Programmieren in Java

Einführung in die (imperative) Programmierung

wissen.leben
WWU Münster

Name: Patrick Förster, Michael Hasseler

Zentrum für
Informationsverarbeitung

Wiederholung

- Wozu Programmiersprachen?
- Maschinencode→Assembler→Compiler→"Höhere Programmiersprachen"

- Syntax: Programmiersprachen gehorchen einer Grammatik
  - Nur grammatikalisch richtige Sätze können „compiliert" werden
  - Jeder Satz kann in sogenannte „Tokens" eingeteilt werden
    - Bezeichner
    - Literale
    - Schlüsselwörter
    - Separatoren
    - Operatoren

wissen.leben
WWU Münster

Wiederholung (Tokens in Java)

- Bezeichner:

```
ZEICHEN=A|...|Z |a|...|z|_|$
ZIFFER=0|...|9
BEZEICHNER=<ZEICHEN>,{ZEICHEN|ZIFFER}
```

```
myVariable
my_variable
myVariable_2
```

~~2_variable~~
~~!&variable~~
~~variable!_2~~

- Literale:
  - Boolesch: **true**, **false**
  - Ganzzahlig: **1, 27, 37, 1000**
  - Gleitkomma: **3.14159265359, 2.71828182846**
  - Zeichen: **'A', '0'**
  - Zeichenketten: **"Hello World"**

- Separatoren:

;	Satzende	{…}	Code-Block
,	Aufzählung	()	Klammerung (Operatoren, Methoden-Signatur, Methoden-Aufruf)

wissen.leben
WWU Münster

Wiederholung (Tokens in Java II)

- Schlüsselwörter:

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

const goto

wissen.leben WWU Münster

Wiederholung (Tokens in Java III)

- Operatoren:

=	Zuweisung	!	Negation	&&	Logisch-Und	|	Bit-Oder (inklusiv)
+	Addition	==	Vergleich	||	Logisch-Oder	^	Bit-Oder (exklusiv)
-	Subtraktion	!=	Nicht-Gleich	?:	Vergleich		
*	Multiplikation	>	Größer	~	Bit-Komplement		
/	Division	<	Kleiner	<<	Shift-Links		
%	Modulo	>=	Größer-Gleich	>>	Shift-Rechts		
++	Inkrement	<=	Kleiner-Gleich	>>>	" (unsigned)		
--	Dekrement	instanceof	„Typ-Check“	&	Bit-Und		

- Jeder Operator hat eine Priorität (vgl. „Punkt vor Strich“-Rechnung)

wissen.leben
WWU Münster

Wiederholung (Code-Rahmen)

- Java-Code:

```
class Program {

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

  Methoden/Funktionen
}
```

  - Der „**class**-Rahmen“ muss vorerst hingenommen werden
    - Der Name der Klasse ist allerdings frei-wählbar
  - Ebenso hinzunehmen: Methode **main** und deren Signatur
  - „imperative Progammierung“ findet in den blauen Bereichen statt

- Der Code muss sich in einer **Datei** befinden, die den **Namen** der **Klasse** hat

Name: Patrick Förster, Michael Hasseler

Datentypen

- Programme basieren immer auf der Manipulation von bestimmten *Daten*
- Daten lassen sich kategorisieren
- Ein *Datentyp* beschreibt eine Menge von Daten der gleichen Art
- Im Code-Bespiel (Folie 6) sind **`String`** und **`void`** Datentypen
- Auch **`Program`** kann man – wie das Kapitel OOP zeigen wird – als Datentyp bezeichnen

- *Werte* sind Elemente eines eindeutigen Datentyps
- In `27 * 37 + 1` sind **`27, 37`** und **`1`** Werte des Datentyps **`int`**

- Jede Programmiersprachen unterscheidet zumindest zwischen:
  - Primitiven Datentypen
  - Zusammengesetzten Datentypen

wissen.leben
WWU Münster

Primitive Datentypen

- Als „primitiv“ wird ein Datentyp bezeichnet, dessen Werte nicht weiter zerlegt werden können
- Die meisten Sprachen bieten Datentypen zur Beschreibung numerischer Werte (am Beispiel von Java)
  - **byte**: {-128...127} (8 Bit)
  - **short**: {- 32768... 32767} (16 Bit)
  - **int**: {-2.147.483.648... 2.147.483.647} (32 Bit)
  - **long**: {-9223372036854775808... 9223372036854775807} (64 Bit)
  - **float**: {-1,4E-45 ... 3,4E+38} (32 Bit)
  - **double**: {-4,9E-324 ... -1,7E+308} (64 Bit}
- In Java können **byte**, **short** und **int** Literale nicht explizit von einander unterschieden werden
- Werte aus dem Bereich **long** werden mit einem „L“ notiert: **999L**
- Werte aus dem Bereich **float** werden mit einem „f“ notiert: **0.5f**

wissen.leben
WWU Münster

Primitive Datentypen (2)

- Weitere primitive Datentypen:
  - **`boolean`**: {**`true, false`**}
  - **`char`**: Unicode-Zeichen (16 Bit)
  - **`String`**: „Buchstabenketten"
- Achtung: Nicht all diese Typen tauchen zwangsläufig in jeder Sprache auf
  - **`boolean`** wird häufig einfach als {0, 1} betrachtet
  - Unicode-Zeichen können auch einfach als numerischer Wert dargestellt werden (in Java ist ‚c' bspw. gleich 99)
- Nicht all diese Type müssen zwangsläufig „primitiv" sein
  - Java kennt bspw. zwar den Datentyp **`String`**, dieser ist allerdings nicht primitiv (mehr später)
- In vielen Sprachen können eigene Teilmengen von **`int`** als neue primitive Datentypen definiert werden (sogenannte: **`enum`**)
  - Java kennt **`enums`**, allerdings sind auch diese wieder nicht primitiv

wissen.leben
WWU Münster

Variablen

- Bisher wurden Daten als Literale angegeben
- *Variablen* können verschiedene Werte zu verschiedenen Zeitpunkten annehmen

```
int n1;
```

→Variable namens **n1**, die Werte des Datentyps **int** aufnehmen kann

- Durch den **=** Operator wird der Variablen ein Wert zugewiesen

```
n1 = 1000;
```

- Definition und erste Zuweisung (*Initialwert*) können kombiniert werden:

```
int n1 = 1000;
```

- Der Wert muss kein *Literal* sein, sondern kann auch das Ergebnis eines Ausdrucks, Methodenaufrufes oder der Wert einer anderen Variablen sein

```
int n1 = 27;
int n2 = 37;
int result = n1 * n2 + 1;
```

wissen.leben
WWU Münster

Typsystem (statisch)

- Java ist *statisch* typisiert:

Wenn eine Variable für Werte eines bestimmten Typs deklariert wurde, kann sie im weiteren Programmverlauf keinen Wert eines anderen Typs annehmen

```
int n1 = 27;
n1 = 1000;  // Ok!
n1 = "Hello World"; // Fehler!
```

- Nur die erste Zuweisung (Deklaration) muss den Typ definieren
- Jeder weitere Zuweisung überschreibt den alten Wert
- Das Literal **1000** ist vom Typ **int**, die zweite Zuweisung ist demnach ok
- **"Hello World"** ist vom Typ **String**, die Zuweisung schlägt also fehl
- Ein so geformtes Programm würde nicht mal compilierbar sein
- Den Typ eines Literals erkennt der Compiler implizit an ihrer Darstellung

Java ist statisch typisiert!

wissen.leben WWU Münster

Typsystem (dynamisch)

- Das Gegenteil zu einem statischen Typsystem ist ein *dynamisches*

> Jede Variable kann für Werte eines beliebigen Typs eingesetzt werden. Der Typ der Variablen ergibt sich zur Laufzeit aus seiner letzten Zuweisung.

- Der Zuweisungsoperator benötigt in diesem Fall keine Angabe eines Typs
- Das vorherige Beispiel wäre in dynamisch typisierten Sprachen lauffähig
- Beispiele sind: JavaScript, PHP, Phyton
- Nachteil solcher Sprachen ist häufig das später Erkennen von Typfehlern:

```
temp = 27;
```

```
temp = "test";
```

```
var2 = 37 * temp;
```

- Angenommen **`temp`** wird mit dem **`int`**-Wert **`27`** initialisiert, die Zuweisung an **`var2`** wäre dann kein Problem, da der Operator **`*`** für **`int`** definiert ist
- Findet allerdings zu irgendeinem Zeitpunkt zuvor eine zweite Zuweisung bspw. mit `"test"` statt, wird die **`*`**-Operation fehlschlagen
- Statisch würde ein Compiler dies bereits vor der Ausführung bemängeln!

wissen.leben WWU Münster

Coercion (Java)

- Trotz des statischen Typsystems verbietet Java nicht alles:

```
int iValue = 37;
long lValue = iValue;
```

```
float fValue = 37.0f;
double dValue = fValue;
```

```
long lValue = 37L;
double dValue = lValue;
```

- Der Compiler merkt selbständig, dass die zweite Zuweisung möglich ist, da der Wert der rechten Seite zu einem Typ auswertet, der Teilmenge des Typs der linken Seite ist
- Diese Umwandlung wird „*implizite Typanpassung*" bzw. *Coercion* genannt

Typ	Mögliche implizite Umwandlung
byte	short, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Cast (Java)

- Die Rückrichtung ist in statischen Typsystemen meist nicht erlaubt:

```
long lValue = 37L;
int iValue = lValue;
```

- Ein Compiler wird bemängeln, dass er nicht automatisch von **long** zu **int** umwandeln kann bzw. will, denn:
  - Die Umwandlung könnte mit einem Genauigkeitsverlust verbunden sein
  - Was ist der Wert von **9223372036854775797L** im **int**-Wertebereich?

- Man kann den Compiler zwingen eine Umwandlung durchzuführen
- Diese Art der *expliziten Umwandlung* wird *Cast* genannt:

```
long lValue = 37L;
int iValue = (int)(lValue);
```

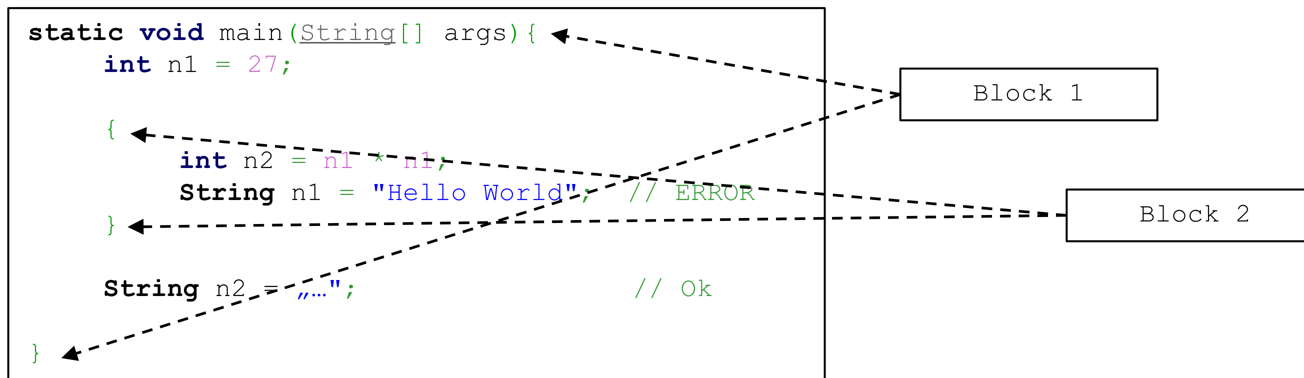
```
float fValue = 37.0f;
char cValue = (char)fValue;
```

- Ein Cast ist nur anwendbar, wenn die beteiligen Typen kompatible sind!

wissen.leben
WWU Münster

Scope

- Jede Variable hat einen *Gültigkeitsbereich* (Scope)
- Nur in diesem Bereich ist die Variable nutzbar
- In statisch typisierten Sprachen kann in diesem Bereich keine weitere Variable mit gleichem Namen definiert werden
- In Java bilden **{ }**-Blöcke Gültigkeitsbereiche
- Diese können geschachtelt werden:



- Einige Programmiersprache kennen „globale“ Variablen

wissen.leben
WWU Münster

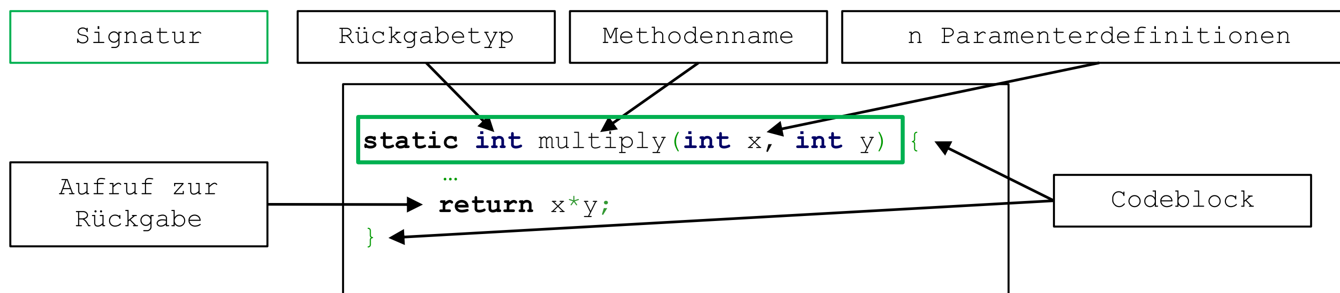
Funktionen

- Angenommen der * Operator wäre nicht definiert
- Durch Reduktion auf + wäre es möglich eine Multiplikation abzubilden
- Der nötige Code müsste für jede weitere Multiplikation wiederholt werden
- Besser: Programmeinheit bereitstellen, die je nach Bedarf mit verschiedenen Werten aufgerufen werden kann
- Eine solche Programmeinheit wird als *Funktion* bezeichnet
- Ihren „Inhalt" bezeichnet man als *Algorithmus*
- Eine Funktion wird allgemein durch einen Namen und eine Menge an Parametern definiert (diese Menge kann auch leer sein)
- In statisch typisierten Sprachen müssen die Parameter typisiert und ein Rückgabetyp angeben werden
- Der Aufruf einer Funktion ist ein Ausdruck und kann damit auf der rechten Seite einer Zuweisung stehen

wissen.leben WWU Münster

Methoden (Java)

- In Java gibt es keine Funktionen als autarkes Sprachkonzeptes
- Funktionen treten als *Methoden* im Zusammenspiel mit einer Klasse auf
- Als imperatives Mittel werden sie hier vorweg besprochen:
  - Methoden werden im Klassenblock definiert
  - Definitionen könnten *nicht* geschachtelt werden (Methode in Methode)



- Das Schlüsselwort **`static`** muss hier vorerst als gegeben hingenommen werden und bedeutet soviel wie „Zur umschließenden Klasse gehörig“
- Der Methodenaufruf erfolgt nach gewohnt mathematischer Notation:

```
multiply(27, 37)
```

Methoden (Java)

- Es darf (in einer Klasse) keine zwei Methoden mit der selben Signatur geben
- Im Beispiel hat die Methode den Rückgabewert vom *Typ* **`int`**
- Das **`return`**-Schlüsselwort gibt an, welchen *Wert* die Rückgabe hat

- Wir sind bereits einer Methode begegnet:

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

- Das Schlüsselwort **`public`** wird später (OOP) besprochen und bedeutet so viel wie „von außen ausführbar"

- Spezieller Datentyp **`void`**:
  - Methoden müssen nicht zwangsläufig eine Rückgabe liefern
  - Der Datentyp **`void`** steht für „Kein Wert"
  - Wird eine Methode mit **`void`** als Rückgabe definiert, muss im Code kein **`return`** deklarieren werden (ein leeres „**`return ;`**" ist allerdings erlaubt)
  - **`void`** kann nicht als Typ einer Variablen deklariert werden
  - Was bedeutet das für **`void`**-Methoden?

wissen.leben
WWU Münster

Kontrollstrukturen

- Zurück zum *-Operator: Theoretisch kann man nun die Codeeinheit zur Berechnung einer Multiplikation in eine Methode auslagern
- Man stößt aber direkt auf das nächste Problem: Die Berechnung ist abhängig vom Multiplikator:

$$a * b = \underbrace{b + b + \ldots + b}_{a} = \sum_{i=1}^{a} b$$

- Bisher wurde allerdings keine Möglichkeit vorgestellt, eine solche Abhängigkeit universell umzusetzen
- Jede Sprache mit einem imperativen Kern stellt sogenannte Kontrollstrukturen zur Verfügung um den Programmfluss in Abhängigkeit von bestimmten Faktoren zu steuern:
    - Bedingte Anweisungen
    - Fallunterscheidung
    - Schleifen
    - (Ausnahme-Behandlung)

wissen.leben
WWU Münster

Bedingte Anweisungen

- Mit der **if**-Anweisung können Code-Fragmente in Abhängigkeit von einer Bedingung zur Ausführung gebracht werden
- Die *Bedingung* muss dabei ein Ausdruck sein, der zum Typ **boolean** auswertet
- Liefert dieser Ausdruck **true** so wird der erste Code-Block ausgeführt, ansonsten der **else**-Teil
- Falls ein Code-Block lediglich einen „Satz" enthält, können die **{**-Klammern weggelassen werden
- Angenommen die Methode **random** liefert einen zufälligen **int**-Wert
- Der Operator **%** liefert den ganzzähligen Rest der Division **random/2** (*modulo*)
- Der Operator **==** vergleicht linke und rechte Seite und liefert **true**/**false**
- Das gesamte Code-Fragmente gibt „Gerade" aus, falls **number** durch zwei teilbar ist, ansonsten „Ungerade"

```
if ( Bedingung ){
    true: Code
}
else {
    false: Code
}
```

```
int number = random();
if (number % 2 == 0) {
    println("Gerade");
}
else {
    println("Ungerade");
}
```

wissen.leben WWU Münster

Boolesche Operatoren

- Im Beispiel wurde bereits der Operator == verwendet, der **true** liefert, falls linker und rechter Operant übereinstimmen, ansonsten **false**
- Weiter wichtige boolesche Operatoren inkl. Wahrheitstabelle:

Und

&	*true*	*false*
true	true	false
false	false	false

Oder

|	*true*	*false*
true	true	true
false	true	false

Ausschließendes Oder(xor)

^	*true*	*false*
true	false	true
false	true	false

Negation

!	
true	false
false	true

- Für **&** bzw. **|** bietet Java die Möglichkeit der Kurzschluss-Semantik an:
  - Wenn der erste Operant für *oder* **true** bzw. für *und* **false** liefert, muss der zweite nicht mehr ausgewertet werden
  - Kurzschluss-Semantik wird mit doppelten oder/und (**&&**/**||**) notiert

```
if (number % 2 == 0 && number > 100) …
```

  - In der Regel sind die Kurzschluss-Operatoren vorzuziehen

wissen.leben WWU Münster

Fallunterscheidung

- Die Kontrollstruktur **`switch`** wird verwendet, um per Fallunterscheidung zu einem bestimmten Code-Fragment zu springen
- Der Ausdruck muss vom Typ **`int/String`** sein
- Es können n-viele Fälle (**`case`**) angegeben werden
- Der hinter **`case`** definierte Ausdruck muss ein *Literal* und ebenfalls vom Typ **`int/String`** sein
- Zur Laufzeit wird das Programm genau an der Stelle fortgesetzt, dessen konstante Fallbeschreibung zur Auswertung des Ausdrucks passt
- Passt kein Fall, wird zum (optionalen) **`default`**-*Fall* gesprungen

```
switch ( Ausdruck ) {
        case Literal 1: {
            Code 1
        }
        ...
        case Literal n: {
            Code n
        }
        default: {
            Code default
        }
}
```

- **Achtung**: Das Programm wird nach dem Sprung zum entsprechenden Fall ebenfalls alle anderen *nachfolgenden* **`case`**-Fragmente ausführen!

wissen.leben
WWU Münster

Break-Anweisung

- In der Regel möchte man die Ausführung eines eine **switch**-Anweiungs auf die Ausführung des passenden **case**-Fragmentes einschränken
- Mit dem Schlüsselwort **break** kann die Ausführung der **switch**-Anweisung abgebrochen werden:

```
int number = random();
switch (number % 3) {
    case 0:
        println("Fall 0");
        break;
    case 1:
        println("Fall 1");
        break;
    case 2:
        println("Fall 2");
    default:
        println("Fall unbekannt");
}
```

number % 3 == 0

```
Fall 0
```

number % 3 == 1

```
Fall 1
```

number % 3 == 2

```
Fall 2
Fall unbekannt
```

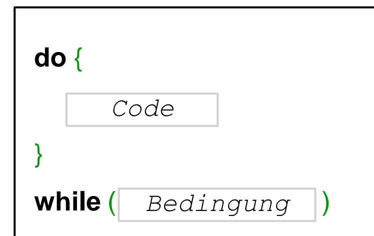
- Theoretisch könnte man jede **switch**-Anweisung durch eine entsprechend geschachtelte **if-else**-Anweisung übersetzten

wissen.leben
WWU Münster

Schleifen

- Mit **switch** und **if** können bedingte Code-Fragmente definiert werden
- Für das Eingangsbeispiel der Multiplikation benötigt man noch eine Möglichkeit Anweisung zu wiederholen
- Die dafür nötigen Kontrollstrukturen sind Schleifen, die bedingt lange über einen Code-Block iterieren

While-Schleife



Do-Schleife

- Die Bedingung muss in jedem Fall zu **boolean** auswerten
- Solange diese nicht **false** liefert, wird der Code wiederholt ausgeführt
- Achtung: Bei der **while**-Schleife wird die Bedingung vor der ersten Ausführung geprüft wird, während bei der **do**-Schleife das Code-Fragment in jedem Fall einmal ausgeführt wird

wissen.leben
WWU Münster

For-Schleife

- Neben der **while**- und der **do**-Schleife existiert noch die **for**-Schleife
- Sowohl *Zuweisung*, *Bedinung* als auch *Anweisung* sind optional

```
for ( Zuweisung ; Bedingung ; Anweisung )
    Code
}
```

- In der Regel wird die **for**-Schleife dazu genutzt, über den Zustand einer in *Zuweisung* initialisierten Start-Variable zu iterieren
- Dabei wird der Zustand der Variablen in *Anweisung* aktualisiert
- *Bedingung* dient wie schon bei **do** und **while** als Abbruchbedingung

```
for (int i = 0; i <= 10; i++) {
        println(i);
}
```

- Hier wird die **int**-Variable **i** mit **0** initialisiert
- Der Operator **++** erhöht **i** in jedem Schritt um
- Die Schleife läuft solange weiter, wie **i** kleiner-gleich zehn ist und gibt somit die Zahlen von **0** bis **10** aus

Rekursive Methoden

- Rekursive Methoden sind solche, die sich selbst aufrufen oder im Laufe ihrer Ausführung durch andere erneut aufgerufen werden
- In vielen funktionalen Programmiersprachen sind rekursive Methode das Mittel zur Iteration
- Beispiel: *Fibonacci-Zahlen: {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...}*

$$fib(n) = \begin{cases} 0 & falls\ n = 0 \\ 1 & falls\ n = 1 \\ fib(n-1) + fib(n-2) & sonst \end{cases}$$

```
static int fib(int n) {
      switch (n) {
            case 0: return 0;
            case 1: return 1;

            default: return fib(n-1) + fib(n-2);
      }
}
```

- Die Methode **`fib(n)`** ruft für den Fall **`n > 1`** zwei mal sich selbst auf
- Bei jedem rekursiven Aufruf wird das Argument **`n`** verringert
- Die Bedingungen **`case 0`** und **`case 1`** garantieren dabei, dass die Selbstaufrufe nicht endlos lang durchgeführt werden (für `n>=0`)

wissen.leben WWU Münster

Rekursive Methoden

- Was passiert bei einem solchen Aufruf (am Beispiel: **`fib(5)`**)

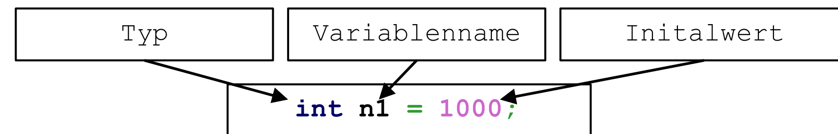
```
fib(4) + fib(3)
(fib(3)+fib(2))+(fib(2)+fib(1))
(((fib(2)+fib(1))+(fib(1)+fib(0))) + (((fib(1)+fib(0))+1)
(((fib(1)+fib(0))+1)+(1+0)) + ((1+0)+1)
((1+0)+1)+1) + (1+1)
(1+1)+1) + 2
(2 + 1) + 2
3 + 2
5
```

- Bevor der erste Aufruf aufgelöst werden kann (**`fib(0 bzw. 1)`**) wird ein Baum an rekursiven Aufrufen aufgebaut (Rekursionstiefe)
- Solange eine Rekursion nicht aufgelöst wurde, muss der Status jedes Aufrufes im Speicher erhalten bleiben
- Eine solche Rekursion ist demnach ein Speicherfresser (ausprobieren!)
- Man spricht von einer „endrekursiven“ Methode, wenn der letzte Aufruf eine Methode auch der letzte Berechnungsschritt ist
- Endrekursive Methoden können in iterative umgewandelt werden (z.B. durch eine Schleife)

wissen.leben
WWU Münster

Wiederholung

- Datentypen:
  - Ein Datentyp bezeichnet eine Menge von Daten der gleichen Art
  - Werte „primitiver Datentypen“ können nicht weiter zerlegt werden
  - Primitive Datentypen in Java
    - Ganzzahlen: **`byte, short, int, long`**
    - Gleitzahlen: **`float, double`**
    - Außerdem: **`boolean, char`**



- Variablen:
  - Platzhalter für Werte
  - Zu einem bestimmten Zeitpunkt hat jede Variable einen bestimmten Typ
  - Java ist „*statisch*“ typisiert

    ➔ Der Typ einer Variablen kann in ihrem Gültigkeitsbereich nach der Definition nicht mehr geändert werden!

wissen.leben
WWU Münster

Wiederholung

- Typumwandlung:
  - Coercion: „*implizite Typumwandlung*“ (durch den Compiler)
  - Cast: „*explizite Typumwandlung*“ (durch den Programmierer)

- Methoden/Funktionen
  - Dienen zur Kapselung eines Algorithmus
  - Aufruf kann auf n-verschiedenen Parametern basieren
  - Methode liefert Ergebnis eines bestimmten Typs
    - „Ergebnislose“ Methoden durch den speziellen Datentyp **`Void`**

- Kontrollstrukturen:
  - **`If`**-Anweisung
  - **`Switch`**-Anweisung
  - Schleifen: **`Do, While, Repeat`**

wissen.leben WWU Münster

Aufgaben

I. Bringen Sie folgenden Klassiker nach Anleitung zur Ausführung:

```
class Program {
   public static void main(String[] args){
      System.out.println("Hello World");
   }
}
```

1. Man legt sich in einem beliebigen Verzeichnis die Datei „*Program.java*“ an und kopiert den Beispiel-Code in diese
2. Start der Eingabeaufforderung (Windows: Start→Zubehör→ Eingabeaufforderung)
3. Hier wechselt man in das eben angelegt Verzeichnis („cd …“)
4. Der Java-Compiler aus dem JDK hört auf dem Namen „javac“:

```
javac Program.java
```

5. Ein Blick in das Verzeichnis („dir“) verrät, dass eine neue Datei namens „Program.class“ angelegt wurde

wissen.leben
WWU Münster

Aufgaben

6. Diese lässt sich nun auf der virtuellen Maschine ausführen:

```
java Program
```

7. Der Aufruf liefert das ersehnte:

```
Hello World
```

II. Schreiben Sie eine Methode **`mult`** zur Multiplikation zwei ganzzahliger Zahlen (**`int`**) ohne den **`mult`**-Operator

III. Schreiben Sie eine Methode **`modulo`**, die den ganzzahligen Rest eine Division auf positiven, ganzen Zahlen größer 0 (**`int`**) liefert

- Die Eingaben sollten überprüft werden und bei einer fehlerhaften Eingabe -1 zurückliefern

JDK

- Oracel bietet unter dem Kürzel JDK (Java Development Kit) alles nötige
  - Compiler
  - JRE
  - API
- http://www.oracle.com/technetwork/java/javase/downloads/index.html

wissen.leben
WWU Münster

Lösung (Aufgabe II)

II. Schreiben Sie eine Methode **mult** zur Multiplikation zwei ganzzahliger Zahlen größer 0 (**int**) ohne den **mult**-Operator

Lösung:

1. Datei erzeugen (z.B. **Vorlesung2**.java)
2. „Programmrahmen" erstellen

```
class Vorlesung2
   public static void main(String[] args){
   }
}
```

Klassenname == Dateiname!

3. Methode erstellen

```
class Vorlesung2 {

   public static int mult(int a, int b){
   }

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

Resultat der Multiplikation auf ganzen Zahlen ist eine ganze Zahl:
Rückgabewertebereich: **int**
Methode erwartet zwei Parameter:
Multiplikand: **a**
Multiplikator: **b**

wissen.leben WWU Münster

Lösung (Aufgabe II)

4. Algorithmus (Vorschrift von Folie 19: $a*b=\underbrace{b+b+...+b}_{a}=\sum_{i=1}^{a} b$)

```
static int mult(int a, int b) {
   int result = 0;
   for (int i = 0; i < a; i++) {
      result = result + b;
   }
   return result;
}
```

Zwischenspeicher für das Resultat

a – mal b addieren

Resultat zurückliefern

5. Achtung: Die **for**–Schleife läuft unendlich falls **a < 0**

```
static int mult(int a, int b) {
   if (a < 0 && b > 0) return mult(b, a);
   if (a < 0 && b < 0) return mult(-1 * a, -1 * b);
   …
}
```

Falls a < 0, b > 0: Kommutativgesetz
Falls a < 0, b < 0: – * – = +

6. Testen:

```
public static void main(String[] args) {
   System.out.println(mult(37, 27));
   System.out.println(mult(-37, 27));
   System.out.println(mult(37, -27));
   …
}
```

Methodenaufruf

wissen.leben
WWU Münster

Lösung (Aufgabe III)

III. Schreiben Sie eine Methode **modulo**, die den ganzzahligen Rest eine Division auf positiven, ganzen Zahlen (**int**) liefert

- Die Eingaben sollten überprüft werden und bei einer fehlerhaften Eingabe -1 zurückliefern

Lösung:

1. Die Methode wird mit in Vorlesung2.java geschrieben
2. Methode erstellen

```
class Vorlesung2 {

   public static int modulo(int a, int b){
   }

   …
}
```

```
Resultat der Multiplikation auf
positiven ganzen Zahlen ist eine
ganze Zahl:
            Rückgabewertebereich: int
Methode erwartet zwei Parameter:
            Dividend: n
            Divisior: mod
```

wissen.leben
WWU Münster

Lösung (Aufgabe III)

3. Algorithmus

```
static int modulo(int n, int mod) {
   if (n == mod) return 0;

   int temp = 0;
   while ((temp + mod) < n) {
      temp += mod;
   }

   return n - temp;
}
```

Falls n == mod, n/n = 1 → Rest 0

Finde die **größte Zahl < n**, die durch **mod teilbar** ist:
Solange **mod addieren, bis** die nächste Addition **> n** ist
Die **Differenz** der **größten Zahl** < n und **n** liefert den **Rest** der ganzzahligen Division

4. Grenzen des Algorithmus:

```
static int modulo(int n, int mod) {
   if (n <= 0 || mod <= 0) {
      return -1;
   }
   …
}
```

n und **mod** sollen **> 0** sein, ansonsten **-1** zurückliefern

wissen.leben WWU Münster