5 Funktionen

- Funktionen
- Funktionen definieren
- Parameter übergeben
- Terminierung und Rückgabe
- Rekursive Funktionen

5.0 Funktionen

Ich bin froh zu diesem Kapitel zu kommen, denn ab hier bekommt das Programmieren Stil und Wiederverwendbarkeit. Ein Funktion ist nichts anderes als ein paar zusammengefasste Anweisungen, die immer wieder aufgerufen werden können. Das verleiht uns die Möglichkeit sehr flexibel und Problemorientiert zu schreiben. Ausserdem wird der Quelltext durch seine Wiederverwendbarkeit möglichst klein gehalten. Das erhöht die Lesbarkeit und der Code ist um einiges weniger auf Fehler anfällig. Ich kann mir das Programmieren ohne Funktionen eigentlich gar nicht mehr vorstellen.

Die ECMA-262 Spezifikation besagt, dass man zwischen eigenen, selbst geschriebenen Funktionen („Programmfunktion") und den in Processing eingebauten Funktionen („Intern-Funktion") unterscheiden sollte. Das bedeutet nichts anderes als, dass Processing bereits einige Funktionen für uns bereithält um uns das Programmier-Leben einfacher zu gestalten. Wenn wir z.B. einen Kreis zeichnen wollen, hält Processing die Funktion *ellipse()* für uns bereit. So können wir einen Kreis Zeichnen ohne zu wissen wie das eigentlich geht. Sonst müssten wir eine „male Kreis" Funktion selbst schreiben. Das ginge zwar auch, würde aber natürlich mehr Mühe auf unserer Seite benötigen. So müssen wir das „Rad" nicht noch einmal erfinden. Von diesen „build-in" Funktionen gibt es einige, die du in der Processing Referenz nachlesen kannst. Also, bevor du dir die Mühe machst eine eigene Funktion zu schreiben, solltest du schnell schauen ob es so etwas nicht schon in Processing existiert. Daher sollte man seine Namensgebung für eigene Funktionen so wählen, dass man nicht bereits eingebaute Funktionen mit dem selben Namen überschreibt.

Während wir lernen eigene Funktionen zu schreiben machen wir uns mit diesen Fundamentalen bekannt:

Funktion Dekleration oder Funktion Definition
: Hier werden die Aktionen gebündelt aufgeschrieben.

Funktion Aufruf
: Das Ausführen einer Funktion oder Aufrufen einer Funktion.

Funktion Argument und Parameter
: Die Bereitstellung von Daten um eine definierte Funktion zu manipulieren. Wird im Funktionsaufruf übergeben.

Funktion Terminierung
: Beenden der Funktion und optional Rückgabe von Ergebnissen und Werten.

Variablen und ihr Gültigkeitsbereich
: Variablen, die im Funktionskörper definiert werden, gelten als lokal und können nicht von ausserhalb der Funktion gelesen oder gesetzt werden. Sie heissen lokale Variablen und werden nach der Ausführung der Funktion vom Speicher entfernt.

5.1 Funktionen definieren

Um eine einfache Funktion zu schreiben braucht es einen Namen und eine Anweisung im Funktionskörper (zwischen den geschweiften Klammern).
Die generische Struktur der Funktion:

```
typ funktionsName (param1,param2,param3,...)
{
   Anweisungen...
}
```

Das Schlüsselwort *void* startet die Dekleration der Funktion. Void heisst, dass kein Rückgabewert erwartet wird. Funktionen können nämlich Werte zurückgeben nachdem sie ausgeführt worden sind. Dazu später mehr. Wenn wir kein Wert zurück geben wollen, müssen wir void benutzen. Danach kommt ein selbst gewählter Name. Dieser Name wird beim Aufruf der Funktion benutzt und unterscheidet die Funktionen voneinander. Hier gelten für Funktion-Namen die selben Regeln wie bei Variablen-Namen. Als nächstes kommt die Parenthese, d.h. die Klammern (). Hier können später verschiedene Parameter eingepflegt und durch Kommas getrennt werden. Jetzt aber, wo wir erstmal keine Parameter benutzen, lassen wir diese einfach leer. Zu guter Letzt brauchen wir noch einen Anweisungsblock, der zwischen den geschweiften Klammern steht. Hier stehen die Kommandos, die ausgeführt werden wenn die Funktion aufgerufen wird.

Schreiben wir doch mal eine einfach Funktion, die uns im Output einen Text ausgibt.

Code:

```
void einfacheFunktion()
{
  println("Das ist einfach!");
}
```

Um die Funktion aufzurufen brauchen wir nur noch den Namen, die Parenthese und das Semikolon, welches die Aktion beendet :

Code:

```
einfacheFunktion();
```

So, hier das komplette Processing Skript.

Code:

```
void einfacheFunktion()
{
  println("Einfache Funktion Ausgefuehrt");
}

void setup()
{
    einfacheFunktion();
}
```

5.2 Parameter übergeben

Die Funktion erlaubt es uns also immer wiederkehrende Aufgaben zusammen zu fassen und von beliebiger Position aufzurufen. Was machen wir jedoch wenn wir zwar immer die gleiche Logik einer Funktion brauchen aber die Werte oder Objekte sich ändern. Wir möchten z.B. immer wieder eine Addition zweiter Werte berechnen. Jedoch ändern sich die Werte die wir addieren wollen. Wir brauchen daher eine Funktion die eine Logik besitzt, aber mit immer variablen Werten arbeitet. Diese Werte (Parameter) können wir dann beim Aufruf der Funktion mitliefern. Sie müssen bei der Funktionsdekleration mit Typ und Namen angeben werden. Bleiben wir erstmal bei dem einfachen Beispiel der Addition von drei verschiedenen Kommazahlen.

Code:

```
void zahlenAddition(float a, float b, float c)
{
   float ergebnis = a + b + c;
   println(ergebnis);   // als Ergebnis sollte hier die 11 stehen
}


void setup()
{
    zahlenAddition(2.2, 3.3, 5.5);
}
```

Hier noch ein anderes Beispiel. Hier sollen per *draw()* willkürlich Striche auf unsere Bühne gezeichnet werden.

Code:

```
void drawLines(float p1x, float p1y, float p2x, float p2y)
{
    line(p1x, p1y, p2x, p2y);
}

void draw()
{
    drawLines(random(100), random(100), random(100), random(100));
}
```

Natürlich kann man auch andere Datentypen übergeben. Hier ein Beispiel mit Übergabe eines Arrays. Hier gelten die gleichen Regeln wie bei Rückgabe von primitiven Datentypen.

Code:

```
int[] array = {10, 20, 30};
void passingArray(int[] arr){
   println(arr);
}
void setup() {
  passingArray(array);
}
```

5.3 Terminierung und Rückgabe

Bisher haben wir bei der Dekleration von Funktionen immer das Schlüsselwort *void* benutzt, somit wurde keine Rückgabe von Werten erwartet. Jetzt aber will ich die Rückgabe von Werten aufzeigen. Doch wie kann man Werte zurückgeben und wozu kann man es gebrauchen?
Anstatt des *void* Schlüsselworts sollten wir bei der Deklaration einer Funktion den Typ des erwarteten Werts angeben. D.h. wenn wir eine Ganzzahl erwarten, schreiben wir ein *int* anstatt des *void*. Um die Rückgabe auszulösen benutzen wir *return.*

Ein kleines Beispiel soll den Sachverhalt aufzeigen. Dazu benutze ich nochmals eine veränderte Form des „zahlenaddition" Codes vom vorletzten Beispiel. Mit dem Unterschied, dass jetzt die Funktion den ausgerechneten Wert zurück gibt. Bei der Dekleration der Funktion unbedingt drauf achten, dass der Typ der Funktion der gleiche ist wie der Wert der von return zurück gegeben wird.
Diesen zurückgegebenen Wert speichern wir in einer Variable des gleichen Typs.

Code:

```
float zahlenAddition(float a, float b, float c)
{
   return a + b + c;
}

void setup()
{
    float ergebnis = zahlenAddition(2.2, 3.3, 5.5);
    println(ergebnis);            // Sollte 11.0 als Ergebnis haben
}
```

Das *return* Schlüsselwort kann nicht nur Werte zurück geben, sondern beendet immer auch die Funktion. Somit wird Code, der nach dem *return* steht niemals ausgeführt werden - somit wird eine Funktion terminiert.

Die Rückgabe von anderen Datentypen folgt den selben Regeln wie die Primitiven. Im folgendem Beispiel übergeben wir ein Array, dividieren alle Zahlen durch zwei und lassen es uns zurück geben.

Code:

```
int[] array = {10, 20, 30};

int[] returnMyArray(int[] array) {
  for(int i = 0; i < array.length; i++) {
    array[i] = array[i] / 2;
  }
  return array;
}

void setup() {
  int[] arr = returnMyArray(array);
  println(arr);
}
```

5.4 Rekursive Funktionen

Eine rekursive Funktion ist eine Funktion, die sich selbst aufruft. Sie benutzt dazu den eigenen Funktionsnamen als Aufruf im Funktionskörper. Hier ein Beispiel um das Prinzip aufzuzeigen.

```
void bigTrouble()
{
   bigTrouble();
}
```

Ärger gibt es wenn man dies ausführen lässt, denn der Computer befindet sich sogleich in einer Endlosschleife und das Programm stürzt ab. So sollte man rekursive Funktionen nur in einer Bedingung ausführen um das Problem zu umgehen.
Ein klassische Methode um eine Rekursion zu benutzen ist die mathematische Berechnung des Faktorial einer Zahl. Das Faktorial von 3 ist 6 (3x2x1=6). Hierzu das Processing Beispiel.

Code:

```
int factorial(int x)
{
  if(x < 0)
  {
    return 0;
  } else if (x <= 1){
    return 1;
  } else {
    return x * factorial(x-1);
  }
}

void setup()
{
    int f =  factorial(5);
    println(f);
}
```

Wem die Mathematik nicht so liegt gibt es hier noch ein anders Beispiel welches horizontale Linien, die mit übergeben werden in einem bestimmten Abstand zeichnet.

Code:

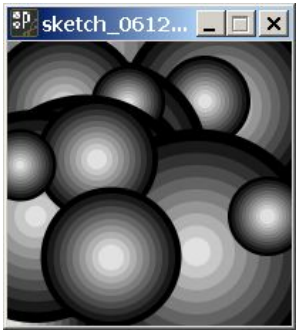
```
void recursiveLines(int lines, int spacing)
{
  line(0,i*spacing,100,i*spacing);
  i++;
  if( i < lines) recursivLines(lines,spacing);
}

void setup()
{
  recursiveLines(10,10);
}
```

Welche Ansatz besser ist, der Rekursive oder der Non-rekursive, kommt ganz auf das Problem an. Manche Probleme lassen sich durch rekursive Lösungen einfacher schreiben, sind aber langsamer als nicht rekursive Ansätze.
Rekursion ist auf jeden Fall besser, wenn man nicht weiss wie tief eine Datenstruktur verschachtelt ist - wie z.B das Auslesen von Verzeichnissen. In einem Verzeichnis können weitere beliebig verschachtelte Verzeichnisse liegen. Eine generelle Lösung ohne Rekursion wäre da schwierig.

5.5 Real life example

Um ein weiteres Beispiel für die Verwendung von Funktionen aufzuzeigen möchte ich gern ein Beispiel aus der Processing Referenz ansprechen. Es geht darum eine Funktion zu schreiben, die Kreise in verschiedenen Grauwerten von innen nach aussen zeichnet. Dabei werden die Grauwerte je weiter die Teilkreise nach aussen gezeichnet werden immer dunkler. Die Position und Grösse sollte zufällig gesetzt werden.



Code:

```
void setup()
{
  size(200, 200);
  background(51);
  framerate(5);
  noStroke();
  smooth();
}

void draw()
{
 draw_target((int)random(200),(int)random(200),(int)random(200),10);
 draw_target((int)random(200),(int)random(200),(int)random(200),10);
 draw_target((int)random(200),(int)random(200),(int)random(200),10);
}

void draw_target(int xloc, int yloc, int sizing, int num)
{
  float grayvalues = 255/num;
  float steps = sizing/num;
  for(int i=0; i<num; i++)
  {
    fill(i*grayvalues);
    ellipse(xloc, yloc, sizing-i*steps, sizing-i*steps);
  }
}
```