E.Tutorial
www.et.ethz.ch

Programmieren mit Java Modul 4

Methoden und Funktionen

Theorieteil

Inhaltsverzeichnis

Begriffe

Methode	Rückgabewert	Exception
Subroutine	Rückgabetyp	
Prozedur	Sichtbarkeit von Variablen	Zufallszahl
Funktion	Überladen	
Parameter	Rekursion	Modularität

Autoren:

Lukas Fässler, Barbara Scheuner, David Sichau

E-Mail:

et@ethz.ch

Datum:

4 April 2025

Version: 1.1

Hash: 98d9199

Trotz sorgfältiger Arbeit schleichen sich manchmal Fehler ein. Die Autoren sind Ihnen für Anregungen und Hinweise dankbar!

1 Modulübersicht

Durch das **Modularitätsprinzip** wird ein Gesamtproblem in getrennte Teilprobleme zerlegt. In diesem Modul lernen Sie Möglichkeiten kennen, wie Sie Anweisungen in **Unterprogrammen** (oder **Subroutinen**) zusammenfassen können. Unterprogramme sind funktionale Einheiten, die von mehreren Stellen in einem Programm aufgerufen werden können. Auf diese Weise muss ein Programmteil nur einmal entwickelt und getestet werden, wodurch sich der Programmieraufwand verringert und der Programmcode verkürzt. Werden beim Aufrufen des Unterprogramms Daten übergeben, werden diese als **Parameter** bezeichnet. In vielen Programmiersprachen werden zwei Varianten von Unterprogrammen unterschieden: jene **mit einem Rückgabewert** (**Funktionen**) und jene **ohne Rückgabewert** (**Prozeduren**). In Java bezeichnet man beide Varianten generell als **Methoden**, die Parameter und Rückgabewert haben können.

2 Methoden

In Java besteht eine Methode aus einem **Kopf (oder Signatur)** mit den Modifikatoren `public static`[1], dem Rückgabetyp, dem Namen, den Parametern sowie dem **Rumpf** in geschweiften Klammern `{}`. Die Ausführung einer Java-Applikation startet meist in der **Haupt** oder **main-Methode**, die sich oft darauf beschränkt, andere Methoden aufzurufen.

2.1 Methoden ohne Rückgabewert (Prozeduren)

Methoden ohne Rückgabewert haben Sie bereits verwendet: Zum Beispiel die Methode `System.out.println()`, welche einen Zeilenumbruch auslöst. Sie können aber auch eigene Methoden schreiben, welche von Ihnen vorgegebene Aufgaben erfüllen. Methoden ohne Rückgabewert sehen wie folgt aus:

Schreibweise:

```
void methodenName() { //Signatur der Methode
    // Anweisungen
}
```

Das Wort `void` gibt dabei an, dass kein Rückgabewert erwartet wird. Die leere Klammer `()` bedeutet, dass keine Parameter übergeben werden.

[1] Die Bedeutung der Modifikationen ergeben sich aus der Objektorientierung von Java und werden in diesem Buch nicht behandelt.

Beispiel: Die folgende Klasse beinhaltet die Haupt-Methode mit dem Namen `main` und die Methode `ausgabe`, welche in der main-Methode aufgerufen wird:

```
public class ProzedurTest{
    public static void main(String[] args){
        ausgabe();
    }
    public static void ausgabe(){  // Signatur der Methode
        System.out.println("die Methode wurde ausgeführt");
    }
}
```

2.2 Methoden mit Rückgabewert (Funktionen)

Methoden mit Rückgabewert werden verwendet, um ein Resultat aus den Anweisungen in der Methode zu gewinnen. Der Datentyp des Rückgabewerts steht dabei direkt vor dem Methodennamen. In der Methode selbst muss sichergestellt werden, dass in jedem Fall ein Wert vom entsprechenden Typ zurückgegeben wird. Um einen Wert zurückzugeben, wird das Wort `return` verwendet.

Schreibweise:

```
rückgabetyp methodenName() { // Signatur der Methode
    // Anweisungen
}
```

Beispiel: Die folgende Klasse beinhaltet eine main-Methode und eine weitere Methode `getAnswer`, welche in der main-Methode aufgerufen wird:

```
public class ProzedurTest{
    public static void main(String[] args){
        int wert = getAnswer();
    }
    public static int getAnswer(){  // Signatur der Methode
        return 42;
    }
}
```

Der Rückgabewert in obrigen Beispiel ist vom Typ Integer. Bei Methoden mit Rückgabewert muss darauf geachtet werden, dass in jedem Fall, der eintreffen könnte, zwingend ein Wert zurückgegeben wird.

Beispiel:

```
boolean hallo(){
    int i = 1;
    if (i == 1){
        return true;
    } else {
        System.out.println("tritt nie ein");
    }
}
```

Auch wenn der else-Fall in dieser Methode nie eintrifft, muss darauf geachtet werden, dass auch in diesem Fall etwas zurückgegeben würde. Korrekt wäre somit:

```
boolean hallo(){
    int i=1;
    if (i==1){
        return true;
    } else {
        System.out.println("tritt nie ein");
        return false;
    }
}
```

2.3 Methoden mit Parametern

Beide Arten von Methoden (also Prozeduren und Funktionen) können in der Signatur Parameter verlangen. Parameter sind Variablen, deren Initialisierung beim Aufruf der Methode geschieht und die zur Erfüllung der Aufgabe der Methode nötig sind.

Beispiel:

```
void addiereUndGibAus(int x, int y){
    int z = x + y;
    System.out.println("Summe: "+z);
}
```

Diese Methode kann nun z.B. mit den Werten 3 und 2 aufgerufen werden mit der Anweisung:

```
addiereUndGibAus(3,2);
```

Methoden mit Rückgabewert können nun auch noch einen Wert zurückgeben.

Beispiel:

```
int addiere (int x, int y){
    int z = x + y;
    return z;
}
```

Diese Methode kann nun z.B. aufgerufen werden mit:

```
int resultat = addiere(3,2);
```

Der Rückgabewert wird hier in der Variable `resultat` gespeichert, deren Datentyp mit dem Rückgabetyp übereinstimmt.

3 Methoden aus der Klasse Math

Die Klasse Math bietet einige Methoden an, welche **mathematische Funktionen** realisieren. All diese Methoden sind Funktionen, weil sie einen Rückgabewert (das Resultat der Berechnung) besitzen. Als Übergabeparameter erwarten sie meist eine oder zwei Zahlen.

Beispiel:

```
public class TestMath{
    public static void main(String[] args){
        double x= 19.7;
        double y= 20.3;
        double groessereZahl = Math.max(x,y);
        double kleinereZahl = Math.min(x,y);
        double abrunden = Math.ceil(x);
    }
}
```

Eine Methode ohne Parameter in der Klasse `Math` ist die Funktion `Math.random()`. Diese Funktion gibt eine **Zufallszahl** zwischen 0 und 1 zurück, wobei die Zahl gleich 0 sein kann, aber immer kleiner als 1 ist. Mit Hilfe dieser Methode kann eine **zufällige ganze Zahl** zwischen 1 und `x` erzeugt werden. Dies geschieht in drei Schritten:

1. Erzeugen einer Zufallszahl zwischen 0 und 1.
2. Multiplikation dieser Zahl mit `x`. Dies ergibt eine Gleitkommazahl Zahl zwischen 0 und `x`.
3. Runden dieser Zahl.

Beispiel:

```
// Generierung ganzer Zahl zwischen 0 und 99 durch abrunden
int zufallsZahl1 = (int)(Math.random()*100);

// Generierung ganzer Zahl zwischen 0 und 100 durch runden
long zufallsZahl2 = Math.round(Math.random()*100);
```

4 Überladen von Methoden

Verschiedene Methoden können denselben Namen haben, wenn sie unterschiedliche Parameter erhalten. Unterschiedliche Rückgabewerte alleine sind nicht ausreichend. Haben zwei Methoden denselben Namen, aber erhalten unterschiedliche Typen oder Anzahl von Parameter, nennt man die **Methoden überladen**. Ein populäres Beispiel, das Sie schon oft verwendet haben, ist die Methode `print()` bzw. `println()`. Dieser Methode können Werte von Typ String, int, double, etc. übergeben werden. Für Sie als Benutzer sieht es aus, als würden Sie immer dieselbe Methode aufrufen. Tatsächlich ist es aber so, dass unterschiedliche Methoden entsprechend dem Datentyp aufgerufen werden.

Wir können also die beiden folgenden Methoden mit identischem Namen `linie` in der gleichen Klasse schreiben:

```
// Signatur mit einem Parameter
public static void linie(int x){
    for (int i=0; i<x; i++){
        System.out.print('-');
    }
}
// Signatur mit zwei Parametern
public static void linie(int x, char c){
    for (int i=0; i<x; i++){
        System.out.print(c);
    }
}
```

Wenn Sie nun die Methode mit nur einem Parameter-Wert aufrufen, wird die erste Methode verwendet, mit zwei entsprechenden Werten die zweite:

```
linie(8);
linie(19, '-');
```

5 Gültigkeitsbereiche von Variablen

Variablen haben eine **beschränkte Lebensdauer**. Einige Variablen werden erst mit dem Beenden des Programms gelöscht, andere schon früher. Eine Variable ist immer für den Anweisungsblock (siehe Modul 2) oder eine Klasse gültig, in dem sie deklariert wurde. Eine Variable, welche am Anfang einer Methode deklariert wird, wird auch wieder gelöscht, wenn die Methode beendet wird. Wenn man den Wert dieser Variablen später noch benötigt, muss er an die aufrufende Methode zurückgegeben werden.

Beispiel:

```
public class VariablenSichtbarkeit{
    public static int global = 20;

    public static void main(String[] args){
        System.out.println("Werte zwischen 1 und "+ global);
        ausgabe();
    }

    public static void ausgabe(){
        int zufall = 0;
        for (int i=0; i<30; i++){
            zufall = (int) (Math.random() * global) + 1;
        }
    }
}
```

Die Variable `global` ist dabei für die ganze Klasse gültig. Die Variable `zufall` ist hingegen nur innerhalb der Methode `ausgabe` gültig und die Variable `i` nur innerhalb der Schleife.

Beispiel: Folgender Code würde, wenn die kommentierte Zeile verwendet würde, zu einer Fehlermeldung führen, da die Variable `zufall` in der `main`-Methode nicht gültig ist:

```
public class VariablenSichtbarkeit2{

    public static void ausgabe(){
        int zufall = (int) (Math.random() * global) + 1;
    }

    public static void main(String[] args){
        ausgabe();
        // System.out.println("Werte von 1 bis "+ zufall);
    }
}
```

6 Rekursion

Bei der **Rekursion** ruft eine Methode sich selber wieder auf. Damit sich die Methode nicht endlos immer wieder selbst aufruft, was die gleichen Konsequenzen wie eine Endlosschleife hätte, benötigt sie eine **Abbruchbedingung**, welche diese Folge von Selbst-Aufrufen stoppt.

Um eine Rekursion zu programmieren, müssen Sie zwei Elemente bestimmen:

- **Basisfall:** in diesem Fall ist das Resultat der Berechnung schon bekannt. Dieser Fall ist die Abbruchbedingung der Rekursion.
- **Rekursiver Aufruf:** es muss bestimmt werden, wie der rekursive Aufruf geschehen soll.

6.1 Beispiel 1: Fakultät

Die Berechnung der Fakultät $f(x) = x!$ von x kann mit einer rekursiven Methode realisiert werden.

- Basisfall: für die Werte 0 und 1 ist die Fakultät 1.
- Rekursion: $x! = x \cdot (x-1)!$ für $x > 1$.

```
int fakultaet(int x){
    if ((x == 0) || (x == 1){ // Basisfall
        return 1;
    } else {
        return x * fakultaet(x-1);  // Rekursiver Aufruf
    }
}
```

6.2 Beispiel 2: Fibonacci

Ein weiteres beliebtes Beispiel für Rekursionen ist die **Fibonacci-Folge**. Diese unendliche Folge von natürlichen Zahlen beginnt mit 0 und 1. Die danach folgende Zahl ergibt sich jeweils aus der Summe der zwei vorangegangenen Zahlen: Die Folge lautet also $0, 1, 1, 2, 3, 5, 8, \ldots$

Bei der Fibonacci-Folge sind bei jedem Schritt zwei rekursive Aufrufe nötig:

$f(n) = f(n-1) + f(n-2)$ für $n \geq 2$ mit den Anfangswerten $f(1) = 1$ und $f(0) = 0$.

- Basisfall: Für den Fall, dass n gleich 0 oder 1 ist, wissen wir, dass 0 bzw. 1 zurückgegeben werden muss.
- Rekursion: Für alle anderen Fälle rufen wir die Funktion wieder auf, wobei wir den übergebenen Wert um jeweils 1 und 2 verringern.

```
int fibonacci(int n) {
    if (n == 0){ // Basisfall 1
        return 0;
    } else if (n == 1){ // Basisfall 1
        return 1;
    } else { // zwei Mal rekursiver Aufruf
        return (fibonacci(n-1) + fibonacci(n-2));
    }
}
```

Beachten Sie, dass Funktion `fibonacci` in Beispiel 2 sich selbst gleich zweimal aufruft. Man spricht auch von *kaskadenförmiger Rekursion*. In Beispiel 1 erfolgt nur ein einzelner Selbstaufruf, was man als *lineare Rekursion* bezeichnet.

7 Fehlerbehandlung mit Exceptions

Exceptions (*Ausnahmen*) werden durch Ausnahmesituationen bei der Programmausführung ausgelöst. Dadurch sollen Programmabbrüche verhindert werden. In Java ermöglicht die **try-catch-Anweisung** das Auffangen und Behandeln von Exceptions innerhalb einer Methode.

Beispiel: Vielleicht wollten Sie schon einmal aus Versehen einen Text „jk“ in eine Zahl umwandeln oder in einem Array auf einen Index zugreifen, der ausserhalb des Arrays liegt. Exceptions müssen nicht zwingend bis zum Benutzer durchdringen (d.h. auf der Konsole erscheinen). Man kann sie auch abfangen (*catch*) und eine Lösung für das Problem suchen. Abgefangen können die Exceptions mit:

```
try {
    // Irgendetwas, dass einen Fehler auslösen könnte
} catch (Exception c) {
    // Lösung des Problems
}
```

Beispiel: Hier könnte der Benutzer anstelle einer ganzen Zahl zum Beispiel ein Zeichen eingeben. Diese Fehlereingabe kann z.B. wie folgt behandelt werden:

```
public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    int zahl = 0;
    System.out.println("Bitte eine ganze Zahl eingeben");
    String wert = scan.nextLine();
    try {
        zahl = Integer.parseInt(wert);
    } catch (Exception e){
        System.out.println("Das ist keine ganze Zahl");
        Zahl = 0;
    }
}
```

7.1 Werfen einer Exception

Sie können das Konstrukt der Exception auch selber verwenden. Wenn Sie in einer eigenen Methode einen Fehler kommunizieren wollen, dann können Sie dort eine Exception werfen (*throw*). Mit der Anweisung `throw new Exception()` lösen Sie einen Fehler aus. Die Signatur einer Methode, welche eine Exception auslösen kann, muss noch durch den Zusatz `throws Exception` (heisst so viel wie *wirft Fehlermeldungen*) ergänzt werden.

Beispiel:

```
public class ExceptionTester{
    public static void main(String[] args) {
        try {
            tester(-3);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    public static void tester(int x) throws Exception{
        if (x < 0){
            throw new Exception();
        }
    }
}
```