	Unterprogramme		AnPr
	Name	Klasse	Datum

1 Allgemeines

Programme werden üblicherweise nicht als ein einziger, fortlaufender Programmcode verfasst, sondern mit geeigneten Mitteln unterteilt und somit strukturiert. Die Methodik hierfür, welche von (fast) allen Programmiersprachen unterstützt wird, ist die der Unterprogramme.
Unterprogramme werden aus folgenden Gründen geschaffen:

- Wiederholungen im Code werden vermieden.
- Programme werden leichter lesbar.
- Die Entwicklung und der Test werden vereinfacht.

2 Eigenschaften von Unterprogrammen

Um die Nutzung von Unterprogrammen so effizient wie möglich zu gestalten, gibt es die Möglichkeit ihnen Eigenschaften zuzuordnen:

Parameter: Damit der Aufrufer die Verarbeitung innerhalb des Unterprogramms beeinflussen kann, gibt es die Option entsprechende Parameter an das Unterprogramm zu übergeben. Hierbei ist wichtig, dass die Parameter in typisierten Programmiersprachen auch Datentypen besitzen, beim Aufruf die Daten also mit dem richtigen Datentyp übergeben werden müssen.

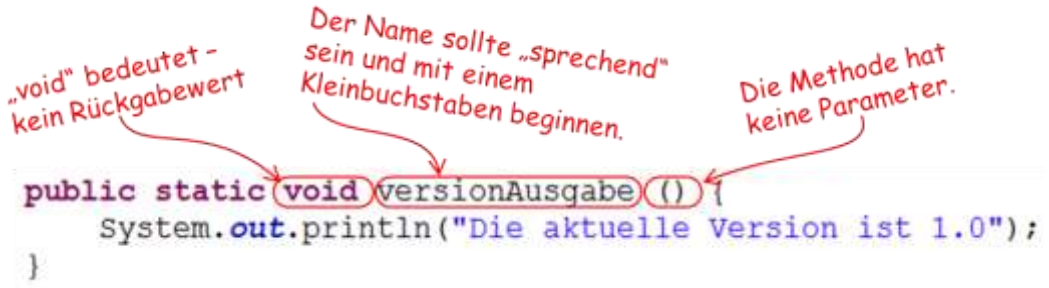
Name: Um einen Aufruf zu ermöglichen, erhält jedes Unterprogramm einen Namen. Dieser sollte möglichst jedem Programmierer den Sinn und die Funktionsweise des Unterprogramms klar machen. Name und Parameter (Anzahl und Reihenfolge der Datentypen) werden auch die „**Signatur**“ einer Methode genannt. Diese Signatur ermöglicht es dem Computer beim Aufruf die richtige Methode zu finden. Es können also mehrere Unterprogramme mit dem gleichen Namen existieren, welche sich lediglich durch die Anzahl und Reihenfolge der Parameterdatentypen unterscheiden. Im Rahmen der Objektorientierung würde man hier von „Überladung“ sprechen – die Methode ist „überladen“.

Rückgabewert: Zur Übermittlung von eventuellen Unterprogrammergebnissen können Unterprogramme einen einzelnen Rückgabewert an den Aufrufer zurückgeben. In prozeduralen Programmiersprachen nennt man Unterprogramme, welche einen Wert zurückgeben, **Funktionen**. Solche die keinen Wert zurückgeben heißen **Prozeduren**. In objektorientierten Programmiersprachen spricht man immer von **Methoden**. Das liegt im Wesentlichen an der Tatsache, dass Methoden Zugriff auf Instanzvariablen haben – doch davon mehr wenn das Thema „Objektorientierung“ angesprochen wird. Da Java eine objektorientierte Programmiersprache ist, sprechen wir ab hier nur noch von Methoden.

3 Aufbau Methodendeklaration

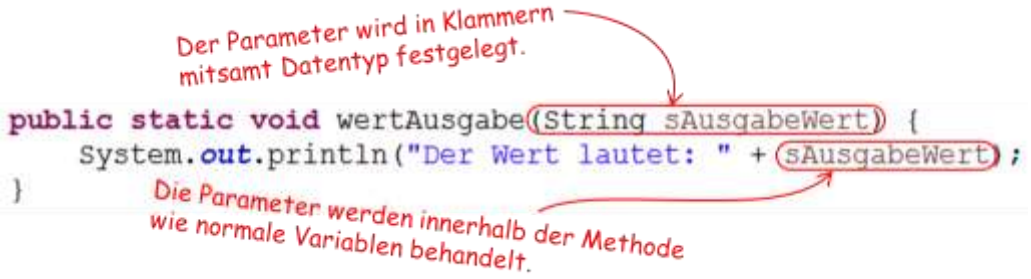
Methoden werden mit dem oben beschriebenen Kopf (also Name, Parameter und Rückgabewert) und dem anschließenden Code, dem „Rumpf“ realisiert. Die Übergabeparameter werden innerhalb der Methode als ganz normale Variablen behandelt, weshalb sie auch im Kopf deklariert werden müssen. Die Zuweisung der Werte erfolgt bei Methodenaufruf. Der Code ist keinen weiteren Beschränkungen unterworfen.

Beginnen wir zuerst mit der einfachsten Form – einer Methode, welche keine Parameter aufweist und keinen Rückgabewert liefert:



Wichtig ist hier das Schlüsselwort „void" (engl. für leerer Raum) in Java, welches indiziert, dass es keinen Rückgabewert gibt. Nach dem Methodennamen folgt die Parameterliste, welche in diesem Beispiel leer ist – die Methode also keine Parameter erwartet. Der Bereich zwischen den beiden geschweiften Klammern nennen wir auch „Methodenrumpf". Dies ist der Bereich, der bei Aufruf der Methode ausgeführt wird.

Im nächsten Beispiel haben wir eine Methode mit einem Parameter (und wieder ohne Rückgabewert):

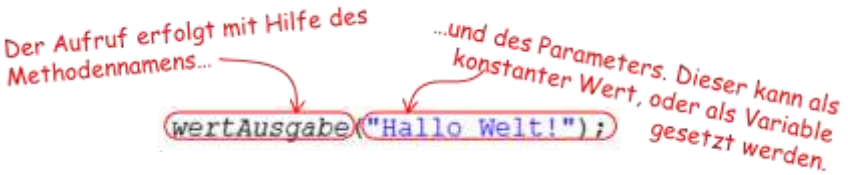


Die Parameter sind im Prinzip nichts anderes wie Variablen und werden innerhalb des Methodenrumpfes auch wie Variablen behandelt. Sie können gelesen und geschrieben werden. Der Gültigkeitsbereich geht immer bis zur schließenden geschweiften Klammer des Methodenrumpfes.

~~Blöde~~ Frage:
Gibt es irgendwelche Namenskonventionen für die Parameter?
Antwort: Die Namen der Parameter unterliegen den gleichen Forderungen wie andere Variablen auch. Rein technisch können sie beliebig benannt werden. Das einzige, was technisch von Interesse ist, sind die Datentypen der Parameter. Diese sind auch beim Aufruf wichtig.

4 Aufruf einer Methode

Der eigentliche Aufruf ist nichts Neues für uns. Wir haben bereits diverse Methoden genutzt – parseInt(), println() etc. Beim Aufruf ist lediglich wichtig, dass wir den Methodennamen und die dazugehörigen Parameter mit dem richtigen Datentypen verwenden. In unserem Beispiel „wertAusgabe" müssen wir den Namen, gefolgt von einem Stringwert verwenden:



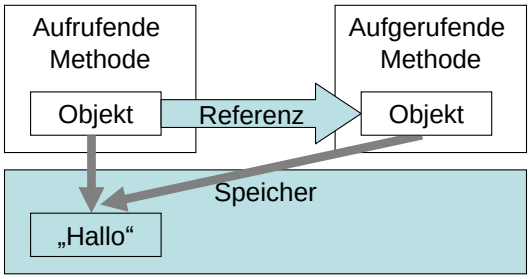
Es ist hierbei erst mal unerheblich, ob der Stringwert aus einer Konstanten (wie oben dargestellt) oder aus einer Variablen kommt.

Sind mehrere Parameter im Spiel, so muss lediglich darauf geachtet werden, dass die Datentypen in der richtigen Reihenfolge auftreten:

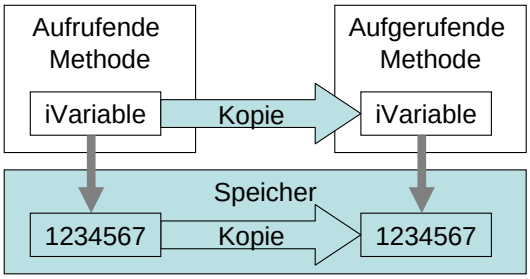
```
                    Die verschiedenen Parameter werden mit Komma getrennt eingetragen:
public static void wertBerechnung(double dZaehler, double dNenner, String sEinheit){
    if (dNenner == 0){
        System.out.println("Nicht definiert");
    } else {
        System.out.println("Das Ergebnis lautet: " + (dZaehler/dNenner) + sEinheit);
    }
}
```

5 Verhalten der Übergabeparameter

Bei der Parameterübergabe ist zu beachten, dass man bei vielen Programmiersprachen die Übergaben „by value“ und „by reference“ unterscheiden muss.



Call by Reference



Call by Value

Bei **Call by Reference** wird die Variable von der aufrufenden Methode nicht übergeben, sondern lediglich eine Referenz auf den Speicherort der Variable. Dies hat zur Konsequenz, dass wenn die aufgerufene Methode den Inhalt der Variable ändert, diese Änderung auch bei der aufrufenden Methode sichtbar ist.

Man kann sich dies so vorstellen, als dass eine Person A ein Dokument vorbereitet und auf den Tisch legt – diese Person A soll das aufrufende Programm symbolisieren. Das aufgerufene Programm wird von Person B symbolisiert und erhält von Person A lediglich die Information wo das Dokument liegt. Person A teilt Person B also mit „Das Dokument liegt auf dem Tisch“. Wenn Person B nun Änderungen an dem Dokument vornimmt, so wird diese Änderung an dem Dokument auf dem Tisch vorgenommen und Person A kann diese Änderungen entsprechend sehen.

Im Gegensatz dazu wird bei **Call by Value** eine echte Kopie der Variable erstellt, welche der aufgerufenen Methode übergeben wird. Somit sind Änderungen an der Variable in der aufrufenden Methode nicht sichtbar.

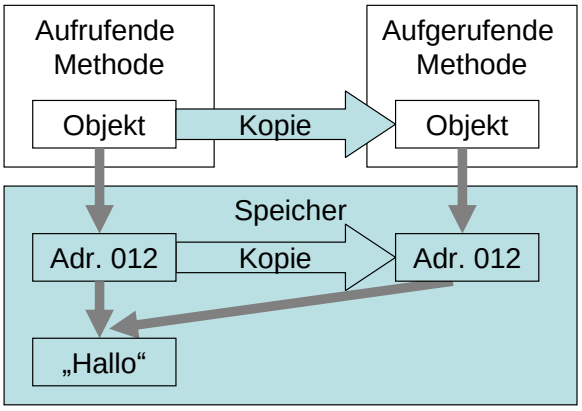
Man kann sich dies so vorstellen, als dass eine Person A ein Dokument vorbereitet und auf den Tisch legt – diese Person A soll wiederum das aufrufende Programm symbolisieren. Das aufgerufene Programm wird wieder von Person B symbolisiert und erhält von Person A nun eine echte Kopie des Dokuments (also aus dem Kopierer). Wenn Person B nun Änderungen an dem Dokument (also der Kopie) vornimmt welches sie von Person A erhalten hat, so wird diese Änderung an dem originalen Dokument auf dem Tisch nicht vorgenommen und Person A kann diese Änderungen nicht sehen.

Fazit: Bei Call by Reference arbeiten sowohl die aufrufende, als auch die aufgerufene Methode mit der gleichen Variablen. Jede Änderung, welche das aufgerufene Programm an dem Variableninhalt macht, wird automatisch auch von dem aufrufenden Programm „gesehen“.
Bei Call by Value arbeiten beide Methoden mit einer eigenen Kopie der Variablen. Eine Änderung durch das aufgerufene Programm wird somit nicht vom aufrufenden Programm „gesehen“.

In Java gibt es keine Wahlmöglichkeit, ob by Value oder by Reference aufgerufen wird – folgende Tabelle zeigt das Verhalten (nicht die tatsächliche Umsetzung) der Java Virtual Machine auf:

Art des Übergabeparameters:	**Verhalten:**
Einfache Datentypen (int, double etc.)	Call by value
Strings	Call by value
Objekte	Call by reference

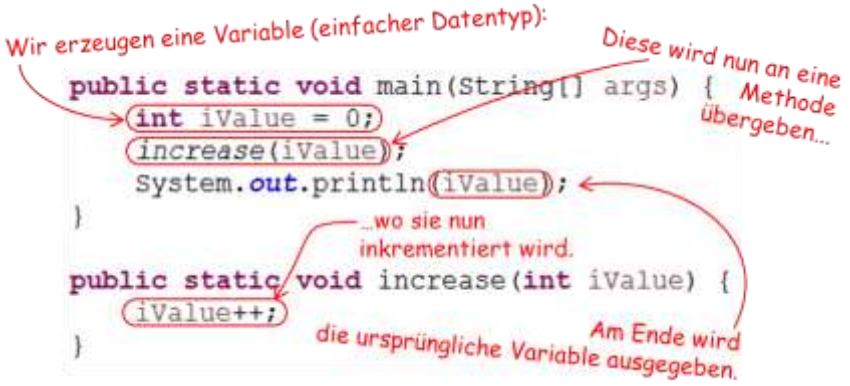
~~Blöde~~ Frage:
Ich habe gelesen, dass Java nur „Call by Value“ unterstützt. Irgendwie passt das aber nicht zur obigen Tabelle, weil es würde ja bedeuten, dass das Verhalten für Objekte so nicht möglich wäre?!?
Antwort: Es ist absolut richtig, dass Java nur „Call by Value“ unterstützt. Der Trick ist, daß wir in Java zwischen einfachen Datentypen und Objekten unterscheiden. Ein einfacher Datentyp ist ein Platz im Speicher, in dem der tatsächliche Wert steht. Wenn nun ein Parameter ein einfacher Datentyp ist, so wird dieser tatsächlich kopiert und als „Value“ übergeben. Eine Objektvariable trägt als einzige Information die ID des Objektes (was vergleichbar mit dem Speicherort oder der Adresse ist). Diese Information wird bei der Übergabe kopiert (also „Call by Value“). Der aufgerufene arbeitet nun mit dieser ID weiter, welche aber auf das gleiche Objekt zeigt, wie das des Aufrufers. Strings wiederum gelten auch als Objekte, werden aber wenn sie einmal im Speicher sind nicht mehr verändert. Jede Anpassung würde zu einer neuen ID führen. Insofern wirkt das Verhalten von Strings wie einfache Datentypen.



Java Umsetzung

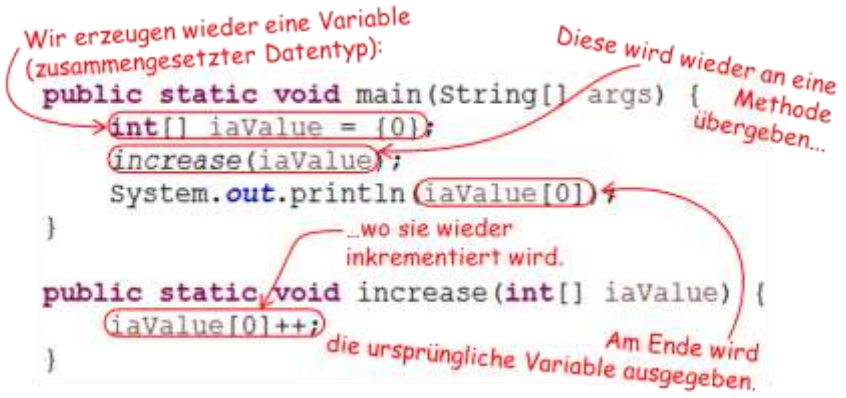
Sehen wir uns das Ganze mal im Versuch an. Beginnen wir mit der Übergabe eines einfachen Datentyps, den wir im Unterprogramm verändern und sehen was passiert:

Wenn wir den Code ausführen, dann sehen wir auf der Konsole den Wert „0“. Dies liegt daran, dass die Variable als Wert übergeben worden ist – die Variable „iValue“ des Unterprogramms ist also eine andere, als die Variable „iValue“ der main – Methode. Insofern wird diese auch nicht durch das Unterprogramm verändert.



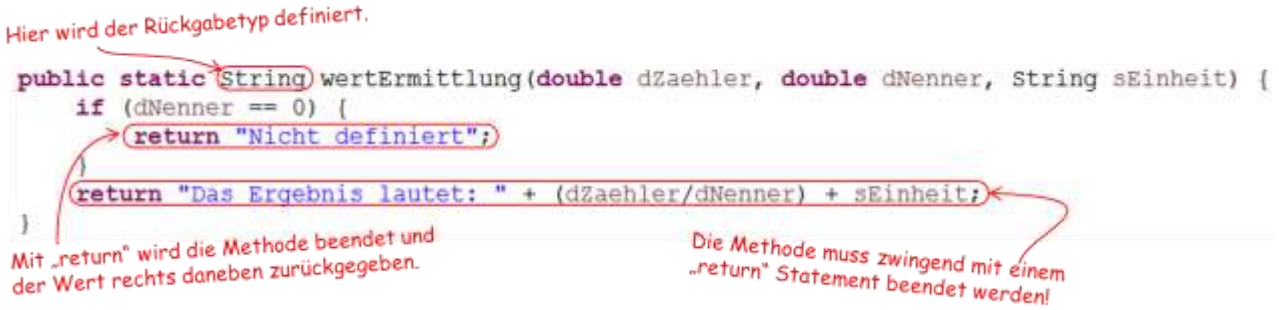
Sehen wir uns den gleichen Code an, bei dem jedoch die Variable als zusammengesetzter Datentyp – sprich Objekt realisiert wurde. Dazu verwenden wir nun anstatt einer int Variablen ein int Array:

Bei der Ausführung dieses Codes sehen wir nun, dass auf der Konsole die „1“ steht. Die Variable iaValue in der main-Methode beinhaltet nun lediglich die ID (bzw. Adresse) des Objekts, welche an die Methode „increase“ weitergegeben wird. Dort wird auf genau dieser Adresse der Wert inkrementiert. Die main-Methode liest nun wieder auf der gleichen Adresse den Wert und sieht ihn inkrementiert.

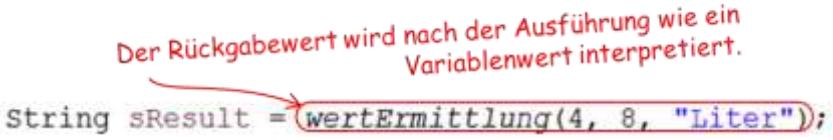


6 Rückgabewerte

In Java kann es immer nur einen Rückgabewert geben. Es ist nicht möglich, bspw. zwei int Variablen gleichzeitig zurückzugeben. Das wichtigste Schlüsselwort hierbei ist „return“. Dies ist die Stelle, welche den Rückgabewert festlegt:



Es muss vom Programmierer sichergestellt werden, dass die Methode auf jeden Fall ein return Statement verarbeitet. Ist dies nicht der Fall, verweigert das JDK das Compillieren!
Weiterhin ist wichtig zu wissen, dass ein return Statement die Verarbeitung des Unterprogramms unterbricht. Alle folgenden Statements werden also nicht mehr ausgeführt. Sehen wir uns nun einen beispielhaften Aufruf an:



Wenn die Methode nun aufgerufen wird, erfolgt also zwingend ein „return“ Statement und der Aufrufer erhält den Wert zurückgeliefert. Er kann somit lesend darauf zugreifen (ähnlich wie er eine Variable auslesen würde).

~~Blöde~~ Frage:
Ich muss in meiner Methode nun zwei Werte zurückgeben. Wie kann ich das realisieren?
Antwort: Hier gibt es mehrere Möglichkeiten. Zum einen kann man mit Arrays als Rückgabewerte arbeiten, zum anderen könnte man eigene Objekte erzeugen, welche alle notwendigen Informationen tragen (dazu müssen wir aber in die Objektorientierung einsteigen). Die dritte Möglichkeit ist, dass man ein Objekt als Übergabeparameter festlegt. Da von diesem ja die Adresse übergeben wird, kann der Aufrufer nach dem Beenden der Methode wieder auf das Objekt zugreifen, weshalb man sich den Rückgabedatentyp sparen kann und somit auch die „return“ Anweisung. Das macht den Code zwar nicht unbedingt lesbarer, aber es funktioniert. Die letzte Option ist auch wieder eine Lösung innerhalb der objektorientierten Programmierung – indem man mit Instanzvariablen arbeitet. All das wird aber später nochmal beim Thema „Objektorientierung“ behandelt.

7 Lizenz