3 Rekursion

In der Natur und in der Mathematik findet man das Thema ***Selbstähnlichkeit*** bzw. sich wiederholende Strukturen, etwa für Schneeflocken oder Fraktale und Julia-Mengen, die interessante grafische Gebilde sind. Man spricht in diesem Zusammenhang von ***Rekursion***, und meint damit, dass Dinge sich wiederholen oder ähneln. Bezogen auf Methoden bedeutet dies, dass diese sich selbst aufrufen. Wichtig dabei ist eine Abbruchbedingung in Form spezieller Eingabewerte, die zum Ende der Selbstaufrufe führt.

3.1 Einführung

Diverse Berechnungen lassen sich hervorragend als rekursive Funktion beschreiben. Ziel dabei ist es, einen komplexeren Sachverhalt in mehrere einfachere Teilaufgabenstellungen herunterzubrechen.

3.1.1 Mathematische Beispiele

Nachfolgend schauen wir uns mit der Fakultät, der Summenbildung und den Fibonacci-Zahlen drei einführende Beispiele zur rekursiven Definition an.

Beispiel 1: Fakultät

Mathematisch ist die ***Fakultät*** für eine positive Zahl n als das Produkt (also die Multiplikation) aller natürlichen Zahlen von 1 bis einschließlich n definiert. Zur Notation wird das Ausrufezeichen der entsprechenden Zahl nachgestellt. Beispielsweise steht 5! für die Fakultät der Zahl 5:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Dies lässt sich wie folgt verallgemeinern:

$$n! = n * (n-1) * (n-2) * ... * 2 * 1$$

Basierend darauf ergibt sich die rekursive Definition:

$$n! = \begin{cases} 1, & n = 0,\ n = 1 \\ n \cdot (n-1)!, & \forall n > 1 \end{cases}$$

Dabei steht das umgedrehte »A« ($\forall$) für »für alle« .

Es ergibt sich für die ersten n folgender Werteverlauf:

n	1	2	3	4	5	6	7	8
n!	1	2	6	24	120	720	5040	40320

Berechnung der Fakultät in Java Schauen wir uns kurz an, wie sich die rekursive Berechnungsvorschrift der Fakultät in eine ebensolche Methode transferieren lässt:

```
public static int factorial(final int n)
{
    if (n < 0)
        throw new IllegalArgumentException("n must be >= 0");

    // rekursiver Abbruch
    if (n == 0 || n == 1)
        return 1;

    // rekursiver Abstieg
    return n * factorial(n - 1);
}
```

Verdeutlichen wir uns, was diese rekursive Definition an Aufrufen erzeugt:

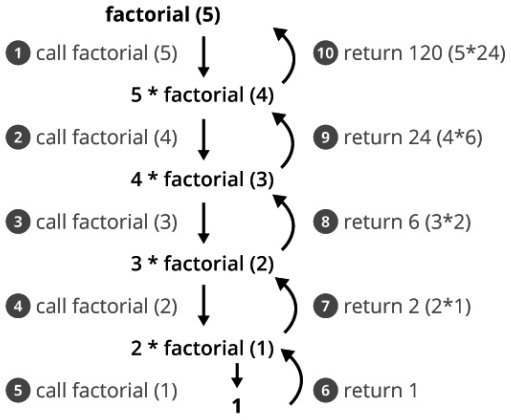

Abbildung 3-1 *Rekursive Aufrufe bei* `factorial(5)`

Beispiel 2: Berechnung der Summe der Zahlen bis n

Mathematisch ist die ***Summe*** für eine Zahl n als die Addition aller natürlichen Zahlen von 1 aufsteigend bis einschließlich n definiert:

$$\sum_{1}^{n} i = n + n - 1 + n - 2 + ... + 2 + 1$$

Das kann man folgendermaßen rekursiv definieren:

$$\sum_{1}^{n} i = \begin{cases} 1, & n = 1 \\ n + \sum_{1}^{n-1} i, & \forall n > 1 \end{cases}$$

Es ergibt sich für die ersten n folgender Werteverlauf:

n	1	2	3	4	5	6	7	8
sum(n)	1	3	6	10	15	21	28	36

Berechnung der Summe in Java Erneut überführen wir die rekursive Berechnungsvorschrift der Summation in eine rekursive Methode:

```
public static int sum(final int n)
{
    if (n <= 0)
        throw new IllegalArgumentException("n must be >= 1");

    // rekursiver Abbruch
    if (n == 1)
        return 1;

    // rekursiver Abstieg
    return n + sum(n - 1);
}
```

Achtung: Beschränkte Aufruftiefe

Bitte bedenken Sie, dass zur Summenbildung immer wieder Selbstaufrufe geschehen. Deswegen kann man hier nur einen Wert um die 10.000 – 20.000 übergeben. Bei größeren Werten kommt es zu einem `StackOverflowError`. Für andere rekursive Methoden gelten ähnliche Beschränkungen bezüglich der Anzahl an Selbstaufrufen.

Beispiel 3: Fibonacci-Zahlen

Auch die ***Fibonacci-Zahlen*** lassen sich hervorragend rekursiv definieren, wobei die Formel schon ein klein wenig komplexer ist:

$$fib(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ fib(n-1) + fib(n-2), & \forall n > 2 \end{cases}$$

Es ergibt sich für die ersten n folgender Werteverlauf:

n	1	2	3	4	5	6	7	8
fib(n)	1	1	2	3	5	8	13	21

Wenn man sich die Berechnungsvorschrift grafisch verdeutlicht, dann wird schnell klar, wie weit sich der Baum der Selbstaufrufe potenziell aufspannt – für größere n wäre der Aufrufbaum viel ausladender, wie es durch die gestrichelten Pfeile angedeutet ist:

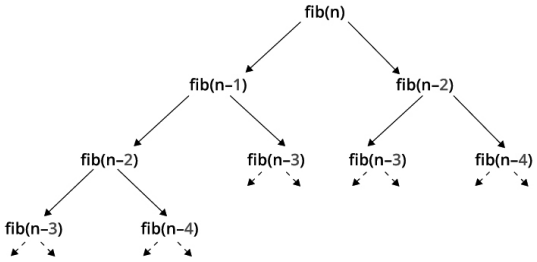

Abbildung 3-2 *Fibonacci rekursiv*

Selbst bei diesem exemplarischen Aufruf erkennt man, dass diverse Aufrufe mehrmals erfolgen, etwa für $fib(n-4)$ und $fib(n-2)$, aber insbesondere dreimal für $fib(n-3)$. Das führt sehr schnell zu aufwendigen und langwierigen Berechnungen. Wie wir dies optimieren können, lernen wir später in Abschnitt 8.1 kennen.

Tipp: Abweichende Definition mit null als Startwert

Es sei noch angemerkt, dass es eine Abwandlung gibt, die beim Wert 0 startet. Dann gilt $fib(0) = 0$ und $fib(1) = 1$ und danach gemäß der rekursiven Definition $fib(n) = fib(n-1) + fib(n-2)$. Dies produziert die gleiche Zahlenfolge wie die obige Definition, nur um den Wert für die 0 ergänzt.

3.1.2 Algorithmische Beispiele

Zur Einführung haben wir uns mathematische Beispiele angeschaut. Darüber hinaus eignet sich Rekursion aber auch sehr gut für algorithmische Aufgabenstellungen, beispielsweise um für ein Array zu prüfen, ob die dort hinterlegten Werte ein Palindrom bilden. Unter einem Palindrom versteht man ein Wort, das sich von vorne und hinten gleich liest, etwa OTTO oder ABBA. Hier ist gemeint, dass die Elemente jeweils paarweise vorne und hinten übereinstimmen. Das gilt z. B. für ein `int[]` mit folgenden Werten: `{ 1, 2, 3, 2, 1 }`.

Beispiel 1: Palindrom – rekursive Variante

Die Prüfung auf Palindrom-Eigenschaft lässt sich rekursiv recht einfach lösen. Schauen wir uns dies als Programm an, nachdem ich kurz den Algorithmus beschrieben habe.

Algorithmus Wenn das Array die Länge 0 oder 1 hat, dann ist es per Definition ein Palindrom. Ist die Länge zwei und größer, dann wird jeweils das äußere linke und äußere rechte Element auf Übereinstimmung geprüft und danach eine Kopie des Arrays verkürzt um je eine Position vorne und hinten erzeugt. Die weitere Prüfung erfolgt im Anschluss auf dem verbliebenen Teilstück, wie es nachfolgendes Listing zeigt:

```
static boolean isPalindromeSimpleRecursive(final int[] values)
{
    // rekursiver Abbruch
    if (values.length <= 1)
        return true;

    int left = 0;
    int right = values.length - 1;

    if (values[left] == values[right])
    {
        // Achtung: copyOfRange, exklusive End
        final int[] remainder = Arrays.copyOfRange(values, left + 1, right);

        // rekursiver Abstieg
        return isPalindromeSimpleRecursive(remainder);
    }

    return false;
}
```

Das beschriebene und realisierte Vorgehen führt jedoch zu relativ vielen Kopien und Extraktionen von Teilarrays. Diesen Aufwand kann man vermeiden, wenn man zwar die Idee beibehält, den Algorithmus jedoch mithilfe eines Tricks minimal abwandelt.

Optimierter Algorithmus Statt der Kopien nutzt man weiterhin das Originalarray und verwendet zwei Positionsmarkierungen `left` und `right`, die initial das gesamte Array umfassen. Nun prüft man, ob der durch diese Positionen referenzierte linke und rechte Wert übereinstimmen. Ist das der Fall, verkleinert man den Prüfbereich auf beiden Seiten um eine Position und ruft das Ganze rekursiv auf. Das wird so lange wiederholt, bis der linke Positionszeiger den rechten überspringt.

Die Implementierung ändert sich wie folgt:

```
static boolean isPalindromeRecursive(final int[] values)
{
    return isPalindromeRecursive(values, 0, values.length - 1);
}

static boolean isPalindromeRecursive(final int[] values,
                                     final int left, final int right)
{
    // rekursiver Abbruch
    if (left >= right)
        return true;

    if (values[left] == values[right])
    {
        // rekursiver Abstieg
        return isPalindromeRecursive(values, left + 1, right - 1);
    }

    return false;
}
```

Vielleicht fragen Sie sich, wieso ich das Ganze nicht kompakter schreibe und weniger `return`-Anweisungen verwende. Bei der Darstellung dieser Algorithmen geht es mir vor allem um Verständlichkeit. Mehrere `return`s sind eigentlich nur dann ein Problem, wenn die Methode sehr lang und unübersichtlich ist.

Tipp: Hilfsmethoden zur Rekursionserleichterung

Die Idee von Positionszeigern in Arrays oder Strings ist bei Lösungen zur Rekursion ein gebräuchliches Mittel zur Optimierung und Vermeidung etwa von Array-Kopien. Damit das Ganze für Aufrufer nicht unkomfortabel wird, bietet sich eine High-Level-Methode an, die eine Hilfsmethode aufruft, die weitere Parameter besitzt. Dadurch ist es möglich, beim rekursiven Abstieg gewisse Informationen mitzugeben. In diesem Beispiel sind es die linke und rechte Grenze, sodass man auf potenziell aufwendige Kopien verzichten kann. Viele nachfolgende Beispiele werden von der generellen Idee Gebrauch machen.

Beispiel 1: Palindrom – iterative Variante

Obwohl eine rekursive Definition eines Algorithmus mitunter ziemlich elegant ist, produziert diese doch durch den rekursiven Abstieg in Form der Selbstaufrufe potenziell einiges an Overhead. Praktischerweise lässt sich jeder rekursive Algorithmus in einen iterativen umwandeln. Schauen wir uns dies für die Palindrom-Berechnung an. Für die iterative Umsetzung verwenden wir zwei Positionszeiger – statt des rekursiven Abstiegs nutzen wir eine `while`-Schleife. Diese bricht ab, wenn alle Elemente geprüft wurden oder falls zuvor eine Abweichung festgestellt wurde:

```
private static boolean isPalindromeIter(int[] values)
{
    int left = 0;
    int right = values.length - 1;
    boolean sameValue = true;

    while (left < right && sameValue)
    {
        sameValue = values[left] == values[right];

        left++;
        right--;
    }

    return sameValue;
}
```

Auch hier noch eine Anmerkung zur Kompaktheit: Diese Methode könnte man wie folgt schreiben, indem man auf die Hilfsvariable verzichtet:

```
static boolean isPalindromeIterativeCompact(final int[] values)
{
    int left = 0;
    int right = values.length - 1;

    while (left < right && values[left] == values[right])
    {
        left++;
        right--;
    }
    // left >= right || values[left] != values[right]
    return left >= right;
}
```

Der Rückgabewert ergibt sich aus der als Kommentar angedeuteten Bedingung, falls $left >= right$ gilt, dann ist `values` kein Palindrom. Bei dieser Variante muss man aber deutlich mehr über die Rückgabe nachdenken – ich bevorzuge erneut Verständlichkeit und Wartbarkeit gegenüber Kürzte oder Performance.

Beispiel 2: Fraktal als Beispiel

Wie eingangs erwähnt, lassen sich mit Rekursion auch Grafiken erzeugen. Nachfolgend wird eine grafisch simple Variante ausgegeben, die den Unterteilungen eines Lineals nachempfunden ist:

```
-
==
-
===
-
==
-
```

Tatsächlich lässt sich das unter Zuhilfenahme von Java-11-Bordmitteln (`repeat(n)`) leicht rekursiv wie folgt formulieren, wobei zweimal ein rekursiver Abstieg erfolgt:

```
static void fractalGenerator(final int n)
{
    if (n < 1)
       return;

    if (n == 1)
        System.out.println("-");
    else
    {
        fractalGenerator(n - 1);
        System.out.println("=".repeat(n));
        fractalGenerator(n - 1);
    }
}
```

Steht Java 11 und damit die Methode `repeat()` nicht zur Verfügung, dann schreibt man sich einfach eine Hilfsmethode `repeatCharSequence()` (vgl. Abschnitt 2.3.7).

Verwendet man statt ASCII-Zeichen etwas aufwendigere Zeichenfunktionen, so kann man mithilfe von Rekursion interessante und ansprechende Gebilde erzeugen, eingebettet in eine Swing-Applikation etwa folgende Schneeflocke.

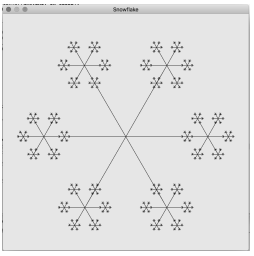

Abbildung 3-3 *Rekursive Grafik mit* `drawSnowflake()`

Diese stilisierte Darstellung einer Schneeflocke lässt sich wie folgt implementieren:

```
public static void drawSnowflake(final Graphics graphics,
                                 final int startX, final int startY,
                                 final int length, final int depth)
{
    for (int degree = 0; degree < 360; degree += 60)
    {
        final double rad = degree * Math.PI / 180;
        final int endX = (int) (startX + Math.cos(rad) * length);
        final int endY = (int) (startY + Math.sin(rad) * length);

        graphics.drawLine(startX, startY, endX, endY);

        // rekursiver Abstieg
        if (depth > 0)
        {
            drawSnowflake(graphics, endX, endY, length / 4, depth - 1);
        }
    }
}
```

3.1.3 Ablauf beim Multiplizieren der Ziffern einer Zahl

Zum Abschluss der algorithmischen Beispiele möchte ich nochmals die einzelnen Schritte und Selbstaufrufe verdeutlichen. Als artifizielles Beispiel nutzen wir dazu das Multiplizieren der Ziffern einer Zahl, auch Querprodukt genannt, etwa für den Wert $257 \Rightarrow 2 * 5 * 7 = 10 * 7 = 70$. Mithilfe von Modulo lassen sich die Extraktion der einzelnen Ziffern und deren Multiplikation recht einfach wie folgt implementieren:

```
static int multiplyAllDigits(final int value)
{
    final int remainder = value / 10;
    final int digitValue = value % 10;

    System.out.printf("multiplyAllDigits: %-10d | remainder: %d, digit: %d%n",
                      value, remainder, digitValue);

    if (remainder > 0)
    {
        final int result = multiplyAllDigits(remainder);

        System.out.printf("-> %d * %d = %d%n",
                          digitValue, result, digitValue * result);
        return digitValue * result;
    }
    else
    {
        System.out.println("-> " + value);
        return value;
    }
}
```

Betrachten wir die Ausgaben für die zwei Zahlen 1234 und 257:

```
jshell> multiplyAllDigits(1234)
multiplyAllDigits: 1234       | remainder: 123, digit: 4
multiplyAllDigits: 123        | remainder: 12, digit: 3
multiplyAllDigits: 12         | remainder: 1, digit: 2
multiplyAllDigits: 1          | remainder: 0, digit: 1
-> 1
-> 2 * 1 = 2
-> 3 * 2 = 6
-> 4 * 6 = 24
$2 ==> 24

jshell> multiplyAllDigits(257)
multiplyAllDigits: 257        | remainder: 25, digit: 7
multiplyAllDigits: 25         | remainder: 2, digit: 5
multiplyAllDigits: 2          | remainder: 0, digit: 2
-> 2
-> 5 * 2 = 10
-> 7 * 10 = 70
$3 ==> 70
```

Es ist gut ersichtlich, wie die rekursiven Aufrufe mit einer immer kürzeren Zahlenfolge geschehen und schließlich das Ergebnis auf Basis der letzten Ziffer in anderer Richtung konstruiert bzw. berechnet wird.

3.1.4 Typische Probleme

Rekursion erlaubt es oftmals, Problemstellungen auf verständliche Weise zu formulieren und zu implementieren. Dabei gibt es jedoch zwei Dinge zu beachten, auf die ich im Folgenden eingehe.

Endlose Aufrufe und `StackOverflowError`

Ein wissenswertes Detail ist, dass die Selbstaufrufe dazu führen, dass diese auf dem Stack zwischengespeichert werden. Für jeden Methodenaufruf wird ein sogenannter Stackframe mit Informationen zur aufgerufenen Methode und deren Parametern auf dem Stack abgelegt. Der Stack ist allerdings in seiner Größe beschränkt und somit können nur eine endliche Zahl an verschachtelten Methodenaufrufen erfolgen – in der Regel aber weit über 10.000. Das hatte ich bereits in einem Praxistipp kurz thematisiert.

Bei sehr vielen rekursiven Aufrufen kann es zu einem `StackOverflowError` kommen. Teilweise tritt die Problematik aber auch auf, weil keine Abbruchbedingung in der Rekursion vorgesehen wurde oder diese falsch formuliert ist:

```
// Achtung: Zur Demonstration bewusst falsch
static void infiniteRecursion(final String value)
{
    infiniteRecursion(value);
}

static int factorialNoAbortion(final int number)
{
    return number * factorialNoAbortion(number - 1);
}
```

Mitunter ist auch einfach der Aufruf falsch, weil kein verringerter Wert übergeben wird:

```
// Achtung: Zur Demonstration bewusst falsch
static int factorialWrongCall(final int n)
{
    if (n == 0)
        return 1;
    if (n == 1)
        return 1;

    return n * factorialWrongCall(n);
}
```

Während man einen direkten endlosen Selbstaufruf noch recht gut optisch erkennen kann, wird dies mit zunehmender Anzahl Zeilen schwieriger: Die fehlende Abbruchbedingung in der Methode `factorialNoAbortion()` mag – spätestens mit etwas Erfahrung und Übung bezüglich Rekursion – auch noch gut erkennbar sein. In der Methode `factorialWrongCall()` ist das nicht mehr so leicht zu ermitteln. Hier muss man schon genauer wissen, wie die Logik sein sollte.

Aus den Beispielen sollten wir zwei Dinge mitnehmen:

1. **Abbruchbedingung** – Eine rekursive Methode muss immer auch mindestens einen Abbruch beinhalten. Aber selbst bei korrekter Definition kann es sein, dass beispielsweise der nicht erlaubte negative Wertebereich nicht überprüft wird. Für `factorial(int)` würde dann ein Aufruf mit einem negativen Wert zu einem `StackOverflowError` führen.
2. **Komplexitätsreduktion** – Eine rekursive Methode muss immer das ursprüngliche Problem in ein oder mehrere kleinere Teilprobleme untergliedern – manchmal ist dies bereits durch den um 1 reduzierten Wert eines Parameters gegeben.

Unerwartete Parameterwerte

Ein ziemlich fieser, weil leicht begangener und nur schwierig ersichtlicher Fehler kann bei Methodenaufrufen, insbesondere auch rekursiven, auftreten, wenn man Parameterwerte um eins erhöhen oder reduzieren möchte. Aus Gewohnheit ist man dazu verleitet, Postinkrement bzw. -dekrement (`++` bzw. `--`) zu verwenden, wie im folgenden Beispiel für die rekursive (leicht ungeschickte) Berechnung der Länge eines Strings, wobei der Parameter `count` die aktuelle Länge enthalten soll:

```
static int calcLengthParameterValues(final String value, int count)
{
    if (value.length() == 0)
        return count;

    System.out.println("Count: " + count);
    final String remaining = value.substring(1);

    return calcLengthParameterValues(remaining, count++);
}
```

Beim Betrachten der Ausgaben für den Aufruf

```
final int length = calcLengthParameterValues("ABC", 0);
System.out.println("length: " + length);
```

sind wir vermutlich überrascht: Statt des um 1 erhöhten Werts, bleibt dieser gleich:

```
Count: 0
Count: 0
Count: 0
length: 0
```

Eine mögliche vorgegebene Maximalzahl kann somit nicht sichergestellt werden. In diesem Beispiel führt nur die Verkürzung der Eingabe zum Abbruch der Rekursion.

Interessanterweise lässt sich der Denkfehler schneller aufdecken, wenn man der guten Programmiertradition folgt und den Parameter `final` deklariert. Dadurch produziert der Compiler direkt eine Fehlermeldung und merkt an, dass die Variable nicht veränderlich ist. Daraufhin kommt man vielleicht eher auf die Idee, als Übergabe den Ausdruck `count + 1` zu wählen. Mit diesem Wissen fällt die Korrektur leicht:

```
static int calcLengthParameterValues(final String value, final int count)
{
    if (value.length() == 0)
        return count;

    System.out.println("Count: " + count);
    final String remaining = value.substring(1);
    return calcLengthParameterValues(remaining, count + 1);
}
```

Gravierendere Auswirkungen Durch die gute Angewohnheit, Parameter als `final` zu definieren, lassen sich die vorgestellte Probleme entschärfen. Nehmen wir trotzdem einmal an, wir hätten bei der schon vorgestellten rekursiven Palindrom-Prüfung auf das `final` verzichtet und zudem etwas sorglos `++` bzw. `--` genutzt:

```
static boolean isPalindromeRecursive(int[] values, int left, int right)
{
    // rekursiver Abbruch
    if (left >= right)
        return true;

    if (values[left] == values[right])
    {
        // rekursiver Abstieg
        return isPalindromeRecursive(values, left++, right-);
    }
    return false;
}
```

Die Auswirkungen sind noch schlimmer als im vorherigen Beispiel, dass wenigstens terminiert, aber einen falschen Wert liefert. Bei der Palindrom-Prüfung erhalten wir stattdessen nach einiger Zeit einen `StackOverflowError`.