

# EINFACHE PROGRAMME ERSTELLEN

## Aufbau, Codierung und Erstellung einfacher Java-Programme

Struktur und Inhalt des Kurses wurden 2012 von Markus Heckner entwickelt. Im Anschluss haben Alexander Bazo und Christian Wolff Änderungen am Material vorgenommen. Die aktuellen Folien wurden von Alexander Bazo erstellt und können unter der [MIT-Lizenz](#) verwendet werden.

## RÜCKBLICK

- Beim Programmieren definieren wir Lösungsstrategien für Probleme (für einen Computer)
- Programmiersprachen dienen dabei als Vermittler zwischen unseren Ideen und den Kapazitäten des Rechners
- *Bouncer* ist ein Roboter, den Sie mit Hilfe der Programmiersprache *Java* steuern können
- Innerhalb eines Java-Programms dienen Methoden zur Gliederung des Programmablaufs und innerhalb einer Methode werden Befehle sequenziell abgearbeitet
- Auf den *Kontrol-*-bzw. *Programmfluss* können wir mit Schleifen und Bedingungen Einfluss nehmen
- Mit Kommentaren können wir unsere Programme für uns und andere ProgrammiererInnen beschreiben

# KURZES FEEDBACK VON IHNEN



<https://pingo.coactum.de/201662> 

# AKTUELLER SEMESTERFORTSCHRITT (WOCHE 2)

Kursabschnitt	Themen		
<b>Grundlagen</b>	Einführung	Einfache Programme erstellen	Variablen, Klassen & Objekte
	Kontrollstrukturen & Methoden	Arrays & komplexe Schleife	
<b>Klassenmodellierung</b>	Grundlagen der Klassenmodellierung	Vererbung & Sichtbarkeit	
<b>Interaktive Anwendungen</b>	Event-basierten Programmierung	String- & Textverarbeitung	
<b>Datenstrukturen</b>	Listen, Maps & die Collections	Speicherverwaltung	Umgang mit Dateien
<b>Software Engineering</b>	Planhaftes Vorgehen bei der Softwareentwicklung		
	Qualitätsaspekte von Quellcode		

# DAS PROGRAMM FÜR HEUTE

- Erkenntnisse aus der ersten Woche
- Hintergrundinformationen zur Programmiersprache Java
- Einfache Programme schreiben und ausführen
- Dekomposition als Problemlösungsstrategie

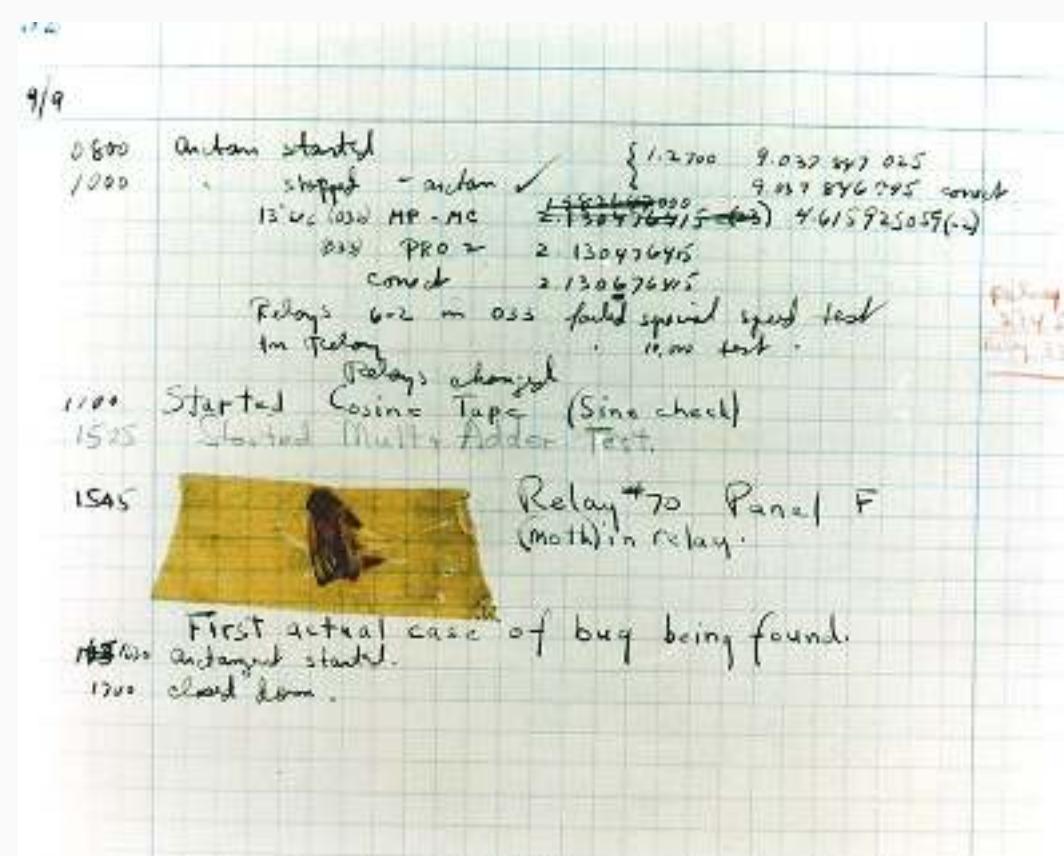
# ERKENNTNISSE AUS DER ERSTEN WOCHE

# BUGS (1/2)

*A software bug is an error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. The process of finding and fixing bugs is termed "debugging" and often uses formal techniques or tools to pinpoint bugs, and since the 1950s, some computer systems have been designed to also deter, detect or auto-correct various computer bugs during operations.*

Quelle: *Computer programming* (Wikipedia) 

## BUGS (2/2)



Bei *Bugs* handelt es sich um logische Fehler, die verhindern, dass Ihr Programm das gewünschte Verhalten zeigt. Sie sind zu unterscheiden von syntaktischen Fehlern, die auf falsche Nutzung der Sprachelemente zurück zuführen sind.

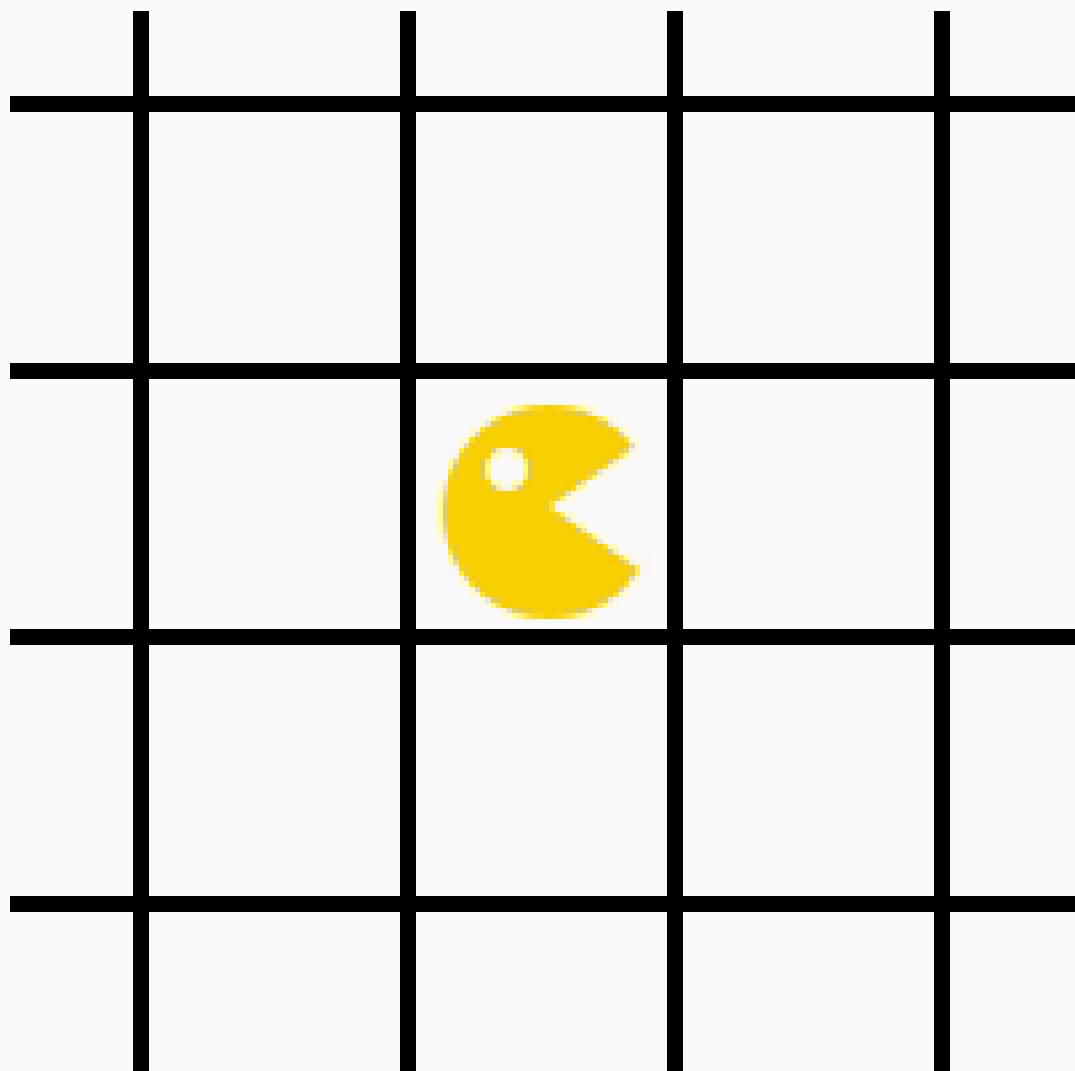
Bildquelle: *Der erste Bug. Dokumentiert am 9. September 1945 von von Grace Hopper, U.S. Naval Historical Center Online Library*  
Photograph NH 96566-KN ⚡

# HÄUFIGE FEHLER IN DER ERSTEN WOCHE (1/2)

Der vergessene Methodenaufruf: Teilschritte werden in einer separaten Methode programmiert, die entsprechende Methode wird aber nie verwendet. **Nur die bounce-Methode wird automatisch gestartet. Alle anderen Methoden müssen explizit dort - oder an anderer Stelle - aufgerufen werden.**

```
1 public class FirstRoom extends BouncerApp {  
2     @Override  
3     public void bounce() {  
4         loadLocalMap("FirstRoom");  
5     }  
6     // BUG: Diese Methode wird definiert, aber nicht innerhalb des Programms verwendet!  
7     private void doSomething() {  
8         bouncer.move();  
9         bouncer.move();  
10    }  
11 }  
12 }
```

# HÄUFIGE FEHLER IN DER ERSTEN WOCHE (2/2)



**Die Endlosschleife:** Schleifen müssen immer so konstruiert sein, dass eine direkte oder indirekte Abbruchbedingung gegeben ist, d.h. die Bedingung, die im Schleifenkopf definiert ist, muss absehbar eintreffen. Tritt die aufgestellte Bedingung nicht ein, wird der Schleifenkopf endlos oft wiederholt. Das führt in der Regel zur Blockade oder zum Absturz Ihres Programms:

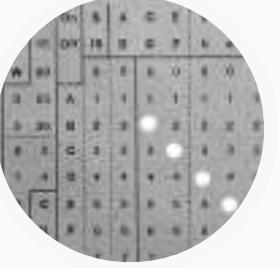
```
while(bouncer.canMoveLeft()) {  
    bouncer.turnLeft();  
}
```

# VOM QUELLCODE ZUM AUSFÜHRBAREN PROGRAMM

# PROGRAMMIEREN: GESTERN UND HEUTE



**1843:** Ada Lovelace schreibt das erste Programm für Babbages *Analytical Engine*



**~ 1900:** Mechanische Rechenmaschinen nutzen Lochkarten als Befehls- und Datenträger: Hollerith-Maschinen (Ursprung der Firma *IBM*)



**~ 1950:** Mit der Verbreitung der (Groß-) Rechner entstehen die ersten höheren Programmiersprachen



**~ 1965-70:** U.a. Margrit Hamilton und Friedrich Bauer prägen den Begriff *Software Engineering*

# ENTWURF VS. AUSFÜHRUNG

Moderne Programmiersprachen abstrahieren den Prozess der Entwicklung eines Programms **vollständig** von dessen späterer Ausführung

- Wir verfassen den **Quellcode** unserer Programme mit Hilfe einer Programmiersprache
- Spezielle Software (vor allem der *Compiler*) übersetzt unsere Befehle in eine Menge an Anweisungen, die unabhängig von der Programmiersprache, vom Rechner *interpretiert* werden können
- Zusammen mit allen weiteren benötigten Dateien, z.B. Grafiken oder verwendeten externen Befehlssätzen (*Bibliotheken*), wird bei dieser Transformation das eigentliche **Programm** erstellt
- Erst in diesem Zustand können wir unser Programm **ausführen**, d.h. starten

# VOM QUELLCODE ZUM AUSFÜHRBAREN PROGRAMM

Für die meisten Programmiersprachen wird im Kern ein ähnlicher Prozess für den beschriebenen Transformationsprozess verwendet:

1. Ausgangspunkt ist der Quellcode
2. Der *Compiler* erstellt eine - plattformabhängige - ausführbare Datei
3. Das Resultat kann *geöffnet* bzw. gestartet werden

```
/**  
 * Simple HelloButton() method.  
 * @version 1.0  
 * @author john doe <doe.j@example.com>  
 */  
HelloButton()  
{  
    JButton hello = new JButton( "Hello"  
    hello.addActionListener( new HelloB  
    // use the JFrame type until support  
    // new component is finished  
    JFrame frame = new JFrame( "Hello Bu  
    Container pane = frame.getContentPane()  
    pane.add( hello );  
    frame.pack();  
    frame.show();           // display t  
}
```

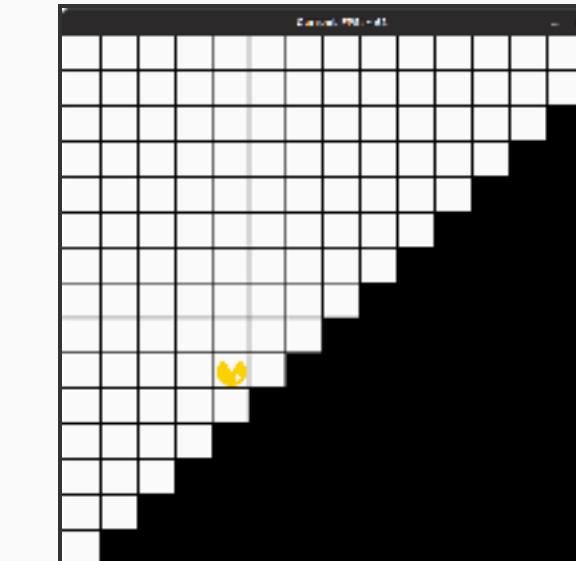


# VOM JAVA-QUELLCODE ZUM AUSFÜHRBAREN PROGRAMM

Java-Programme sind an sich plattformunabhängig und können theoretisch auf unterschiedlichen Betriebssystemen ausgeführt werden. Dadurch wird der Transformationsprozess komplexer:

1. Ausgangspunkt ist wieder der Quellcode
2. Der *Compiler* erstellt eine - plattformunabhängige - Version des Codes, den *Bytecode*
3. Das erstellte Programm kann auf jedem System ausgeführt werden, auf dem eine *Laufzeitumgebung* für Java installiert ist

```
/**  
 * Simple HelloButton() method.  
 * @version 1.0  
 * @author john doe <doe.j@example.com>  
 */  
HelloButton()  
{  
    JButton hello = new JButton( "Hello"  
    hello.addActionListener( new HelloBt  
    // use the JFrame type until support  
    // new component is finished  
    JFrame frame = new JFrame( "Hello Bu  
    Container pane = frame.getContentPane()  
    pane.add( hello );  
    frame.pack();  
    frame.show();           // display  
}
```



# GRUNDSÄTZLICHES ZUR PROGRAMMIERSPRACHE JAVA

- Die Programmiersprache Java wurde Anfang der 1990er-Jahre entworfen und wird seit Mitte der 90er-Jahre von der Firma *Sun* bzw. *Oracle* standardisiert
- Java, bzw. Programme, die in Java geschrieben sind, sind plattformunabhängig: *Write once, run everywhere*
- Java-Programme laufen nicht *direkt*, sondern erfordern eine Zwischenschicht (*Virtual Machine*): Die *Java Runtime*
- Solche *Runtime Environments* gibt es für zahlreiche Plattformen: Windows, Mac, Linux, Android(!), ...

# HELLO WORLD

*A "Hello, World!" program generally is a computer program that outputs or displays the message "Hello, World!". Such a program is very simple in most programming languages, and is often used to illustrate the basic syntax of a programming language. It is often the first program written by people learning to code.*

Quelle: *Computer programming* (Wikipedia) 

# HELLO JAVA

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

## WICHTIGE DETAILS ZUM HELLO WORLD-PROGRAMM (1/2)

Unsere *Bouncer*-Programme sind **spezielle** Java-Programme, für **allgemeine** Programme gelten einige andere/zusätzliche *Regeln*:

- Jedes Java-Progam muss über eine `main`- Methode verfügen, die beim Starten des Programm automatisch aufgerufen wird (Vgl. Bouncer: Die `main`-Methode ist vor Ihnen *versteckt*, ist aber vorhanden und sorgt u.a. für den automatischen Aufruf der `bounce`-Methode)
- Die *Signatur* (`public static void main(String[] args)`) der `main`-Methode muss exakt wie beschrieben aussehen
- Ihr Programm *startet* in der ersten Zeile der `main`-Methode

## WICHTIGE DETAILS ZUM HELLO WORLD-PROGRAMM (1/2)

- Die Programmiersprache Java beinhaltet (ähnlich wie *Bouncer*) eine Menge an eingebauten Befehlen, die Sie in Ihren Programmen verwenden können
- Der Befehl `System.out.println()` gibt z.B. den Text, den Sie in den Anführungszeichen übergeben, aus
- Ausgaben werden in der Regel auf der aktuellen *Standardausgabe* Ihres Betriebssystem ausgegeben
- Wir können unsere Programme direkt in IntelliJ starten. Um dies auch außerhalb der Entwicklungsumgebung zu ermöglichen, sind aber weitere Schritte notwendig

# HELLO WORLD 2.0

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4         if(1 == 1) {  
5             System.out.println("Math still works. Let's start ...");  
6         }  
7         for(int i=0; i < 42; i++) {  
8             System.out.println("Hello World");  
9         }  
10    }  
11  
12 }
```

Die Schleifen und Bedingungen, die Sie beim Programmieren von *Bouncer* verwenden, sind *universell* auf andere JAVA-Programme übertragbar. Das gleiche gilt für alle anderen Syntax- und Sprachelemente, die wir im Laufe des Semesters kennen lernen werden. Spezielle Befehle, z.B. `bouncer.move()` sind nur innerhalb der *Bouncer*-Programme verfügbar.

# DEKOMPOSITION ALS GRUNDLEGENDE PROBLEMLÖSUNGSSTRATEGIE

# PROGRAMMIERPROBLEME LÖSEN

- Im Laufe des Semesters lösen Sie immer komplexer werdende Problemstellungen
- Auch wenn sich diese Probleme inhaltlich unterscheiden, können wir häufige ähnliche Strategien für die Problemanalyse und die Implementierung der jeweiligen Lösung verwenden
- Für alle Aufgaben gilt: Wir versuchen das zugrundeliegende Problem vollständig zu verstehen, bevor wir eine Lösungsstrategie entwerfen und implementieren
- Dabei können wir u.a. das Prinzip der *Dekomposition* (engl. *Decomposition*) anwenden

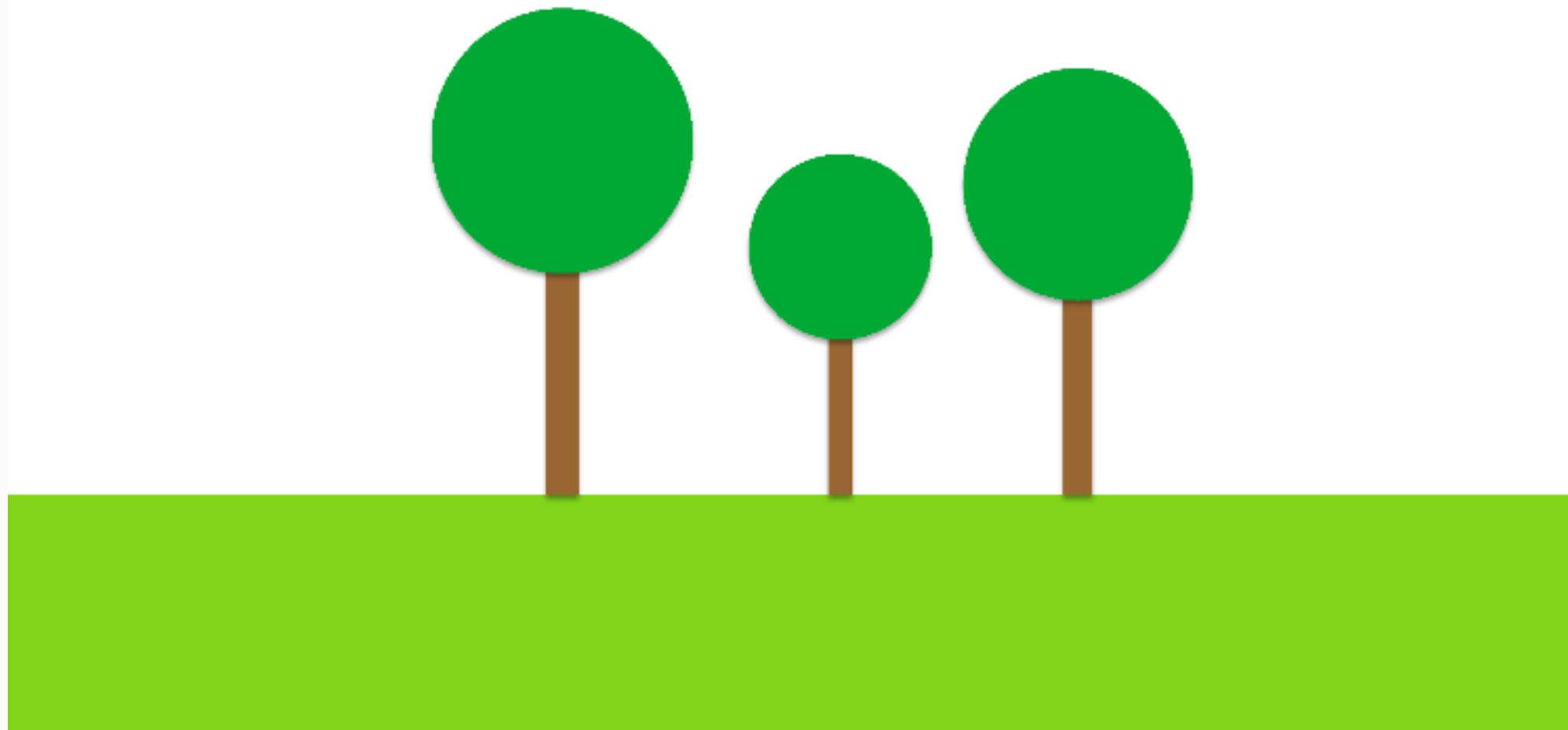
# DEKOMPOSITION

*Decomposition in computer science, also known as factoring, is breaking a complex problem or system into parts that are easier to conceive, understand, program, and maintain.*

Quelle: *Decomposition (computer science)* (Wikipedia) 

Dekomposition bezeichnet sowohl eine Strategie für die vorangehende Problemanalyse als auch für den Vorgang, gut strukturierten und aufgeteilten Code (Vgl. *Methoden*) zur Umsetzung der gefundenen Lösung zu schreiben. Die Ursprünge für diese *Philosophie* lassen sich im Paradigma der **Strukturierten Programmierung**  finden.

# BEISPIEL: DER BAU EINES HAUSES (1/6)



# BEISPIEL: DER BAU EINES HAUSES (2/6)

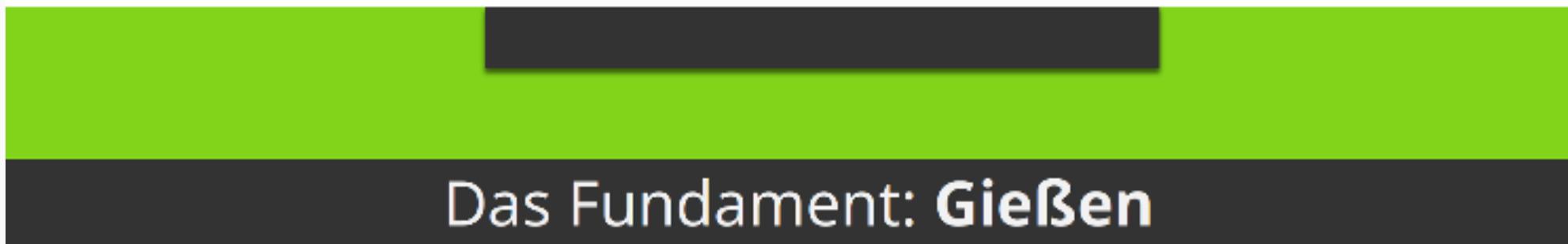


Das Fundament: **Planieren**

# BEISPIEL: DER BAU EINES HAUSES (3/6)



# BEISPIEL: DER BAU EINES HAUSES (4/6)



# BEISPIEL: DER BAU EINES HAUSES (5/6)



# BEISPIEL: DER BAU EINES HAUSES (6/6)



# DEKOMPOSITION UND *TOP DOWN*

In der Regel wird bei der Analyse eines Problems *von oben nach unten (top down)* gearbeitet. Ausgehend vom vollständigen Problem, spezifizieren wir immer detailliertere und spezialisiertere Einzelaufgaben:

- Hauptaufgabe: **Haus bauen**
- Unteraufgaben:
  - **Fundament erstellen** mit Teilaufgaben *Planieren, Ausheben und Gießen*
  - **Gebäude bauen** mit Teilaufgaben *Mauern bauen und Dach decken*

Beim Umsetzen der Lösung kann dann auch *von unten nach oben (bottom up)* gearbeitet werden.

## DEKOMPOSITION AM BEISPIELE BOUNCER

Schon in der letzten Woche haben wir dieses Prinzip angewendet, um Aufgaben wie den Hürdenlauf oder das Erklimmen der Treppe zu lösen!

```
1 public void bounce() {  
2     loadMap("Stairs");  
3     climbStairs();  
4 }  
5 private void climbStairs() {  
6     bouncer.turnLeft();  
7     while(bouncer.canMoveForward()) {  
8         climbOneStair();  
9         bouncer.turnLeft();  
10    }  
11    turnRight();  
12 }  
13 private void climbOneStair() {  
14     bouncer.move();  
15     turnRight();  
16     bouncer.move();  
17 }  
18 private void turnRight() {  
19     bouncer.turnLeft();  
20 }
```

# ZUSAMMENFASSUNG

- Java ist eine plattformunabhängig Programmiersprache: Zum Ausführen von Java-Programmen wird eine plattformspezifische *Virtual Machine* benötigt, die Teil der Java-Laufzeitumgebung ist
- Programme unterscheiden sich in der Entwicklungsphase (*Quellcode*) und der Ausführungsphase (*Binary* oder *intermediate Code*)
- Klassen sind das zentrale Strukturelement eines Java-Programms und bei Programmstart, wird immer die essentielle `main`-Methode ausgeführt
- *Decomposition* und die *Top Down*-Analyse sind zentrale Konzepte, um Programmierprobleme strukturiert zu lösen und übersichtlichen und wartbaren Code zu erstellen

## VIELEN DANK FÜR IHRE AUFMERKSAMKEIT. WENN SIE MÖCHTEN, SEHEN WIR UNS IM ANSCHLUSS IN DER ZENTRALÜBUNG!

Fragen oder Probleme? In allgemeinen Angelegenheiten und bei Fragen zur Vorlesung wenden Sie sich bitte an Alexander Bazo ([alexander.bazo@ur.de](mailto:alexander.bazo@ur.de)). Bei organisatorischen Fragen zu den Studienleistungen und Übungsgruppen schreiben Sie bitte Florin Schwappach ([florin.schwappach@ur.de](mailto:florin.schwappach@ur.de)). Bei inhaltlichen Fragen zu den Übungen, Beispielen und Studienleistungen schreiben Sie uns unter [mioop@mailman.uni-regensburg.de](mailto:mioop@mailman.uni-regensburg.de).

## QUELLEN

Eric S. Roberts, *The art and science of Java: an introduction to computer science, New international Edition*, 1. Ausgabe, Pearson, Harlow, UK, 2014.