Einführung in die Programmierung

9. Programmierung mit Methoden und Ausdrücken

Prof. Dr. Marcel Luis

Westf. Hochschule

WS 2022/2023

Methoden

- Methoden helfen, Code-Duplizierungen zu vermeiden.
- Methoden erlauben, längere Algorithmen so zu strukturieren, dass sie besser lesbar, verständlich und änderbar sind.
- Objektorientierte Programmierung benötigt Methoden, um Operationen auf Objekten zu realisieren (dazu später mehr).
- Methoden sind für Programmierung in einem funktionalen Stil (zur Realisierung von Funktionen) zwingend erforderlich.

Statische Methoden

Statische Methoden stellen Berechnungsfunktionalität bereit, die unter einem Namen aufgerufen werden kann.

Deklaration einer statischen Methode

Statische Methode zur Realisierung der Betragsfunktion:

```
public static int betrag(int n) {

    return (n >= 0) ? n : -n;
}
```

Anwendung der Methode

Anwendung der Methode zur Berechnung von $\frac{|n|}{|n+1|}$:

```
betrag(n) / (double) betrag(n + 1)
```

Drei Aspekte bzgl. statischer Methoden

- Deklaration einer Methode
- Syntax der Anwendung einer Methode
- Semantik der Anwendung einer Methode

Deklaration einer statischen Methode

Schema der Deklaration einer statischen Methode:

```
Modifikatoren Ergebnistyp Methodenname(
        Typ_1 FormalerParameter_1,
        Typ_2 FormalerParameter_2,
        ...,
        Typ_n FormalerParameter_n) {

    ...
    return Ergebnisausdruck;
}
```

- Modifikatoren: zunächst immer `public static`
- Methodenkopf: Legt Typ, Namen und Parameter der Methode fest.
- Methodenrumpf: Legt Berechnungsvorschrift fest. Kann Anwendung der formalen Parameter enthalten.

Methodennamen

- wie bei allen Namen: kurz und prägnant, vor allem aber ausdrucksstark
- in der Regel: Verb oder Verb-Phrase (s. `https://google.github.io/styleguide/javaguide.html#s5.2.3-method-names`)
  - Ausnahme: Methode hat Typ `boolean`; dann Form einer Frage, die mit ja (`true`) oder nein (`false`) beantwortet werden kann (wie bei Variablennamen)
  - mögliche Ausnahme: Methode liefert „eine Sache“; dann kann Verb entfallen (z. B. `laenge` statt `gibLaenge`)
- Schreibweise *lowerCamelCase* (wie bei Variablennamen)

Methodennamen

Gute Methodennamen

`berechneDifferenz`	Alltagsschreibweise: berechne Differenz
`gibText`	Alltagsschreibweise: gib Text
`setzeBetrag`	Alltagsschreibweise: setze Betrag
`fuegeHinzu`	Alltagsschreibweise: füge hinzu
`enthaeltWert`	Frage, die mit ja oder nein zu beantworten ist.
`istPrimzahl`	Frage, die mit ja oder nein zu beantworten ist.
`maximum`	Zwar kein Verb, jedoch als *max*-Funktion ein eingeführter mathematischer Begriff. `gibMaximum` wäre hier nicht besser.

Methodennamen

Schlechte Methodennamen

`hilf`	Soll wohl eine „Hilfsmethode“ sein, aber was genau macht Sie?
`berechne`	Es ist unklar, was berechnet wird.
`wandleUm`	Es soll wohl etwas umgewandelt werden, aber von *was* in *was*?
`pruefe`	Verb, aber unklar, was geprüft wird.

Formale Parameter oder Parametervariablen

- Parametervariablen sind spezielle lokale Variablen.
- Alle lokalen und Parametervariablen einer Methode müssen *unterschiedlich* heißen.
- Eine Methode muss keine Parametervariable besitzen. Der Methodenkopf sieht dann so aus:

  Modifikatoren *Ergebnistyp* *Methodenname*()

Ergebnistyp einer Methode

- Anwendung einer Methode ist ein Ausdruck.
- Typ des Ausdrucks ist der Ergebnistyp im Methodenkopf.
- Typ des Ergebnisausdrucks muss mit Ergebnistyp verträglich sein.

Verträglichkeit von Ergebnistyp und Typ des Ergebnisausdrucks

Sei `float` der Ergebnistyp einer statischen Methode und `a` eine Variable vom Typ `long`.

Zulässige Ergebnisausdrücke

`a / 2F`	gleiche Typen
`a / 2`	`long` ist implizit in `float` konvertierbar

Unzulässige Ergebnisausdrücke

`2.5`	nicht implizit konvertierbar
`a / 2.0`	nicht implizit konvertierbar

Syntax der Anwendung einer Methode

Ohne Qualifizierung

Es sei in einer Klasse eine Methode mit diesem Kopf deklariert:

`public static` *Ergebnistyp* *Methodenname*`(`Typ_1 p_1`, ...,` Typ_n p_n`)`

Anwendung dieser Methode in derselben Klasse:

Methodenname`(`$Ausdruck_1$`, ...,` $Ausdruck_n$`)`

Anwendung ist korrekt, wenn

- Typen von $Ausdruck_1$ bis $Ausdruck_n$ mit den entsprechenden Typen Typ_1 bis Typ_n der formalen Parameter gleich sind, oder
- implizit in diese konvertierbar sind.

Syntax der Anwendung einer Methode

Mit Qualifizierung

Es sei in einer Klasse K eine Methode mit diesem Kopf deklariert:

`public static` *Ergebnistyp Methodenname*`(`Typ_1 p_1`, ..., `Typ_n p_n`)`

Anwendung dieser Methode aus einer *anderen* Klasse:

K`.`*Methodenname*`(`$Ausdruck_1$`, ..., `$Ausdruck_n$`)`

Das Voranstellen des Klassennamens heißt *Qualifizierung*.

Anwendung einer Methode mit Qualifizierung

Anwendung einer statischen Methode aus der Klasse `Math`:

```
Math.abs(n) / (double) Math.abs(n + 1)
```

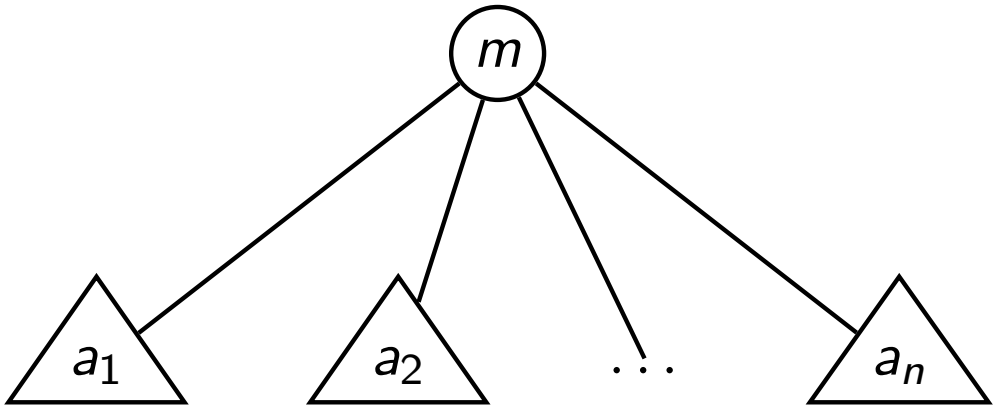
Anwendung einer Methode ist ein Ausdruck

- Anwendung $m(a_1, a_2, \ldots, a_n)$ einer Methode ist ein Ausdruck.
- Typ des Ausdrucks ist Ergebnistyp der Methode m.

Anwendung einer Methode

Baumdarstellung

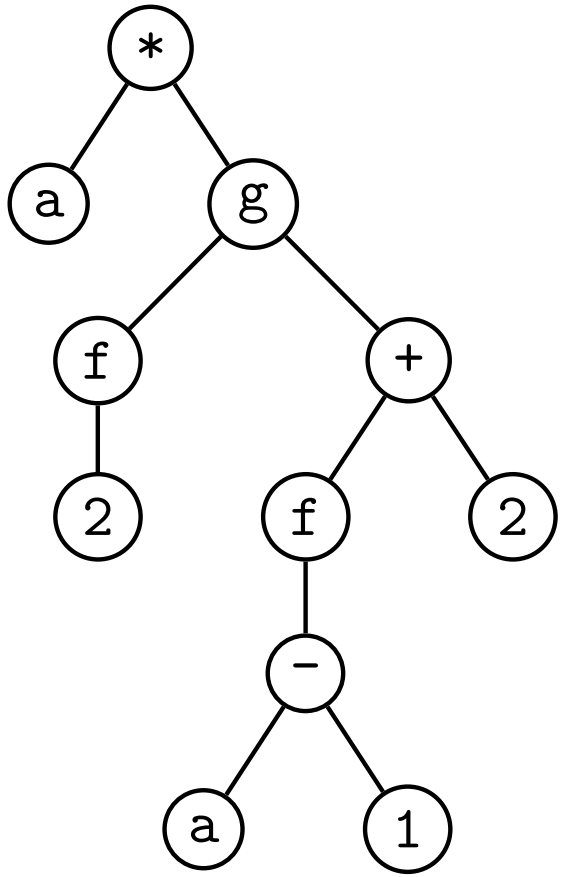
Baumdarstellung der Methodenanwendung $m(a_1, a_2, \ldots, a_n)$:



Anwendung einer Methode

Beispiel Baumdarstellung

Baumdarstellung des Ausdrucks `a * g(f(2), f(a - 1) + 2)`:



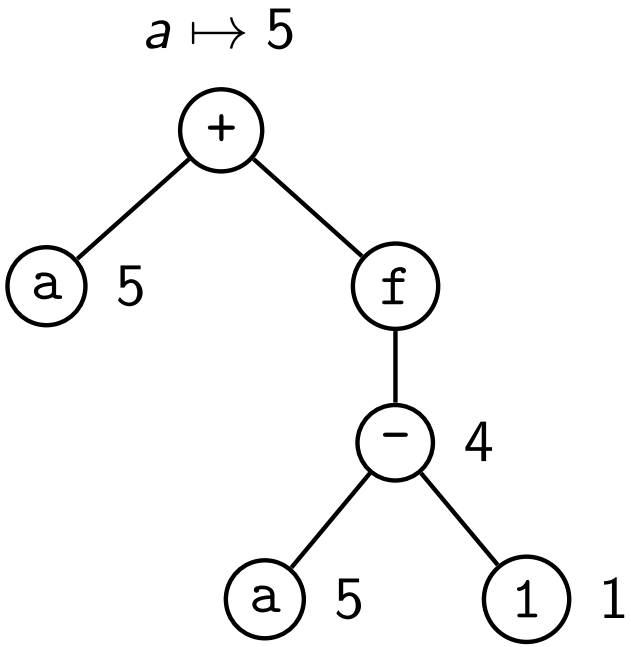
Semantik der Anwendung einer Methode

Beispiel: Auswertung durch mehrere Bäume

Definierte Methode

```
public static int f(int n) {
    return n * n - 1;
}
```

Auswertung des Ausdrucks `a + f(a - 1)` für $a = 5$.



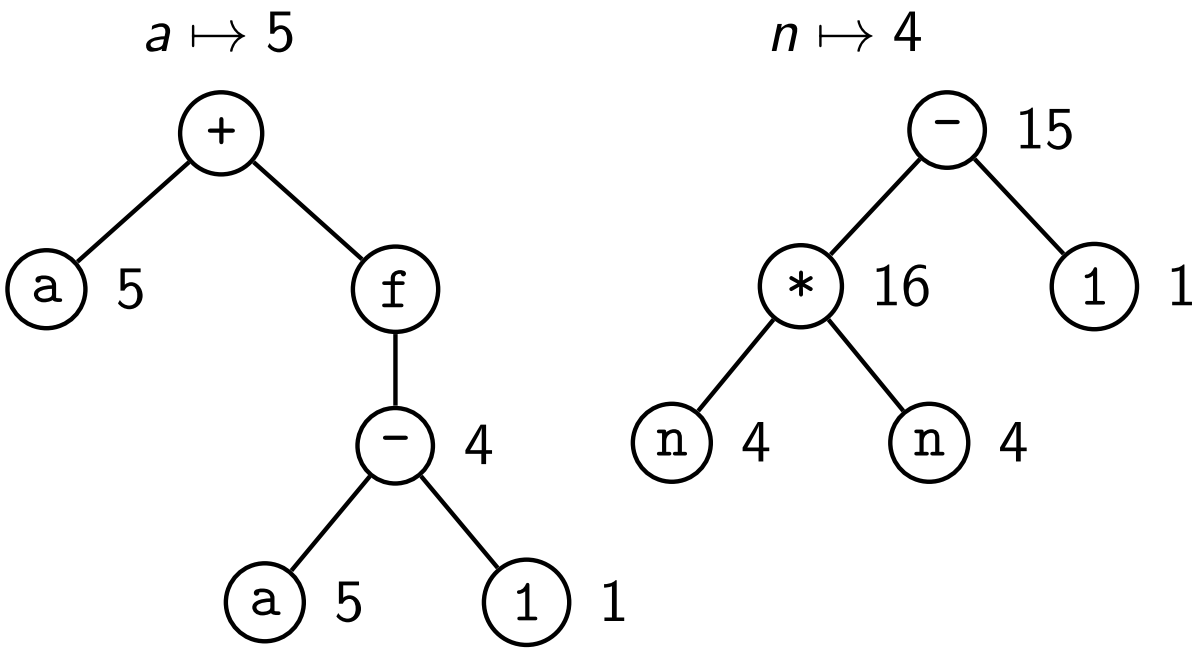
Semantik der Anwendung einer Methode

Beispiel: Auswertung durch mehrere Bäume

Definierte Methode

```
public static int f(int n) {
    return n * n - 1;
}
```

Auswertung des Ausdrucks `a + f(a - 1)` für $a = 5$.



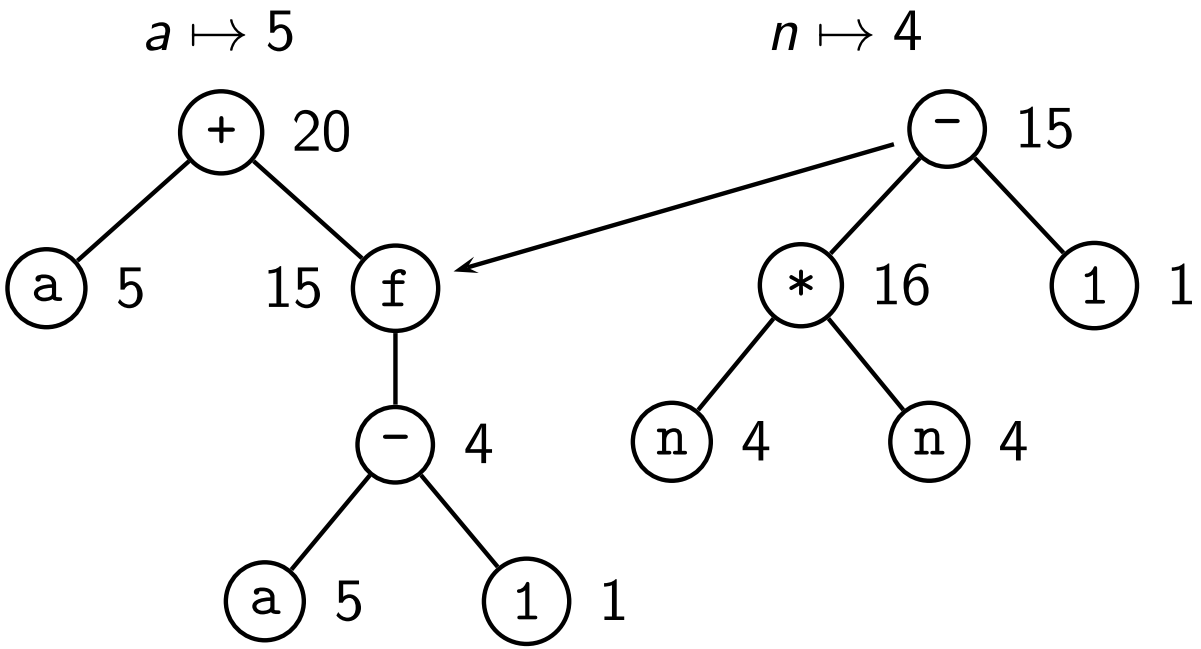
Semantik der Anwendung einer Methode

Beispiel: Auswertung durch mehrere Bäume

Definierte Methode

```
public static int f(int n) {
    return n * n - 1;
}
```

Auswertung des Ausdrucks `a + f(a - 1)` für $a = 5$.



Semantik der Anwendung einer Methode

1. Es werden *alle* Teilausdrücke ausgewertet. Deren Werte sind die aktuellen Parameter der Anwendung (oder des Aufrufs) der Methode.
2. Die aktuellen Parameter werden implizit den formalen Parametern der Methode zugewiesen.
3. Mit dieser Belegung der formalen Parameter wird die Methode ausgeführt.
4. Ihr Ergebniswert (also der Wert des Ergebnisausdrucks hinter `return`) ist der Wert des Methodenaufrufs.

Semantik der Anwendung einer Methode

Beispiel: Auswertung durch Gleichungskette

Definierte Methode

```
public static int f(int n) {
    return n * n - 1;
}
```

Auswertung des Ausdrucks `a + f(a - 1)` für $a = 5$.

$$
\begin{aligned}
& (a + f(a-1))_{a \mapsto 5} \\
= \;& 5 + f(4) \\
= \;& 5 + (n * n - 1)_{n \mapsto 4} \\
= \;& 5 + (4 * 4 - 1) \\
= \;& 5 + 15 \\
= \;& 20
\end{aligned}
$$

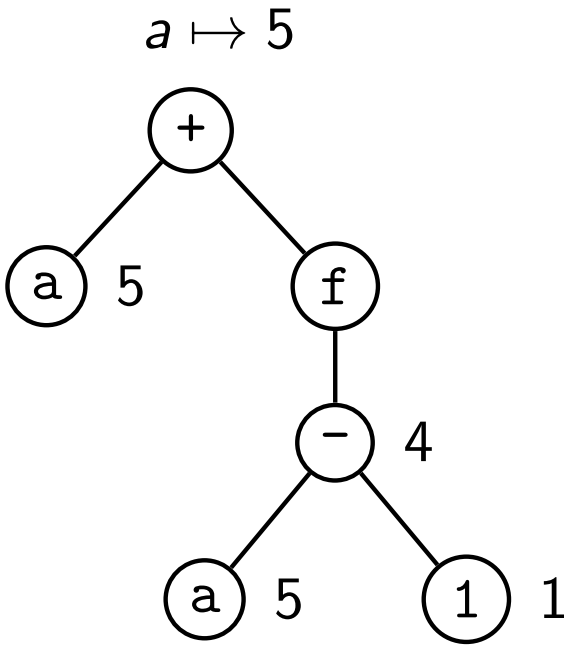
Semantik der Anwendung einer Methode

Beispiel: Auswertung durch Baumeinbettung

Definierte Methode

```
public static int f(int n) {
    return n * n - 1;
}
```

Auswertung des Ausdrucks `a + f(a - 1)` für $a = 5$.



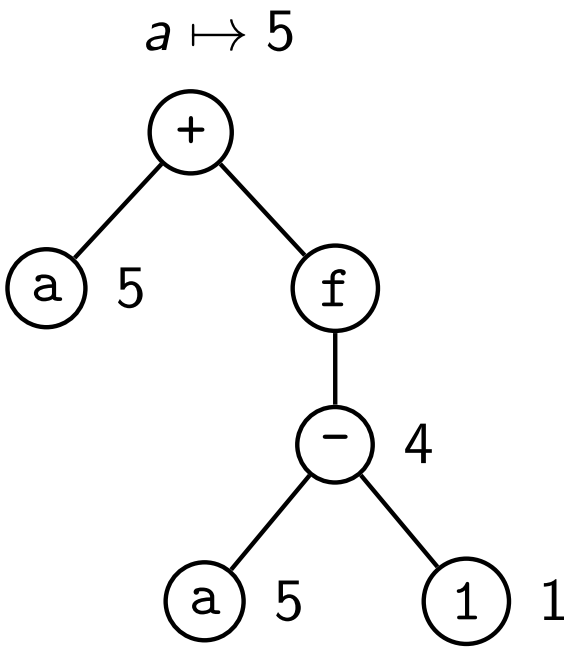
Semantik der Anwendung einer Methode

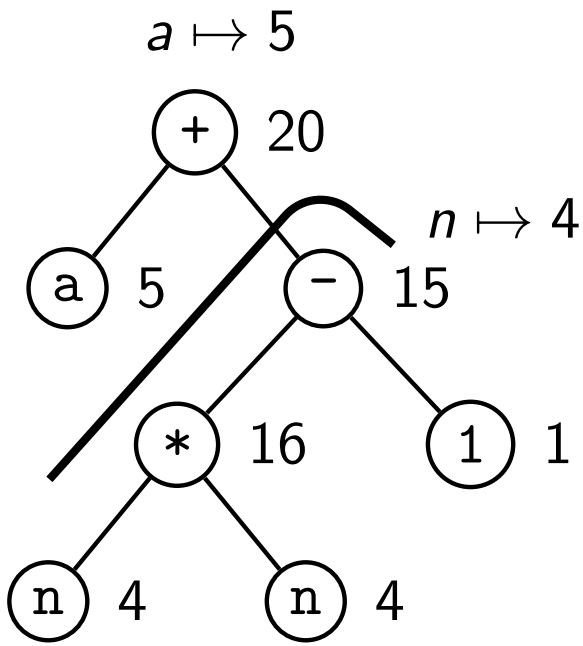
Beispiel: Auswertung durch Baumeinbettung

Definierte Methode

```
public static int f(int n) {
    return n * n - 1;
}
```

Auswertung des Ausdrucks `a + f(a - 1)` für $a = 5$.





Überladen von Methoden

- Im Allgemeinen gibt man *verschiedenen* Methoden *verschiedene* Namen.
- Haben verschiedene Methoden *ähnliche* Funktionalität, kann man sie unter bestimmten Voraussetzungen gleich benennen.
- *Überladen* von Methoden erlaubt, verschiedene Methoden einer Klasse *gleich* zu benennen.

Überladung

Voraussetzungen

Zwei Methoden einer Klasse dürfen gleich benannt sein, wenn die Listen ihrer Parameter*typen* ungleich sind. Die Listen sind ungleich, wenn eine der beiden Bedingungen vorliegt:

- Die Listen sind ungleich lang.
- Die Listen sind gleich lang, aber es gibt eine Position i, sodass der i-te Typ in der einen Liste ungleich dem i-ten Typ in der anderen Liste ist.

Für die Frage der Überladung spielen *keine* Rolle:

- Namen der formalen Parameter
- Ergebnistypen der Methoden

Dokumentation von Methoden

Javadoc

- Jede Methode einer Klasse (und auch die Klasse selbst) muss eine Dokumentation nach dem Javadoc-Standard besitzen.
- Ist dies der Fall, kann man mit Hilfe des Programms `javadoc`, das Teil des JDK ist, eine gut lesbare HTML-Dokumentation der Klasse erzeugen.
- Sinn dieser Dokumentation ist, *Anwendern* der Klasse zu erläutern, welche Funktion die Klasse und welche Funktion jede Methode der Klasse besitzt.
- Es geht bei der Dokumentation somit um *Anwendungsaspekte*, nicht um die interne technische Realisierung der Klasse und ihrer Methoden.

Dokumentation von Methoden

Schlüsselwörter

Innerhalb der Javadoc-Beschreibung einer Methode können Schlüsselwörter verwendet werden, die bei der Erzeugung der HTML-Dokumentation besonders berücksichtigt werden und dort zu speziellen Formatierungen führen.

@param Parametername	Beschreibung der Bedeutung eines formalen Parameters
@return	Beschreibung des Ergebniswerts

Dokumentation von Methoden

Beispiel aus der Klasse `Math`

```
/**
 * Returns the absolute value of an {@code int} value.
 * If the argument is not negative, the argument is returned.
 * If the argument is negative, the negation of the argument is returned.
 *
 * <p>Note that if the argument is equal to the value of
 * {@link Integer#MIN_VALUE}, the most negative representable
 * {@code int} value, the result is that same value, which is
 * negative.
 *
 * @param   a   the argument whose absolute value is to be determined
 * @return  the absolute value of the argument.
 */
public static int abs(int a) {

    return (a < 0) ? -a : a;
}
```

Praktisches Beispiel

Rechnen mit Uhrzeiten

In einer Klasse `Uhrzeit` sollen zwei Methoden realisiert werden, um mit Uhrzeiten (Stunde, Minute, Sekunde) zu rechnen.

- Zu einer Uhrzeit eine Anzahl Stunden, Minuten und Sekunden hinzuzählen.
- Differenz zwischen zwei Uhrzeiten in Sekunden ermitteln.

Die Uhrzeiten sollen durch ganze Zahlen im Format `SSMMss` repräsentiert werden.

Beispiele

- 23 Uhr, 45 Minuten und 7 Sekunden wird als 234507 dargestellt.
- 0 Uhr, 0 Minuten und 12 Sekunden wird z. B. als 12 dargestellt.

Wann schreibt man Methoden?

Ein paar Empfehlungen ...

Man schreibt Methode, um ...

einen Teil des Algorithmus als eigenständige „Berechnungseinheit“ auszugliedern (s. Uhrzeit-Beispiel `berechneSekSeitMitternacht`).

- Diese Einheit sollte auch *alleine* einen Sinn haben, der sich in einem passenden Methodennamen ausdrücken lässt.
- Stellen wir uns den Algorithmus als Tätigkeit vor, die wir ausführen sollen, dann ist diese Berechnungseinheit eine Unterstützung, die jemand *leisten kann* und die wir durch Aufruf der Methode *in Anspruch nehmen*.

  > *„Gib mir bitte für beide Zeiten die entsprechenden Sekunden seit Mitternacht, dann kann ich die Differenz der Zeiten berechnen.“*

- Die Berechnungseinheit realisiert i. d. R. *keine* „Trivialberechnung“ wie Addition zweier Argumente o. ä.

Wann schreibt man Methoden?

Ein paar Empfehlungen ...

Man schreibt Methode, um ...

doppelten Programmcode zu vermeiden (s. Uhrzeit-Beispiel `berechneSekSeitMitternacht`).

- Programmcode, dessen Dopplung vermieden wird, sollte eine sinnvolle eigenständige Berechnungseinheit darstellen, z. B. *Runden auf zwei Stellen* oder *Rechnungsbetrag für Grund- und Verbrauchspreis ermitteln*.
- Keine separaten Methoden, um Dopplungen von „Trivialberechnungen“ zu vermeiden.
- Ist Aufruf einer separaten Methode länger als der dadurch ersetzte Code, dann eher *keine* separate Methode (es sei denn, der Programmcode wird dadurch klarer).

Rekursiv definierte Methode

Definition (Rekursiv definierte Methode)

Eine Methode ist rekursiv definiert (oder einfach: eine Methode ist rekursiv), wenn ihr Rumpf *direkt* oder *indirekt* eine Anwendung derselben Methode enthält.

Direkte Rekursion Der Rumpf enthält *unmittelbar* eine Anwendung derselben Methode.

Indirekte Rekursion Der Rumpf enthält die Anwendung einer *anderen* Methode, die – auch über den Weg weiterer Methoden – einen Aufruf der Anfangsmethode enthält.

Rekursiv definierte Methode

Beispiel Fakultät

Definition (Fakultät)

Die Fakultät $n!$ einer natürlichen Zahl $n \in \mathbb{N}_0$ ist das Produkt aller natürlichen Zahlen von 1 bis n.

$$n! = 1 \cdot 2 \cdots n = \prod_{k=1}^{n} k$$

Die Fakultät von 0 ist nach dieser Definition als 1 definiert.

Aufgabe

Definiere eine Methode

```
public static long fakultaet(int n)
```

sodass durch den Aufruf `fakultaet(`n`)` die Fakultät von n berechnet wird.

Rekursiv definierte Methode

Beispiel Fakultät

Beobachtungen

- Je größer n, desto mehr Multiplikationen müssen ausgewertet werden, um die Fakultät von n zu berechnen.
- Eine Realisierung der Methode durch einen *festen* Multiplikatonsausdruck ist nicht möglich.
- Aber: Das Problem, $n! = 1 \cdot 2 \cdots (n-1) \cdot n$ zu berechnen, enthält ein *Teilproblem derselben Art*.

$$n! = \underbrace{1 \cdot 2 \cdots (n-1)}_{(n-1)!} \cdot n = (n-1)! \cdot n$$

- Da wir eine Methode `fakultaet` zur Berechnung der Fakultät haben (werden), können wir diese zur Lösung des Problems einsetzen.

Rekursiv definierte Methode

Beispiel Fakultät

Von der Beobachtung zur Idee der Realisierung

$$n! = \underbrace{1 \cdot 2 \cdots (n-1)}_{(n-1)!} \cdot n = (n-1)! \cdot n$$

```
public static long fakultaet(int n) {

    return fakultaet(n - 1) * n;
}
```

Aber:

Nach dieser Definition wäre `fakultaet(0)` nicht 1.
Wir müssen den Fall 0 separat berücksichtigen.

Rekursiv definierte Methode

Beispiel Fakultät

Methode `fakultaet`

Annahme: die Methode wird nicht auf negative Parameter angewendet.

```
public static long fakultaet(int n) {

    return n == 0
        ? 1
        : fakultaet(n - 1) * n;
}
```

Rekursive Methoden

Auswertung

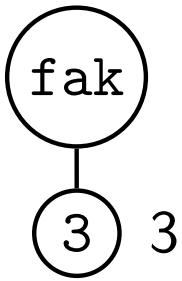
Auswertung rekursiv definierter Methoden

Die Auswertung von Anwendungen rekursiver Methoden erfolgt nach *denselben* Regeln wie bei nicht-rekursiven Methoden.
Das heißt, man kann sich die Semantik veranschaulichen durch

- Auswertung durch mehrere Bäume
- Auswertung durch Gleichungskette
- Auswertung durch Baumeinbettung, d. h. die „Baumvariante“ der Gleichungskette

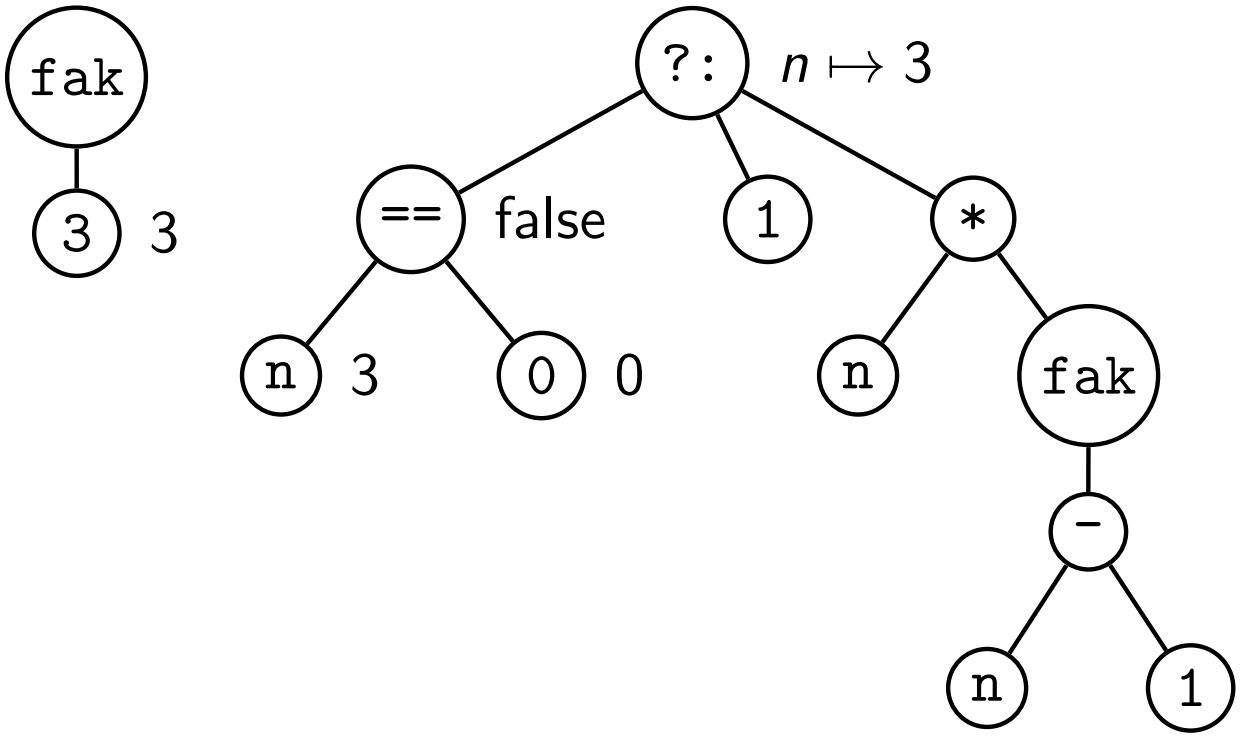
Rekursive Methoden

Auswertung am Beispiel der Fakultätberechnung *fakultaet*(3)



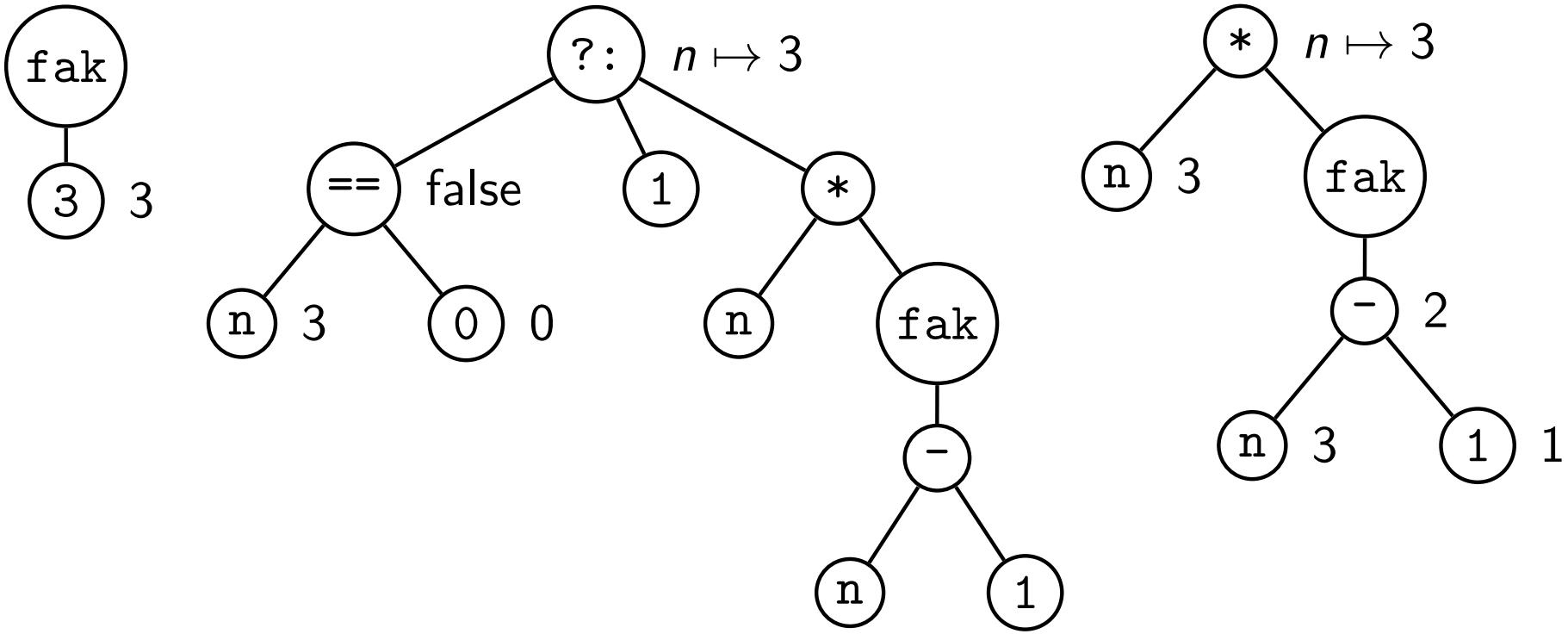
Rekursive Methoden

Auswertung am Beispiel der Fakultätberechnung *fakultaet*(3)



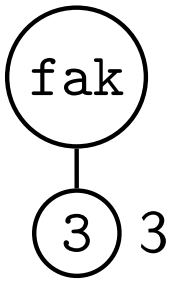
Rekursive Methoden

Auswertung am Beispiel der Fakultätberechnung *fakultaet*(3)



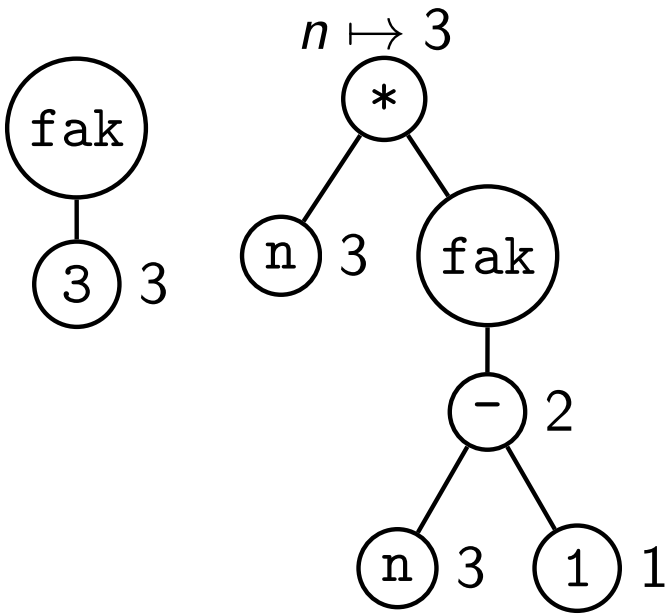
Rekursive Methoden

Auswertung von *fakultaet*(3), bedingte Ausdrücke aus Platzgründen weggelassen



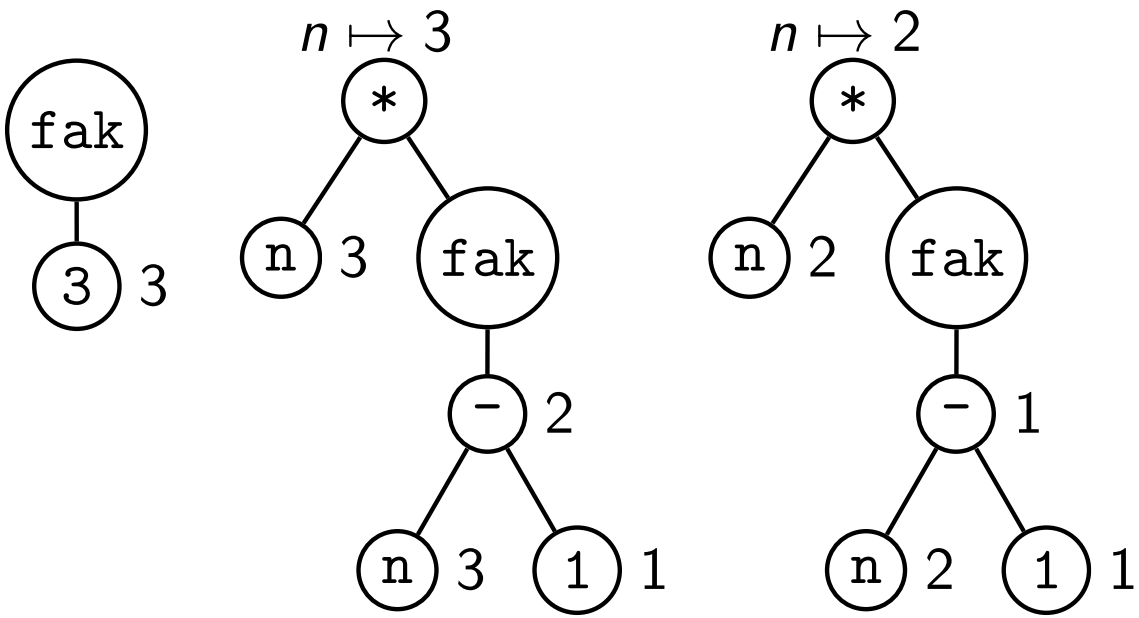
Rekursive Methoden

Auswertung von *fakultaet*(3), bedingte Ausdrücke aus Platzgründen weggelassen



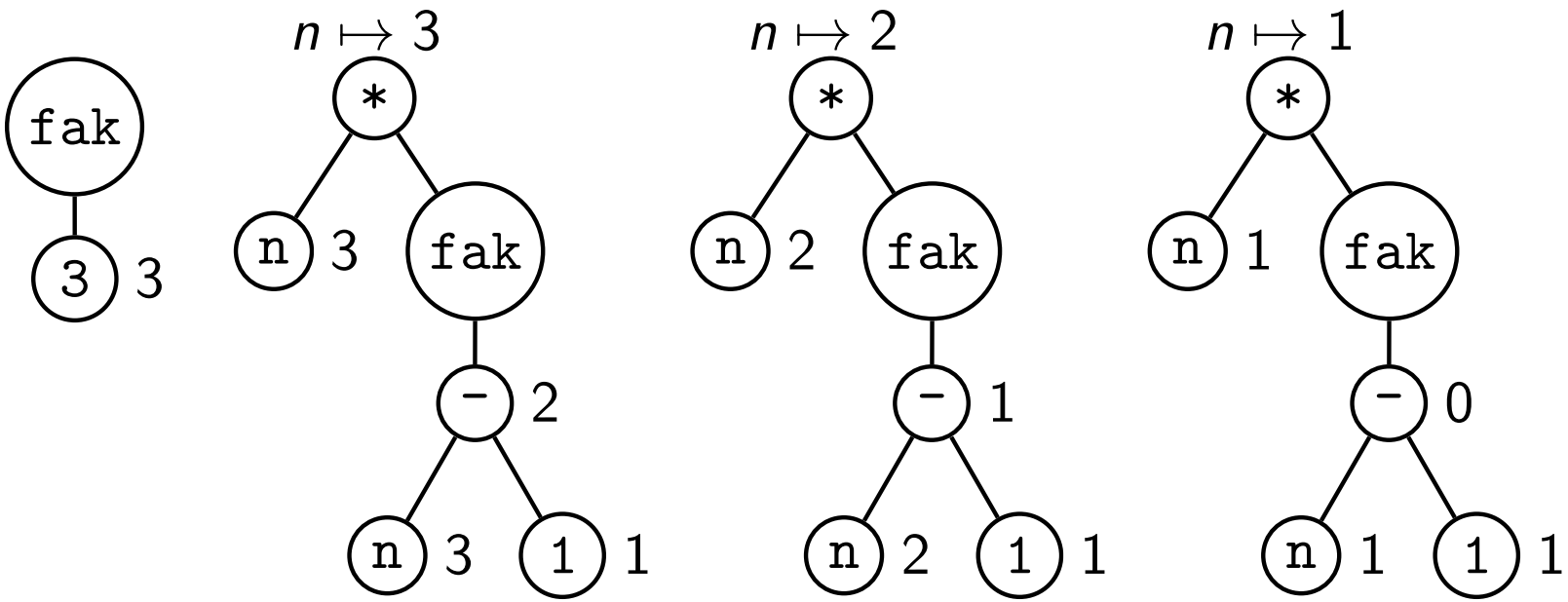
Rekursive Methoden

Auswertung von *fakultaet*(3), bedingte Ausdrücke aus Platzgründen weggelassen



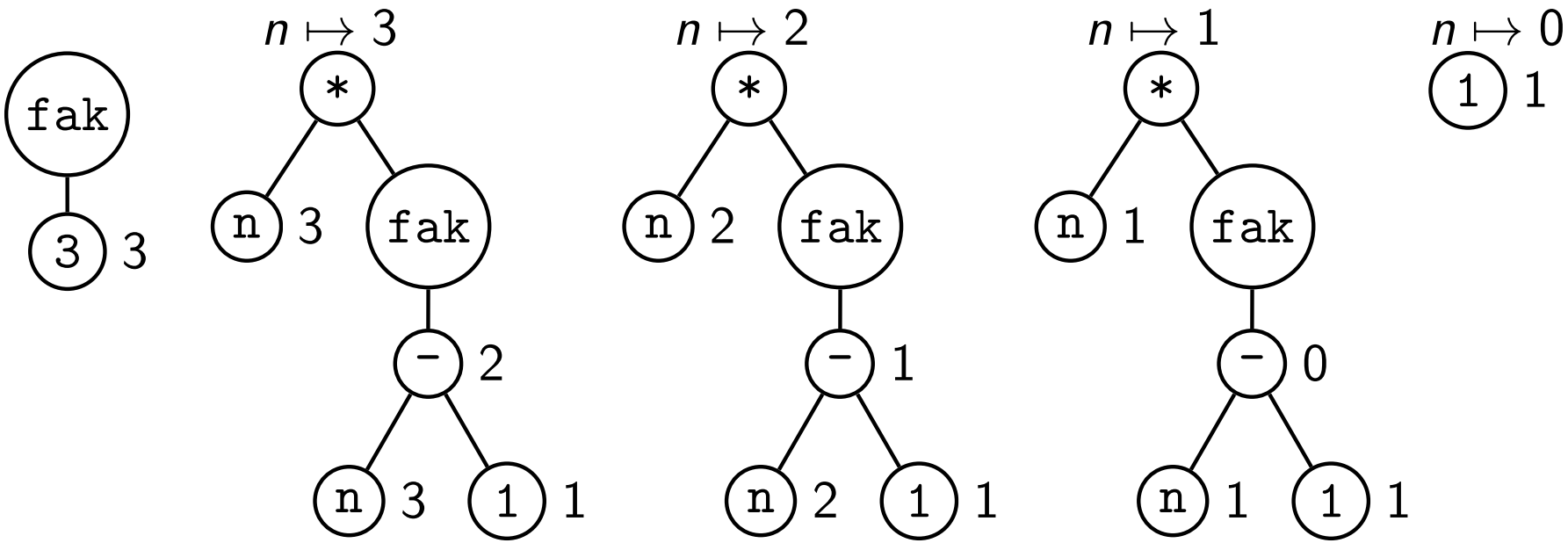
Rekursive Methoden

Auswertung von *fakultaet*(3), bedingte Ausdrücke aus Platzgründen weggelassen



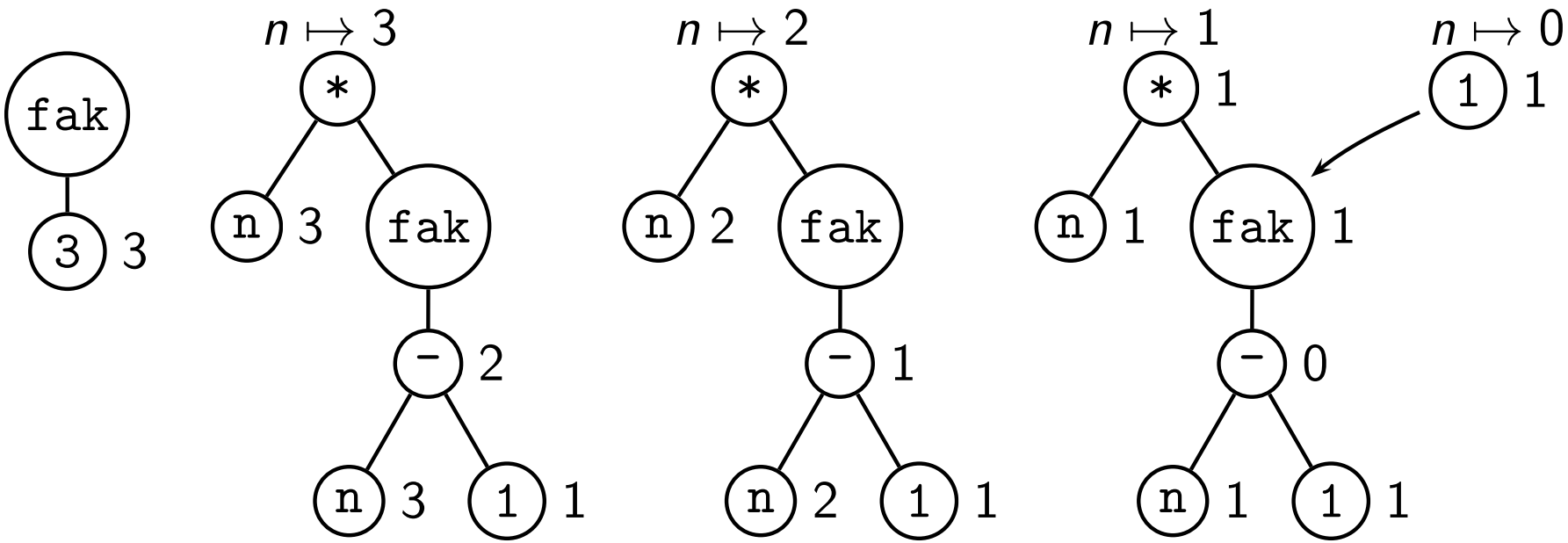
Rekursive Methoden

Auswertung von *fakultaet*(3), bedingte Ausdrücke aus Platzgründen weggelassen



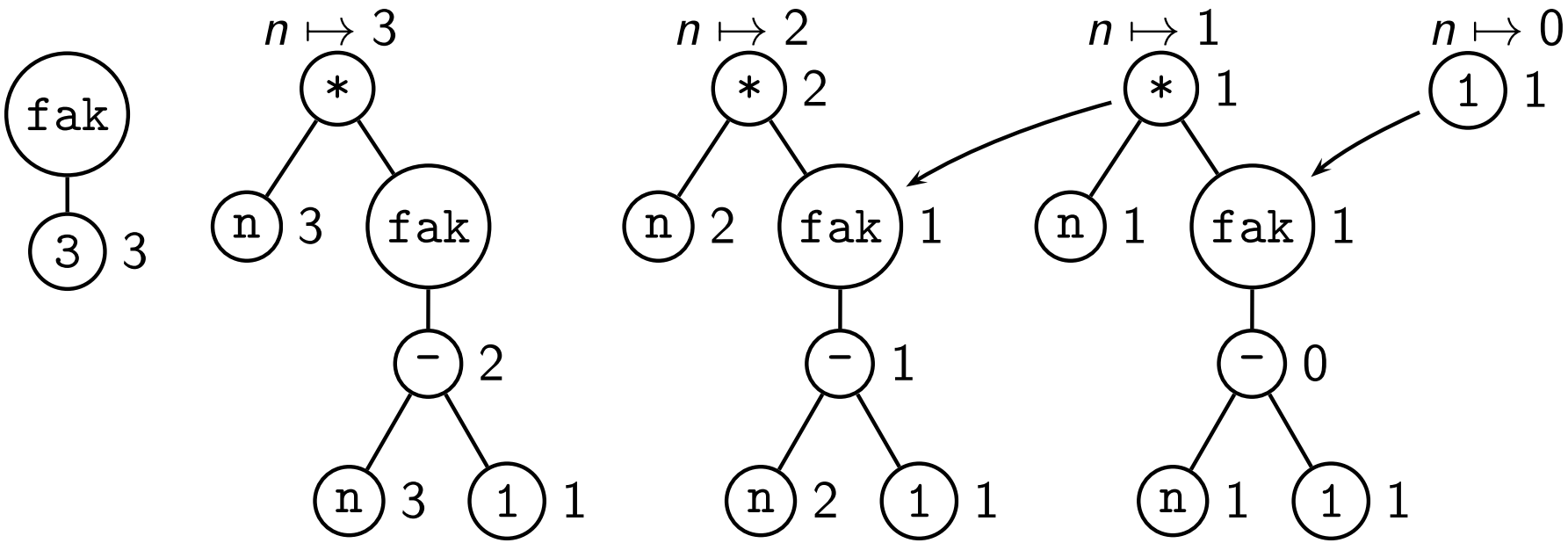
Rekursive Methoden

Auswertung von *fakultaet*(3), bedingte Ausdrücke aus Platzgründen weggelassen



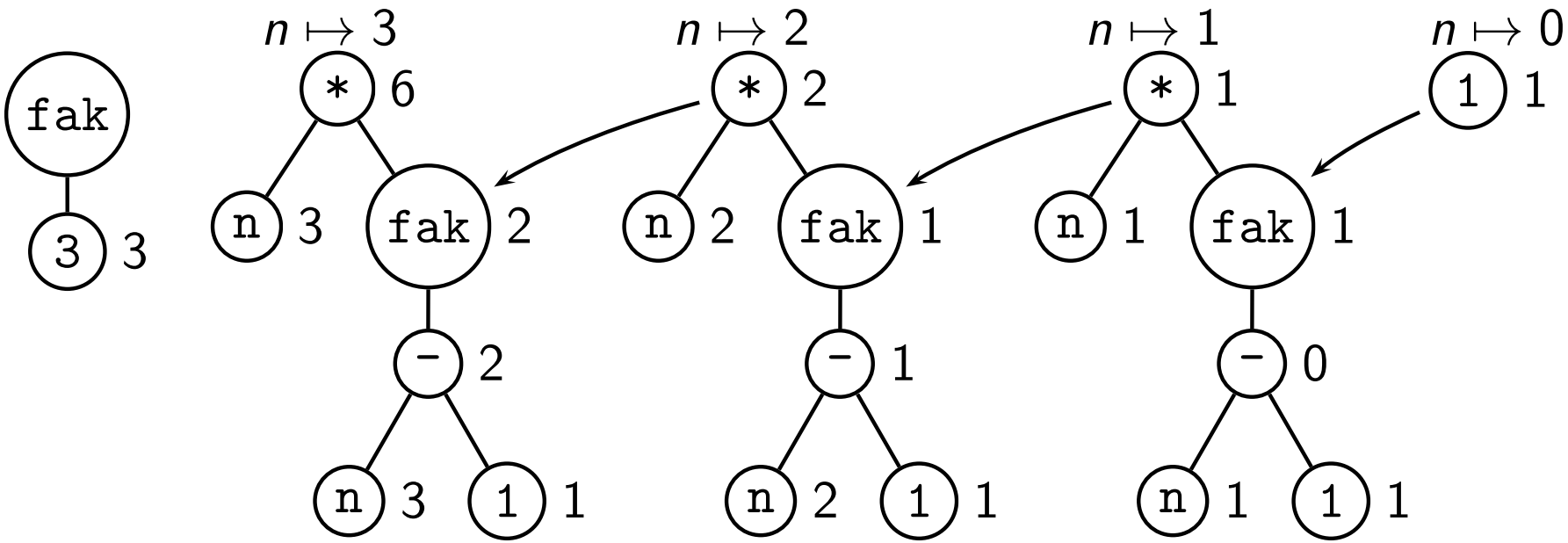
Rekursive Methoden

Auswertung von *fakultaet*(3), bedingte Ausdrücke aus Platzgründen weggelassen



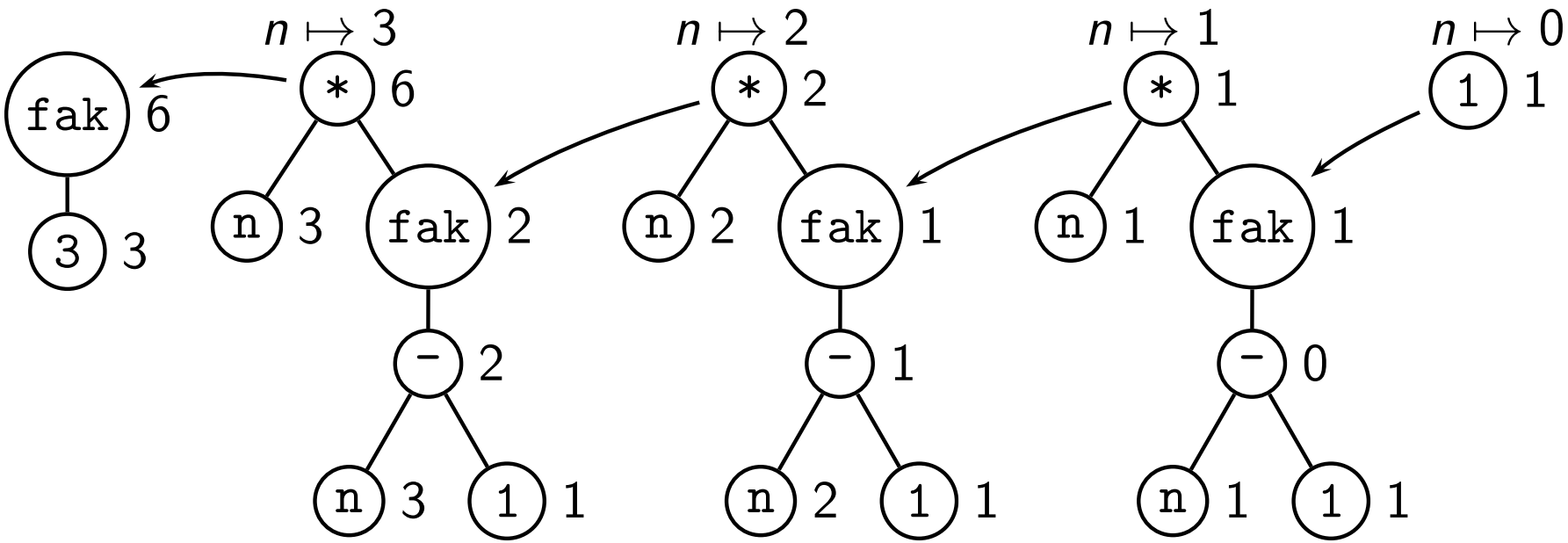
Rekursive Methoden

Auswertung von *fakultaet*(3), bedingte Ausdrücke aus Platzgründen weggelassen



Rekursive Methoden

Auswertung von *fakultaet*(3), bedingte Ausdrücke aus Platzgründen weggelassen



Rekursive Methoden

Auswertung durch Gleichungskette

$$\mathit{fakultaet}(3)$$

Rekursive Methoden

Auswertung durch Gleichungskette

$$
\begin{aligned}
& \mathit{fakultaet}(3) \\
= \; & \big(\texttt{(n == 0) ? 1 : n * fakultaet(n - 1)}\big)_{n \mapsto 3} \\
= \; & \big(\texttt{n * fakultaet(n - 1)}\big)_{n \mapsto 3}
\end{aligned}
$$

Rekursive Methoden

Auswertung durch Gleichungskette

$$
\begin{aligned}
& \mathit{fakultaet}(3) \\
= \; & (\texttt{(n == 0) ? 1 : n * fakultaet(n - 1)})_{n \mapsto 3} \\
= \; & (\texttt{n * fakultaet(n - 1)})_{n \mapsto 3} \\
= \; & 3 * \mathit{fakultaet}(2)
\end{aligned}
$$

Rekursive Methoden

Auswertung durch Gleichungskette

$$
\begin{aligned}
& \mathit{fakultaet}(3) \\
= \; & (\texttt{(n == 0) ? 1 : n * fakultaet(n - 1)})_{n \mapsto 3} \\
= \; & (\texttt{n * fakultaet(n - 1)})_{n \mapsto 3} \\
= \; & 3 * \mathit{fakultaet}(2) \\
= \; & 3 * (\texttt{n * fakultaet(n - 1)})_{n \mapsto 2} \\
= \; & 3 * (2 * \mathit{fakultaet}(1))
\end{aligned}
$$

Rekursive Methoden

Auswertung durch Gleichungskette

$$
\begin{aligned}
& \mathit{fakultaet}(3) \\
= \; & (\texttt{(n == 0) ? 1 : n * fakultaet(n - 1)})_{n \mapsto 3} \\
= \; & (\texttt{n * fakultaet(n - 1)})_{n \mapsto 3} \\
= \; & 3 * \mathit{fakultaet}(2) \\
= \; & 3 * (\texttt{n * fakultaet(n - 1)})_{n \mapsto 2} \\
= \; & 3 * (2 * \mathit{fakultaet}(1)) \\
= \; & 3 * (2 * (\texttt{n * fakultaet(n - 1)})_{n \mapsto 1}) \\
= \; & 3 * (2 * (1 * \mathit{fakultaet}(0)))
\end{aligned}
$$

Rekursive Methoden

Auswertung durch Gleichungskette

$$
\begin{aligned}
& \mathit{fakultaet}(3) \\
= \; & (\texttt{(n == 0) ? 1 : n * fakultaet(n - 1)})_{n \mapsto 3} \\
= \; & (\texttt{n * fakultaet(n - 1)})_{n \mapsto 3} \\
= \; & 3 * \mathit{fakultaet}(2) \\
= \; & 3 * (\texttt{n * fakultaet(n - 1)})_{n \mapsto 2} \\
= \; & 3 * (2 * \mathit{fakultaet}(1)) \\
= \; & 3 * (2 * (\texttt{n * fakultaet(n - 1)})_{n \mapsto 1}) \\
= \; & 3 * (2 * (1 * \mathit{fakultaet}(0))) \\
= \; & 3 * (2 * (1 * (\texttt{(n == 0) ? 1 : n * fakultaet(n - 1)})_{n \mapsto 0})) \\
= \; & 3 * (2 * (1 * 1))
\end{aligned}
$$

Rekursive Methoden

Auswertung durch Gleichungskette

$$
\begin{aligned}
& \mathit{fakultaet}(3) \\
= \; & (\texttt{(n == 0) ? 1 : n * fakultaet(n - 1)})_{n \mapsto 3} \\
= \; & (\texttt{n * fakultaet(n - 1)})_{n \mapsto 3} \\
= \; & 3 * \mathit{fakultaet}(2) \\
= \; & 3 * (\texttt{n * fakultaet(n - 1)})_{n \mapsto 2} \\
= \; & 3 * (2 * \mathit{fakultaet}(1)) \\
= \; & 3 * (2 * (\texttt{n * fakultaet(n - 1)})_{n \mapsto 1}) \\
= \; & 3 * (2 * (1 * \mathit{fakultaet}(0))) \\
= \; & 3 * (2 * (1 * (\texttt{(n == 0) ? 1 : n * fakultaet(n - 1)})_{n \mapsto 0})) \\
= \; & 3 * (2 * (1 * 1)) \\
= \; & 6
\end{aligned}
$$

Vom Problem zur rekursiven Lösung

Was macht eine rekursive Methode aus?

Eine rekursive Methode stützt sich (direkt oder indirekt) auf sich selbst ab.
Sie macht das jedoch nicht in *jedem* Fall, denn sonst würde die Auswertung nicht terminieren.
Es gibt in einer rekursiven Methode deshalb in der Regel

1. mindestens einen nicht-rekursiven Fall, für den die Auswertung direkt terminiert, z. B.

$$fakultaet(0) = 1$$

2. mindestens einen rekursiven Fall, für den sich die Berechnung auf die Lösung eines „einfacheren“ oder „kleineren“ Falls abstützt, z. B.

$$fakultaet(n) = n * fakultaet(n-1)$$

Vom Problem zur rekursiven Lösung

Fragen, die Sie zur rekursiven Lösung führen

Für die rekursive Lösung einer gegebenen Art von Problemen müssen Sie die rekursiven und nicht-rekursiven Fälle identifizieren.
Stellen Sie sich dazu folgende Fragen:

1. Was ist ein *einfaches* Problem dieser Art, und wie kann ich es „direkt“ lösen?
2. Wenn das Problem *nicht einfach* ist:

   Gibt es darin ein *kleineres* Problem *derselben* Art, sodass ich mit dessen Lösung – wenn sie mir jemand gäbe – und nur noch *wenig zusätzlichem Aufwand* das ursprüngliche Problem lösen kann?

   Benötigt wird eine „Regel“, damit die Lösung eines gegebenen Problems auf die Lösung eines kleineren Problems derselben Art zurückgeführt werden kann.

Vom Problem zur rekursiven Lösung

Fragen, die Sie zur rekursiven Lösung führen (Beispiel Fakultät)

Frage: Was ist ein *einfaches* Fakultät-Problem, und wie kann ich es „direkt“ lösen?

Antwort: Die Berechnung der Fakultät von 0 ist einfach. Die Fakultät von 0 ist 1.

Vom Problem zur rekursiven Lösung

Fragen, die Sie zur rekursiven Lösung führen (Beispiel Fakultät)

Die Berechnung der Fakultät von $n > 0$ ist *nicht einfach*. (Jedenfalls nicht so, dass man i. Allg. die Lösung ohne Rechnung direkt angeben kann.)

Frage: Gibt es in diesem Problem ein *kleineres* Fakultät-Problem, sodass ich mit dessen Lösung – wenn sie mir jemand gäbe – und nur noch *wenig zusätzlichem Aufwand* die Fakultät von n berechnen kann?

Antwort: Ja, die Berechnung der Fakultät von n enthält als Teilproblem die Berechnung der Fakultät von $n-1$. Hätte ich diese Lösung, müsste ich sie nur noch mit n multiplizieren.

Vom Problem zur rekursiven Lösung

Fragen, die Sie zur rekursiven Lösung führen (Beispiel Fakultät)

Die Berechnung der Fakultät von $n > 0$ ist *nicht einfach*. (Jedenfalls nicht so, dass man i. Allg. die Lösung ohne Rechnung direkt angeben kann.)

Frage: Gibt es in diesem Problem ein *kleineres* Fakultät-Problem, sodass ich mit dessen Lösung – wenn sie mir jemand gäbe – und nur noch *wenig zusätzlichem Aufwand* die Fakultät von n berechnen kann?

Antwort: Ja, die Berechnung der Fakultät von n enthält als Teilproblem die Berechnung der Fakultät von $n-1$. Hätte ich diese Lösung, müsste ich sie nur noch mit n multiplizieren.

... und wer gibt mir die Lösung des Teilproblems?

Ich erhalte sie durch Anwendung der eigenen Methode.

Vom Problem zur rekursiven Lösung

Arten der Rekursion

- Abhängig von den Antworten auf diese Fragen gelangen Sie zu rekursiven Lösungen, die bestimmten Schemas folgen und jeweils charakteristische Eigenschaften besitzen.
- Wir behandeln folgende Schemas:
  - Lineare Rekursion
  - Endrekursion (iterative Rekursion, engl. *tail recursion*)
  - Baumrekursion

Lineare Rekursion

Definition (Lineare Rekursion)

Eine Methode heißt linear rekursiv, wenn sie in jedem Zweig ihres bedingten Ausdrucks höchstens *eine* Anwendung derselben Methode enthält.

Beispiel

Ein Beispiel ist die oben definierte Methode `fakultaet`.

Lineare Rekursion

Beispiel Quersumme

Problem

Gegeben: natürliche Zahl

Gesucht: Quersumme dieser Zahl (bezogen auf die Dezimaldarstellung)

Die zwei Fragen, deren Antworten uns zur rekursiven Lösung führen

1. Was ist ein *einfaches* Quersummen-Problem, und wie kann ich es „direkt“ lösen?
2. Wenn das Problem *nicht einfach* ist: Gibt es darin ein *kleineres* Quersummen-Problem, sodass ich mit dessen Lösung – wenn sie mir jemand gäbe – und nur noch *wenig zusätzlichem Aufwand* das ursprüngliche Quersummen-Problem lösen kann?

Lineare Rekursion

Beispiel Quersumme

Was ist ein *einfaches* Quersummen-Problem, und wie kann ich es „direkt“ lösen?

Für einstellige Zahlen ist das Quersummen-Problem einfach. Die Quersumme ist die Zahl selbst.

Lineare Rekursion

Beispiel Quersumme

Wenn das Problem *nicht einfach* ist: ...

1. Ein nicht-einfaches Quersummen-Problem ist die Berechnung der Quersumme einer mehrstelligen Zahl, z. B. 4713.
2. Gibt es darin ein *kleineres* Quersummen-Problem, sodass ich mit dessen Lösung – wenn sie mir jemand gäbe – und nur noch *wenig zusätzlichem Aufwand* das ursprüngliche Quersummen-Problem lösen kann?
   - Quersumme von 713; mit ihr müsste ich nur noch 4 hinzuaddieren.
   - Quersumme von 471; mit ihr müsste ich nur noch 3 hinzuaddieren.
3. Wir entscheiden uns für das zweite Teilproblem, denn das Abschneiden der letzten Ziffer, um von 4713 zu 471 zu kommen, geht unabhängig von der Länge der Zahl immer durch Division mit 10.
4. Jetzt nicht darüber rätseln, wie man an die Quersumme von 471 kommt. (Die erhalten wir durch Anwendung unserer gesuchten Methode.)

Lineare Rekursion

Beispiel Quersumme

Die Antworten auf die Fragen als Java-Methode

```
public static int quersumme(int a) {
    return (a < 10) ? a : quersumme(a / 10) + a % 10;
}
```

- `a < 10` unterscheidet zwischen einfachem und nicht-einfachem Fall.
- Ja-Zweig des bedingten Ausdrucks ist Lösung des einfachen Problems.
- `a / 10` berechnet zu nicht-einfachem Problem das etwas kleinere Teilproblem.
- `quersumme(a / 10)` löst dieses kleinere Problem.
- `+ a % 10` macht aus Lösung des kleineren Problems die Lösung des ursprünglichen Problems.

Lineare Rekursion

Beispiel Summe ganzer Zahlen

Problem

Gegeben: ganze Zahlen m und n

Gesucht: Summe aller ganzen Zahlen von m bis n

Die zwei Fragen, deren Antworten uns zur rekursiven Lösung führen

1. Was ist ein *einfaches* Summen-Problem, und wie kann ich es „direkt“ lösen?
2. Wenn das Problem *nicht einfach* ist:
   Gibt es darin ein *kleineres* Summen-Problem, sodass ich mit dessen Lösung – wenn sie mir jemand gäbe – und nur noch *wenig zusätzlichem Aufwand* das ursprüngliche Summen-Problem lösen kann?

Endrekursion

Einführung

- Durch linear rekursive Methode `quersumme` wird geschachtelter Additionsausdruck aufgebaut.
- Dieser Ausdruck kann erst nach Erreichen des nicht-rekursiven Falls ausgewertet werden.
- Endrekursiv definierte Methoden führen einen Teil der Berechnung bei jedem rekursiven Aufruf durch. Bei Erreichen des Rekursionsendes ist das Ergebnis schon „fertig“.

Endrekursion

Einführung

Anderes Schema zur Berechnung der Quersumme:

Quersumme von	schon addierte Ziffernwerte
2031	0
203	1
20	4
2	4
0	6

- Zahl, deren Quersumme berechnet werden soll, wird fortlaufend „verkürzt“.
- Gleichzeitig werden Werte der „abgeschnittenen“ Ziffern aufaddiert.
- Wird Zahl 0 erreicht, ist „Zwischensumme“ die gesuchte Quersumme.

Endrekursion

Berechnungsschema als Methode

Dieses Berechnungsschema lässt sich rekursiv ausdrücken:

```
public static int quersumme(int a, int summme) {
    return (a == 0)
           ? summe
           : quersumme(a / 10, summe + a % 10);
}
```

- Berechnung der Quersumme einer Zahl *a* erfolgt durch `quersumme(a, 0)`.
- Bei `quersumme` handelt es sich um Spezialfall der linearen Rekursion, die sogenannte *Endrekursion* (engl. *tail recursion*).

Endrekursion

Definition

Eine linear rekursive Methode heißt *endrekursiv*, wenn die Anwendung derselben Methode in den Zweigen des bedingten Ausdrucks die *letzte* Operation ist (deshalb die Bezeichnung *End*rekursion).

- Endrekursive Methoden können in imperativer Programmierung durch Schleifenanweisungen (Iterationen) realisiert werden.
- Endrekursion wird auch *iterative* Rekursion genannt.

Endrekursion

Sprachregelung

- Endrekursion ist Spezialfall der linearen Rekursion.
- Zur leichteren Unterscheidung bezeichnen wir mit *linear rekursiv* künftig nur Methoden, die nicht *gleichzeitig* auch endrekursiv sind.

Endrekursion

Vom Problem zur endrekursiven Lösung

- Gegeben sei ein Problem einer bestimmten Art.
- Überlegen Sie, wie Sie das Problem auf systematische Art „tabellarisch“ lösen können, sodass
  - jede Zeile der Tabelle einer Anwendung der endrekursiven Methode entspricht,
  - die Werte jeder Zeile den Werten der Parametervariablen entsprechen,
  - und sich aus der letzten Zeile das Ergebnis ermitteln lässt.
- Häufig benötigen Sie bei einer endrekursiven Lösung zusätzliche Parameter für „Zwischenwerte“.

Endrekursion

Beispiel Primzahltest

Problem:

Gegeben:	eine natürliche Zahl
Gesucht:	Ist dies eine Primzahl?

Tabelle mit drei Spalten ebnet Weg zur endrekursiven Lösung:

n:	Zahl, deren Primzahleigenschaft getestet wird
teiler:	möglicher Teiler von n
teiler gefunden?	gibt an, ob bisher schon ein Teiler von n gefunden wurde

Endrekursion

Berechnungsablauf Primzahltest

Berechnungsablauf für Zahl, die keine Primzahl ist:

n	teiler	teiler gefunden?
35	2	nein
35	3	nein
35	4	nein
35	5	nein
35	6	ja
...	...	...
35	35	ja

Berechnungsablauf für Primzahl:

n	teiler	teiler gefunden?
17	2	nein
17	3	nein
...	...	...
17	17	nein

Endrekursion

Methode zum Primzahltest

```
public static boolean istPrimzahl(int n, int teiler,
                                  boolean teilerGefunden) {

    return (teiler == n)
           ? !teilerGefunden
           : (n % teiler == 0)
             ? istPrimzahl(n, teiler + 1, true)
             : istPrimzahl(n, teiler + 1, teilerGefunden);
}
```

Endrekursion

Verbesserung des Berechnungsablaufs

- Wird Teiler gefunden, müssen weitere mögliche Teiler nicht mehr getestet werden.
- Damit entfällt Notwendigkeit für Parameter „teiler gefunden?“.
- Es müssen keine Teiler größer als $\sqrt{n}$ getestet werden. Auf diese Verbesserung verzichten wir, damit der Programmcode übersichtlicher bleibt.

n	teiler
35	2
35	3
35	4
35	5

n	teiler
17	2
17	3
...	...
17	17

Endrekursion

Verbesserte Methode

```
public static boolean istPrimzahl(int n, int teiler) {

    return (teiler == n)
           ? true
           : (n % teiler == 0)
             ? false
             : istPrimzahl(n, teiler + 1);
}
```

Oder noch kürzer so:

```
public static boolean istPrimzahl(int n, int teiler) {

    return (teiler == n)
           || (n % teiler != 0) && istPrimzahl(n, teiler + 1);
}
```

Baumrekursion

Definition (Baumrekursion)

Eine Methode heißt baumrekursiv, wenn ein Zweig des bedingten Ausdrucks mindestens zwei Anwendungen derselben Methode enthält.

Baumrekursion

Fibonacci-Funktion

Mathematische Definition

Für alle $n \in \mathbb{N}_0$ gilt:

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

Realisierung durch baumrekursive Methode

```
public static int fib(int n) {
    return (n <= 1)
           ? n
           : fib(n - 1) + fib(n - 2);
}
```

Baumrekursion

Auswertung baumrekursiver Methoden

„Form“ des Ausdrucks bei Auswertung

- Die Auswertung einer *baumrekursiven* Methode führt zu einem „breit verzweigten“ Ausdruck, da jede Anwendung der Methode zwei oder mehr Anwendungen derselben Methode nach sich zieht.
- Zum Vergleich: Bei Auswertung einer *linear rekursiven* Methode entsteht ein „linearer“ Ausdruck, da jede Anwendung der Methode höchstens *eine* weitere Anwendung dieser Methode nach sich zieht.

Baumrekursion

Auswertung baumrekursiver Methoden

Konsequenzen für den Berechnungsaufwand

- Der breit verzweigte Ausdruck kann einen hohen (z. B. exponentiellen) Berechnungsaufwand bedeuten.
  - Der exponentielle Berechnungsaufwand kann unvermeidbar sein (s. Türme von Hanoi).
  - Der exponentielle Berechnungsaufwand kann durch eine andere Art der Realisierung vermeidbar sein (s. Übung zu Fibonacci).
- Trotz der breiten Verzweigung kann der Berechnungsaufwand „vergleichsweise gering“ bleiben (s. Übung zu Sortieren durch Mischen).

Baumrekursion

Türme von Hanoi

Aufgabenstellung

- Es gibt drei *Plätze*.
- Auf einem Platz liegen übereinander n Scheiben unterschiedlicher Größe. Eine größere Scheibe liegt *immer* unterhalb einer kleineren. Die anderen beiden Plätze sind frei.
- Ziel des Spiels ist es, alle Scheiben auf einen anderen Platz zu bewegen. Es darf stets nur eine Scheibe gleichzeitig bewegt werden und niemals eine größere Scheibe auf einer kleineren liegen.
- Gesucht ist eine Methode, die alle notwendigen Einzelschritte berechnet, um n Scheiben vom Platz *start* zu einem Platz *ziel* zu bewegen.

Baumrekursion

Türme von Hanoi

Realisierung durch baumrekursive Methode

```
public static String bewegeScheiben(int anzahl,
        int start, int ziel) {

    int zwischenplatz = 6 - start - ziel;
    return anzahl == 0
           ? ""
           : bewegeScheiben(anzahl - 1, start, zwischenplatz)
             + "Bewege Scheibe von " + start
             + " nach " + ziel + "\n"
             + bewegeScheiben(anzahl - 1, zwischenplatz, ziel);
}
```

Baumrekursion

Baum aus Ziffernfolge generieren

Aufgabenstellung

- Die Ziffern einer ganzen Zahl sollen als Baum angeordnet werden.
- Die mittlere Ziffer wird die Wurzel. Ist die Anzahl der Ziffern geradzahlig und gibt es somit keine mittlere Ziffer, wird die Ziffer links der Mitte zur Wurzel.
- Die Ziffern links davon werden nach gleicher Art zum linken Teilbaum, die Ziffern rechts davon zum rechten Teilbaum.
- Teilbäume werden durch Klammern eingeschlossen und der resultierende Baum als Zeichenkette repräsentiert.

Baumrekursion

Baum aus Ziffernfolge generieren

Beispiel

Zahl	Baum
1	`1`
12	`1(2)`
123	`(1)2(3)`
1234567	`((1)2(3))4((5)6(7))`
12345678	`((1)2(3))4((5)6(7(8)))`
123456789	`((1)2(3(4)))5((6)7(8(9)))`