

# 7. Unterprogramme

## 7.1 Motivation für Unterprogramme

Im Alltag werden Tätigkeiten oft “vereinfacht” beschrieben: Reden wir zum Beispiel vom Kochen, Lesen, Laufen oder Telefonieren, dann bestehen diese Tätigkeiten zwar aus einer Abfolge von Einzelschritten, aber wir können uns trotzdem darunter etwas vorstellen. Mit nur einem Wort kann man sich so eine Abfolge von verschiedenen Einzelschritten vorstellen. Dieses Zusammenfassen von Einzelschritten - hier zu einer “Tätigkeit” - ist nichts anderes als zu abstrahieren. Durch diese Abstraktion verstehen wir - in der Regel ausreichend genau - was insgesamt geschieht, ohne die einzelnen Details genauer zu kennen.

Im Pacman Beispiel ist dies ähnlich: Im draw-Bereich des Pacman Programms gehören jeweils einige Programmzeilen (vgl. “Einzelschritte”) inhaltlich zusammen. Sie machen also etwas Ähnliches oder erfüllen gemeinsam eine größere Aufgabe (vgl. “Tätigkeit”): Sie zeichnen etwa das Spielfeld mit den Futterpunkten, den Pacman bestehend aus Körper, Mundöffnung, Auge, oder einen Geist, der in sich wieder aus verschiedenen geometrischen Objekten aufgebaut ist (siehe nachfolgende Abbildung links).

Wenn man nun das Pacman-Programm aufgrund dieser inhaltlichen Zusammengehörigkeiten ansieht, dann wird zunächst der setup-Bereich ausgeführt. Im draw-Bereich können wir also inhaltlich drei Abschnitte erkennen: *Futterpunkte/Spielfeld*, *Pacman* und *Ghost* (siehe nachfolgende Abbildung links).

Diese Einteilung des Programms in Abschnitte kann uns dabei helfen, einen besseren Überblick über den Ablauf und den Gesamtaufbau unseres Programms zu erhalten. Wenn wir uns nur die Abschnitte anschauen, sieht der Code auf den ersten Blick gleich übersichtlicher aus. Dies vereinfacht es, die Funktionalität eines Programms zu erfassen, für so einen Überblick braucht man nicht den gesamten Programmcode Zeile für Zeile lesen und nachvollziehen.

Wenn wir also das Pacman-Programm auf dem Level der Code-Abschnitte ansehen (siehe nachfolgende Abbildung rechts), sind wir eine Abstraktionsebene höher gewechselt. Wir können erkennen, dass es Abschnitte gibt, die einen Pacman zeichnen und einen Geist und wie diese Abschnitte im Programm angeordnet sind. Wie sie jedoch im “Inneren” im Detail realisiert sind, wird verschleiert. Wir erhalten somit einen besseren Überblick über das gesamte Programm, aber weniger Detailinformation.

In der Programmierung setzen wir diese Strukturierung mit sogenannten **Unterprogrammen** um. Mit Unterprogrammen fassen wir eben solche Codeabschnitte zusammen, versehen sie mit einem aussagekräftigen Namen und stellen sie zur weiteren Verwendung zur Verfügung. Auf diese Weise strukturieren wir mit Unterprogrammen Programmcode und machen diesen leichter lesbar. Außerdem ersparen wir uns damit Schreibaarbeit, denn einmal geschriebene Unterprogramme können beliebig oft an verschiedenen Stellen verwendet werden.

Betrachtet man den gesamten Code dann auf Basis dieser Struktur mit den Unterprogrammen und schaut nicht in die Details, was jedes einzelne macht, dann kann dies als ein Beispiel für Code-Abstraktion in der Programmierung wahrgenommen werden.

[illegible]

Man nennt Unterprogramme auch **Funktionen**, **Prozeduren** oder auch **Methoden**, je nachdem in welchem Kontext sie vorkommen und was sie genau machen. Umgangssprachlich hat sich in Processing für die Bezeichnung *Unterprogramm* der Begriff **Funktion** eingebürgert, weshalb wir in den Unterlagen daher Unterprogramme synonym als **Funktionen** bezeichnen.

Wenngleich in diesem Handout der Fokus auf dem Programmieren eigener Funktionen liegt, sehen wir uns zunächst an, wie eine Funktion ausgeführt (man sagt auch: aufgerufen) wird:

## 3 Beispiele für Funktionsaufrufe:

<code>fill(255, 0, 0);</code>	Funktion mit Parametern
<code>ellipse(50, 50, 50, 50);</code>	Funktion mit Parametern
<code>noFill();</code>	Funktion ohne Parameter

Zum **Ausführen einer Funktion** ist also deren Funktionsname anzugeben und nachfolgend innerhalb runder Klammern keine, einen, oder mehrere sogenannte Parameter (durch Beistriche getrennt). Abgeschlossen wird der Funktionsaufruf, wie in Processing bei Anweisungen üblich, durch einen Strichpunkt. Das Ausführen einer Funktion wird **Funktionsaufruf bzw. Aufrufen einer Funktion** genannt. Das kennen Sie bereits in Processing, auch wenn Sie vielleicht diese Begriffe nicht gekannt haben.

## 7.2 Eigene Unterprogramme programmieren

Wie werden nun eigene Funktionen programmiert? So wie Variablen deklariert werden müssen, müssen auch Funktionen dem Computer bekannt gemacht werden. Das bedeutet, auch sie müssen **deklariert** bzw. **definiert** werden, bevor sie in Programmen dann verwendet bzw. aufgerufen werden können.

Im Folgenden sehen Sie zwei Beispiele für Definitionen selbstprogrammierter Funktionen:

```
void drawRedCircle() {
    fill(255, 0, 0);
    ellipse(50, 50, 50, 50);
}
```

Der Aufruf `drawRedCircle()`; zeichnet einen roten Kreis mit Durchmesser 50 Pixel an der Position (50, 50).

Die folgende Funktion `maximum(...)` berechnet das Maximum dreier ganzer Zahlen a, b und c:

```
int maximum(int a, int b, int c) {
    int max = a;

    if (max < b) {
        max = b;
    }

    if (max < c) {
        max = c;
    }
    return max;
}
```

Die **Definition** einer Funktion in Processing besteht aus vier Teilen, die im Laufe dieses Kapitels im Detail erläutert werden:

- Datentyp des Rückgabewerts (Rückgabetyt)
- Funktionsname
- Parameter innerhalb von Klammernpaar ( )
- Anweisungsblock (Funktionsrumpf)

### Beispiel

```
int
maximum
(int a, int b, int c)
{...}
```

Wenn wir nun eigene Funktionen programmieren möchten, dann können wir anhand folgender Fragestellungen strukturiert vorgehen:



1. Welche Aufgabe soll die Funktion erfüllen?

Anhand dieser Fragestellung wählen wir einen aussagekräftigen Namen für die Funktion. Denn der Funktionsname soll das, was die Funktion macht, so gut wie möglich beschreiben. Für erlaubte Funktionsnamen gelten im Übrigen die gleichen Regeln wie für Variablennamen.

2. Soll die Funktion einen Rückgabewert liefern? Wenn ja, von welchem Typ?

Der Rückgabewert wird links vom Funktionsnamen geschrieben. Er gibt an, welche Art von Ergebnis eine Funktion liefert. Beispielsweise liefert die Funktion `maximum()` eine ganze Zahl (`int`) als Ergebnis, welches die größte Zahl der übergebenen drei ganzen Zahlen ist. Zeichnen wir mit der Funktion etwas, wie in der Abbildung oben etwa einen Ghost, dann brauchen wir keinen Rückgabewert, den wir an anderer Stelle im Programm weiterverwenden wollen. In diesem Fall ist der Typ des Rückgabewerts `void` (kein Rückgabewert).

3. Welche Informationen/Werte benötigt die Funktion zur Lösung der Aufgabe?

Mit dieser Frage überlegen wir, ob das Unterprogramm beim Aufruf bestimmte Werte benötigt, um die Aufgabe erfüllen zu können.

Wollen wir etwa das Maximum von drei Zahlen berechnen, werden diese drei Zahlen zur Ausführung des Unterprogramms benötigt. Soll etwa ein Kreis gezeichnet werden, sind Werte betreffend dessen Position und Größe notwendig.

Diese Werte, die eine Funktion zur Ausführung der Aufgabe benötigt, werden Parameter genannt. Mit Parametern kann man das Verhalten einer Funktion ein wenig steuern bzw. flexibler gestalten. Sie werden innerhalb runder Klammern unmittelbar nach dem Funktionsnamen geschrieben. Dazu wird in der Definition der Funktion für jeden Parameter vor seinem Namen auch sein Datentyp angegeben. Falls die Funktion keine Parameter benötigt, wird innerhalb der runden Klammern einfach nichts angegeben, so wie bei der Funktion `drawRedCircle()`.

4. Wie wird die Aufgabe erfüllt?

Diese Fragestellung zielt darauf ab, wie wir zum Ergebnis gelangen. Hier wird -

innerhalb geschwungener Klammern - also jener Code programmiert, der die Aufgabe der Funktion löst.

Datentyp des Rückgabewerts (1), Funktionsname (2) und Parameter (3) bilden zusammen den sogenannten **Funktionskopf**:

```
rückgabeTyp funktionsName(parameter1, parameter2, ..., parameterN)
```

Der Grundaufbau einer Funktion ist insgesamt also so gegliedert:

```
rückgabeTyp funktionsName(parameter1, parameter2, ..., parameterN) {  
    // Funktionsrumpf, der die auszuführenden Anweisungen enthält  
}
```

**Achtung: Die Definitionen von Funktionen werden in Processing AUSSERHALB der setup() und draw() Bereiche geschrieben!**

## 7.3 Funktionen ohne Rückgabewert und Parameter

Betrachten wir zunächst eine sehr einfache Variante der Funktionen - nämlich Funktion ohne Rückgabewert und ohne Parameter. Eine sehr einfache selbstgeschriebene Funktion könnte so aussehen:

```
void drawRedCircle() {
    fill(255, 0, 0);
    ellipse(50, 50, 50, 50);
}
```

Diese Funktion zeichnet - wie man am Funktionsnamen schon erkennen kann, einen roten Kreis. Dieser wird fix an der Position (50, 50) gezeichnet. Der Name der Funktion ist `drawRedCircle()`, es gibt keine Parameter, die runden Klammern werden aber jedenfalls geschrieben. Dass die Funktion keinen Rückgabewert besitzt, erkennen Sie am Rückgabebetyp `void`. `void` bedeutet so viel wie "leer".

Wenn Sie diesen Code in Processing kopieren und das Programm starten, dann passiert noch nicht viel. Denn genau wie die Processing Funktionen `rect()`, `ellipse()` usw., muss auch `drawRedCircle()` aufgerufen werden, damit der rote Kreis auch tatsächlich gezeichnet wird. Dazu wird der Funktionsname mit nachfolgenden runden Klammern, z.B. in `draw()`, geschrieben.

Hinweis: In diesem Beispiel ist auch ersichtlich, dass die **Definition der Funktion** `drawRedCircle()` **außerhalb der `setup()` und `draw()` Bereiche** programmiert wird.

```
void setup() {
}

void draw() {
    drawRedCircle(); // Funktionsaufruf
}

// Definition der Funktion
void drawRedCircle() {
    fill(255, 0, 0);
    ellipse(50, 50, 50, 50);
}
```

Das "Hinschreiben" des Funktionsnamen, um die entsprechende Funktion auszuführen, nennt man **Funktionsaufruf**. Die Funktion wird also von dieser Stelle im Programm aufgerufen und damit der Code innerhalb des Funktionsrumpfes ausgeführt. Wird jetzt das Programm ausgeführt, dann wird ein roter Kreis gezeichnet.

In diesem kleinen Beispiel scheint eine Funktion noch Mehrarbeit zu sein. Sobald bei komplexeren Programmen jedoch mehr Code im `draw()` Bereich steht, ist es übersichtlicher, zusammenhängenden Code in Funktionen auszulagern, sodass der `draw()` Bereich nur wenige Funktionsaufrufe und die wichtigsten Berechnungen beinhaltet. Damit kann die gesamte Funktionalität des Programms in `draw()` schnell erfasst werden, ohne seitenweise Codezeilen lesen zu müssen.

Funktionen können nicht nur in `setup()` und `draw()` aufgerufen werden. Auch innerhalb von anderen Funktionen sind Funktionsaufrufe möglich, wie das nachfolgende Beispiel illustriert:

```
void setup() {
  size(500, 500);
  background(255);
}

void draw() {
  drawRobot(); //Funktionsaufruf von drawRobot()
}

//Funktionsdefinition drawRobot()
void drawRobot() {
  drawRobotHead(); //Funktionsaufruf von drawRobotHead()
  drawRobotBody(); //Funktionsaufruf von drawRobotBody()
}

//Funktionsdefinition drawRobotHead()
void drawRobotHead() {
  //head
  fill(200);
  rect(200, 100, 100, 100);

  //eyes
  fill(250, 250, 0);
  ellipse(230, 130, 20, 20);
  ellipse(270, 130, 20, 20);

  //mouth
  fill(0);
  rect(230, 160, 40, 20);
}

//Funktionsdefinition drawRobotBody()
void drawRobotBody() {
  fill(200);
  rect(240, 200, 20, 20); //neck
  rect(180, 220, 140, 200); //body
  rect(160, 220, 20, 150); //left arm
  rect(320, 220, 20, 150); //right arm
}
```



In diesem Beispiel wird die Funktion `drawRobot()` in `draw()` aufgerufen. `drawRobot()` selbst enthält wieder zwei Funktionsaufrufe: `drawRobotHead()` und `drawRobotBody()`. An diesem Beispiel lässt sich auch schön die Abstraktion erkennen. Für den Leser reicht es bereits den Funktionsaufruf im `draw()` zu lesen, um zu verstehen, was hier im Groben passiert - es wird ein Roboter gezeichnet.

Liest man dann die Funktion `drawRobot()` genauer durch, dann erhält man die Information, dass der Roboter aus Kopf und Körper besteht. Geht man dann noch weiter und betrachtet die Funktionen `drawRobotHead()` und `drawRobotBody()`, dann sieht man im Detail, wie Körper und Kopf in sich gezeichnet werden. Je weiter man also in die Funktionen hineinschaut, desto mehr Details erfährt man. Betrachtet man nur die Details, dann weiß man vielleicht nicht, ob beispielsweise der Roboterkopf nur "herumliegt" oder eigentlich ein Teil eines gezeichneten "ganzen Roboters" ist.

Funktionen sind jedoch nicht nur zur besseren Strukturierung da, sondern können auch mehrfach an verschiedenen Stellen des Programms verwendet, d.h. aufgerufen, werden:

Das folgende Beispiel zeichnet mehrere unterschiedlich große und farbige Kreise und Vierecke. Die fürs Zeichnen notwendigen Variablen werden global deklariert und in `setup()` initialisiert. Die Variable `numBodyParts` enthält dabei die Anzahl der noch zu zeichnenden Kreise und Vierecke. Des Weiteren ruft `setup()` die `updateValues()`-Funktion auf, um zufällige Werte für die Variablen `radius` und `diameter` festzulegen.

Das eigentliche Zeichnen passiert in der Funktion `bodyParts()`, die pro Funktionsaufruf einen Kreis und ein Viereck zeichnet und die Anzahl an Figuren, die noch zu zeichnen sind, um 1 verringert. Sie verwendet auch `updateValues()`, um die Koordinaten und Größe der nächsten zwei Figuren zu berechnen. In `draw()` selbst passiert nicht viel: Es wird lediglich die Funktion `bodyParts()` genau `numBodyParts`-mal aufgerufen.

An diesem Beispiel sehen Sie, dass die Funktion `updateValues()` an zwei unterschiedlichen Stellen verwendet wird.

```
float radius;
float diameter;
float x;
float y;
int numBodyParts;

void setup()
{
    size(600, 400);
    x = 0.25f * width;
    y = 0.5f * height;
    updateValues();
    numBodyParts = 6;
}
```

```

void draw()
{
  if(numBodyParts > 0) // wir brauchen keine Schleife,
                      // da draw wiederholt ausgeführt wird.
  {
    bodyParts();
  }
}

void bodyParts()
{
  int r = int(random(255));
  int g = int(random(255));
  int b = int(random(255));

  fill(r, g, b);
  ellipse(x, y, diameter, diameter);

  fill(255 - r, 255 - g, 255 - b);
  rect(x - radius, y, diameter, diameter);

  float oldRadius = radius;
  updateValues();
  x = x + radius + oldRadius;
  numBodyParts--;
}

void updateValues()
{
  radius = random(40) + 10;
  diameter = radius * 2;
}

```

## 7.4 Funktionen mit Parametern

Funktionen sind aber noch mächtiger. Sie können noch mehr, als nur Programmcode auslagern oder zusammenfassen. Durch den Einsatz von **Parametern** lässt sich die Wirkung einer Funktion variieren und damit individueller gestalten. Parameter sind Variablen innerhalb von Funktionen. Diese besonderen Variablen erhalten ihre Werte nicht durch direkte Zuweisung, sondern die Werte werden beim Funktionsaufruf **übergeben**. Dadurch geben sie der Funktion ein wenig Spielraum, sodass eine Funktion nicht nur auf ein sehr konkretes Problem zugeschnitten ist, sondern ähnliche Probleme durch Aufruf der gleichen Funktion (aber mit jeweils anderen Parametern) lösbar sind.

Beispielsweise hat die Processing-Funktion `ellipse()` vier Parameter. Die ersten zwei geben die Position der Ellipse an und die letzten zwei Parameter die Breite und die Höhe. Der Funktionskopf für diese Funktion sieht wie folgt aus:

```
void ellipse(float a, float b, float c, float d)
```

Der Aufruf der Funktion

```
ellipse(10, 20, 50, 30);
```

bestimmt hier bereits welche Werte die Parameter `a` (`= 10`), `b` (`= 20`), `c` (`= 50`) und `d` (`= 30`) erhalten. Dadurch, dass diese Werte beim Aufruf bestimmt werden können, ist es möglich die Ellipse an verschiedene Orte zu platzieren und in verschiedenen Größen darzustellen. Ist die Länge und Breite der Ellipse gleich, erhalten wir die Sonderform der Ellipse, einen Kreis.

Beispiel: Der rote Kreis aus dem vorhergehenden Beispiel soll an verschiedenen Positionen gezeichnet werden können. Um das zu ermöglichen, führen wir zwei Parameter ein, einen für die x-Koordinate und einen für die y-Koordinate der gewünschten Position des Kreises. Wir erweitern dazu den Funktionskopf um zwei Parameter:

```
void drawRedCircle(int x, int y) {
    fill(255, 0, 0);
    ellipse(x, y, 50, 50);
}
```

Die Parameter sind vom Typ `int`, da ganzzahlige Koordinaten ausreichen. Statt den fixen Koordinaten (50, 50) verwenden wir nun die Parameter `x` und `y`. Beim Funktionsaufruf ist es jetzt möglich, Koordinaten zu übergeben und rote Kreise an verschiedenen Positionen zu zeichnen:

```
void draw() {
    drawRedCircle(25, 100);
}
```

```
drawRedCircle(75, 40);
}
```

Im `draw()` wurde diesmal zweimal die Funktion `drawRedCircle()` aufgerufen, einmal soll der Kreis an der Stelle (25, 100) gezeichnet werden und einmal an der Stelle (75, 40).

Mit nur kleinen Änderung ist es jetzt möglich diesen Kreis mehrmals an verschiedenen Stellen zu zeichnen. Diese Funktion kann noch mit weiteren Parametern erweitert werden. Z.B. kann auch die Größe des Kreises als Parameter gesetzt werden oder auch der Rotton (oder allgemein die Farbe) des Kreises.

```
void drawRedCircle(int x, int y, int size, int redValue) {
    fill(redValue, 0, 0);
    ellipse(x, y, size, size);
}
```

Es ist auch möglich die Parameter an sich noch komplexer berechnen zu lassen, bevor diese an die Funktion übergeben werden. So lassen sich zum Beispiel einfache Berechnungen innerhalb des Funktionsaufrufs durchführen:

```
void draw() {
    int size = 50;

    drawRedCircle(size - 25, 100);
    drawRedCircle(size + 25, 40);
}
```

Hier wird zuerst die Berechnung innerhalb der Klammern durchgeführt. Anschließend wird das jeweilige Ergebnis als Parameterwert übergeben und die Funktion damit ausgeführt.

Auch Funktionsaufrufe können als Parameter eingesetzt werden. Hier wird der Rückgabewert der Funktion als Parameterwert an die neue Funktion übergeben. Wie genau so eine Funktion mit Rückgabewert aussieht, wird im übernächsten Abschnitt im Detail beschrieben.

## 7.5 Funktionen und Rückgabewerte

Der erste Teil des Funktionskopfs ist der Rückgabotyp. Der Rückgabotyp gibt Auskunft, welche Art von Information von dieser Funktion als “Ergebnis” zurückgeliefert wird. Der Rückgabewert steht an erster Stelle im Funktionskopf. Betrachten wir noch einmal die Funktion `drawRedCircle()`.

```
void drawRedCircle(int x, int y, int size, int redValue)
```

Der Funktionskopf für diese Funktion beginnt mit dem Schlüsselwort `void`. `void` bedeutet “leer”, bzw. hier so viel wie “kein Rückgabotyp”, das heißt, diese Funktion liefert keinen Rückgabewert bzw. keine Information zurück. Funktionen, die keinen Wert zurückliefern, nennt man auch **Prozeduren**. Keinen Rückgabewert braucht man zum Beispiel zum Zeichnen geometrischer Objekte im Sketch-Fenster, wie `line()`, `rect()`, `triangle()`, `text()`, oder auch wenn nur Einstellungen im Speicher verändert werden oder etwas ausgegeben wird, z.B. `fill()`, `noFill()`, `background()` oder `print()`.

Ein Beispiel für eine in Processing verfügbare Funktion *mit* Rückgabewert ist die Funktion `random()`, siehe auch [https://processing.org/reference/random\\_.html](https://processing.org/reference/random_.html)

Gemäß der Dokumentation erzeugt die Funktion `random()` eine Zufallszahl. Sie erwarten sich von der Funktion also, wenn sie aufgerufen wird, dass diese eine Zufallszahl zurückliefert, die dann weiter verwendet werden kann. Der vorletzte Absatz in dieser Dokumentation beinhaltet:

**Returns**      `float`

*Returns* gibt in der Processing Dokumentation immer den Datentyp des Rückgabewerts der Funktion an. Bei `random()` ist der Rückgabewert also vom Datentyp `float`. Das bedeutet, dass die Zufallszahl, die die Funktion generiert und die Sie dann als Ergebnis zurückgeliefert bekommen, eine Gleitkommazahl ist.

Weitere Beispiele für Processing Funktionen mit Rückgabotyp sind etwa:

```
int round(float num)
    rundet eine Gleitkommazahl num auf die nächstgelegene ganze Zahl
    (kaufmännisches Runden) und liefert diese ganze Zahl zurück

int abs(int num)
    berechnet den Absolutbetrag einer ganzen Zahl num:  $|num|$ 
    und liefert diesen als ganze Zahl zurück

float sqrt(float num)
    berechnet die Quadratwurzel einer Gleitkommazahl num  $\sqrt{num}$ 
    und liefert diese als Gleitkommazahl zurück

float pow(float n, float e)
    berechnet  $n^e$ 
    und liefert das Ergebnis dieser Berechnung als Gleitkommazahl zurück
```

Von all diesen Funktionen erwarten Sie ein Ergebnis, das Sie in Form eines Wertes eines bestimmten Datentyps, z.B. als `float` oder `int`, erhalten. Alle oben genannten Funktionen liefern einen Wert. Diesen können Sie zum Beispiel in einer Variable speichern und dann weiter verarbeiten.

Beispiel: Die folgende Funktion summiert in einer Schleife alle Zahlen von 0 bis inklusive der übergebenen Zahl auf und gibt das Ergebnis zurück. Da die Summanden alle vom Typ `int` sind, wird auch das Ergebnis, welches in `sum` gespeichert ist, wieder vom Typ `int` sein.

```
int sumNumbers(int end) {  
    int sum = 0;  
  
    for (int i = 1; i <= end; i++) {  
        sum += i;  
    }  
  
    return sum;  
}  
  
void setup() {  
    int result = sumNumbers(5);  
    println(result);  
}
```

Ein wichtiges Schlüsselwort ist in diesem Zusammenhang `return`. Jede Funktion, die einen Rückgabebetyp hat, hat im Rumpf mindestens ein `return`, um Processing mitzuteilen, welcher Wert zurückgeliefert werden soll. Im Beispiel bedeutet `return sum`; dass der Wert in der Variable `sum` zurück geliefert werden soll. Den Wert, der zurückgegeben wird, nennt man **Rückgabewert**.

Eine besondere Eigenschaft von `return` ist, dass nicht nur der Rückgabewert übergeben wird, sondern auch der Ablauf der Funktion an dieser Stelle beendet wird und im Programm an die Stelle nach dem Funktionsaufruf zurückgesprungen wird und es von dort aus weiter ausgeführt wird.

## 7.6 Typische Fehler bei Funktionen

In diesem Abschnitt werden die häufigsten Denkfehler und Fehlermeldungen im Zusammenhang mit Rückgabetypen und Rückgabewerten vorgestellt.

### Datentypkonflikt beim Rückgabewert

Der Rückgabewert muss immer vom gleichen Datentyp sein, wie er im Funktionskopf angegeben ist oder implizit gecastet werden können. Verändert man beispielsweise den Datentyp der Variable `sum` im obigen Beispiel auf `float`, dann meldet Processing

Type mismatch: cannot convert from float to int

weil Processing bei `return sum;` versucht, den Rückgabewert in den Datentyp `int` umzuwandeln (`float` kann nicht implizit auf `int` gecastet werden).

Alle Datentypen, die Sie bisher kennen, können als Rückgabetypen verwendet werden. Wenn die Funktion eine Zahl zurückliefert, dann sollte der Rückgabetypp `int` oder `float` sein. Falls die Funktion eine "Ja"/"Nein" bzw. "Wahr"/"Falsch" Frage beantwortet, dann ist der Typ `boolean` ein geeigneter Rückgabetypp.

### Speichern von Rückgabewerten

Wenn Sie eine Funktion mit Rückgabewert aufrufen, denken Sie daran, den Rückgabewert dann auch zu speichern bzw. direkt zu verwenden! Benötigen Sie den Rückgabewert nur an der Stelle des Funktionsaufrufs, können Sie ihn direkt verwenden. Möchten Sie jedoch den Rückgabewert mehrfach weiter verwenden, speichern Sie ihn in einer Variable:

Beispiel: Die Funktion `sumOfEvenNum()` liefert die Summe aller geraden Zahlen zwischen dem Startwert `start` und dem Endwert `end` inklusive.

```

void setup() {
  sumOfEvenNum(1, 5);
}

//sumOfEvenNum Definition
int sumOfEvenNum(int start, int end) {
  int sum = 0;

  for (int i = start; i <= end; i++) {
    if (i % 2 == 0) { //wenn i eine gerade Zahl ist
      sum += i;      //addiere zur Summe die gerade Zahl i dazu
    }
  }

  return sum;
}
  
```

Das erwartete Ergebnis nach dem Funktionsaufruf im `setup()` ist 6 (weil  $2 + 4 = 6$ ). Beim Ausführen erscheint dieser Wert aber weder im Sketch-Fenster noch in der Konsole.

Sie müssen das Ergebnis, das Sie von der Funktion erhalten, auch explizit speichern (1) oder zumindest verwenden (2):

- (1) Ergebnis wird in einer Variable, hier `int result`, gespeichert und ist somit wiederverwendbar:

```
void setup() {  
  int result = sumOfEvenNum(1, 5);  
  println(result);  
}
```

- (2) Ergebnis wird direkt verwendet, z.B. innerhalb von `println` (und somit auf der Konsole ausgegeben):

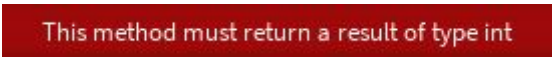
```
void setup() {  
  println(sumOfEvenNum(1, 5));  
}
```

In diesem Fall wird der Rückgabewert nicht gespeichert. Daher kann er nur an dieser einen Stelle einmalig verwendet und nicht weiterverwendet werden.

Falls Sie also nach einem Funktionsaufruf ein Ergebnis erwarten, aber keines sehen, könnte das eine mögliche Fehlerquelle sein.

## Kein Rückgabewert / Fehlendes return

Eine Fehlermeldung, die beim Erstellen einer Funktion mit Rückgabewert auftauchen könnte, ist:



This method must return a result of type int

Diese Fehlermeldung weist darauf hin, dass für die Funktion (Processing verwendet hier ausnahmsweise die Bezeichnung `method` statt wie üblich `function`) ein Rückgabewert fehlt, in diesem Beispiel etwa vom Typ `int`. Diese Fehlermeldung tritt auf, wenn innerhalb des Funktionsrumpfs überhaupt kein `return` steht. Es ist hier also im Funktionsrumpf die `return` Anweisung hinzuzufügen. Auch unter folgendem Umstand kann dieser Fehler auftreten:

Beispiel: Die Funktion `isEven()` liefert einen Wahrheitswert. Falls der Wert in `n` eine gerade Zahl ist, liefert die Funktion als Antwort `true`.



```
boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
  }
}
```

Auch hier wird dieselbe Fehlermeldung angezeigt, obwohl ein `return` vorhanden ist. Denn was passiert, wenn `n` keine gerade Zahl ist - dann wird die `if`-Anweisung übersprungen und das `return` gar nicht erst ausgeführt. Processing "erreicht" somit das `return` nicht. Damit gäbe es keinen Rückgabewert. Aus diesem Grund wird auch in diesem Fall die Fehlermeldung "This method must return a result of type boolean" ausgegeben. Es ist also sicherzustellen, dass - besonders bei Schleifen, Verzweigungen und verschachtelten Funktionen, in jedem möglichen Programmablauf jedenfalls eine `return` Anweisung ausgeführt wird.

Um den Fehler zu beheben, wäre im obigen Beispiel folgender Ansatz möglich.

```
boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
  } else {
    return false;
  }
}
```

Aber auch dieser Ansatz ist zulässig:

```
boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
  }
  return false;
}
```

Diese Variante ist möglich, da nach der Ausführung von `return` die Funktion an jener Stelle gleich abbricht und der Rest der Funktion nicht mehr ausgeführt wird. Daher wird nur für ungerade Zahlen der Wert `false` zurückgeliefert. Ist `n` gerade, so wird das `return` innerhalb der `if` Anweisung ausgeführt und die Funktion wird beendet ohne, dass das zweite `return` ausgeführt wird.

Es geht aber auch noch kürzer:

```
boolean isEven(int n) {
  return n % 2 == 0;
}
```

Das Ergebnis des Vergleichs (das ja stets ein boolean Wert ist!) wird ohne Zwischenspeicherung als Ergebnis geliefert.

Allerdings gibt es auch Fälle, wo (rein logisch gesehen) in jedem Fall ein Rückgabewert geliefert wird, aber dennoch Processing jammert, dass ein Rückgabewert fehlt:

```

boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
  } else if (n % 2 != 0) {
    return false;
  }
}

```

Für uns Menschen ist klar, dass `n` nur entweder gerade oder ungerade sein kann und dass daher unsere Funktion `isEven` in beiden Fällen einen Wahrheitswert zurückliefert. Processing "irrt" in diesem Fall, weil Processing die Bedingungen bei seiner Programmanalyse nicht auswertet, sondern nur nach Programmzweigen ohne `return` Ausschau hält. So könnte zum Beispiel die erste Bedingung auch `if (n % 2 == 0)` und die zweite Bedingung `else if (n % 3 == 0)` lauten. In diesem Fall würden wir einen `else`-Zweig zwingend benötigen, um immer einen Rückgabewert für diese Funktion gewährleisten zu können.

Jedenfalls weist solch eine Fehlermeldung, wenn der Code für uns Menschen fehlerfrei erscheint, Processing ihn jedoch anders interpretiert, meist auf einen schlechten Programmierstil hin. Grundsätzlich sollte man versuchen immer alle möglichen auftretenden Fälle zu berücksichtigen, um sicherzustellen, dass jedenfalls eine `return`-Anweisung erreicht wird.

## Unreachable code

Die Fehlermeldung

Unreachable code

wird dann angezeigt, wenn Processing erkennt, dass es Programmcode gibt, der aufgrund des Funktionsabbruchs an einer `return` Anweisung nie ausgeführt werden kann. Das kann beispielsweise ein Codeblock sein, der direkt nach einem `return` folgt.

```

boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
    println("I returned true");
  } else {
    return false;
    println("I returned false");
  }
}

```

Auch im folgenden Fall erkennt Processing, dass Codeabschnitte nie erreicht werden können:

```
boolean isEven(int n) {  
    if (n % 2 == 0) {  
        return true;  
    } else {  
        return false;  
    }  
  
    println("I returned something");  
}
```

Sie sollten sich aber auch hier nicht darauf verlassen, dass Processing immer jeden un erreichbaren Code erkennt. Im folgenden Fall wird kein Fehler angezeigt:

```
boolean isEven(int n) {  
    if (n % 2 == 0) {  
        return true;  
    } else if (n % 2 != 0) {  
        return false;  
    }  
  
    println("I returned something");  
    return false;  
}
```

Der `if`-Block und der `else if`-Block decken gemeinsam alle Möglichkeiten ab. Es wäre daher nicht möglich, egal welchen Wert `n` hat, dass das letzte `println()` und das `return false;` ausgeführt wird. Dennoch wird dieser Fall nicht von Processing erkannt. In diesem Fall kaschiert der erste Fehler wie bereits im Abschnitt "Fehlendes Return" beschrieben, den zweiten Fehler, dass bestimmte Codeteile nie erreicht werden können.

## 7.7 Funktionen $\longleftrightarrow$ `draw()` und `setup()`

Betrachtet man die beiden Bereiche `draw()` und `setup()` genauer, dann wird man vermutlich eine Ähnlichkeit zu den gerade erlernten Funktionen erkennen. Und tatsächlich sind sowohl `draw()` als auch `setup()` im Grunde auch Funktionen, ganz präzise: Prozeduren. Beide besitzen keinen Rückgabetyt und benötigen keine Parameter. Aber warum werden diese ausgeführt, ohne dass man den Befehl explizit wo hinschreibt, wie es auch bei allen anderen Funktionen der Fall war?

Das liegt daran, dass der Aufruf durch Processing im Hintergrund selbst geschieht. Wie auch in vielen anderen Programmiersprachen, braucht Processing einen Einstiegspunkt, von dem an beginnend das Programm ausgeführt wird. Es ist eine für uns Programmierer nicht sichtbare Funktion, die zunächst einmal die Funktion `setup()` aufruft, ohne dass Sie es sehen. Und anschließend in einer für uns nicht sichtbaren Schleife, die die `draw()`-Funktion wiederholt aufruft. Diese Schleife wird solange ausgeführt, bis der User z.B. das Fenster schließt. Sie sehen also, auch hier findet bereits eine Abstraktion statt.

Der ungefähre Code im Hintergrund könnte vereinfacht beispielsweise so aussehen:

```
setup();
boolean quit = false;

// Falls der User das Programm noch nicht beendet hat
while(!quit) {
    checkUserInput(); // prüft, ob der User das Programm beenden will
    draw(); // Ihr Code wird ausgeführt
    switchCanvas(); // das in draw() Gezeichnete wird sichtbar gemacht
}
```

Zuerst wird die `setup()` Funktion ausgeführt. Danach wird eine `boolean` Variable `quit` deklariert und initialisiert, die speichert, ob das Programm beendet werden soll oder nicht. In einer Schleife werden mit der Funktion `checkUserInput()` bei jedem Durchlauf die Usereingaben überprüft. In dieser Funktion wird beispielsweise auch geprüft, ob der User das Programm durch den Stop-Button beenden möchte. Falls dies der Fall ist, wird die Variable `quit` auf `true` gesetzt und die Schleife wird danach nicht mehr ausgeführt. Nachdem die Interaktionen mit dem User überprüft wurden, wird die `draw()` Funktion ausgeführt und dann alles, was intern gezeichnet wurde mit der Funktion `switchCanvas()` auch am Sketch-Fenster sichtbar gemacht. Vergessen Sie nicht, dass das Ergebnis von `draw()` daher erst nach dessen Ausführung im Sketch-Fenster ausgegeben wird. Es wird also NICHT jeweils unmittelbar nach Ausführung eines einzelnen (Zeichen-)Befehls innerhalb `draw()`, wie z.B. `rect()`, `arc()`, usw. aktualisiert.

Dieser Code ist nur eine Annäherung an das, was in Processing ablaufen könnte. Da wir keinen Einblick in den Code haben, wissen wir nicht, was genau im Hintergrund von Processing ausgeführt wird.

## 7.8 Sichtbarkeit von Parametern und Funktionsvariablen

Gerade im Bezug auf Funktionen spielt die Sichtbarkeit der Variablen eine wichtige Rolle. Auch hier gilt die Regel: Alles was innerhalb der geschwungenen Klammern deklariert wird, ist nur innerhalb dieser geschwungenen Klammern sichtbar und verwendbar. Auf eine Variable, die innerhalb einer Funktion deklariert wurde, kann nicht in einer anderen Funktion zugegriffen werden. Dies haben Sie bereits in Kapitel 3 anhand von `draw()` und `setup()`, die ja ebenso Funktionen sind, kennen gelernt. Besonders im Zusammenhang mit Parametern und Rückgabewerten ist dies zu berücksichtigen.

Beispiel: Gehen Sie folgendes Beispiel durch und überlegen Sie sich, welche Werte für die Variablen `a` und `b` in `println(a, b);` ausgegeben werden.

```
void setup() {
  int a = 5;
  int b = 10;
  addNumbers(a, b);
  println(a, b);
}

void addNumbers(int a, int b) {
  a += b;
}
```

Am Anfang kommt man vielleicht in die Versuchung zu glauben, die Variable `a` müsse den Wert 15 haben. Schließlich wird der Wert von `a` an die Funktion `addNumbers()` übergeben und innerhalb der Funktion verändert. Dies hat aber keine Auswirkung auf die Variablen `a` und `b` der `setup()`-Funktion. Denn die **übergebenen Werte** werden für die Funktion **kopiert**. Die Veränderung findet daher in der Kopie statt, nicht aber in der ursprünglichen Variable! Daher sind auch Rückgabewerte notwendig, um Veränderungen wieder zurückzuliefern.

Soll das Ergebnis von `addNumbers(a, b);` in die Variable `a` in `println(a, b);` "aktualisiert" werden, dann wäre eine korrekte Version für dieses Beispiel folgende:

```
void setup() {
  int a = 5;
  int b = 10;
  a = addNumbers(a, b);
  println(a, b);
}

int addNumbers(int a, int b) {
  a += b;
  return a;
}
```

```
    return a;  
}
```

Hier wird in `addNumbers()` der in der Kopie von `a` berechnete Wert mit `return` zurückgeliefert. Entsprechend muss der Rückgabebetyp angepasst werden. In `setup()` wird der Rückgabewert an die Variable `a` zugewiesen. `a` erhält dadurch diesmal tatsächlich den in `addNumbers()` berechneten Wert.

## 7.9 Dokumentation von Funktionen

Die Dokumentation einer Funktion mag zuweilen als lästige Arbeit empfunden werden, aber sie ist unerlässlich und von großer Bedeutung, denn sie hilft anderen Programmiererinnen und Programmierern dabei, die Funktion besser zu verstehen und korrekt anzuwenden, ohne den Code Zeile für Zeile zu lesen. Es ist daher ratsam, auf ein sorgfältiges Dokumentieren der selbst programmierten Funktionen von Beginn an Wert zu legen. Nicht zuletzt hilft die Dokumentation auch Ihnen selbst als Programmiererin oder Programmierer - etwa, wenn Sie Programme nach längerer Zeit wieder verwenden möchten und sich nicht mehr an jedes Detail erinnern.

Bei einfachen Funktionen sagt ein aussagekräftiger Funktionsname bereits vieles über die Funktion aus. Trotzdem gehört eine Dokumentation zum guten Programmierstil. Spätestens bei komplexeren Funktionen, wo der Funktionsname nicht mehr alles Wichtige ausdrücken kann und mögliche Parameter nicht selbsterklärend sind, sind Funktionen genau zu dokumentieren, um den Anwenderinnen und Anwendern die Verwendung der Funktion zu erleichtern.

### Was ist Dokumentation?

Die Dokumentation einer Funktion beinhaltet nicht nur eine Beschreibung, was die Funktion macht, sondern auch, welche Parameter sie verwendet, wofür diese stehen und Informationen zu ihrem Rückgabewert. Außerdem sollten alle **Vorbedingungen und besonderen Eigenheiten**, die alleine vom Funktionsnamen nicht abgeleitet werden können, notiert werden. Einen Anhaltspunkt zur Dokumentation liefert die Processing API. Diese besteht nämlich zu einem guten Teil aus der Dokumentation jener Funktionen, die in Processing verfügbar sind.

Als mögliches Beispiel die Dokumentation zur Processing Funktion `size()`:

[https://processing.org/reference/size\\_.html](https://processing.org/reference/size_.html)

Die Beschreibung der Funktion beinhaltet neben ihrer eigentlichen Funktion auch Anwendungsbeispiele sowie Hinweise zu besonderen Eigenheiten: Im Falle von `size()` etwa kann man aus der Dokumentation herauslesen, dass diese Funktion nicht überall und jederzeit und nicht beliebig oft aufgerufen werden kann.

Je besser solche Eigenheiten dokumentiert werden, desto weniger wird es den Anwender/die Anwenderin später überraschen, wenn etwas nicht wie gedacht funktioniert.

## Wie dokumentiert man?

Dokumentationen werden in Form von Kommentaren über den zugehörigen Funktionskopf geschrieben. So ist sie auf den ersten Blick verfügbar und ersichtlich.

Wichtig ist, dass sie alle relevanten Informationen enthält, um die Funktion zu verstehen und korrekt einzusetzen. Bei komplexen Funktionen ist außerdem darauf zu achten, keine Eins-zu-Eins-Übersetzung des Codes zu schreiben (z.B. "Es wird eine Variable leftOffset angelegt für den Offset von links. Dann wird Variable topOffset angelegt ... " etc.). Besser ist es die Wirkung der Funktion als Gesamtes in knapper Form so prägnant wie möglich zusammen zu fassen.

Die Dokumentation für die Funktion "maximum" könnte in etwa so aussehen:

```
/*
  Gibt die größte von drei gegebenen ganzen Zahlen zurück
  a: erste Zahl
  b: zweite Zahl
  c: dritte Zahl
*/
int maximum(int a, int b, int c) {
  int max = a;

  if (max < b) {
    max = b;
  }

  if (max < c) {
    max = c;
  }

  return max;
}
```

## Dokumentieren mit JavaDoc

Mit speziellen Tools (z.B. Doxygen<sup>1</sup>), können aus Kommentaren von Funktionen auch formatierte und strukturierte Dokumentationen als HTML Seiten oder LaTeX Manuals generiert werden und die Ausgabe als RTF (MS-Word), PostScript oder hyperlinked PDF unterstützen. Dafür sind diese Kommentare in einer bestimmten Form zu beschreiben. Ein weit verbreiteter Stil für solche Generatoren ist der JavaDoc Style<sup>2</sup> von Oracle.

Der allgemeine Aufbau einer Funktionsdokumentation im JavaDoc Style ist im Folgenden anhand des obigen Beispiels dargestellt:

<sup>1</sup> <http://www.stack.nl/~dimitri/doxygen/>

<sup>2</sup> <http://www.oracle.com/technetwork/articles/java/index-137868.html>



```
/**
 * Gibt die größte von drei gegebenen Zahlen zurück
 *
 * (funktioniert für positive und negative Werte)
 *
 * @param a erste Zahl
 * @param b zweite Zahl
 * @param c dritte Zahl
 * @return Maximum der drei Zahlen
 */
int maximum(int a, int b, int c) {
    int max = a;

    if (max < b) {
        max = b;
    }

    if (max < c) {
        max = c;
    }

    return max;
}
```

Der Dokumentations-Kommentarblock wird in JavaDoc mit `/**` eingeleitet.

Es werden nach dem Schrägstrich jetzt **zwei Sterne** verwendet anstatt des herkömmlichen einzelnen Sterns für einen herkömmlichen Kommentarblock.

Die erste Zeile beinhaltet eine Kurzbeschreibung der Funktion. Der nächste Absatz enthält detaillierte Informationen. Hier werden z.B. besondere Eigenschaften der Funktion dokumentiert. Danach folgen die Beschreibungen der Parameter. Für jeden Parameter ist ein separater Eintrag anzulegen, welcher mit `@param` eingeleitet wird. Dem folgt der Name und die Verwendung des Parameters. Als letztes wird der Rückgabewert kurz beschrieben. Dieser wird mit `@return` eingeleitet. Gibt es keinen Rückgabewert (steht als Rückgabotyp also `void`) oder werden keine Parameter benötigt, dann werden die entsprechenden Zeilen ausgelassen.