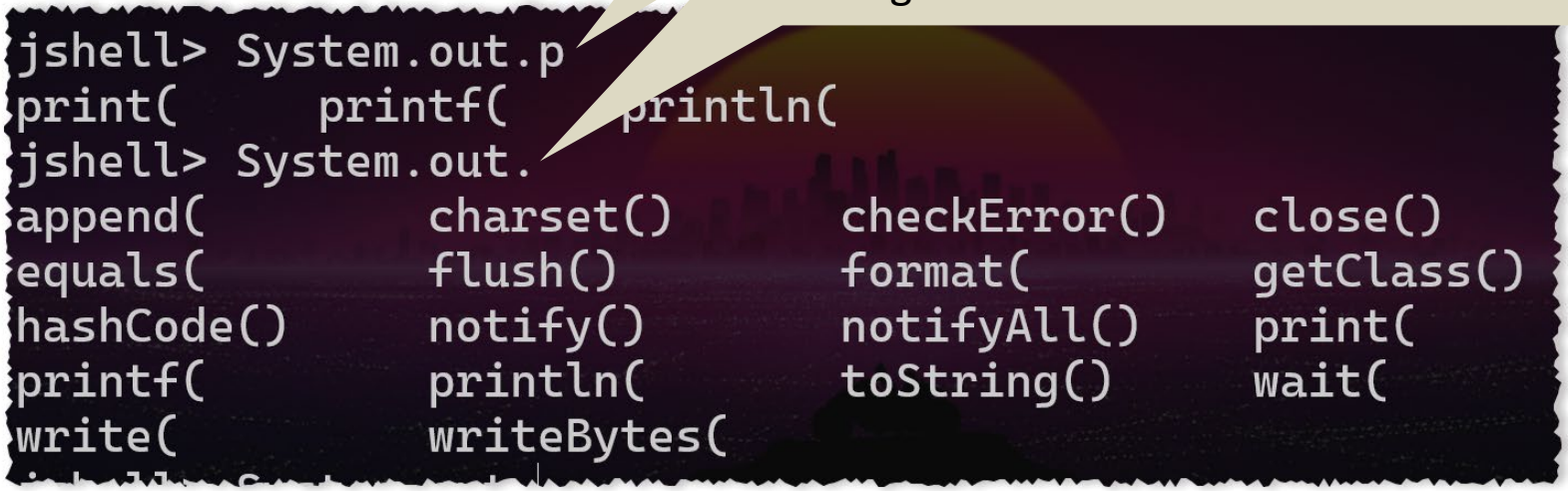
```
jshell> void greet() {
   ...>     System.out.println("Hello");
   ...>     System.out.println("Hello again");
   ...> }
|  created method greet()

jshell> greet()
Hello
Hello again

jshell>
```

Methodenaufruf und erzeugte Ausgabe

JShell REPL: Autovervollständigung

Tab(ulator)-Taste

- Listet existierende Befehlsoptionen auf
- Vervollständigt, falls eindeutig
- Zeigt Hilfe zu Standardmethoden an



Java-Programme Erstellen und Ausführen

- Zwei Möglichkeiten
  - Für jede Datei → skaliert, für grössere Programme
  - Für einzelne Anweisungen → flexible, für schnelle Experimente
- Regeln für Java-Programme bleiben dieselben
  - Scheint manchmal nicht der Fall zu sein – liegt daran, dass die REPL den Code im Hintergrund in ein «vollständiges» Java-Programm einbettet
- Fokus dieser Vorlesung: Arbeiten mit Dateien, da essenziell für komplexere Programme

2. Erste Java-Programme

2.1 Einführung: Vom Quelltext zur Ausgabe

2.2 Methoden I: Programmstrukturierung

2.3 Einfache Berechnungen

2.4 Input und Random

2.5 Logische Aussagen über Programm(segment)e

Methoden: Was?

- Methode: Benannte Sequenz von Anweisungen

```
1  public class HelloWorld {
2
3      public static void main(String[] args) {
4          System.out.println("Hello World!");
5          System.out.println("Hello World again!");
6      }
7  }
```

- Beispiel: `main` (im Program `HelloWorld`)
  - Methode enthält Anweisungen
  - Diese werden beim Methodenaufruf ausgeführt
    (`main` wird beim Programmstart automatisch aufgerufen)

Methoden: Warum?

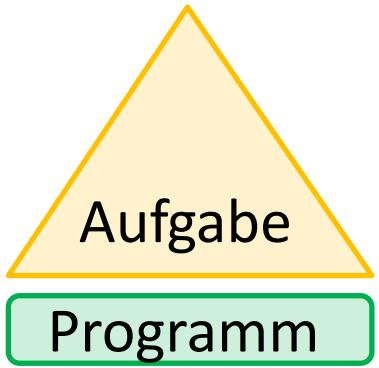
- Methoden *strukturieren* die Anweisungen
  - Anstatt alle Anweisungen in einer Methode (`main`) unterzubringen
- Methoden erlauben es, *Wiederholungen* zu vermeiden
  - Nur einmal im Programm(text)
  - Aber mehrfach aufrufbar und somit ausführbar
- Jede Methode stellt eine neue, spezialisierte Anweisung dar
  - Idee hinter Programmierbibliotheken

Programmentwicklung: Grobes Rezept

1. Gesamtaufgabe in Teilaufgaben *zerlegen*
   - Erfordert Problemverständnis und Problemlösestrategien
2. Teilaufgaben in separaten Methoden *lösen*
   - Teilaufgaben sind kleiner, Lösungen daher einfacher zu implementieren
3. Teillösungen zu Gesamtlösung *zusammensetzen*
   - Siehe Punkt 1

Einstiegsbeispiel

- Aufgabe: Programm, das Text ausgibt



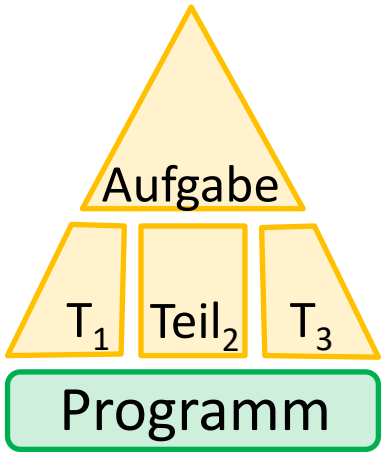
- Gesucht:

Ausgabe

```
--------

Warnung: sichern Sie die Daten

--------

Lange Erklärung

--------

Warnung: sichern Sie die Daten

--------
```

Zerlegen in Teilaufgaben T_1, T_2, …, T_n

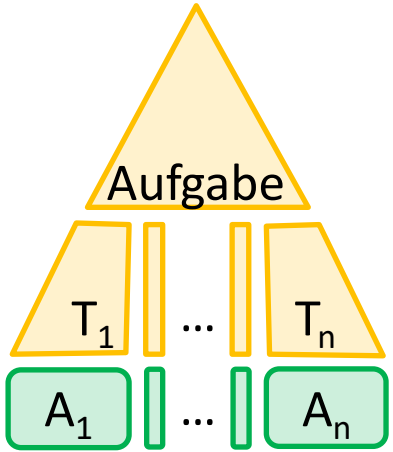
- Wie zerlegen? Wie weit/kleinteilig zerlegen?
  - Keine fixen Regeln, die immer zum Erfolg führen
  - Keine objektiv beste Zerlegung
  - Stattdessen: Heuristiken
    - Jede Methode sollte *eine* klare Aufgabe haben
    - Ausbildung: Hinweise («hints»), Vorgaben
    - Beruf: Erfahrung, Diskussionen



- Zerlegung
  - Bisher nur relevant für Menschen, aber dem Computer egal
  - Neu auch relevant für KI-gestützte Entwicklung (fokussierte Prompts)

Einstiegsbeispiel: 1. Zerlegung

- Aufgabe: Programm, das Text ausgibt



Ausgabe

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

- 1. Zerlegung
  - Teilaufgabe T_i: i-te Zeile in einer `println()` Anweisung ausgeben
  - Anweisung A_i für Teilaufgabe T_i
  - A_i hintereinander ausführen → Gesamtausgabe erhalten

Code 1. Zerlegung

```
public class PrintExampleV1 {
  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 Erklärung");
    System.out.println("\n--------\n");
    System.out.println("Warnung: sichern Sie die Daten\n");
    System.out.println("\n--------\n");
  } // Ende main
}
```

Code 1. Zerlegung

Java

```java
public class PrintExampleV1 {
  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 Erklärung");
    System.out.println("\n--------\n");
    System.out.println("Warnung: sichern Sie die Daten\n");
    System.out.println("\n--------\n");
  } // Ende main
}
```

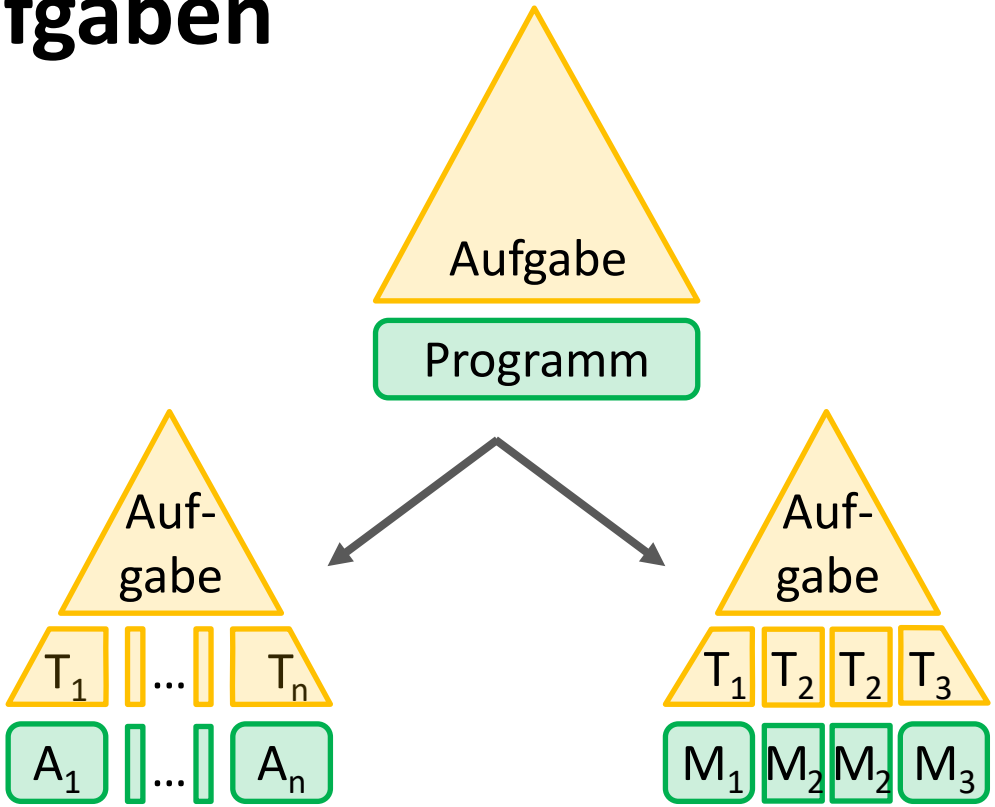
Ausgabe

```
--------

Warnung: sichern Sie die Daten

--------

Lange Erklärung

--------

Warnung: sichern Sie die Daten

--------
```

- Beobachtung: Redundanz (Wiederholungen im Programm)
- Gesucht: Zerlegung, die die *Struktur der Aufgabe* widerspiegelt

Zerlegen in Teilaufgaben

- Sich *wiederholende Teilaufgaben* erkennen, so dass die Anweisungen für (viele) T_i *wiederverwendbar* sind
- Anweisungen für T_i in Methode M_i
- Methoden (statt einzelner Anweisungen) hintereinander ausführen



v1: zu kleinteilig

v2: besser?

Einstiegsbeispiel

- Aufgabe: Programm, das Text ausgibt
- 2. Zerlegung
  - Teilaufgabe T_1 : drucke Warnung
  - Teilaufgabe T_2 : drucke Erklärung
- Resultierendes Programm
  - M_1; M_2; M_1;

```
--------

Warnung: sichern Sie die Daten

--------
```

```
Lange Erklärung
```

```
--------

Warnung: sichern Sie die Daten

--------
```

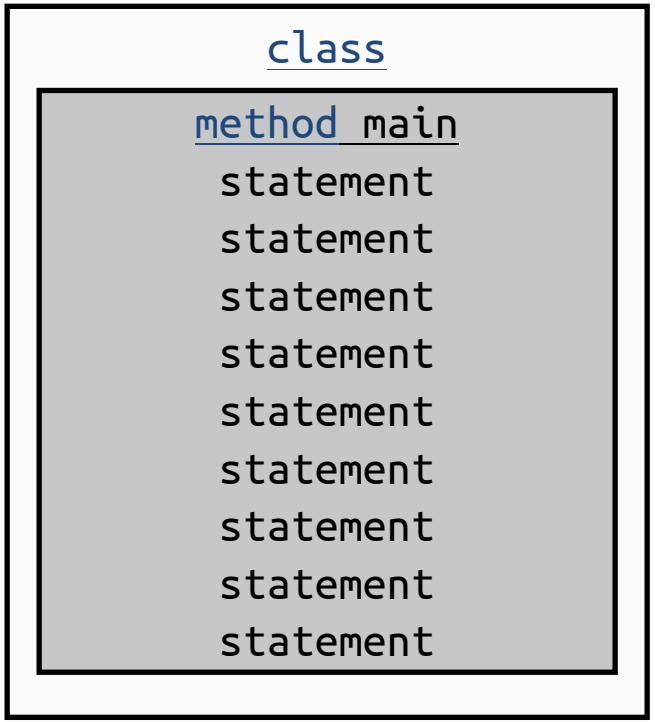
Code 2. Zerlegung

```
public class PrintExampleV2 {
  public static void main(String[] args) { // Gesamtlösung
    printWarning();
    printExplanation();
    printWarning();
  }

  public static void printWarning() {       // T1 bzw. M1
    System.out.println("\n--------\n");
    System.out.println("Warnung: sichern Sie die Daten\n");
    System.out.println("\n--------\n");
  }

  public static void printExplanation() {  // T2 bzw. M2
    System.out.println("Lange Erklärung");
  }
}
```

Wie entstehen Methoden?

Möglichkeit A: Im Nachhinein

1. Entwickeln des Algorithmus
   - Problemverständnis, Lösungsidee
   - Aufteilung in Teilprobleme
2. Implementation des Algorithmus
   - In einem Stück, z.B. alles in `main()`

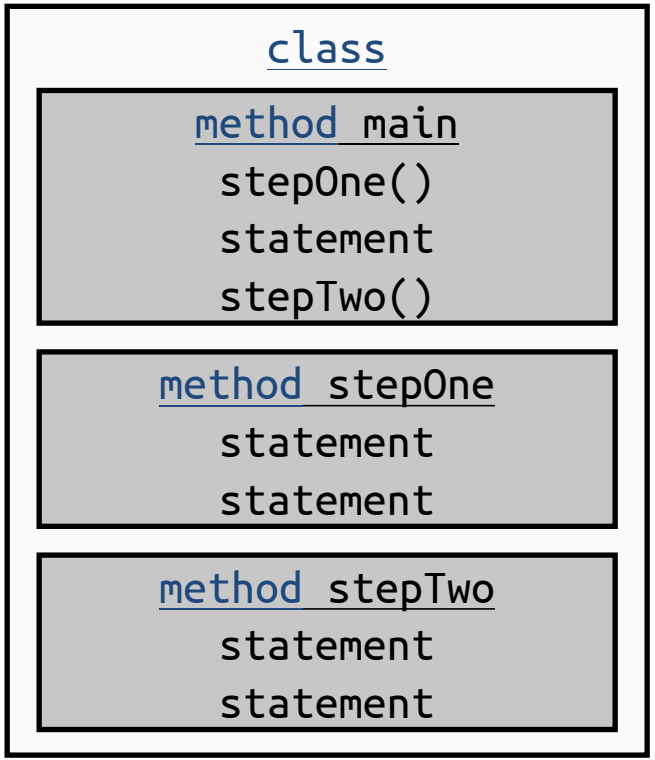


Wie entstehen Methoden?

Vorteile?
Nachteile?

Möglichkeit A: Im Nachhinein

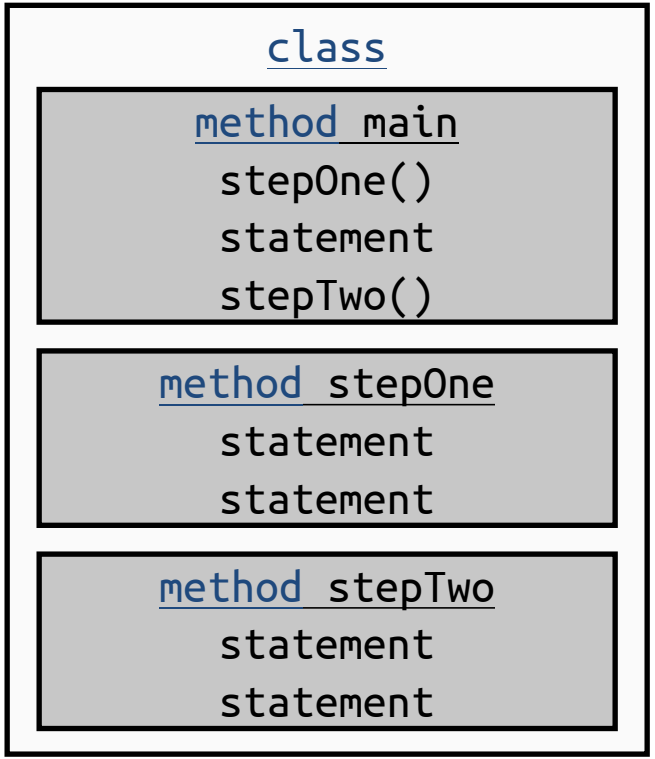
1. Entwickeln des Algorithmus
   - Problemverständnis, Lösungsidee
   - Aufteilung in Teilprobleme
2. Implementation des Algorithmus
   - In einem Stück, z.B. alles in `main()`
3. Code nachträglich aufteilen
   - Wiederverwendbare Funktionalität identifizieren, Redundanz vermeiden



Wie entstehen Methoden?

Möglichkeit B: Teilproblemen folgend

1. Entwickeln des Algorithmus
   - Problemverständnis, Lösungsidee
   - Aufteilung in Teilprobleme
     - Wiederkehrende Teilprobleme direkt identifizieren
2. Implementation des Algorithmus
   - In einzelnen, von Teilproblemen abgeleiteten Methoden



Wie entstehen Methoden?

Möglichkeit B (Teilproblemen folgend) ist grundsätzlich vorzuziehen

- Strukturiert den Entwicklungsprozess (kleine, überschaubare Einheiten statt einem langen Codeblock)
- Vermeidet Redundanz (statt diese im Nachhinein zu entfernen)
- Ermöglicht Teamarbeit (für uns nicht wichtig, aber in der Praxis essenziell)

Möglichkeit A kann aber

- Für den Anfang einfacher sein
- «Lokal» für die Entwicklung einzelner Methoden genutzt werden

Methodenaufrufe

```
public class PrintExampleV2 {
  public static void main(String[] args) {
    printWarning();
    printExplanation();
    printWarning();
  }

  public static void printWarning() {
    ...
  }

  public static void printExplanation() {
    ...
  }
}
```

- **Beobachtung:** Methodenaufruf im Format `methName();`
  - Nur für *statische* Methoden (Schlüsselwort `static`)
  - Nicht-statische Methoden schauen wir uns später an (Aufrufformat `obj.methName();`)
- Aber wie läuft ein Methodenaufruf eigentlich ab …?

Zunächst: Linearer Kontrollfluss

Java

```
public class ControlFlowExample1 {
  public static void main(String[] args) {
    System.out.println("Zeile 1");
    System.out.println("Zeile 2");
    System.out.println("Zeile 3");
  }
}
```

Ausgabe

```
Zeile 1
Zeile 2
Zeile 3
```

- Sequenzen von Anweisungen werden, wie im Programmtext stehend, *von oben nach unten** ausgeführt
- Dies nennt sich **linearer Kontrollfluss** («**linear control flow**»)

* Verzweigungen und Schleifen lernen wir später kennen

Kontrollfluss mit Methodenaufrufen

```
main() {
  statement1;
  name();
  statement2;
}

name() {
  statement3;
  statement4;
}
```

- Die Anweisung `name()` stellt einen **Methodenaufruf** («**call**», «**invocation**») dar
- Wird sie ausgeführt dann
  1. Spring die Ausführung (der Kontrollfluss) an den Anfang der entsprechenden Methode
  2. Werden die Anweisungen innerhalb der Methode von oben nach unten (linear) ausgeführt
  3. Springt die Ausführung zurück hinter den (nun beendeten) Methodenaufruf

Kontrollfluss mit Methodenaufrufen

```
main() {
  statement1;
  name();
  statement2;
}

name() {
  statement3;
  statement4;
}
```

Effektiv werden die Anweisungen hier daher wie folgt ausgeführt:

1. `statement1;`
2. `statement3;`
3. `statement4;`
4. `statement2;`

Kontrollfluss mit Methodenaufrufen

Hinweis: Die Reihenfolge der Methoden im Programmtext ist (in Java) irrelevant

```
main() {
  statement1;
  name();
  statement2;
}

name() {
  statement3;
  statement4;
}
```

=

```
name() {
  statement3;
  statement4;
}

main() {
  statement1;
  name();
  statement2;
}
```

Grösseres Beispiel: Schrittweise Ausführung

Java

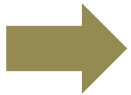
```
public class ControlFlowExample2 {
 public static void main(String[] args) {
    System.out.println("Anfang");
    message1();
    System.out.println("Mitte");
    message2();
    System.out.println("Ende");
  }
  public static void message1() {
    System.out.println("1. Nachricht");
  }
  public static void message2() {
    message1();
    System.out.println("2. Nachricht");
  }
}
```

Ausgabe

```
Anfang
1. Nachricht
Mitte
1. Nachricht
2. Nachricht
Ende
```

Grösseres Beispiel: Mögliche Erklärungshilfe

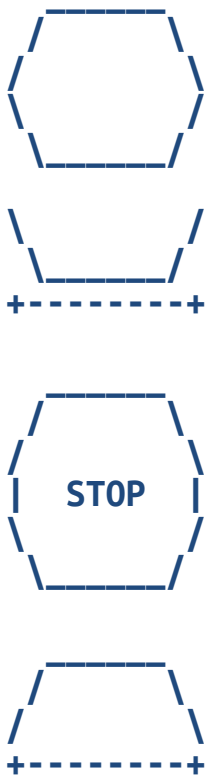
```
main(…) {
  …println("Anfang");
  msg1();
  …println("Mitte");
  msg2();
  …println("Ende");
}

msg1() {
  …println("1. N't");
}

msg2() {
  msg1();
  …println("2. N't");
}
```



```
main(…) {
  …println("Anfang");
  // msg1()
    …println("1. N't");
  …println("Mitte");
  // msg2()
    // msg1()
      …println("1. N't");
    …println("2. N't");
  …println("Ende");
}
```

Die Darstellung rechts visualisiert indirekt den *Aufrufstapel* — lernen wir später kennen

Eine neue Aufgabe …



Schreiben Sie ein Programm,
dass diese Figuren ausgibt

Entwicklungsschritte

Version 1: Ohne Ausnutzen der Struktur

1. Programm mit leerer `main`-Methode erstellen
2. Erwünschte Ausgabe in `main` kopieren; für jede Zeile eine entsprechende `println`-Anweisung
3. Programm ausführen; erwartete und erhaltene Ausgabe vergleichen

Programm 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(" /      \\");
```

⋮

⋮

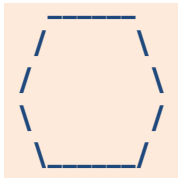
```
    System.out.println("/        \\");
    System.out.println("|  STOP  |");
    System.out.println("\\        /");
    System.out.println(" \\______/");
    System.out.println();
    System.out.println("  ______");
    System.out.println(" /      \\");
    System.out.println("/        \\");
   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

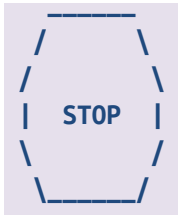
Entwicklungsschritte

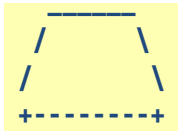
Version 2: Mit Ausnutzen der Struktur, mit Redundanz

1. Vorhandene Strukturen identifizieren
2. `main`-Methode in separate Methoden aufteilen, basierend auf obiger Strukturierung
3. Programm ausführen; erwartete und erhaltene Ausgabe vergleichen

Entwicklungsschritte







Strukturen in dieser Figur (von oben nach unten)

- «Sechseck» → Methode `hexagon()`
- «Wanne» → `tub()`
- «Stopschild» → `stopSign()`
- «Hut» → `hat()`

Programm Version 2

```
public class Figures2 {
  public static void main(String[] args) {
    hexagon();
    tub();
    stopSign();
    hat();
  }
```

```
  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 tub() {
    System.out.println("\\        /");
    System.out.println(" \\______/");
    System.out.println("+--------+");
    System.out.println();
  }
```

Programm 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();
  }
```

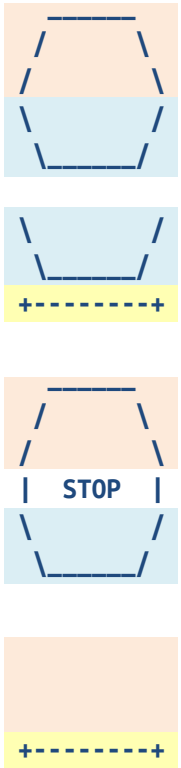
```
  public static void hat() {
    System.out.println("  ______");
    System.out.println(" /      \\");
    System.out.println("/        \\");
    System.out.println("+--------+");
  }
} // class "Figures2"
```

Entwicklungsschritte

Version 3: Mit Ausnutzen der Struktur, ohne Redundanz

1. Vorhandene (Teil-) Strukturen und Redundanzen identifizieren
2. Methoden so erstellen, dass Redundanzen (soweit möglich/sinnvoll) vermieden werden
3. Code kommentieren
4. Programm ausführen

Redundanz in der Ausgabe



Mehrfach auftretende Teilstrukturen

- Obere Hälfte Hexagon → Methode `hexagonTop()`
- Untere Hälfte Hexagon → `hexagonBottom()`
- Trennlinie → `line()`

Programm Version 3

```
// Author: Maija Meikäläinen, 24-135-678
// Prints several figures, split up into
// methods that capture common
// (sub)structures.
public class Figures3 {
  public static void main(String[] args) {
    hexagon();
    System.out.println();
    tub();
    System.out.println();
    stopSign();
    System.out.println();
    hat();
  }
```

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

  // Draws a hexagon's bottom half.
  public static void hexagonBottom() {
    System.out.println("\\        /");
    System.out.println(" \\______/");
  }
```

Programm Version 3 (Fortsetzung)

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

  // Draws a tub-shaped figure.
  public static void tub() {
    hexagonBottom();
    line();
  }

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

```
  // Draws a stopSign.
  public static void stopSign() {
    hexagonTop();
    System.out.println("|  STOP  |");
    hexagonBottom();
  }

  // Draws a hat-shaped figure.
  public static void hat() {
    hexagonTop();
    line();
  }
} // class "Figures3"
```

Übungsaufgabe

Schreiben Sie ein Programm, dass diese Rakete ausgibt:

2. Erste Java-Programme

2.3 Einfache Berechnungen

2.3.1 Einführung: Typen, Variablen, Werte

2.3.2 Ausdrücke über Basistypen

2.3.3 Variablendeklaration und -initialisierung

(Sinnvolles) Arbeiten mit Daten

- Programme operieren auf Daten, z.B. eines Boots
  - Höhe, Länge, Tiefe; Kaufpreis; Name; Standort; …
- Nicht alle Operationen (Berechnungen) sind sinnvoll
  - ✓ Länge mal Breite mal Höhe
  - ✓ 75% des Kaufpreises
  - ✗ Finde kürzesten Weg zwischen Standort und vorgestern
  - ✗ Starte eine Hausbootzucht
- Ziel von Programmiersprachen:
  Sinnvolle Berechnungen ermöglichen, unsinnige verhindern

Bootgrafik erzeugt mit *Image Creator in Bing*

Typen: Was?

- Ein **Typ** («**type**») beschreibt eine *Kategorie von Werten/Daten*
  - Ähnlich zu mathematischen Mengen
  - Z.B. $\mathbb{Z}$ (ganze Zahlen) vs. $\mathbb{Q}$ (rationale Zahlen)

- Typen beschreiben *Eigenschaften von Daten*
  - Bestimmt die Operationen, die für diese Werte möglich sind
  - Z.B. Typen «Boat» vs. «Katze» — welche Eigenschaften/Operationen teilen sie, welche unterscheiden sie?

Typen: Warum?

- Viele Programmiersprachen erfordern explizite Typangaben
  - Sicherheit (Probleme verhindern/erkennen) — für EProg wichtig
  - Dokumentation — für EProg wichtig
  - Effizienz — für EProg nicht/weniger wichtig
- Sicherheit: Typen verhindern Fehler
  - Kann AHV-Nummer nicht zum Gehalt addieren
- Dokumentation: Typen spezifizieren die Teile einer Berechnung
  - Was wird erwartet (Preis? Standort?), was wird berechnet?

Primitive Typen in Java

Acht **primitive Typen** («**primitive types**»): für Zahlen, Buchstaben und Wahrheitswerte. Beispiele:

Name	Beschreibung	Beispiele
`int`	ganze Zahlen	`-2147483648`, `-3`, `0`, `42`, `2147483647`
`long`	grosse ganze Zahlen	`42`, `-3`, `0`, `92233720368547758O7`
`double`	reelle Zahlen	`3.1`, `-0.25`, `9.4e3`
`char`	(einzelne) Buchstaben	`'a'`, `'X'`, `'?'`, `'\n'`
`boolean`	Wahrheitswerte	`true`, `false`

Wer es genau wissen möchte: Java Language Specs §4.2

Typen: Von Bitfolgen zu Daten

- Die Programmiersprache legt fest, wie ein Typ implementiert ist
  - Implementiert = Darstellung der Werte und Definition der Operationen

- *Interne Darstellung* aller Werte durch Nullen und Einsen
  - 97 → 01100001
  - "ab" → 01100001 01100010

ASCII-Tabelle

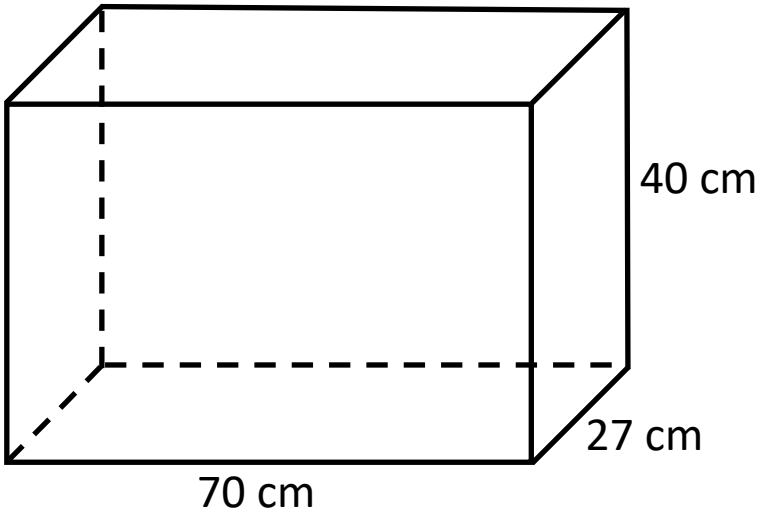
"a"	97	01100001
"b"	98	01100010

- Was *bedeutet* 01100001? → Typen wichtig
  - `"a".toUpperCase()` sinnvoll, aber `97.toUpperCase()` nicht

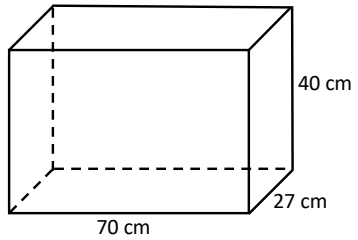
Anwendung: Oberfläche eines Quaders

- Gegeben: Dimensionen eines Quaders
- Aufgabe: Java-Programm zur Berechnung der Oberfläche
- Oberfläche in cm^2:

$$2 \cdot (70 \cdot 27 + 27 \cdot 40 + 40 \cdot 70)$$



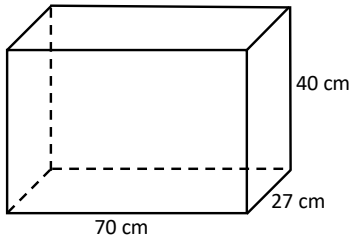
Entsprechendes Programm




```
public class Quader {
  // Oberflächenberechnung für einen bestimmten Quader
  public static void main(String[] args) {
    System.out.print("Die Oberfläche beträgt ");
    System.out.print(2 * (70 * 27 + 27 * 40 + 40 * 70));
    System.out.println(" cm2");
  }
}
```

Ausgabe: `Die Oberfläche beträgt 11540 cm2`

Fehler sind schnell gemacht




```
public class Quader {
  // Oberflächenberechnung für einen bestimmten Quader
  public static void main(String[] args) {
    System.out.print("Die Oberfläche beträgt ");
    System.out.print(2 * (70 * 27 + 27 * 40 + 50 * 70));
    System.out.println(" cm2");
  }
}
```

Unaufmerksamkeit …

Ausgabe: `Die Oberfläche beträgt 12940 cm2`

Variablen

- **Variablen** («**variables**») geben Werte (z.B. `40`) sinnvolle Namen (z.B. `height`)
  - Konzept aus der Mathematik bekannt
  - In Berechnungen werden Variablen durch ihre Werte «ersetzt»
    - Z.B. `height * height` steht (hier) für `40 * 40` und ergibt daher `1600`
  - Aber Programmvariablen können sich ändern (später mehr)

- Erforderlich für Variablen in Java: Name *und* Type
  - Hier wären ganzzahlige Werte sinnvoll → `int`-getypte Variable

Programm mit Variablen

```
public static void main(String[] args) {
  int length;
  int height;
  int depth;
  length = 70;
  height = 40;
  depth = 27;
  System.out.print("Die Oberfläche beträgt ");
  System.out.print(2 * (length*depth + depth*height + height*length));
  System.out.println(" cm2");
}
```

// Deklarationen

// Initialisierung durch Wertzuweisungen

Ausgabe: `Die Oberfläche beträgt 11540 cm2`

Variablendeklaration

Eine Variable muss **deklariert** («**declared**») werden
(d.h. bekannt gemacht werden, angekündigt werden)

- **Syntax:** `type name;`
  - Für einen gültigen Java-Typ *type* (wir lernen viele Typen kennen)
  - Variable *name* kann nun jeden Wert vom Typ *type* speichern
- **Beispiel:** `int height;`
- Variablenname (fast) frei wählbar
  - Regel: Kein Java-Schlüsselwort, muss mit einem Buchstaben anfangen
  - Konvention: Camel-Case, mit kleinem Buchstaben beginnend

Variablendeklaration und -initialisierung

- Eine Variable kann deklariert und direkt **initialisiert** («**initialised**»> (d.h. einen Wert bekommen)
  - **Syntax:** *`type name = expression;`*
    - Für einen Typ *`type`* und einen dazu passenden Ausdruck *`expression`* (mehr zu Ausdrücken später, zunächst Ausdruck ≈ Wert)
  - **Beispiel:** `int height = 40;`
- Das Zeichen = bedeutet **Zuweisung** («**assignment**»)
  - Effekt: Variable hat ab jetzt diesen Wert
  - Eine Zuweisung ist *keine* mathematische Gleichheit, auch wenn der Operator leider so aussieht. Später mehr dazu.

Programm mit Variablen

```
public static void main(String[] args) {
  int length = 70;
  int height = 40;
  int depth = 27;
  System.out.print("Die Oberfläche beträgt ");
  System.out.print(2 * (length*depth + depth*height + height*length));
  System.out.println(" cm2");
}
```

// Deklaration und direkte Initialisierung

Ausgabe: `Die Oberfläche beträgt 11540 cm2`

Standard-Initialisierung für Basistypen?

```
public static void main(String[] args) {
  int length;
  System.out.print(length);
}
```

```
Compilation problem:
  Local variable x
  not initialized
```

- Lokale Variablen von primitivem Typ (etwas anderes kennen wir bisher nicht) haben keinen Standardwert, sondern *müssen explizit initialisiert* werden
- Wird versucht, den Wert einer (solchen) uninitialisierten Variable zu nutzen, gibt es einen Compilerfehler

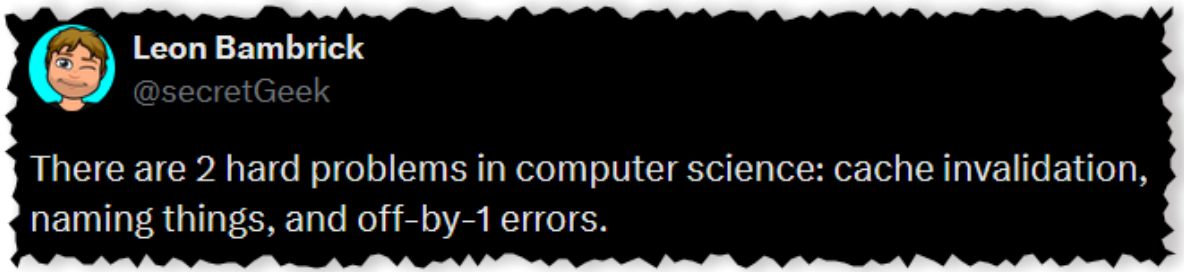
Die JShell verhält sich hier anders und initialisiert solche Variablen mit einem Standardwert, z.B. `0` für `int`

Variablennamen als Dokumentation

- Variablennamen sind Computern egal — aber Menschen nicht
- Gute Namen helfen beim Verstehen eines Programs
  - Selbstbeschreibend
    (z.B. `length`, `lunchPreference`, `twintAccount`)
  - Halbwegs kurz
    (`lengthInMM` vs. `lengthInTheMetricSystemUnitMillimeter`)
  - Unterscheidbar (im Kontext)
    (`name1`, `name2` vs. `firstName`, `middleName`)

Variablennamen als Dokumentation

- Gute Namen zu finden ist nicht (immer) einfach
  - Insbesondere über Berechnungen mit Zwischenwerten hinweg (`salary`, `salaryAfterRaise`, `salaryAfterRaiseWithChristmasBonus`, …)
- Kurze Name (z.B. `i`, `x`, `n`) OK für kurzlebige Variablen
- Namen sind wichtig! Aber seien Sie pragmatisch, nicht dogmatisch



Typen als Dokumentation

- Java unterstützt *Typinferenz* («*type inference*»), d.h. Typinformationen können *manchmal* weggelassen werden. Bsp.:

```
int score = 99;
String name = "Ghüs";
```

```
var score = 99;
var name = "Ghüs";
```

- **Empfehlung:** Typen explizit angeben → Dokumentation

```
var result = findHomeCityForLegi("24-135-987");
```

Welche Werte kann `result` nun haben — Wahrheitswerte ((un)bekannt), Zeichenketten (Name), Zahlen (Postleitzahl)?

Wer definiert Typen?

Gültige Java-Typen sind

- Immer verfügbare Basistypen (eingebaute Typen)
  - Primitive Typen, z.B. `int`
  - Typen aus Javas Standardbibliothek, z.B. `String`
- Benutzerdefinierte Typen (später mehr dazu)
  - Von uns
  - Aus Bibliotheken Dritter

2. Erste Java-Programme

2.3 Einfache Berechnungen

2.3.1 Einführung: Typen, Variablen, Werte

2.3.2 Ausdrücke über Basistypen

2.3.3 Variablendeklaration und -initialisierung

Basistypen

```
…
int depth = 27;

…println(2 * (length*depth + depth*height + height*length));
```

- `27` und `2 * (length*depth + …)` sind **Ausdrücke** («**expressions**»)
- Was sind die Regeln für Ausdrücke?
  - Zunächst für Zahlen (`int`, `double`)
  - EBNF-Regel *expr*

EBNF-Beschreibung Ausdruck (*expr*)

```
<integer>  ⟵ [<sign>] {<digit>}
<number>   ⟵ <integer> | <integer> . {<digit>} | <sign> . {digit}
<op>       ⟵ + | - | * | / | %
<atom>     ⟵ <number> | <identifier>
<term>     ⟵ ( <expr> ) | <atom>
<expr>     ⟵ <term> {<op> <term>}
```

1. (Unvollständige) Beschreibung für Java-Ausdrücke
   - Keine Überraschungen ... wie in der Mathematik/Schule
   - *number* beschreibt Zahl-**Literale** («number **literals**»)
2. Beschreibt nur die Syntax — keine Typen, keine Semantik

Arithmetische Operatoren

- **Operatoren** («**operators**»): Bilden Ausdrücke aus Teilausdrücken

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

- Müssen festlegen, was «*X* ⊗ *Y*» *bedeutet*
  (⊗ Operator; *X*, *Y* `int`-getypte Ausdrücke)
  - Was ist das Ergebnis (Semantik)?
  - Was für einen Typ hat das Ergebnis?

Arithmetische Operatoren: Ergebnisse

Operatoren `+`, `*`, `-`: Wie in der Mathematik — zumindest *fast*

- Zahlen intern als *endliche* Bitfolge repräsentiert: Für `int`/`long` sind es 32/64 Bits. D.h. es gibt eine kleinste und grösste Zahl.
- Verhalten: *MIN* – 1 gleich *MAX*, *MAX* + 1 gleich *MIN*
- Unter-/Überlauf, wie bei einer Uhr
- Wird in *Digitaltechnik* (2. Semester) im Detail besprochen

```
jshell> Integer.MAX_VALUE
$1 ⟹ 2147483647

jshell> Integer.MAX_VALUE + 1
$2 ⟹ -2147483648

jshell> Integer.MIN_VALUE
$3 ⟹ -2147483648
```

Arithmetische Operatoren: Typen

- Für Operatoren `+`, `*`, `-`, `/`, `%`: *TypA* ⊗ *TypA* ergibt *TypA*
- Beispiele
  - `2 * 3` ergibt `6` mit Typ `int` (weil `2` und `3` `int`s sind)
  - `-(7)` ergibt `-7` mit Typ `int` (weil `7` ein `int` ist)
- Naheliegende Fragen
  1. Welchen `int`-Wert liefert, z.B. `5 / 2`? → jetzt
  2. Welchen Typ liefert *TypA* ⊗ *TypB*? → später

Ganzzahlige Division mit /

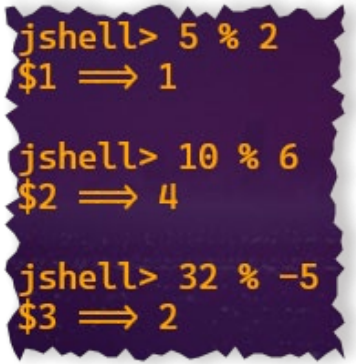
- Division zweier ganzer Zahlen ergibt wieder eine ganze Zahl
- Die mathematisch anfallenden Nachkommastellen werden *abgeschnitten* («*truncated*»)
- Beispiele
  - `5 / 2` ergibt `2` (nicht 2,5)
  - `10 / 6` ergibt `1` (nicht 1,666…)
  - `32 / -5` ergibt `-6` (nicht -6,4)

```
jshell> 5 / 2
$1 ⟹ 2

jshell> 10 / 6
$2 ⟹ 1

jshell> 32 / -5
$3 ⟹ -6

jshell> -32 / -5
$4 ⟹ 6
```

- Division durch 0 führt zum Programmabbruch (Laufzeitfehler)

Rest ganzzahliger Division mit %

- Der Modulo-Operator `%` liefert den Rest ganzzahliger Division
- Beispiele
  - `5 % 2` ergibt `1`
  - `10 % 6` ergibt `4`
  - `32 / -5` ergibt `2`



- Es gilt stehts:
  `(a / b) * b + (a % b)` ergibt `a`
- Division durch 0 führt zum Programmabbruch

Einsatzmöglichkeiten Modulo-Operator

- Bestimme die letzte Ziffer einer ganzen Zahl
  - `230857 % 10` ist `7`
- Bestimmte letzten 4 Ziffern
  - `658236489 % 10000` ist `6489`
- Entscheide, ob eine Zahl gerade ist
  - `7 % 2` ergibt `1`
  - `42 % 2` ergibt `0`

Fragen: Was ist das Ergebnis von …

1. `10 / 4`
2. `-23 / 11`
3. `8 % 20`
4. `2 % 2`
5. `11 % -4`
6. `-11 % -4`
7. `-11 % 4`

Ausdrücke mit mehreren Operatoren

- Zunächst: Mehrfach der gleiche Operator ⊗

- Was bedeutet «*X* ⊗ *Y* ⊗ *Z*»?
  - `7 + 5 + 3` → `(7 + 5) + 3`
  - `64 / 8 / 2` → `(64 / 8) / 2` (nicht `64 / (8 / 2)`)

- Gegeben «*X* ⊗ *Y* ⊗ *Z*», zu welchem Operator gehört *Y*: dem linken, dem rechten?

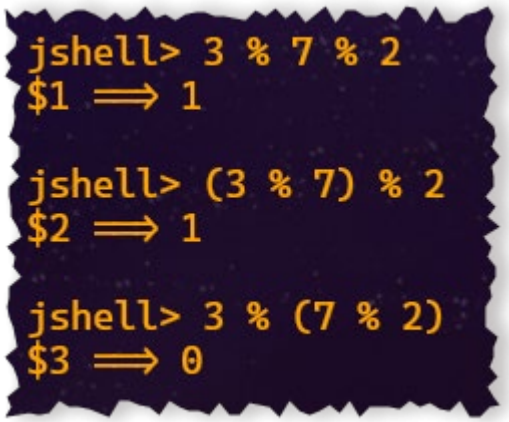
Assoziativität

- Die **Assoziativität** («**associativity**») eines Operators bestimmt, wie ein Operand verwendet wird

- Ist ein Operator ⊗
  - **Linksassoziativ** («**left-associative**») → $X \otimes Y \otimes Z$ gleich $(X \otimes Y) \otimes Z$
  - **Rechtsassoziativ** («**right-associative**») → $X \otimes Y \otimes Z$ gleich $X \otimes (Y \otimes Z)$

Assoziativität

- Beispiele für Operatorassoziativität
  - Linksassoziativ: +, *, /, %
  - Rechtsassoziative: später



- Nebenbemerkung: In der Mathematik ist ein Operator *assoziativ*, falls gilt, dass $(X \otimes Y) \otimes Z$ gleich $X \otimes (Y \otimes Z)$

Ausdrücke mit verschiedenen Operatoren

- Was bedeutet «$X \otimes Y \odot Z$» für zwei verschiedenen Operatoren $\otimes$ und $\odot$?
  - Beispiel: `2 + 6 * 5` → `2 + (6 * 5)` (nicht `(2 + 6) * 5`)

- Operand *Y* wird mit dem Operator mit der höheren **Präzedenz** («**precedence**») verknüpft
  - Auch **Bindungsstärke** oder **Rangordnung** genannt
  - Bei den uns bisher bekannten Operatoren gilt «Punkt vor Strich»
  - D.h. `*`, `/`, `%` binden stärker als `+`, `-`

Implizite Klammerung

1. Operand wird vom Operator mit höherer *Präzedenz* gebunden
2. Wenn zwei Operatoren *dieselbe Präzedenz* haben, dann entscheidet die Assoziativität
3. Wenn zwei Operatoren *dieselbe Präzedenz und Assoziativität* haben, dann werden die (Teil-)Ausdrücke von links nach rechts ausgewertet

- Explizite Klammern verwenden
  - Wenn eine andere Auswertung gewünscht
  - Wenn Ausdrücke komplexer werden (→ Lesbarkeit)

Implizite Klammerung: Beispiel

			`6 + x / 2 * y`
/ und * binden (gleich) stärker als +			
/ links von *,	daher / zuerst	→	`6 + (x / 2) * y`
	danach *	→	`6 + ((x / 2) * y)`
zuletzt +		→	`(6 + ((x / 2) * y))`

```
jshell> 6 + 5 / 2 * 9
$1 ⟹ 24

jshell> 6 + ((5 / 2) * 9)
$2 ⟹ 24
```

```
jshell> 6 + 5 / (2 * 9)
$3 ⟹ 6

jshell> (6 + 5) / (2 * 9)
$4 ⟹ 0

jshell> ((6 + 5) / 2) * 9
$5 ⟹ 45
```

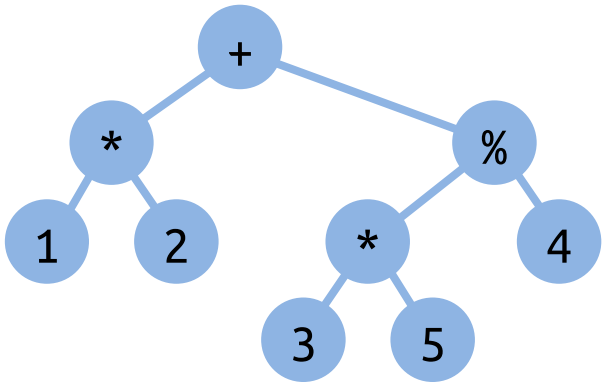
Ausdrucksbäume

Ein (vollständig geklammerter) Ausdruck lässt sich als **Ausdrucksbaum** («**expression tree**») darstellen

Ausdruck

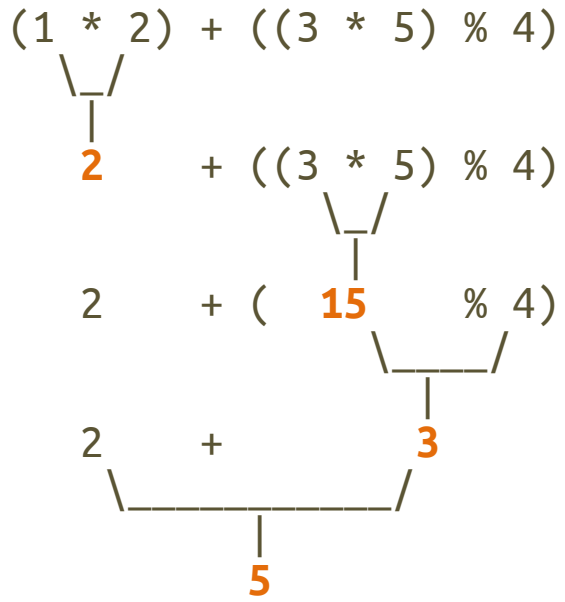
```
(1 * 2) + ((3 * 5) % 4)
```

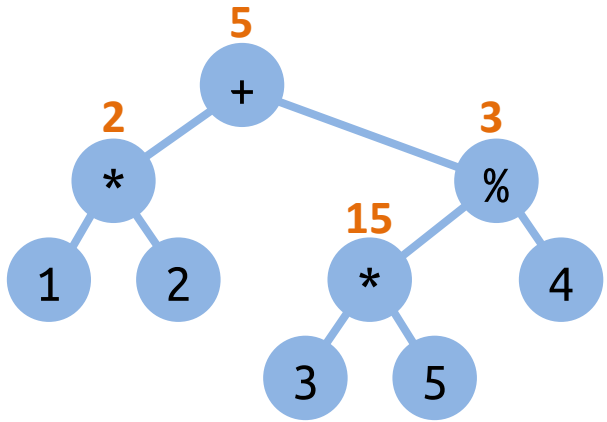
Baum



Evaluationsreihenfolge

Die Operanden eines Operators werden von links nach rechts **ausgewertet**/**evaluiert** («**evaluated**») ➜ Ergebnisberechnung





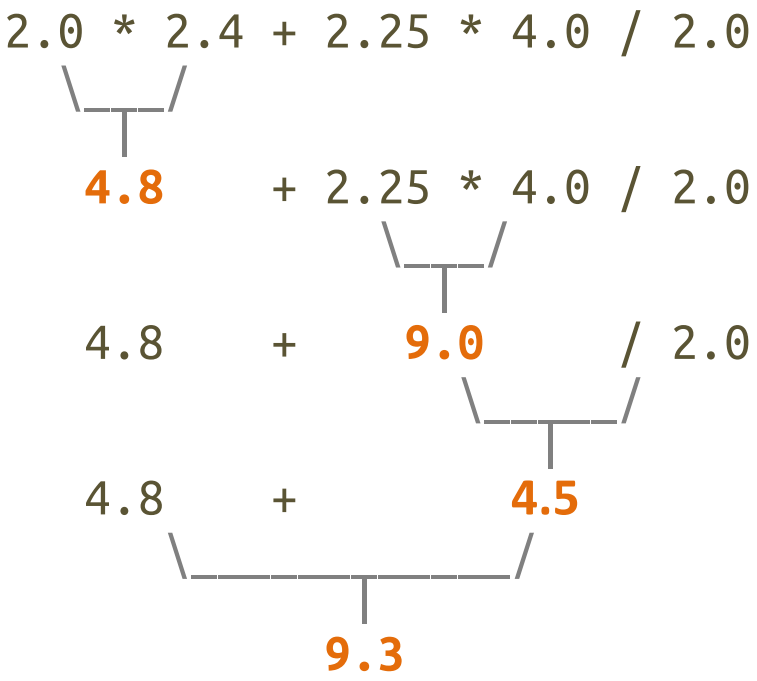
Fragen: Wozu evaluieren diese Ausdrücke?

1. `9 / 5`
2. `695 % 20`
3. `7 + 6 * 5`
4. `7 * 6 + 5`
5. `248 % 100 / 5`
6. `6 * 3 - 9 / 4`
7. `6 + 18 % (17 - 12)`

Typ `double` für Kommazahlen

- Beispielliterale vom Typ `double`:
  - `6.022`, `-42.0` — wie ganze Zahlen, aber mit einem Punkt
  - `1e2, 1e-3, 2.143e17` — *Exponentenschreibweise* X`e`Y steht für $X \cdot 10^Y$
- Arithmetische Operatoren `+`, `-`, `*`, `/`, `%` auch für `double` definiert
  - Division nun wie erwartet: `15.0 / 2.0` ergibt `7.5`
- Assoziativität und Präzedenz wie bisher

Beispielevaluation



`Double` ungleich reelle Zahlen

- Werte vom Typ `double` werden intern mit 64 Bit dargestellt → endliche Darstellung
  - Grösste/kleinste Werte
  - Rundungsfehler (Lücken im Wertebereich)

```
jshell> Double.MAX_VALUE
$1 ⟹ 1.79769313486231570E308

jshell> Double.MIN_VALUE
$2 ⟹ 4.9E-324
```

- `double`-Werte heissen **Fliesskommazahlen** («**floating point numbers**»). Grobe Idee:
  - $0{,}01 \cdot 10^7 = \mathbf{1{,}0 \cdot 10^5} = 100 \cdot 10^3$
    Normalisierung durch Exponentenanpassung («Komma fliesst»)
  - 64 Bit für das Paar (*Vorkommazahl*, *Exponent*) — platzeffiziente Speicherung

```
jshell> 1.0 + 0.1
$3 ⟹ 1.1

jshell> 1.1 + 0.1
$4 ⟹ 1.2000000000000002
```

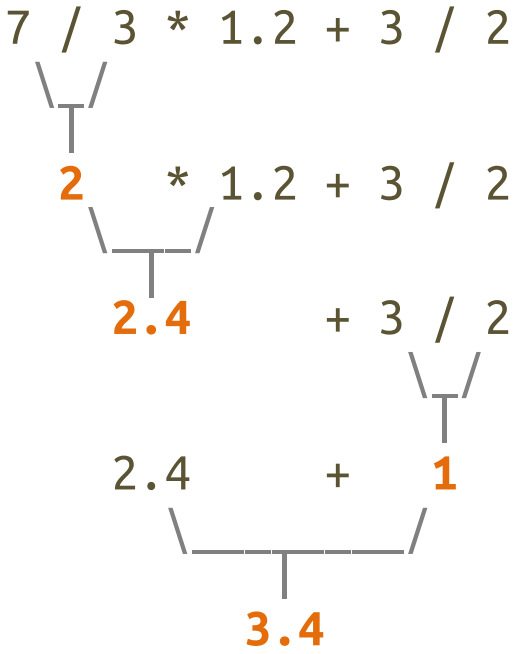
- Details in *Digitaltechnik* (2. Semester) und *Numerische Methoden* (3. Sem.)

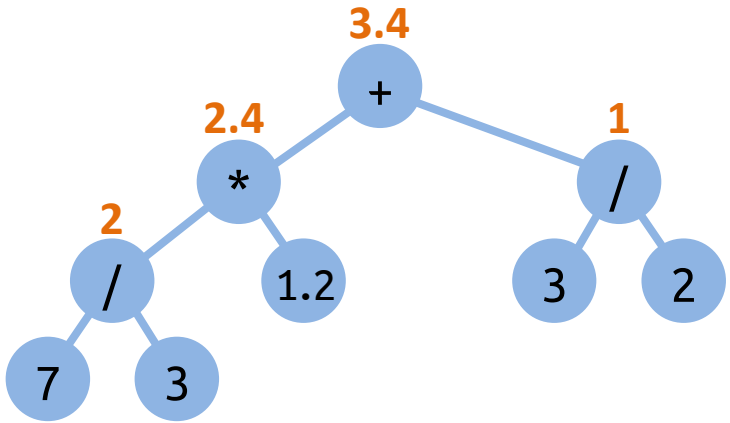
Ausdrücke mit verschiedenen Typen

- Für Operatoren +, *, -, /, %: *TypA* ⊗ *TypA* ergibt *TypA* ✓
- **Welchen Typ liefert der *gemischte Ausdruck TypA* ⊗ *TypB*?**
  - `3 / 2.0` ergibt …? Das gleiche wie `3.0 / 2`?
- Kombination `int`/`long` und `double` ergibt `double`
  - `3 / 2.0` und `3.0 / 2` ergeben `1.5`
  - `4.0 / 2` ergibt `2.0` (nicht den `int` 2)
- **Umwandlung** («**conversion**») vom kleineren zum grösseren Typ
  - Separat für jeden Operator bzw. dessen Operanden
  - Implizit (automatisch) – explizite Typumwandlungen später

Beispielevaluation

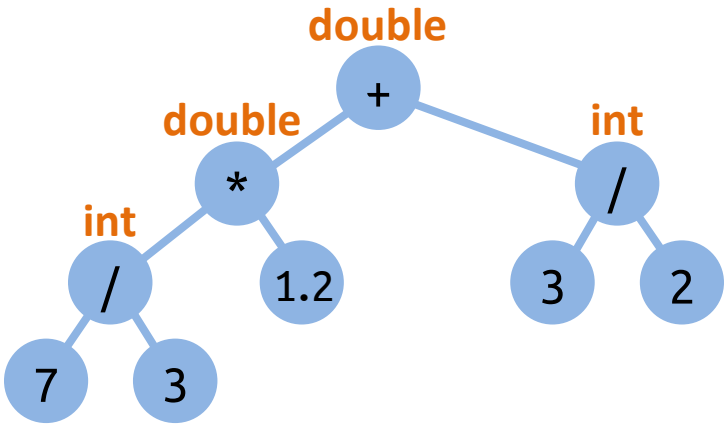
Umwandlung geschieht für jeden Operator





Typherleitung

Baumstruktur auch praktisch für die Typherleitung



Fragen: Wozu evaluieren diese Ausdrücke?

1. `4 + 8 / 3.0 * 6 + 5`
2. `9.0 / (2.0 / 3) + 7`
3. `7 % 3 * 2 + 4.0 * 3 / 2`
4. `9 / (2 / 3) + 7`
5. `20 % 8 + 15 / 27 / (3 % 6)`

Explizite Typumwandlungen

- Gesehen: Impliziten Umwandlungen bei Operatoren
  - Beispiel: `1.0 / 4` ergibt `double`
- Explizite Typumwandlung heissen «**Cast**» («**Type Cast**»)
- Anwendungen
  - Präzision erhöhen: z.B. `int` zu `double` casten, damit eine Division das gewünschte exakte Ergebnis liefert
  - Präzision senken: z.B. um ungewollte Nachkommenstellen abzuschneiden

Explizite Typumwandlungen

- **Syntax:** (*type*) *expression*
- **Beispiele**
  - `(double) 19 / 5` ergibt `3.8` (statt `3`)
  - `(int) (2.1e3 / 1.3e4 * 100)` ergibt `16` (statt `16.15`…)
- (*type*) ist ein Operator
  - Der «cast operator»
  - Binder stärker (höhere Präzedenz) als arithmetische Operatoren
  - Unär und *rechtsassoziativ* — wandelt *nur den Operanden rechts daneben* um
- Für fast alle primitiven Typen verfügbar

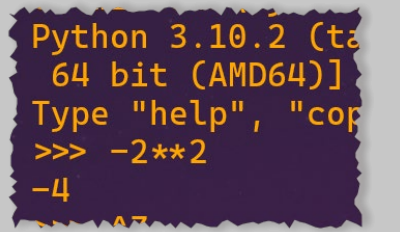
Cast-Operator

- Cast-Operator bezieht sich nur auf seinen rechten Operanden
- Was ergeben …?
  - `(double) 1 + 1 / 2` ergibt `1.0`
  - `1 + (double) 1 / 2` ergibt `1.5`
  - `(double) (2 + 1 + 1) / 3` ergibt `1.333…`

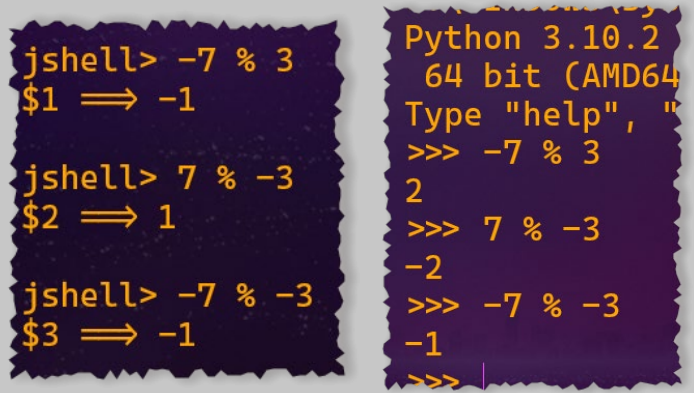
Abschluss numerische Ausdrücke

Die Folien dieser Vorlesung zeigen die Regeln (z.B. Präzedenzen) für *Java.* Für andere Programmiersprachen gelten (eventuell) andere Regeln. Beispiele:

1. Python vs. Excel:
   Präzedenz Negation und Potenzierung



1. Java vs. Python:
   Modulo mit negativen Zahlen



Abschluss numerische Ausdrücke

- Java kennt noch weitere numerische Datentypen und Literalformate
  - Regeln und Verhalten *analog* zu bisher Gezeigtem (`int` und `double`), *unterscheiden* sich in unterstützten Wertebereichen
  - Details für uns nicht weiter relevant → bei Bedarf nachschlagen
- Beispiele (Auswahl)
  - Typ `short`, Wertebereich -128 to 127 (keine eigenen Literale)
  - Typ `long`, Wertebereich -2^{63} to $2^{63} - 1$ (Literale mit Suffix `L`, z.B. `9876543210L`)
  - Typ `char` (Zeichen, später mehr), Wertebereich 0 bis 65535 (Literale z.B. `'g'`, `'\u042E'`)
- Weitere Integer-Literalformate (Auswahl)
  - Mit Dezimaltrenner, z.B. `10_000`, `1_000_000`
  - Hexadezimal (z.B. `0xFF`)

Relevante Java-Spezifikation auf docs.oracle.com: Typen, Literale

Zwischenstand Typen und Operationen

1. Primitive Typen – Beispiele `int` und `double`
   - Haben Werte (`4`, `2.17`) und Operationen (Division mittels `/`) gesehen
2. Typen aus der Java-Standardbibliothek – Beispiel `String`
   - Haben Werte bereits gesehen (`"hello"`, `"2 + 3"`)
   - Jetzt: Verkettung mittels *Plus-Operator*
   - Später: Operationen mittels *Methoden* (`"hello".toUpperCase()`)
3. Später: Selbst entwickelte Typen

Zeichenketten verketten

- **Verkettung** (**Konkatenation**, «**concatenation**») von String: Der Operator + erlaubt es, zwei Strings zusammenzufügen
  - `"You" + "Tube"` ergibt `"YouTube"`
  - `":" + "-" + ")"` ergibt `":-)"`

- Verkettung kann verschiedene Typen kombinieren
  - Was sollte/könnte `3 + " Grad Celsius"` ergeben?

Einschub: Standarddarstellung als String

- In Java gibt es für *jeden* Wert eine Darstellung als String
  - Gilt für *alle* Typen
  - Standarddarstellung immer vorhanden
    → kann jeden Wert mittels `println(…)` ausgeben
  - Standarddarstellung möglicherweise nicht ideal; kann für selbstentwickelte Typen auch geändert werden
- Für primitive Typen und Strings wie zu erwarten, z.B.
  - Für `42` und `-36.7` wie erwartet `"42"` und `"-36.7«`
  - Mit `int x = 77` für `x` wie erwartet `"77"`

Zurück zur Zeichenkettenverkettung

- Verkettung eines Strings mit einem Wert anderen Typs: Verkettung mit der Standarddarstellung des anderen Werts
  - `3 + " Grad Celsius"` ergibt `"3 Grad Celsius"`
  - `"Olympia " + 2024` ergibt `"Olympia 2024«`

- Praktisch für die Erzeugung von Ausgabe

```
double grade = 5.25 + 0.25;
System.out.println("Prüfungsnote ist " + grade);
  // erzeugt Ausgabe "Prüfungsnote ist 5.5"
```

Fragen: Wozu evaluieren diese Ausdrücke?

Auswertungsregeln wie gehabt

- `12 - 3 + 5`
- `2 + 3 + " Zehen"`
- `"Nummer " + 3 + 2`
- `"Nummer" + 2 * 3`
- `"Note " + (4.8 + 5.2) / 2`
- `"Note " + 4.8 + 5.2 / 2`

2. Erste Java-Programme

2.3 Einfache Berechnungen

Variablen – Bereits bekannt

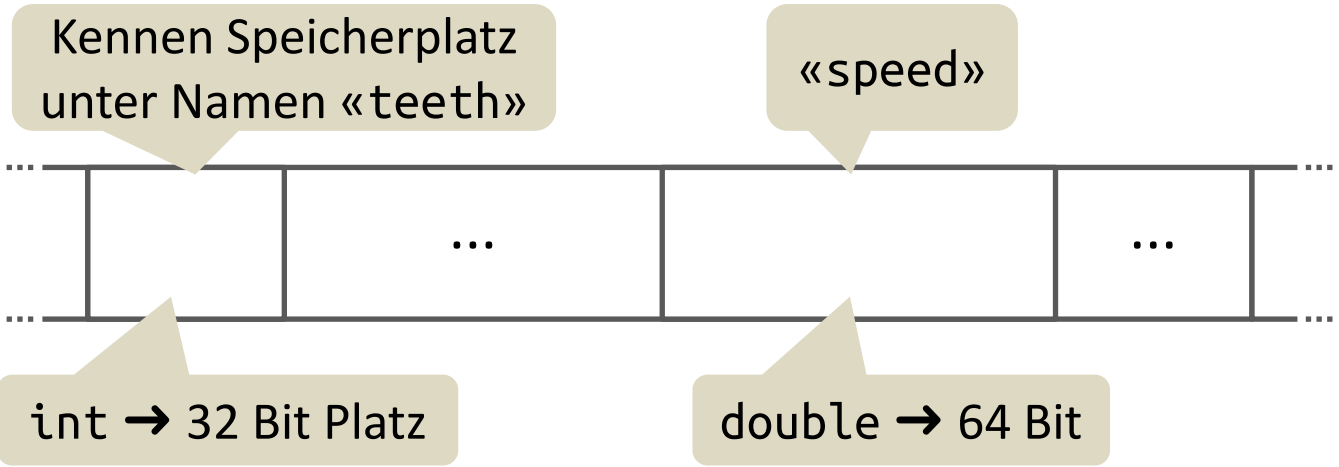
```
double inch;
inch = 6.4;
double centimeter = inch * 2.54;
System.out.println( inch + "\" sind " + centimeter + "cm");
inch = 22;
System.out.println( inch + "\" sind " + inch * 2.54 + "cm");
```

Deklarationen (Ankündigung/Bekanntmachung mit Typinformation)
Initialisierung (erstmalige Wertzuweisung; Schreibzugriff)
Zuweisung (Schreibzugriff)
Auswertung (Evaluation; Lesezugriff)

Deklaration

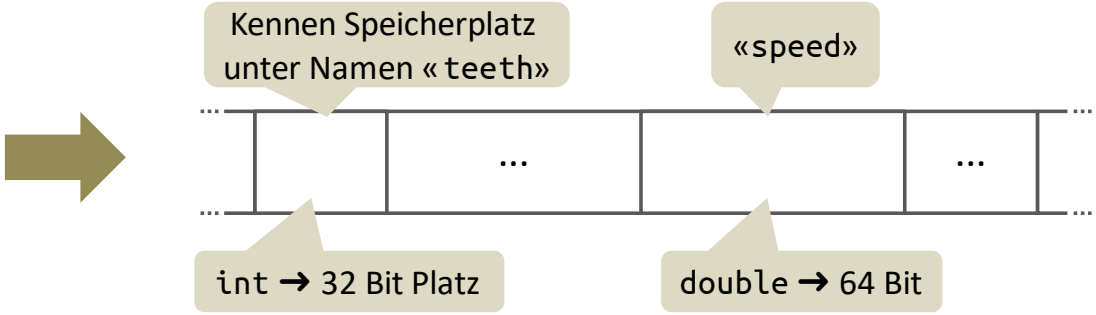
Deklaration einer Variable: Reserviert Speicherplatz gross genug für Werte des entsprechenden Typs

```
int teeth;
teeth = 28;
double speed = 0.83;
```



Deklaration

```
int teeth;
teeth = 28;
double speed = 0.83;
```



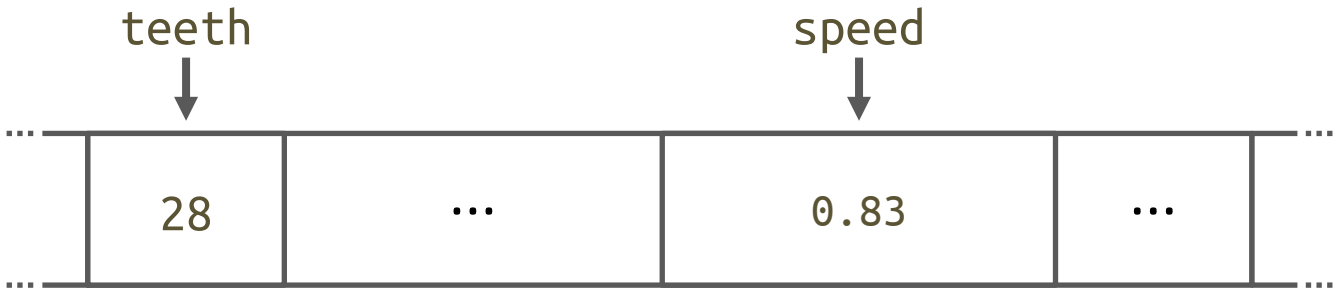
Deklaration einer Variable: Reserviert Speicherplatz gross genug für Werte des entsprechenden Typs

- Intuitive Vorstellung eines «Speicherbands mit benannten Zellen» für uns ausreichend
- Weitere technische Details z.B. in *Compiler Design*

Initialisierung

Initialisierung einer Variable: Erstmaliges Schreiben (Zuweisen) eines Werts in den reservierten Speicherplatz

```
int teeth;
teeth = 28
double speed = 0.83
```



Mögliche Fehler

- Fehlende Deklaration (jede Variable muss deklariert werden)

```
// int teeth;
teeth = 28;
```

```
Unresolved compilation problem:
   teeth cannot be resolved to a variable
```

- Nutzung vor Deklaration

```
teeth = 28;
int teeth;
```

```
Unresolved compilation problem:
   teeth cannot be resolved to a variable
```

- Mehrfachdeklaration (Typen irrelevant)

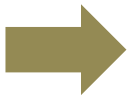
```
int teeth;
long teeth;
```

```
Unresolved compilation problem:
   Duplicate local variable teeth
```

Mögliche Fehler

- Fehlende Initialisierung
  (lokale Variablen haben keinen Standardwert)

```
int teeth;
println(teeth);
```



```
Unresolved compilation problem:
   The local variable teeth may not have
   been initialized
```

- Initialisierung mit Wert falschen Typs

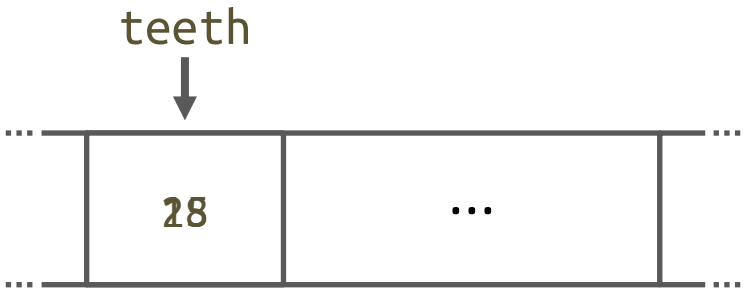
```
int teeth;
teeth = "28";
```

```
Unresolved compilation problem:
   Type mismatch: cannot convert from
   String to int
```

Zuweisung

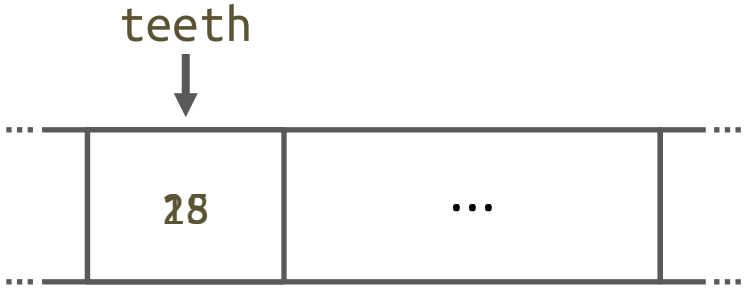
Zuweisung einer Variable: Schreiben eines (neuen) Werts in den entsprechenden Speicherplatz

```
int teeth = 28;
teeth = (13 + 17) / 2;
```



Evaluation

Auswerten einer Variable: In Ausdrücken evaluiert eine Variable zu dem Wert, den die Speicherzelle zu *diesem Zeitpunkt* hat



Java

```
int teeth = 28;
println( teeth );
teeth = (13 + 17) / 2;
println( teeth + 1 );
```

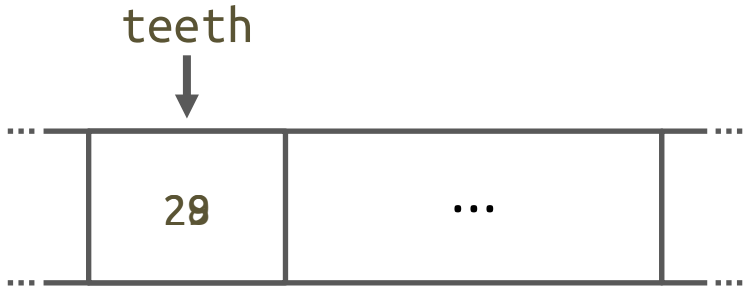
Ausgabe

```
28
16
```

Evaluation + Zuweisung

Variable evaluiert zum *aktuellen* Wert
➜ neuer Wert kann von diesem abhängen

```
int teeth = 28;
teeth = teeth + 1;
```



Zuweisung ist keine Gleichheit!

- Mathematik: $x = x + 1$ («x ist gleich sich selbst plus 1») sinnfrei
- Informatik: Sinnvoll, da eine Zuweisung einen *Effekt* hat (Zustandsänderung)

2. Erste Java-Programme

2.1 Einführung: Vom Quelltext zur Ausgabe

2.2 Methoden I: Programmstrukturierung

2.3 Einfache Berechnungen

2.4 Input und Random

2.5 Logische Aussagen über Programm(segment)e

Interaktive Programme

- Programm kann auf Benutzer (Ausführer:in des Programms) reagieren
  - Z.B. Lesen eines Input-Texts
    - Aufforderung zur Eingabe eines Inputs
    - Einlesen des durch Benutzer getippten Inputs in Variable
    - Ausgabe von Text auf Bildschirm
- Verschiedene Input-Quellen möglich, z.B.
  - Konsole («console»)
  - Datei
  - Website
  - Datenbank
  - Und viele mehr

Konsole («console»)

- Früher:
  mit Computer verbundenes externes Gerät

- Heute:
  Fenster im User Interface
  - Standard-Input (`System.in` in Java)
  - Standard-Output (`System.out` in Java)

Eclipse verwendet ein Fenster um `System.in` und `System.out` zu zeigen!

Input ist komplex

- Unterschiedliche Input-Quellen
- Verhalten von Benutzer ist nicht vorhersehbar oder kontrollierbar
  - Fehlerhafte Eingaben müssen abgefangen werden
- Umwandeln der Darstellung auf Konsole zu Darstellung in Programm (z.B. dezimal zu binär)
  - Mehr dazu in «Digital Design and Computer Architecture» im Frühling

In Java: `Scanner` bietet diese Funktionalität für Input an

- bei Output kümmert sich z.B. `println` darum

Scanner: Syntax

- `Scanner` aus Bibliothek `java.util` importieren
  - `import java.util.Scanner;`
- `Scanner`-Objekt konstruieren mit Angabe der Input-Quelle
  - `Scanner name = new Scanner(System.in);`

Ein neues Objekt (eines nicht-primitiven Typs) kann mit dem Keyword `new` erstellt werden!

Input-Quelle

- Aufruf einer Methode mit Punktnotation («dot notation»)

Scanner: Eine Auswahl von Methoden

Method	Description
`nextInt()`	reads an `int` from the user and returns it
`nextDouble()`	reads a `double` from the user …
`next()`	reads a one-word `String` from the user …
`nextLine()`	reads a one-line `String` from the user …

```
Scanner myConsole = new Scanner(System.in);
int alter = myConsole.nextInt();
System.out.println("Ihr Alter: " + alter);
```

Scanner: Dokumentation

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Scanner.html

nextInt

```
public int nextInt()
```

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:

the int scanned from the input

- Alle Informationen und noch viel mehr
  - Ignorieren Sie (vorerst), was Sie nicht verstehen

Scanner: Ablauf der Eingabe

- Ausgabe eines Prompts (Eingabeaufforderung) an Benutzer
  - Erscheint im Konsolenfenster
  - Information an Benutzer, dass (und welche) Eingabe erwartet wird
- Eingabe des Inputs durch Benutzer
  - Unterbruch des Programms bis Eingabe erfolgt ist
  - Eingabe wird beendet durch Enter-Taste
  - Erscheint im gleichen Konsolenfenster wie Ausgabe
- Programm wird fortgesetzt

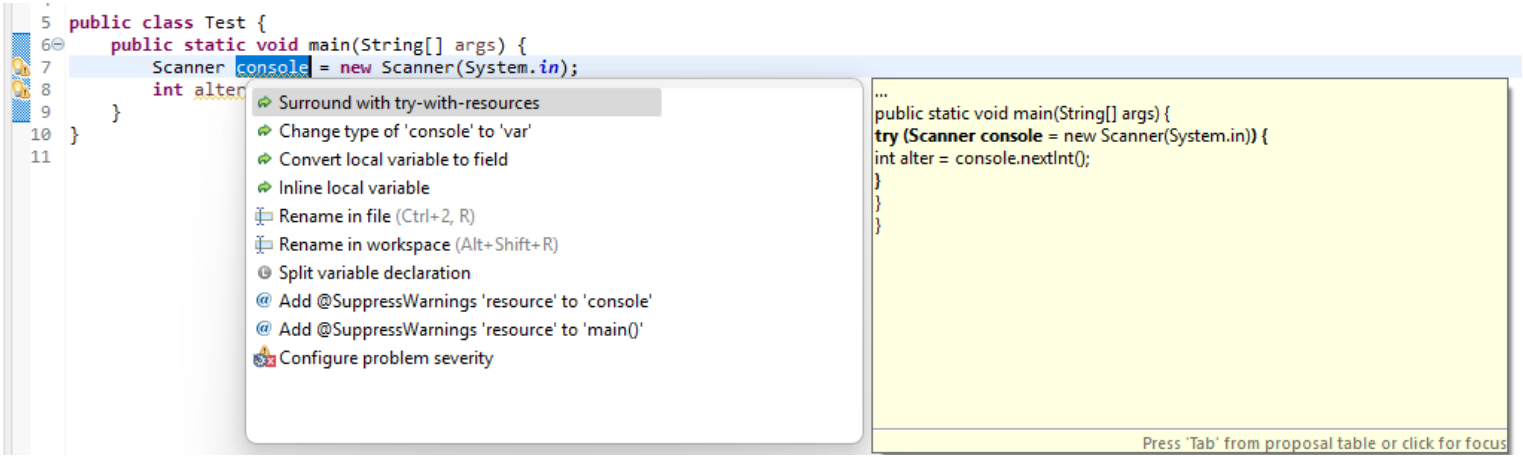
Scanner: Beispiel 1

```
import java.util.Scanner;
public class UserInputExample {
   public static void main(String[] args) {
        Scanner myConsole = new Scanner(System.in);
        System.out.print ("Wie alt sind Sie? "); // Prompt
        int alter = myConsole.nextInt();
        int jahre = 65 - alter;
        System.out.println(jahre + "Jahre bis zur Pensionierung");
   }
}
```

```
Wie alt sind Sie? 32
33 Jahre bis zur Pensionierung
```

Scanner: Warnung zu Resource Leak

- geöffnete Ressourcen (Dateien, …) müssen wieder geschlossen werden
- geschieht z.B. in einem `try`-Block (mehr dazu später)



- Scanner ohne diesen Block führt zu Warnung
- Warnung kann für den Moment ignoriert werden!

Scanner: Beispiel 2

```
import java.util.Scanner;
public class UserInputExample {
   public static void main(String[] args) {
        Scanner myConsole = new Scanner(System.in);
        System.out.print("Zwei Zahlen bitte: "); // Prompt
        int a = myConsole.nextInt();
        int b = myConsole.nextInt();
        int product = a * b;
        System.out.println(product);
   }
}
```

mehrere Zahlen in einer Zeile (bis Enter) lesen

```
Zwei Zahlen bitte: 6 7
42
```

Beispiel: Falscher Input

```
System.out.print("Wie alt sind Sie? ");
int alter = myConsole.nextInt();
```

Falscher Input-Typ

```
Wie alt sind Sie? Zweiunddreissig
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:964)
        at java.util.Scanner.next(Scanner.java:1619)
        at java.util.Scanner.nextInt(Scanner.java:2284)
        ...
```

Eingabeelement («token»)

- Folge von Zeichen, die vom Scanner am Stück gelesen werden
  - Scanner muss wissen, wo Beschreibung aufhört (und anfängt)
  - Separiert durch Zwischenraum («whitespace»)
    - Leerzeichen («space», «blank»), Tabulator («tab»), neue Zeile («new line»)
  - Kann kontextabhängig sein
    - `45 18 $2.50 hello world`
    - `$2.50 "hello world" 1.456E12 "45 18"`
  - Erwartete Zeichen hängen von Methode ab
    - Z.B. Unterschied zwischen `nextInt()` und `nextDouble()`
  - Braucht Beschreibung von legalen Zeichenfolgen (EBNF!)
    - Fehlermeldung (zur Laufzeit), falls Token nicht den richtigen Typ/die richtige Form hat!

Beispiel: `next()`

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
name = name.toUpperCase();
System.out.println(name + " has " + name.length() + " letters ");
```

Liest ein Wort bis zum Zwischenraum

```
What is your name? Minnie
MINNIE has 6 letters
What is your name? Minnie Mouse
MINNIE has 6 letters
What is your name? "Minnie Mouse"
"MINNIE has 7 letters
```

Beispiel: `nextLine()`

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.nextLine();
name = name.toUpperCase();
System.out.println(name + " has " + name.length() + " letters " +
```

Liest eine Zeile

```
What is your name? Minnie
MINNIE has 6 letters
What is your name? Minnie Mouse
MINNIE MOUSE has 12 letters
What is your name? "Minnie Mouse"
"MINNIE MOUSE" has 14 letters
```

Zufallszahlen («random numbers»)

random {adjective}
zufällig
willkürlich [wahllos]
zufallsbedingt
Zufalls-
dem Zufall überlassen

(aus EN-DE-Wörterbuch)

- Oft als Ersatz für Zahlen-Input

(Pseudo-)Zufallszahlengenerator `Random`

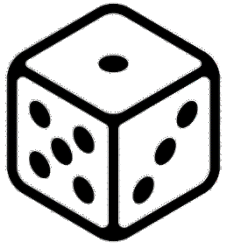
- `Random` aus Bibliothek `java.util` importieren
  - `import java.util.Random;`
- `Random`-Objekt konstruieren
  - `Random name = new Random();`
- Verschiedene Methoden für unterschiedliche Modi

Method	Description
`nextInt()`	returns a ”random integer”
`nextInt(n)`	returns a ”random” integer in the range [0, `n`) in other words: returns a number from {0, …, `n` – 1} at ”random”
`nextDouble()`	returns a “random real” number in the range [0.0, 1.0)

Häufige Szenarien

- Zufällige ganze Zahl aus [`1`, `n`]
  - `1 + rand.nextInt(n)`
- Zufällige ganze Zahl aus [`min`, `max`]
  - `min + rand.nextInt(max - min + 1)`
- Zufällige reelle Zahl aus [`min`, `max`]
  - `min + rand.nextDouble() * (max - min)`
- Zufälliges Objekt aus einer Menge von `n` Objekten
  - Beliebige Nummerierung der Objekte mit [`1`, `n`] oder [`0`, `n-1`]

Beispiel 1: Würfel



```java
import java.util.Random;
public class DiceThrow {
    public static void main(String[] args) {
        Random rand = new Random();
        int face = 1 + rand.nextInt(6);
        System.out.println("You threw a " + face);
    }
}
```

Beispiel-Output: `You threw a 6`

Beispiel 2: Zufällige Note in [1.0, 6.0]

```
import java.util.Random;
public class RandomGrade {
   public static void main(String[] args) {
        Random rand = new Random();
        double r = 5.0 * rand.nextDouble(); // 0 <= r <= 5
        double grade = 1.0 + r;
        System.out.println("You got a " + grade);
   }
}
```

Beispiel-Output: `You got a 4.2`

Beispiel 3: Zufällige gerade Zahl zwischen 4 und 12

```
import java.util.Random;
public class RandomGrade {
   public static void main(String[] args) {
        Random rand = new Random();
        int r = 2 * rand.nextInt(5) + 4;
        System.out.println(r);
   }
}
```

Beispiel-Output: 6

2. Erste Java-Programme

2.5 Logische Aussagen über Programm(segment)e

2.5.1 Motivation, Aussagen, logisches Schliessen

2.5.2 Hoare-Tripel für Zuweisungen

Vogelperspektive Softwareentwicklung

- Grosse Programme werden aus einzelnen *Modulen* (Methoden, Klassen, Bibliotheken, …) zusammen-gesetzt ➜ Modulare Entwicklung

- Voraussetzung: Es muss klar *spezifiziert* werden, was ein Modul leisten soll
  - Für Modulentwickler/innen: Wann ist das Modul vollständig bzw. was fehlt noch?
  - Für Modulnutzer/innen: Wie ist das Modul zu benutzen? Was kann ich erwarten?

Bild mit Handy-Komponenten: ifixit.com

Beispielspezifikationen

```
// PRE:  x ≥ 0
// POST: result² = x
double sqrt(double x) {
  ...
  // viel und komplexer Code
  ...
}
```

```
// PRE:  true
// POST: ∀ i. v_i ≤ v_{i+1}
void sort(vector v) {
  ...
  // viel und komplexer Code
  ...
}
```

Spezifikationen in «mathematischer Sprache» (Logik → *Diskrete Mathematik*), damit eindeutig und beweisbar

- **Vorbedingung** («**precondition**»): Was muss vor dem Aufruf gelten?
- **Nachbedingung** («**postcondition**»): Was gilt nach dem Aufruf?
- Hier als spezielle Java-Kommentare ins Programm geschrieben

result bezieht sich auf das Ergebnis der Berechnung (den Rückgabewert der Methode)

Vor- und Nachbedingungen: Zwei Perspektiven

```
// PRE:  x ≥ 0
// POST: result² = x
double sqrt(double x)
```

Methodenentwickler

```
double sqrt(double x) {
  // Kann x ≥ 0 annehmen
  // ... Code ...
  // Muss result² = x garantieren
}
```

Methodennutzer

```
// Muss x ≥ 0 garantieren
y = sqrt(x);
// Kann y² = x annehmen
```

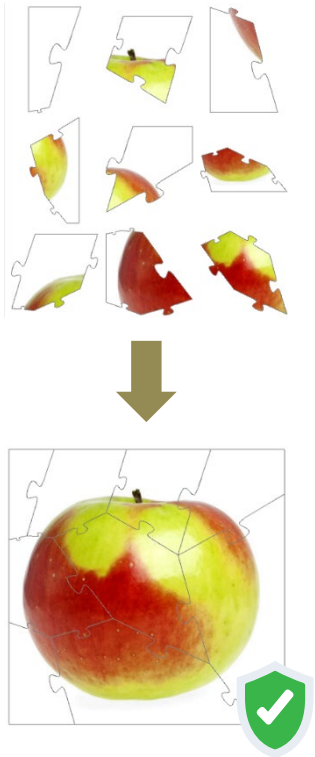
Eine Spezifikation aus Vor- und Nachbedingung etabliert quasi einen *Vertrag*:

- Entwickler/in muss nicht alle Nutzungen kennen; verlässt sich auf die Vorbedingung
- Nutzer/in muss die Implementation nicht kennen; verlässt sich auf die Nachbedingung

Nutzen spezifizierter Software

- **Vision**
  - Jedes Modul erhält eine formale Spezifikation
  - Programm entsteht durch Kombination passender Module
  - *Mathematischer Beweis*, dass das gesamte Programm das gewünschte Ergebnis liefert/sich wie gewünscht verhält

- **Für uns nützlich: Denken in Spezifikationen hilft beim Programmieren und kann die Entwicklung lenken**
  - Wir nutzen die Ideen und Konzepte informell/intuitiv
  - Formale Theorie folgt dann in *Formal Methods* (4. Semester)



Apfel-Puzzle: mathematica.stackexchange.com

Einstieg: Aussagen über Programmsegmente

```
int u = ...; // u hat irgendeinen Wert
int x = 17;
int y = 42;
int z = u + x + y;
```

- Wir wollen eine Aussage über obige Berechnung tätigen
  - Muss Variable (Endergebnis) `z` involvieren
  - Beziehen sich immer auf einen *bestimmten Programmpunkt* (Zeile)
- Diese Aussage ist die *Nachbedingung* des obigen Codes
  - Beispiel: `z > 0` (gilt dies?)

(Logische) Aussagen

- **Aussage** («**assertion**»): Behauptung, die entweder wahr oder falsch ist
  - Wir fragen dann oft «Ist die Aussage wahr?» oder «Gilt sie?»
  - «Logisch» heisst hier «im Sinne der mathematischen Logik»
- Diverse Beispiele:
  - EProg ist ein Fach im Basisjahr Informatik
  - Seebach ist der Hauptort des Kantons Zürich
  - 11 ist eine Primzahl
  - 13 ist eine gerade Zahl
  - $x \geq 0$ *(hängt von x ab)*
  - x geteilt durch 2 ergibt 8 *(hängt von x ab)*

(Logische) Aussagen

- Aussage: Behauptung, die entweder wahr oder falsch ist
  - Nicht alle Aussagen sind wahr
  - Sind evtl. nur unter bestimmten Bedingungen wahr (z.B. «x % 5 ist 3»)

- Wichtig ist, dass es Sinn macht zu fragen, ob die Aussage wahr oder falsch ist
  - Für uns: Im Kontext der Entwicklung eines Programms

```
int u = ...;
int x = 17;
int y = 42;
int z = u + x + y;
// Gilt «z > 0»?
```

Aussagen in EProg

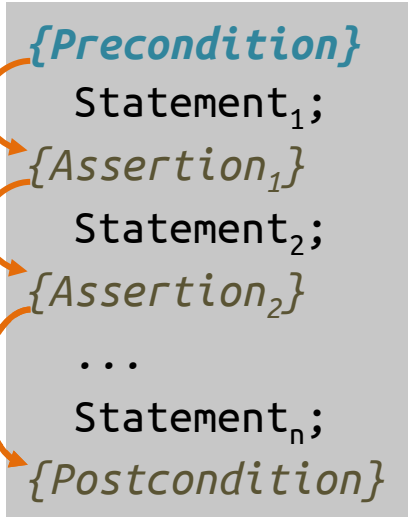
- Wir schreiben Aussagen als Kommentare in den Quelltext
  - Kommentare ➜ keine Bedeutung für Java, für uns aber schon
  - Geschweifter Klammern «{…}» sind übliche Notation (Lehrbücher, *Formal Methods*)
  - Die Kommentarzeichen «//» lassen wir aus Platzgründen i.d.R. weg

```
// {x > 0 und x < y}
  z = x * y;
// {z > 0}
```

```
{x > 0 und x < y}
...
```

- Wir nehmen mathematische Zahlen an: $\mathbb{N}$ statt `int`, d.h. keine Überläufe (aber weiterhin ganzzahlige Division, d.h. `5 / 2` ist `2`)

Vorwärtsschliessen: Vorgehen



- **Start:** Wählen (wissen, raten) einer sinnvollen *Vorbedingung* (`Precondition` = $\texttt{Assertion}_0$)
- **Schrittweise:** Herleiten der nächsten Aussage ($\texttt{Assertion}_i$) durch Einbeziehen des *Effekts* der nächsten Anweisung ($\texttt{Statement}_i$)
- **Ziel:** Herleiten einer *gültigen Nachbedingung* (`Postcondition` = $\texttt{Assertion}_n$)

- Vorwärts = «Welche Garantien (Nachbedingung) kann mein Code, unter der gewählten Vorbedingung, geben?»

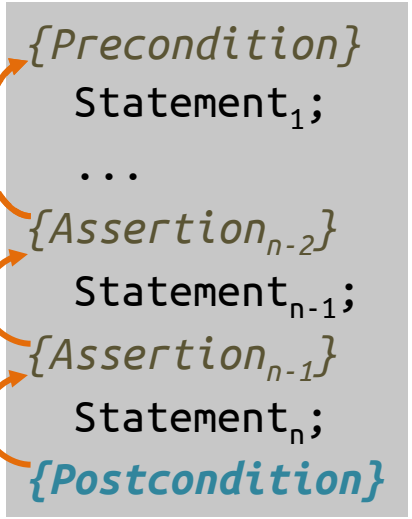
Vorwärtsschliessen: Beispiel

In Aussagen nutzen wir «==» für (mathematische) Gleichheit

```
{u > 0}
  x = 17;
{u > 0 ∧ x == 17}
  y = 42;
{u > 0 ∧ x == 17 ∧ y == 42}
  z = u + x + y;
{u > 0 ∧ x == 17 ∧ y == 42 ∧ z == u + 17 + 42}
```

- Wir wählen als *Vorbedingung* $u > 0$
- Pro Schritt: Wenn vorher Aussage A_{i-1} gilt, welche Aussage A_i gilt dann nach Ausführung des nächsten Statements S_i?
- Dann gilt die *Nachbedingung* $z > 59$ (warum?)

Rückwärtsschliessen: Vorgehen



- **Start:** Wählen (wissen, raten) einer sinnvollen *Nachbedingung*
- **Schrittweise:** Herleiten der vorherigen Aussage (Assertion$_i$) durch Einbeziehen des *Effekts* der nächsten Anweisung (Statement$_i$)
- **Ziel:** Herleiten einer *notwendigen und hinreichenden Vorbedingung*

- Rückwärts = «Welche Vorbedingung braucht mein Code, damit er die gewählten Garantien (Nachbedingung) geben kann?»

Rückwärtsschliessen: Beispiel

```
{u + 17 + 42 > 0}
  x = 17;
{u + x + 42 > 0}
  y = 42;
{u + x + y > 0}
  z = u + x + y;
{z > 0}
```

- Wir wählen als *Nachbedingung* $z > 0$
- Pro Schritt: Wenn nach Statement S_i Aussage A_i gelten soll, welche Aussage A_{i-1} muss dann vor der Ausführung von S_i gelten?
- Dann muss die *Vorbedingung* $u > -59$ gelten (warum?)

Vor-/Rückwärts: Gemeinsamkeit

Vorw.

```
{u > 0}
  x = 17;
{u > 0 ∧ x == 17}
  y = 42;
{u > 0 ∧ x == 17 ∧ y == 42}
  z = u + x + y;
{u > 0 ∧ x == 17 ∧ y == 42 ∧ z == u + 17 + 42}
```

Rückw.

```
{u > -59}
  x = 17;
{u + x + 42 > 0}
  y = 42;
{u + x + y > 0}
  z = u + x + y;
{z > 0}
```

Beide Paare aus Vor- und Nachbedingung («Verträge») sind *gültig*:
Wenn vor der Programmausführung die Vorbedingung gilt,
dann gilt nach der Ausführung die Nachbedingung.

Vor-/Rückwärts: Unterschiede

Vorw.

```
{u > 0}
  x = 17;
{u > 0 ∧ x == 17}
  y = 42;
{u > 0 ∧ x == 17 ∧ y == 42}
  z = u + x + y;
{u > 0 ∧ x == 17 ∧ y == 42 ∧ z == u + 17 + 42}
```

Rückw.

```
{u > -59}
  x = 17;
{u + x + 42 > 0}
  y = 42;
{u + x + y > 0}
  z = u + x + y;
{z > 0}
```

Vorwärts

- Erscheint anfangs «natürlicher», da es dem (linearen) Kontrollfluss folgt
- Hält viele Details in den Aussagen fest, die letztendlich irrelevant sind

Rückwärts

- Code «rückwärtszulesen» erfordert Übung, aber führt zu einer neuen Sicht auf das Programm
- Grosser praktischer Nutzen: Sie (müssen) verstehen, was jede Anweisung zum Erreichen des gewünschten Endzustands beiträgt

Aussagen zur Laufzeit: `assert`-Anweisung

- In Java-Programmen können Aussagen durch `assert`-Anweisungen ausgedrückt werden
  - **Syntax:** `assert` *expr*;
    - Für einen gültigen Java-Ausdruck *expr* (mit Typ `boolean` ➜ später mehr)
    - Daher keine geschweiften Klammern ({…}) um die Aussage
  - **Beispiel:** `assert x > 0;`
  - **Semantik:** Programmabbruch, falls Aussage nicht hält
- Hilfreich, um eigene Annahmen zu überprüfen

2. Erste Java-Programme

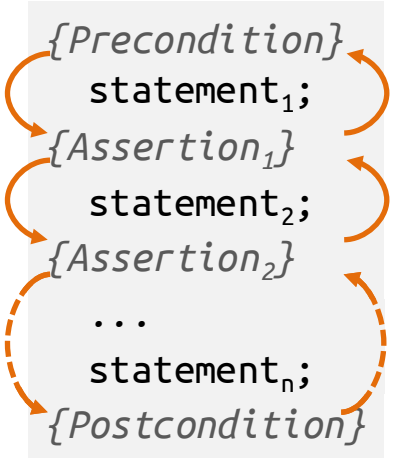
2.5 Logische Aussagen über Programm(segment)e

2.5.1 Motivation, Aussagen, logisches Schliessen

2.5.2 Hoare-Tripel für Zuweisungen

Hoare-Logik: Hintergrund

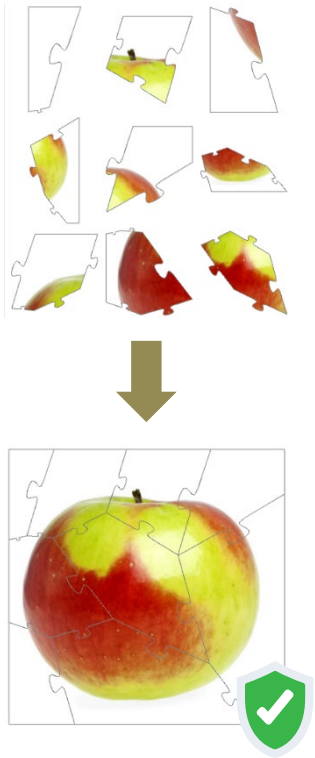
- Tony Hoare entwickelte in den 1970er Jahren einen formalen Ansatz, um Eigenschaften von Programmen *beweisen* zu können → **Hoare-Logik** («**Hoare logic**»)
  - Genutzt von z.B. Amazon, Microsoft, Airbus;
    aber auch von kleineren Firmen in CH, ZH
  - Formale Beweise, plus Tools, in *Formal Methods* (4. Semester)



- Im (Arbeits-)Alltag genügt es oft, weniger detailliert zu argumentieren, als es Hoare-Logik erfordert
- **Unsere Ziele: Grundlegende Konzepte einführen und üben, so dass**
  1. **Sie für einfache Programme (relativ) genau argumentieren können**
  2. Ihnen der Einstieg in *Formal Methods* etwas leichter fällt

Logik und Programmentwicklung

- Sich mit Aussagen über Programmen zu beschäftigen ist eine gute Schulung, *systematisch zu programmieren*
  - Wir können Aussagen machen über Zustände (der Ausführung) eines Programms (und später auch eines Objekts)
  - D.h. wir können den *Effekt* eines Programms (semi-)formal beschreiben
- Wichtig für die Definition von Schnittstellen (zwischen Modulen), wenn wir entscheiden müssen, welche Bedingungen erfüllt sein müssen (z.B. um eine Methode aufzurufen)



Hoare-Tripel: Syntax

- Ein **Hoare-Tripel** («**Hoare triple**»), auch *3-Tupel* genannt, besteht aus zwei Aussagen und einem Programmsegment:

```
{P} S {Q}
```

- Bestandteile eines Tripels
  - P ist die Vorbedingung (Precondition, z.B. *u > -59*)
  - S ist eine Anweisung (Statement; bzw. mehrere Anweisungen)
  - Q ist die Nachbedingung (Postcondition, z.B. *z > -59*)

Hoare-Tripel: Bedeutung

- Ein Hoare-Tripel `{P} S {Q}` ist **gültig** wenn (und nur wenn):
  - Für *jeden* Zustand, für den P gültig ist, ergibt die Ausführung von S immer einen Zustand, für den Q gültig ist.
  - Informell: Wenn P wahr ist vor der Ausführung von S, dann muss Q nachher wahr sein
- Andernfalls ist das Hoare-Tripel **ungültig**.

Hoare-Tripel etablieren

- Wie etablieren (≈ beweisen) wir ein Hoare-Tripel `{P} S {Q}`?
- Für jede Java-Anweisung gibt es *genaue Regeln*, die eine Vor- und Nachbedingung (für diese Anweisung) in Beziehung setzen
  - Regel für Zuweisungen
  - Regel für aufeinander folgende Anweisungen
  - Regel für Verzweigungen, Schleifen (später)
  - …
- Regeln existieren als Vorwärts- und Rückwärts-Version
  - Wir lernen nur die Rückwärts-Regeln kennen, da relevanter

Hoare-Logik-Regel für Zuweisungen

- **Ziel:** Wir wollen ein Tripel `{P} X = E {Q}` etablieren, also zeigen, dass es **gültig** ist
  - P und Q sind Platzhalter für *beliebige* Aussagen
  - X ist ein Platzhalter für eine *beliebige* Java-Variable
  - E ist ein Platzhalter für einen *beliebigen* Java-Ausdruck

- **Regel** (**Vorgehen**, um **Gültigkeit** zu zeigen)
  1. Erhalte `Q'` durch: Ersetzen aller Vorkommen von X in Q durch E
  2. Zeige, dass P ⇒ `Q'` gilt (Implikation, d.h. das `Q'` gilt wenn P gilt)

Erklärung

`{P} X = E {Q}`

1. Bilden von `Q'` durch Ersetzen aller `X` in `Q` durch `E`
2. Zeigen, dass $P \Rightarrow Q'$ gilt

Intuition: Warum ergibt diese Regel Sinn?

- **Schritt 1** (syntaktische Ersetzung von X durch E)
  1. Nach der Zuweisung soll Q gelten
  2. Die Zuweisung ändert den Wert von X auf E – sonst ändert sich nichts
  - → Wenn wir die Aussage Q mit E statt X formulieren (= `Q'`), muss diese Aussage bereits vor der Zuweisung gegolten haben
- **Schritt 2** (Implikation)
  1. Gerade argumentiert: Wenn `Q'` vor der Zuweisung gilt, gilt danach Q
  2. $P \Rightarrow$ `Q'` bedeutet, dass P eine stärkere Vorbedingung als `Q'` ist
  - → P ist daher eine hinreichende Vorbedingung (damit am Ende Q gilt)

Beispiel 1

```
{b + 25 > 30}
  a = b + 25;
{a > 30}
```

`{P} X = E {Q}`

1. Bilden von `Q'` durch Ersetzen aller `X` in `Q` durch `E`
2. Zeigen, dass `P` ⇒ `Q'` gilt

- Tripel ist **gültig**
  - `X` ist *a*, `E` ist *b + 25*,
  - `Q` ist *a > 30*, `P` ist *b + 25 > 30*
  - `Q'` ist *b + 25 > 30*
  - `P` impliziert `Q'` trivialerweise, denn `P` ist gleich `Q'`

Erinnerung: Wir gehen in Aussagen von mathematischen Zahlen aus, d.h. b + 25 führt *nie* zu einem Überlauf

Beispiel 2

```
{b > 5}
  a = b + 25;
{a > 30}
```

`{P} X = E {Q}`

1. Bilden von `Q'` durch Ersetzen aller `X` in `Q` durch `E`
2. Zeigen, dass $P \Rightarrow Q'$ gilt

- Tripel ist **gültig**
  - X ist a, E ist $b + 25$
  - Q ist $a > 30$, P ist $b > 5$
  - `Q'` ist $b + 25 > 30$
  - $(b > 5) \Longrightarrow (b + 25 > 30)$ (Beweis: einfache Formelumstellung)

Beispiel 3

```
{b > 0}
  a = b + 25;
{a > 30}
```

`{P} X = E {Q}`

1. Bilden von `Q'` durch Ersetzen aller `X` in `Q` durch `E`
2. Zeigen, dass `P` ⇒ `Q'` gilt

- Tripel ist **nicht gültig**
  - `X` ist *a*, `E` ist *b + 25*
  - `Q` ist *a > 30*, `P` ist *b > 0*
  - `Q'` ist *b + 25 > 30*
  - `P` impliziert `Q'` *nicht* (Gegenbeispiel: wenn `b` den Wert `2` hat)

Fragen I

Welche der folgenden Tripel sind (un)gültig?

Bsp4

```
{x > 0}
  y = x + 1;
{y > 1}
```

Bsp5

```
{x > 100}
  y = x + 1;
{y > 1}
```

Bsp6

```
{v ≠ 1}
  w = v * v;
{w ≠ v}
```

Fragen II

Welche der folgenden Tripel sind (un)gültig?

Bsp7

```
{age > 94}
  age = age + 1;
{age > 95}
```

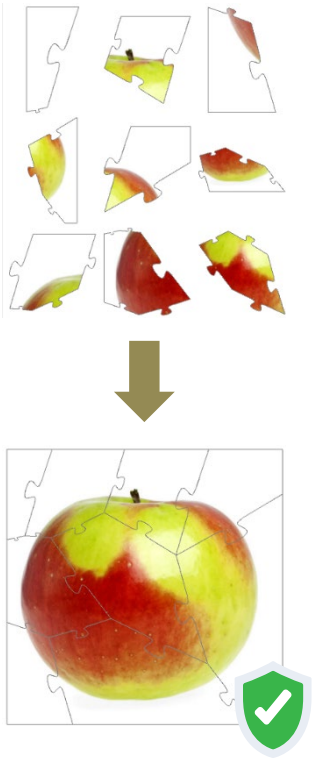
Bsp8

```
{y % z == 0}
  x = y / z;
{y == x * z}
```

Erinnerung: In Aussagen nutzen wir «==» für Gleichheit

Aussagen kombinieren

- Beobachtung Programmentwicklung
  - Gegeben zwei Programmsegmente S_1 und S_2, ergibt die *Kombination* S_1`;` S_2 ein grösseres Programm
  - D.h. wir entwickeln grössere Programme aus kleineren (Modularität)
- Analog für Tripel
  - Gegeben {P_1} S_1 {Q_1} und {P2} S2 {Q2}
  - Wir möchten diese zu {P1} S1`;` S2 {Q2} kombinieren



Hoare-Logik: Aussagen kombinieren

- Wenn zwei Aussagen aufeinander treffen, muss die vorherige Aussage die darauf folgende *implizieren*

```
{P1}
  S1;
{Q1}
{P2}
  S2;
{Q2}
```

Die Kombination links erfordert, dass

1. Tripel $\{P_1\}\ S_1\ \{Q_1\}$ gültig ist
2. Tripel $\{P_2\}\ S_2\ \{Q_2\}$ gültig ist
3. $Q_1 \Longrightarrow P_2$ gilt

- Analog
  - `{P1}{P2} S; {Q}` falls $P_1 \Longrightarrow P_2$
  - `{P} S; {Q1}{Q2}` falls $Q_1 \Longrightarrow Q_2$

Hoare-Logik: Aussagen kombinieren

$\{P_1\}$
  S_1;
$\{Q_1\}$
$\{P_2\}$
  S_2;
$\{Q_2\}$

Die Kombination links erfordert, dass

1. Tripel $\{P_1\}\ S_1\ \{Q_1\}$ und $\{P_2\}\ S_2\ \{Q_2\}$ gültig sind
2. $Q_1 \Longrightarrow P_2$ gilt

Intuition

- Wenn P_2 ausreicht, um S_2 so auszuführen, dass nachher Q_2 gilt …
- … und wenn Q_1 stärker ist als P_2 …
- … dann reicht auch Q_1 aus, um S_2 erfolgreich auszuführen

Hoare-Logik-Regel für Anweisungsfolgen

- Das Tripel $\{P\}\ S_1; S_2\ \{Q\}$ ist **gültig**, genau dann wenn
  - Aussage R existiert, so dass
  1. $\{P\}\ S_1\ \{R\}$ gültig ist
  2. $\{R\}\ S_2\ \{Q\}$ gültig ist

Beispiel

$\{P\}\ S_1; S_2\ \{Q\}$

Finde R, so dass gilt

1. $\{P\}\ S_1\ \{R\}$
2. $\{R\}\ S_2\ \{Q\}$

```
{x > 0} y = x + 1; z = y * y {z > y}
```

Mit R gleich $y > 1$ können wir Gültigkeit zeigen:

```
{x > 0}
  y = x + 1;
{y > 1}
  z = y * y
{z > y}
```

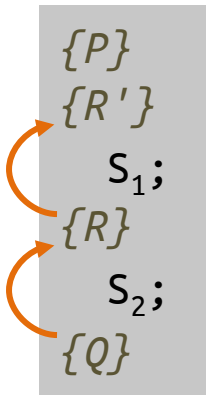
1. Tripel $\{x > 0\}\ y = x + 1\ \{y > 1\}$ ist gültig
   (Zuweisungsregel: $x > 0$ impliziert $x + 1 > 0$)
2. Tripel $\{y > 1\}\ z = y + y\ \{z > y\}$ ist gültig
   (Zuweisungsregel: $y > 1$ impliziert $y * y > y$)

Vorgehen für Anweisungsfolgen

- **Situation:** Entscheide, ob $\{P\}\ S_1; S_2\ \{Q\}$ gültig ist

- **Empfohlenes Vorgehen:**

1. Wende bekannte Regeln an, um Folgendes zu erhalten:

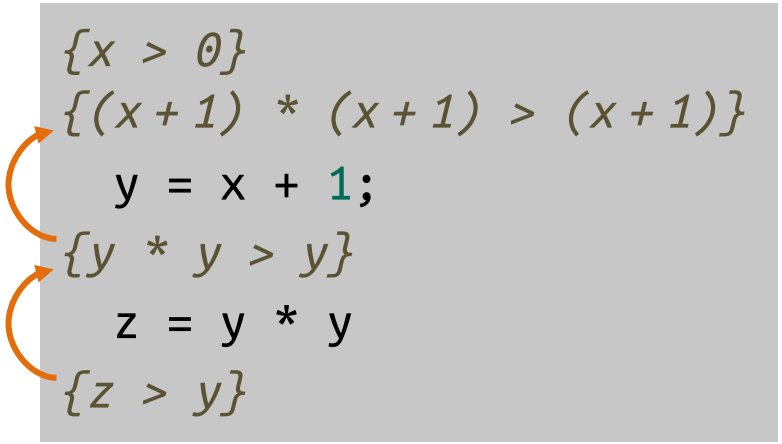


2. Zeige Implikation $P \Longrightarrow R'$

Beispiel

- Frage: Ist `{x > 0} y = x + 1; z = y * y {z > y}` gültig?

1. Regeln anwenden:



2. Implikation zeigen:

$(x > 0)$

$\Longrightarrow$

$(x+1) * (x+1) > (x+1)$

hält, denn für $x \geq 2$ ist $x^2 > x$

- Antwort daher: Ja

Frage

- Welche der folgenden Tripel sind gültig?

```
{true}
  x = y;
  z = x;
{y == z}
```

True als Vorbedingung bedeutet «keinerlei Anforderung» (Anfangswerte von x, y und z sind irrelevant)

```
{x == 7 ∧ y == 5}
  t = x;
  y = x;
  x = t;
{y == 7 ∧ x == 5}
```

Welche Aussagen wollen wir?

- Gleicher Code, unterschiedliche (gültige) Tripel

A1
```
{a > 0 ∧ b > 0}
  x = a + b;
{x > 0}
```
vs.
A2
```
{a + b > 0}
  x = a + b;
{x > 0}
```

B1
```
{true}
  x = y + 1;
{x > y}
```
vs.
B2
```
{true}
  x = y + 1;
{x == y + 1}
```

- **Frage:** Gibt es «bessere» und «beste» Aussagen bzw. Tripel?
  - Welches A-Tripel ist «besser»? Welches B-Tripel? Warum?
  - Was könnte eine allgemeine Definition von «besser» sein?
  - Besser für … wen eigentlich?

Stärkere und schwächere Aussagen

- Wir können *stärkere* und *schwächere* Aussagen unterscheiden
  - Wenn $P_1 \Longrightarrow P_2$ gilt, dann ist P_1 stärker als P_2 (und P_2 schwächer als P_1)
- Beispiele

A1
```
{a > 0 ∧ b > 0}
  x = a / b;
{x > 0}
```

vs.

A2
```
{a + b > 0}
  x = a + b;
{x > 0}
```

- $(a > 0 \land b > 0) \Longrightarrow (a + b > 0)$, d.h. A1 hat die stärkere Vorbedingung

B1
```
{true}
  x = y + 1;
{x > y}
```

vs.

B2
```
{true}
  x = y + 1;
{x == y + 1}
```

- $(x == y + 1) \Longrightarrow (x > y)$, d.h. B2 hat die stärkere Nachbedingung

Beste Tripel? Zwei Perspektiven

Perspektive 1: Sie möchten *existierenden Code nutzen*
(d.h. wir nehmen an, dass der Code «fixiert» ist)

```
{P?}
  code
{Q?}
```

- Je *schwächer die Vorbedingung*, desto öfter nutzbar (breiter einsetzbar) ist der Code
  - Beispiel: $\{a + b > 0\}$ erlaubt den Einsatz in mehr Situationen (ist weniger «anspruchsvoll») als $\{a > 0 \wedge b > 0\}$
- Je *stärker die Nachbedingung*, desto mehr Garantien bekommen Sie (desto mehr wissen Sie) als Code-Nutzer/in
  - Beispiel: Mit $\{x == y + 1\}$ wissen Sie, dass der Abstand zwischen x und y genau eins ist, anders als bei $\{x > y\}$

Beste Tripel? Zwei Perspektiven

Perspektive 2: Sie müssen Code *für jemanden* schreiben
(wir nehmen an, dass die Anforderungen/das Tripel «fixiert» sind)

```
{P}
  code?
{Q}
```

- *Stärkere Vorbedingung* ➜ mehr Garantien für Sie, d.h. Ihre Aufgabe wird einfacher
  - Intuitives Beispiel: «Entwickeln Sie eine Texterkennung für englische Buchstaben» vs. «… für Buchstaben»
- *Schwächere Nachbedingung* ➜ Potenziell mehr Wahlfreiheit, daher eventuell einfacher, für Sie
  - Intuitives Beispiel: «Endergebnis ist grösser als *n*» vs. «Endergebnis ist die nächste Primzahl grösser als *n*»

Automatisierung

- Es gibt (semi-)automatische* Algorithmen, um
  1. Die *schwächste Vorbedingung*, für gegebenen Code und Nachbedingung, zu berechnen
  2. Die *stärkste Nachbedingung*, für gegebenen Code und Vorbedingung, zu berechnen

```
{P?} code {Q}
```

```
{P} code {Q?}
```

- Zum Beispiel berechnet die kennengelernte Regel für Zuweisungen («ersetze Vorkommnisse der zugewiesenen Variablen …») die schwächste Vorbedingung

* Für relative einfache Programme; Problem letztendlich unentscheidbar (→ *Theoretische Informatik*), daher braucht es menschliche Hilfe. Details für EProg nicht relevant.