3. Basiskonzepte von Java

- Die in Abschnitt 3 vorgestellten Konzepte von Java sind allgemein gültig und finden sich so oder so ähnlich in eigentlich jeder gängigen Programmiersprache.

  → Abschnitt 3.2.3, Klassen nur in objektorientierten Programmiersprachen.

- Die Unterschiede in den Details (vor allem in der Syntax) sind allerdings oft sehr groß.
- Wir werden uns für deb Rest dieser Vorlesung voll auf Java konzentrieren.
- Aber (hoffentlich) in dem Bewusstsein, dass Java letztendlich nur ein Beispiel ist.
- Wenn man diese Konzepte einmal an einem Beispiel wie Java verstanden hat, wird man sie in jeder anderen Programmiersprache leicht wiederfinden und durchschauen.

3.1. Programmfluss

- Java–Quelltext wird durch einen Compiler wie `javac` in eine idealisierte Form von Maschineninstruktionen überführt (*Java Byte Code*).
- Durch einen Interpreter wie `java` wird das Programm in seiner Form als Java Byte Code ausgeführt.
- Im Prinzip läuft das wie die Ausführung von ”echtem“ Maschinencode durch die Hardware ab.
- Andere Java–Compiler übersetzen Java–Quelltext auch wahlweise entweder in Maschinencode statt in Java Byte Code, der dann von der Hardware als ”Interpreter“ auszuführen ist
  - (wie es bei vielen anderen Programmiersprachen die Regel ist, z.B. Ada, C, C++, Cobol, Fortran, Pascal).

Abarbeitung eines Programms

- Eine interne Uhr zerlegt die Zeit in einzelne Taktzyklen, und in jedem Taktzyklus wird eine Maschineninstruktion abgearbeitet (etwas idealisiert formuliert!).
- Im Prinzip werden die Maschineninstruktionen eines ausführbaren Programms Schritt für Schritt nach dieser Uhr in der Reihenfolge ihrer Speicheradressen abgearbeitet.
- Durch Sprunginstruktionen (und ausschließlich dadurch!) kann von dieser Abfolge beliebig abgewichen werden.
- Zum Beispiel ergibt sich eine Schleife, wenn mit einer bedingten Sprunganweisung zu einer vorher schon einmal abgearbeiteten Instruktion zurückgesprungen wird.

→ Sprungbedingung ≡ Abbruchbedingung

Programmfluss in Java

- Die Abarbeitung von Java Byte Code durch einen Interpreter läuft im Prinzip genauso ab wie die Abarbeitung von Maschinencode im Von–Neumann–Modell.
- Jede elementare Anweisung eines Java–Quelltextes wird im allgemeinen in ein oder mehrere Instruktionen in Java Byte Code (bzw. Maschinencode) übersetzt.
- Konstrukte wie Verzweigungen und Schleifen werden in jeweils spezifische Konstellationen von bedingten und unbedingten Sprüngen übersetzt.

Begriff Programmfluss:

- Zu jedem Zeitpunkt der Abarbeitung eines Programms wird eine einzelne Instruktion abgearbeitet.
- Die Abfolge der abgearbeiteten Instruktionen ist der dynamische *Programmfluss* durch den statischen *Programmtext* hindurch.

3.1.1. Ausdrücke

Ein Ausdruck konstituiert sich durch

- einen *Rückgabetyp*,
- einen *Rückgabewert* und
- optionale Seiteneffekte.

Beispiel I:

```
double x = (3+6) / 2;
```

(3+6) / 2 → Ausdruck

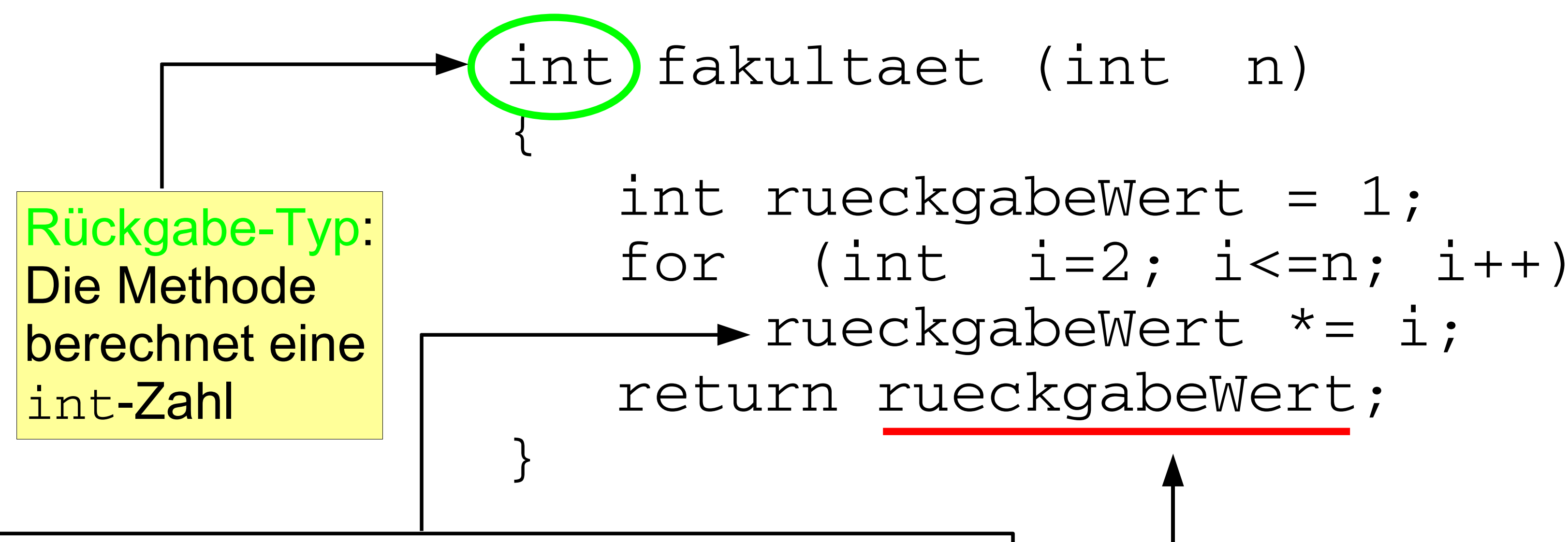
Erläuterungen:

- Rückgabewert: 4
- Rückgabetyp: **`int`** (nicht `double` !)
- Seiteneffekt: Bei der Zuweisung an die `double`–Variable wird dieser `int`–Wert in einen `double`–Wert konvertiert.

Rückgabe-Werte von Methoden

- Eine Methode, die nicht `void` ist, kann in Form eines Ausdrucks aufgerufen werden.
- Der Rückgabetyp einer solchen Methode ist der Typ, der am Anfang des Methodenkopfes steht.
- Der Rückgabewert ist der Wert des Ausdrucks hinter dem Statement `return`
  - **WICHTIG:** `return` beendet die Abarbeitung der Methode!
  - Eine Methode kann mehrere `return`-Statements enthalten (aber die Methode wird abgebrochen, sobald der Programmfluss das erste `return` erreicht)

Beispiel



Anmerkung:
Operator *= weist der Variablen au[f]
linken Seite das Produkt aus ihrem
eigenenmomentanen Wert und dem
des Ausdrucks auf der rechten Seit[e]

Analog sind +=, -= und /= definiert.

Rückgabe-Wert:
Der Wert, der am Ende der Methode in der Variablen `rueckgabeWert` gespeichert ist.

Jeder andere Ausdruck, der einen Wert des Rückgabe-Typs retourniert ist zulässig!

Beispiel

```
int fakultaet (int  n)
{
   int rueckgabeWert = 1;
   for  (int  i=2; i<=n; i++)
       rueckgabeWert *= i;
   return rueckgabeWert;
}
```

Beispiele für Aufruf der Methode:

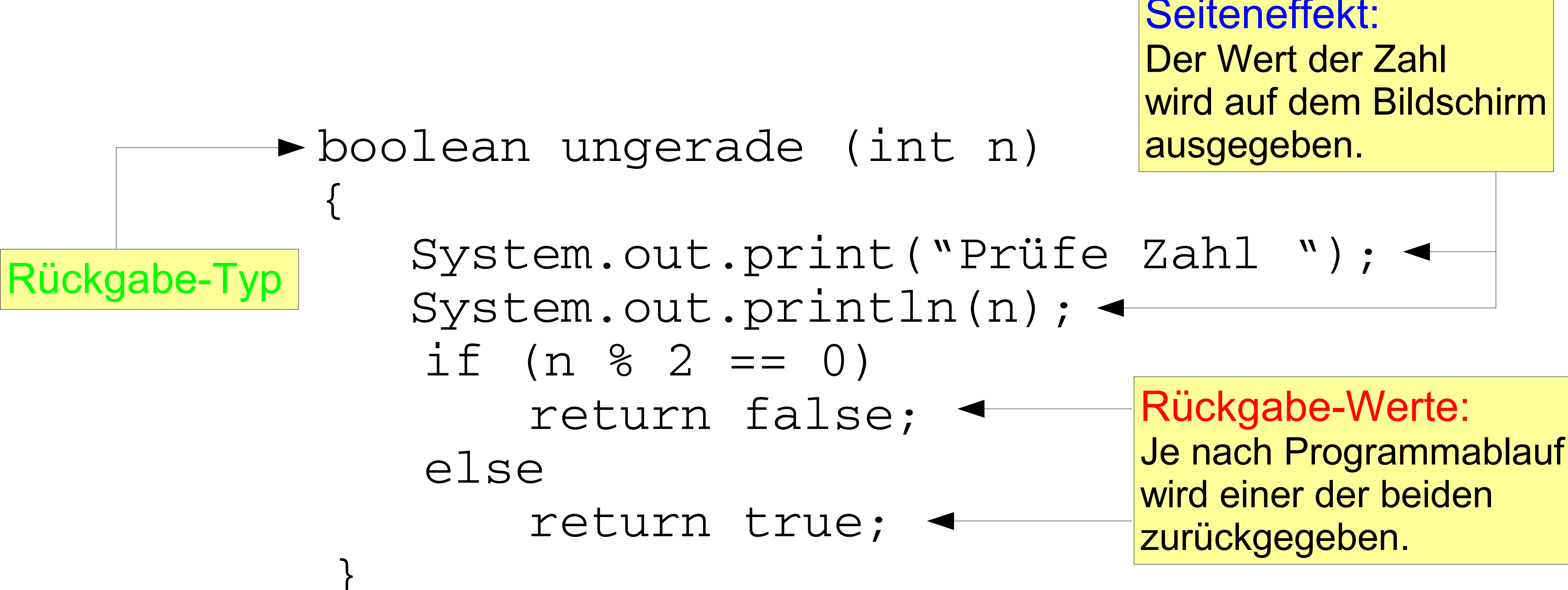
```
int fak5 = fakultaet(5);

if (fakultaet(n) == 6) {
   ...
}
```

Seiteneffekte

- Methodenaufrufe können auch *Seiteneffekte* haben:
  - ◊ Alle Effekte einer Methode auf die ”Umwelt“ außerhalb der Methode.
  - ◊ *Ausnahme*: Die Rückgabe eines Wertes wird nicht als Seiteneffekt, sondern sozusagen als der ”Haupteffekt“ der Methode angesehen.
- *Beispiel*: Das Schreiben eines Wertes auf den Bildschirm ist ein typischer Seiteneffekt.

Beispiel



Beispiel für Aufruf der Methode:

```
if (ungerade(x)) { ... }
```

Wiederholung: Bildschirmausgabe

`System.out.print(Argument)` Schreibt das Argument auf den Bildschirm. Das Argument darf dabei von folgenden Typen sein:

- Zeichenketten zwischen doppelten Anführungsstrichen (genauer: alles vom Typ `String`)
- Alle eingebauten Typen (`int,float,double,char,boolean`) (werden eigentlich auf Strings konvertiert)
- Strings können mittels + “addiert” (aneinandergehängt) werden.

Beispiel:

```
System.out.print("Der Wert von x ist " + x);
```

`System.out.println(Argument)` Tut dasselbe, jedoch beendet es die Ausgabe mit einem Zeilenwechsel.

- Dadurch wird die nächste Ausgabe in eine neue Zeile geschrieben.

3.1.2 Verzweigungen und Schleifen

Allgemeiner Zusammenhang zwischen Programmstruktur und Programmfluss:

- In Programmteilen, in denen keine Verzweigungen, Schleifen, Methodenaufrufe, `return`s etc. vorkommen, folgt der Programmfluss strikt der sequentiellen Programmstruktur.
- *Das heißt*: Die Anweisungen werden sequentiell in der Reihenfolge abgearbeitet, in der sie im Quelltext auftreten.
- Verzweigungen und Schleifen sind in Java die einfachsten, grundlegenden Möglichkeiten, den Programmfluss von dieser sequentiellen Struktur des Quelltextes zu lösen.
- Diese Konstrukte lassen sich auf bedingte und unbedingte Sprungbefehle zurückführen.

if-Statement

```
if ( x == 1 )
   y = y + 1; // If-Zweig
```

Semantik:

1. Stelle fest, ob die Speicherzelle x momentan den Wert 1 enthält.
2. Falls nein, springe zur Instruktion Nr. 4 (bedingter Sprung)
3. Führe die Maschineninstruktionen für den `if`–Zweig aus (d.h. erhöhe den Wert in der Speicherzelle `y` um 1).
4. ...

if-else Statement

```
if ( x == 1 )
   y = y + 1; // If-Zweig
else
   y = y - 1; // Else-Zweig
```

Semantik:

1. Stelle fest, ob die Speicherzelle x momentan den Wert 1 enthält.
2. Falls ja, springe zur Instruktion Nr. 5 (bedingter Sprung).
3. Führe die Maschineninstruktionen für den "`else`"–Zweig aus (d.h. vermindere den Wert in der Speicherzelle `y` um 1).
4. Springe zur Instruktion Nr. 6 (unbedingter Sprung).
5. Führe die Maschineninstruktionen für den "`if`"–Zweig aus (d.h. erhöhe den Wert in der Speicherzelle `y` um 1).
6. ...

while-Schleife

```
while ( x > 0 )
    x = x - 1; // Schleifenrumpf
```

Semantik:

1. Stelle fest, ob die Speicherzelle `x` momentan einen Wert größer als `0` enthält.
2. Falls nein, springe zur Instruktion Nr. 5.
3. Führe die Maschineninstruktionen für den Schleifenrumpf aus. (d.h. vermindere den Wert in der Speicherzelle `x` um `1`).
4. Springe zur Instruktion Nr. 1.
5. ...

do-while-Schleife

```
do
   x = x - 1;      // Schleifenrumpf
while ( x > 0 );
```

Semantik:

1. Führe die Maschineninstruktionen für den Schleifenrumpf aus.
2. Stelle fest, ob die Speicherzelle x momentan einen Wert größer als `0` enthält.
3. Falls ja, springe zur Instruktion Nr. 1.
4. ...

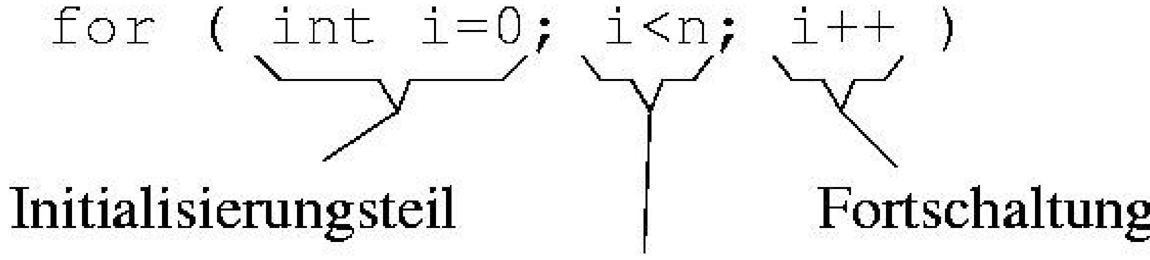
→ **<u>Semantischer Unterschied</u>** <u>zur While–Schleife:</u>
Der Schleifenrumpf wird mindestens einmal durchlaufen!

for-Schleife

Eine For–Schleife ist äquivalent zu einer While-Schleife mit

- Initialisierungsteil vor der eigentlichen Schleife,
- gleicher Fortsetzungsbedingung und
- Fortschaltung am Ende des Rumpfes.

Beispiel:



Zur Syntax:

Initialisierungsteil, Fortsetzungsbedingung und Fortschaltung müssen immer durch Semikolons voneinander getrennt werden.

for-Schleife

- Beispiel einer For–Schleife:

```
for ( int i=0; i<n; i++ )
    A[i] = i;
```

Semantik:

1. Setze den Inhalt der Speicherzelle `i` auf den Wert `0`.
2. Stelle fest, ob der momentane Wert von `i` kleiner als `n` ist.
3. Falls nein, springe zur Instruktion Nr. 7.
4. Setze den Inhalt von `A[i]` auf den Wert von `i`.
5. Erhöhe den Wert von `i` um `1`.
6. Springe zur Instruktion Nr. 2.
7. ...

for-Schleife

- Beispiel einer For–Schleife:

```
for ( int i=0; i<n; i++ )
    A[i] = i;
```

- Umsetzung in einen äquivalenten Java–Quelltext mit While–Schleife:

```
int i = 0;        // Initialisierung
while (i<n)       // Fortsetzungsbedingung
{
    A[i] = i;
    i++;          // Fortschaltung
}
```

Steuerung des Programm-Flusses im Schleifenrumpf

- Bei jeder der drei Schleifenarten (While, Do–While, For) kann die Ausführung der Schleife im Schleifenrumpf durch spezielle Anweisungen noch feiner gesteuert werden.

- *Konkret*:
  - `break`: Bricht die ganze Schleife ab, d.h. springt zur ersten Anweisung nach der Schleife
  - `continue`: Bricht die aktuelle Ausführung des Schleifenrumpfs ab und lenkt den Programmfluss zur Auswertung der Fortsetzungsbedingung zurück.

Nonsens-Beispiel

```
while ( x > 0 )
{
   y = y + 1; // (a)
   z = z + 2; // (b)
   if ( y > 100 )
       break;
   if ( z == 200 )
       continue;
   x = x - 1; // (c)
}
```

Nonsens-Beispiel

Semantik:

1. Stelle fest, ob die Speicherzelle `x` momentan einen Wert größer als `0` enthält.
2. Falls nein, springe zur Instruktion Nr. 10.
3. Führe die Maschineninstruktionen für die Anweisungen (a) und (b) aus.
4. Stelle fest, ob die Speicherzelle `y` momentan einen Wert größer als `100` enthält.
5. Falls ja, springe zur Instruktion Nr. 10 ("`break`").
6. Stelle fest, ob die Speicherzelle `z` den Wert `200` enthält.
7. Falls ja, springe zur Instruktion Nr. 1 (`continue`).
8. Führe die Maschineninstruktionen für die Anweisung (c) aus.
9. Springe zur Instruktion Nr. 1.
10. ...

Blöcke von Anweisungen

- Geschweifte Klammern fassen mehrere Anweisungen zu einer zusammen.

- Eine einzelne Anweisung kann (ohne Effekt) durchaus ebenfalls in geschweifte Klammern gesetzt werden.

- Will man in einem `if`–Zweig, einem zugehörigen `else`–Zweig oder einem Schleifenrumpf nicht nur eine, sondern mehrere Anweisungen ausführen lassen, muss man sie in geschweiften Klammern zusammenfassen.

- Auch für die Zuordnung eines `else`–Zweigs zum richtigen `if`–Konstrukt sind solche Klammerungen von Bedeutung.

Beispiel

```
if ( x > 0 )
{
   if ( x < 1 )
         System.out.println(“x zwischen 0 und 1”);
}
else if ( x < 0 )
{
   if ( x > -1 )
         System.out.println(“x zwischen -1 und 0”);
}
else
   System.out.println("x ist 0" );
```

→ Schreibausgabe nur im Fall `-1 < x < 1`
falls `x <= -1` oder `x >= 1` passiert nichts.

Vergleich mit und ohne Klammerung

```
if ( x > 0 )
   if ( x > 1 )
      System.out.println("x > 1");
else
      System.out.println("0 < x <= 1");
```

→ Der "`else`"–Zweig gehört zum zweiten "`if`".
→ Im Fall `x<=0` wird nichts geschrieben.

```
if ( x > 0 ) {
    if ( x > 1 )
       System.out.println("x > 1");
}
else
       System.out.println("x <= 0");
```

→ Der "`else`"–Zweig gehört zum ersten `if`.
→ Im Fall, dass `x>0` und `x<=1` ist, wird nichts geschrieben.

3.1.3. Methoden-Aufrufe

- Wird eine Methode aufgerufen, so wird der Programmfluss in diese Methode hineingelenkt.
- Mit `return` wird der Programmfluss wieder an die aufrufende Stelle zurückgelenkt.
- Bei einer `void`–Methode braucht ganz am Ende des Quelltextes der Methode kein `return` zu stehen, und der Programmfluss wird trotzdem zurückgelenkt.

Beispiel 1

```
public void meineMethode ( int n )
{
  if ( n < 0 )
     return;
  System.out.println (n);
}
```

Erläuterungen:

- Falls `n` negativ ist, wird der Programmfluss durch das `return` sofort an die aufrufende Stelle zurückgelenkt.
- *Ansonsten*: Sobald der Programmfluss am Ende des Quelltextes der Methode ankommt, wird er auch ohne `return` wieder zurückgelenkt.
- Natürlich könnte man auch ein `return` als letzte Anweisung am Ende der Methode einfügen.

Beispiel 2

```
public int meineMethode ( int n )
{
   if ( n < 0 )
       return -n;
   System.out.println (n);
   return n;
}
```

Erläuterungen:

- Ist eine Methode nicht `void`, so muss mit `return` zugleich ein Wert des Datentyps, der vor dem Methodennamen anstelle von `void` steht, zurückgegeben werden.
- Insbesondere darf hier das `return` (im Gegensatz zur `void`–Methode auf der letzten Folie) auch am Ende des Quelltextes der Methode nicht fehlen.

Beispiel 3

```
public int meineMethode ( int n )
 {
    System.out.println (n);
    return n;
 }
 ...

 int m = 3;
 meineMethode ( m );
```

Erläuterungen:

- Auch eine Methode, die nicht `void` ist, kann wie eine `void` Methode aufgerufen werden.
- Der Rückgabewert geht dann verloren, d.h. man ist nur an den Seiteneffekten der Methode interessiert!

Beispiel 4

```
public boolean printFakultaet ( int n )
 {
   if ( n <= 0 )
      return false;
   int fak = 1;
   for ( int i=2; i<=n; i++ )
      fak *= i;
   System.out.println ( fak );
   return true;
 }
...
printFakultaet(10);
```

Erläuterungen:

- Der Rückgabewert gibt an, ob `n` positiv ist und somit die Fakultät von `n` berechnet werden kann.
- Wenn man aber wie im Beispiel *weiß*, dass `n` positiv ist, braucht man den Rückgabewert nicht.

Beispiel 4

```
public boolean printFakultaet ( int n )
 {
   if ( n <= 0 )
      return false;
   int fak = 1;
   for ( int i=2; i<=n; i++ )
      fak *= i;
   System.out.println ( fak );
   return true;
 }
```

Alternativer Aufruf:

```
if (!printFakultaet(n))
  System.out.println("Fakultät von n nicht berechnet!")
```

Übersetzung von Methoden

- Der Code zu einer Methode wie `meineMethode` oder `fakultaet` bildet einen separaten Codeblock mit fester Anfangsadresse.

- An allen Stellen im Source File, an denen der Compiler einen Aufruf der Methode `meineMethode` findet, setzt er im Code eine unbedingte Sprunganweisung zu dieser Anfangsadresse ein.

- Vor dieser Sprunganweisung setzt der Compiler noch Code ein, mit dem die Parameter der Methode an die Stelle kopiert werden, wo sie von der Methode erwartet werden.

  → Konkret `int n` in den Beispielen 1-4.

- Bei einer Methode mit Rückgabewert muss am Ende noch der Rückgabewert an die Stelle kopiert werden, an der er im aufrufenden Code erwartet wird.

Rücksprung aus Methoden

Problem:

- Eine Methode wie `meineMethode` kann ja durchaus an mehreren Stellen im Code aufgerufen werden.
- Woher ”weiß“ die Methode eigentlich, wohin der Programmfluss mit `return` jeweils zurückspringen soll?

Antwort:

- Zusätzlich zu den Parametern bekommt eine Methode eine *Rücksprungadresse* als weitere Information.
- Vor dem Sprung zur Anfangsadresse der Methode setzt der Compiler daher noch zusätzlichen Code ein, mit dem die Rücksprungadresse an der Stelle abgelegt wird, wo sie von der Methode erwartet wird.
- Jedes `return` wird vom Compiler in Instruktionen übersetzt, die diese Adresse lesen und einen Sprung dorthin ausführen.

main

- Nach dem Starten eines Java Programms wird immer eine spezielle Routine namens `main` aufgerufen

- Definition:

```
public static void main (String[] args) {
    // Definition der Methode
}
```

- Die Methode ist `void`, d.h. ohne Return-Wert

- Der Methode kann eine Menge von Argumenten übergeben werden, die per Definition alle Strings sind (können natürlich vom Programm selbst in andere Typen konvertiert werden).

- Die genaue Bedeutung von `public static` kommt später.

Der Run-Time-Stack

Problem:

- Eine Methode kann intern wieder eine andere Methode aufrufen, die intern wieder eine andere Methode aufruft, die intern wieder eine andere Methode aufruft usw.
- Im allgemeinen ”steckt“ der Prozess also in mehreren Methoden gleichzeitig.
- Nur die ganz zuletzt aufgerufene Methode ist allerdings aktiv in Abarbeitung.
- Die anderen Methoden ”warten“ darauf, dass sie durch Rücksprung reaktiviert werden.
- Die Daten, die jede dieser Methoden mit der jeweils aufrufenden Codestelle austauscht (also Parameter, Rücksprungadresse und Rückgabewert) müssen nach einem einheitlichen Schema organisiert sein, so dass jede Methode ”weiß“, welches genau ihre Daten sind.

Beispiel

```
int f1 ()
 {
    return 1;
 }
int f2 ()
 {
    return 2;
 }
int f3 ()
 {
    return f2()+f1();
 }
```

Erläuterungen:

- Wenn `f3` aufgerufen wird, gibt es eine kurze Zeitspanne, in der der Prozess in `f1`, `f2` und `f3` zugleich ”steckt“.
- In dieser Zeitspanne ist aber nur `f1` bzw. `f2` aktiv in Bearbeitung.

Hierarchie von Aufrufen

- Zusätzlich steckt der Prozess in dieser Zeitspanne auch in der Methode, die ihrerseits `f3` aufgerufen hat, in der Methode, die die letztere aufgerufen hat usw.

- Anfang dieser Aufrufhierarchie: Eine Methode, die ihrerseits von außerhalb des Programms aufgerufen wird, zum Beispiel:
  - ◊ Methode `main` wird vom Interpreter `java` direkt aufgerufen
  - ◊ Methode `paint` von Klasse `Applet` und ihren Erweiterungen wird von `mozilla` und `appletviewer` direkt aufgerufen.

- Wenn man innerhalb seiner eigenen Methoden noch vordefinierte Standardmethoden wie z.B. `System.out.println` aufruft, können darin unsichtbar noch etliche weitere Methodenaufrufe stattfinden.

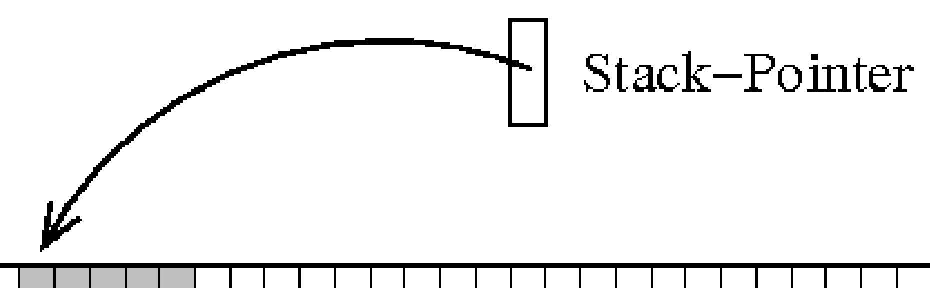
Stack Pointer

- Zu jedem Java–Prozess gibt es einen reservierten Speicherbereich, in dem die Parameter, Rücksprungadressen und Rückgabewerte jeder momentan aufgerufenen Methode nach strikten Regeln abgelegt sind.
- Die Daten zu einem einzelnen Methodenaufruf werden in einem zusammenhängenden Segment abgelegt.
- Die Segmente werden in der zeitlichen Reihenfolge der zugehörigen Methodenaufrufe im Speicher abgelegt.
- Die Anfangsadresse des jeweils zuletzt angelegten Segments wird in einem eigens dafür vorgesehenen Register namens *Stack–Pointer* gespeichert.
- Der Compiler fügt vor dem Methodenaufruf noch Code ein, mit dem der Stack–Pointer um die Größe des Segments hochgesetzt wird, und bei `return` entsprechend Code zur Zurücksetzung des Stack–Pointers.

Abarbeitung des Beispiels

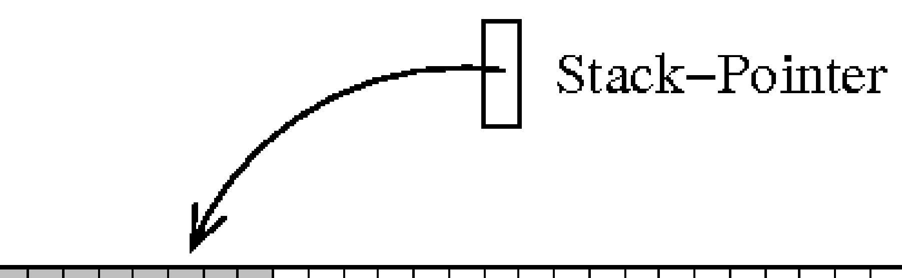
```
int f1() {
    return 1;
}
int f2() {
    return 2;
}
int f3() {
    return f2()+f1();
}
```

Nach Programmstart
(`main` wurde aufgerufen)



Abarbeitung des Beispiels

Nach Aufruf von `f3` aus `main`

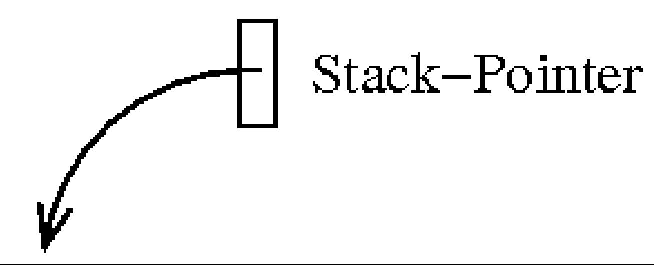



```
int f1() {
    return 1;
}
int f2() {
    return 2;
}
int f3() {
    return f2()+f1();
}
```

Abarbeitung des Beispiels

Nach Aufruf von `f2` aus `f3`

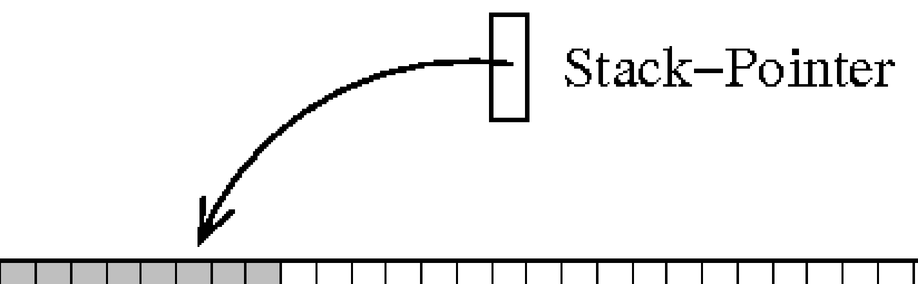
```
int f1() {
    return 1;
}
int f2() {
    return 2;
}
int f3() {
    return f2()+f1();
}
```



Abarbeitung des Beispiels

Nach Beendigung von `f2` in `f3`

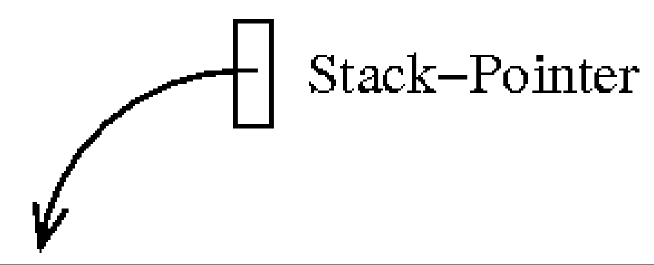
```
int f1() {
    return 1;
}
int f2() {
    return 2;
}
int f3() {
    return f2()+f1();
}
```

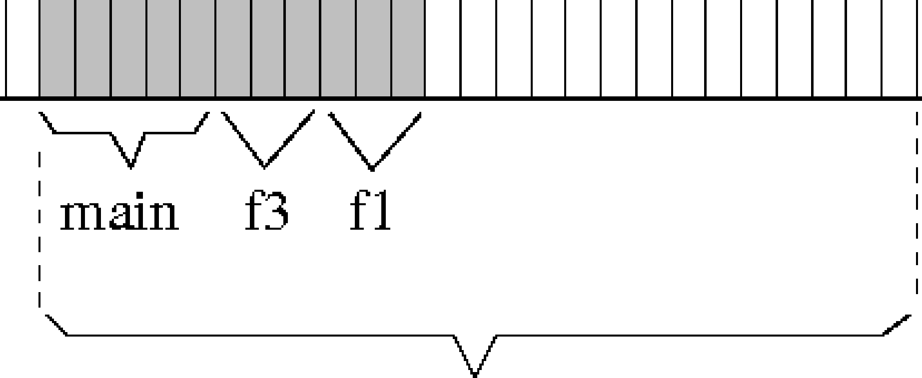


Abarbeitung des Beispiels

Nach Aufruf von `f1` aus `f3`

```
int f1() {
    return 1;
}
int f2() {
    return 2;
}
int f3() {
    return f2()+f1();
}
```

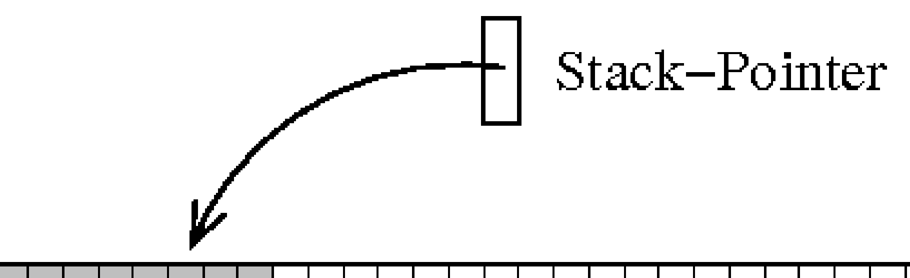




Reservierter Bereich

Abarbeitung des Beispiels

Nach Beendigung von `f1` in `f3`

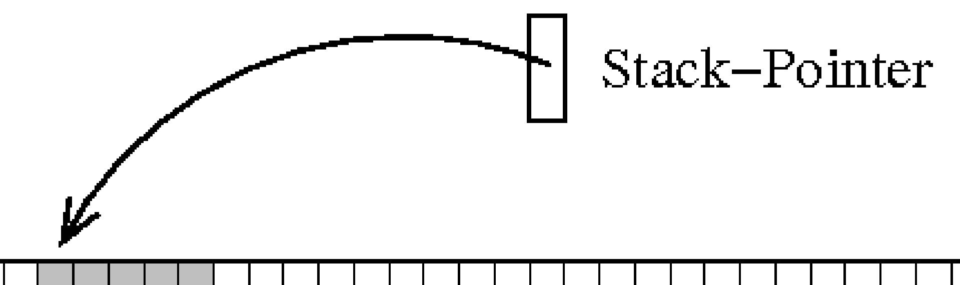



```
int f1() {
    return 1;
}
int f2() {
    return 2;
}
int f3() {
    return f2()+f1();
}
```

Abarbeitung des Beispiels

Nach Beendigung von `f3` in `main`

```
int f1() {
    return 1;
}
int f2() {
    return 2;
}
int f3() {
    return f2()+f1();
}
```



Run-Time Stack

- *Run Time* = Laufzeit (des Programms)

- *Stack* = Stapel

→ Der Run–Time–Stack ist ein ”Stapel“, auf den während der Abarbeitung des Programms durch Methodenaufrufe laufend Segmente draufgepackt und durch `return`s wieder weggenommen werden.

Zusammensetzung des Segments für einen Methodenaufruf:

- Ein Stück Speicher für jeden Parameter der Methode.

- Ein Stück Speicher für die Rücksprungadresse.

Beispiel

```
int f ( char c, double d )
 {
     return c+1;
 }
...
int i = f ( 'a', 3.14 );
```

Inhalt des Segments in diesem Beispiel:

- Ein Speicherstück der Größe eines `char` für `c`.
- Ein Speicherstück der Größe eines `double` für `d`.
- Ein Speicherstück für die Rücksprungadresse.

Beispiel

```
int f ( char c, double d )
 {
     return c+1;
 }
...
int i = f ( 'a', 3.14 );
```

1. Erhöhe den momentanen Wert des Stack–Pointers um die Größe des momentan obenaufliegenden Segments (→ Anfang des neuen Segments).
2. Schreibe den Wert des Zeichen–Literals `'a'` in das für Parameter `c` reservierte Speicherstück im neuen Segment.
3. Schreibe den Wert 3.14 in das für Parameter `d` reservierte Speicherstück im neuen Segment.
4. Schreibe die Adresse der nächsten Instruktion, die auf den Methodenaufruf "`f('a',3.14);`" folgt, in den für die Rücksprungadresse reservierten Speicherplatz im neuen Segment.

Umsetzung des return

1. Schreibe den Wert von `c+1` in ein dafür freigehaltenes Register.
2. Vermindere den momentanen Wert des Stack–Pointers um die Größe des obenaufliegenden Segments.
   → Segment ist ”aus dem Spiel“.
      (Seine Inhalte werden nie wieder angeschaut und bei späterem Bedarf durch Einrichtung neuer Segmente bei Methodenaufrufen überschrieben.)
   → Alle lokalen Variablen verlieren Gültigkeit!
3. Springe zu der Adresse, die in diesem ”aus dem Spiel geratenen“ Segment an der für die Rücksprungadresse reservierten Stelle steht.

Bei der Instruktion an dieser Rücksprungadresse:

1. Kopiere den Wert aus dem oben genannten Register in die Speicherzelle `i`.

```
int i = f ( 'a', 3.14 );
```

Rekursive Methoden

```
void meineRekursiveMethode ( int n )
{
   if ( n < 0 )
      return;
   System.out.print ( n + " " );
   meineRekursiveMethode ( n - 1 );
   System.out.print ( n + " " );
   return;
 }
```

Terminologie:

Eine Methode, die sich selbst aufruft, heißt *rekursiv*.

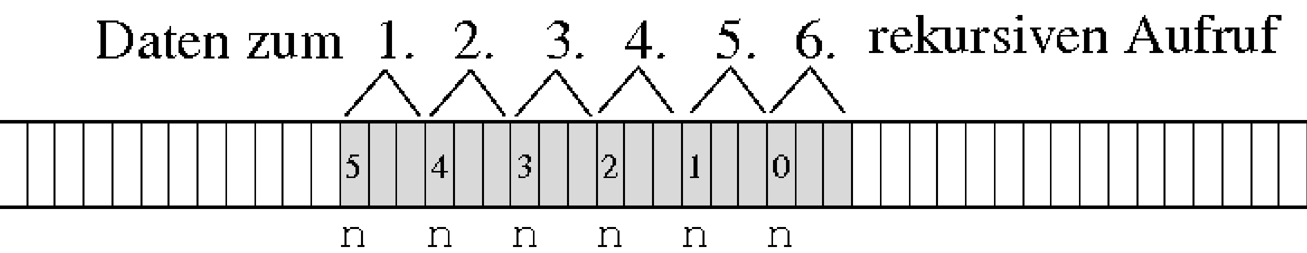
Ergebnis des Aufrufs ”`meineRekursiveMethode(5)`“:

```
5 4 3 2 1 0 0 1 2 3 4 5
```

Frage: Wie lässt sich dieses Ergebnis erklären?

Abarbeitung von Rekursionen

- Es gibt zwar im Quelltext von `meineRekursive-Methode` nur eine einzige Variable namens `n`.
- Aber durch die Rekursion werden nacheinander mehrere Segmente auf den Run–Time–Stack gebracht, die alle zu Aufrufen von `meineRekursiveMethode` gehören.
- In jedem dieser Segmente findet sich ein Speicherstück für `n`.
- Wie bei nichtrekursiven wechselseitigen Methodenaufrufen gehört das jeweils als letztes eingerichtete Segment zum momentan aktiv in Bearbeitung stehenden Methodenaufruf.



Beispiel für Rekursion

```
int fakultaet ( int n )
 {
   if ( n == 1 )
       return 1;
   return n * fakultaet (n-1);
 }
```

Erläuterungen:

- Funktionen wie die Fakultät werden typischerweise rekursiv definiert:
  - ◊ $1! = 1;$
  - ◊ $n! = n \cdot (n - 1)!$ für $n > 1$.
- Die rekursive Methode ist die unmittelbare Umsetzung dieser Definition.
- Achtung: Der Aufrufparameter muß $n \geq 1$ genügen!

Fibonacci-Zahlen

```
public int fib(int n) {
  if (n<2)
      return 1;
  else
      return fib(n-1)+fib(n-2);
}
```

Erläuterungen:

- Die sogenannten Fibonacci-Zahlen werden rekursiv folgendermaßen definiert:
  - ◊ $fib(0) = 1$
  - ◊ $fib(1) = 1$
  - ◊ $fib(n) = fib(n-1) + fib(n-2)$
- Die rekursive Methode ist die unmittelbare Umsetzung dieser Definition.

Zum Namen “Rekursion”

- „Rekursion“ und ”rekursiv“ stammen vom lateinischen ”recurrere“ = ”zurücklaufen“.
- In diesem Kontext könnte man vielleicht sagen: ”zu sich selbst zurückkommen“.
- Genau das passiert ja auch bei Rekursion: Die Methode kommt auf sich selbst zurück.