2 Kleine Einleitung für Nicht-Programmierer

„Lerne nicht eine Programmiersprache, sondern das Programmieren."

Diese kurze Einleitung ist nicht auf Vollständigkeit geschrieben, sondern soll nur einen Einblick in die Materie „Programmierung" geben. In den folgenden Kapiteln werden die hier beschriebenen Subkapitel nochmals intensiver besprochen.

- Kommentare
- Sag „Hallo"
- Ausdrücke
- Bedingungen
- Schleifen
- Funktionen
- Das Display Fenster
- Processing Programmfluss

Alle Programme jeder Programmiersprache folgen den selben Konzepten und lassen sich in fünf verschiedene Grundebenen aufteilen.

° Input (Interaktion)
Useraktionen oder variable Daten.

° Calculation
Mathematische Berechungen und Ausdrücke.

° Veryfy
Überprüfen von Bedingungen und Zuständen

° Loop
Wiederholen von Aufgaben mit meisst verschiedenen Parametern.

° Output
Bildschirm Darstellung oder Ausgaben wie z.B.
Speichern von Daten, interagieren mit anderen Programmen oder Aktionen.

Aber wie sag ich es dem Computer?

Da das Programmieren in einer philosopischen Weise Unerschöpflich ist, hat man verschiedene Ansätze die Dinge zu betrachten. Es ist irgendwie, sowie, eine kleine Welt nach seinen eigenen persönlichen Gesetz-mässigkeiten zu erbauen, zu pflegen und je nachdem einzelne Teile wiederum zu zerstören. Da kann absolutes Chaos herrschen oder klare formelle Formen und Konzepte.
Das Programmieren ist ein Kommunizieren mit dem Computer. Der eigenen Kreativität sind keine Grenzen gesetzt.
Jedoch gibt es Limitationen der einzelnen Programmiersprachen, die man meisst versucht durch „work-a-rounds" zu umgehen. Da ist einiges an Spontanität und Improvisationstalent gefordert.

Da es immer viele verschiedene Möglichkeiten gibt zum gleichen Ziel zu gelangen, entscheidet letztlich ein übersichtlicher, gut durchstrukturierter und dokumentierter Programmfluss. Damit erleichtert sich die Arbeit mit dem Code; kann besser mit Problem und Bugs umgehen; nach langen Pausen ist der Einstieg in die Struktur schneller, und andere Kollegen, die am gleichen Projekt mitarbeiten, können den kryptischen Code und die wirren Konzepte auch besser entziffern.
Natürlich gilt es immer, mit einem Auge auf die Performance des Programms zu schauen. Denn wer hat schon Lust, sich eine Applikation, wie z.B. ein „Jump and Run" Spiel mit „FPS 1.5" anzuschauen, jawohl – niemand!

Daher sollte man gerade als Anfänger, die Dinge, die man programmiert genauer anschauen. Die Zeit messen, die es braucht, zwischen dem Start und der Lösung von Aufgaben.
Somit bekommt man ein Gespür dafür, wie lange Rechenoperationen brauchen um bei verschieden Herangehensweisen ausgewertet und dargestellt werden.
Im weiteren Verlauf wird es auch ein Beispiel dazu geben.

Nun gut, Programme senden und empfangen also Informationen, und die Programmiersprache hält die Werkzeuge dafür bereit. Die Programmiersprache versteht aber leider kein Englisch oder Französisch, sondern hat ihre eigene Syntax und ihre eigene Grammatik. Sie ist zum Glück im Vergleich zur menschlichen Sprache auch einfacher gestrickt.

Nehmen wir mal an, der Computer könnte deutsch verstehen,
und wir würden ihm sagen:
Lass einen Ball auf dem Bildschirm hin und her hüpfen! Der Computer könnte es trotzdem nicht ausführen, denn mit dem Wort Ball wüsste er nichts anzufangen. Ihm würden auch wesentliche Eigenschaften fehlen, wie z.B. Farbe, Durchmesser, Position, Geschwindigkeit, Abprallverhalten, 2D -3D etc. Hm, solle viele Fragen?

Projezieren wir die Aufgabe mal in die echte Welt und Zeichnen einen Kreis auf ein Papier und beobachten uns dabei. Zuerst wählen wir den Stift mit dem gezeichnet werden soll, der bestimmt die Linienstärke. Einen zweiten Stift haben wir schon im Blickfeld, um die Füllfarbe bereitzustellen. Ganz automatisch wählen wir eine Position an dem wir den Kreis zeichnen wollen, überlegen uns eine Grösse und zeichnen los. Ja, das ist ein gutes Konzept um einen Kreis zeichnen zu wollen.

So sieht demnach unsere reelle Prozedur so aus:
1. Liniendicke (definieren)
2. Linienfarbe (definieren)
3. Füllfarbe (definieren)
5. Loszeichen (mit einer bestimmten Position und Grösse)

Genau dieses Konzept übertragen wir nun auf die Programmiersprache Processing und werden den Ball in der gleichen Reihenfolge beschreiben.

Dazu wählen wir die Werkzeuge die Processing dafür bereithält.

```
strokeWeight(2);                    // Definiert Liniendicke
stroke(255, 255, 255);              // Linienfarbe
fill(255, 0, 0);                    // Füllfarbe
ellipse(56, 46, 55, 55);            // Ellipse zeichnen
```

Was passiert hier?
Das Programm wird von oben nach unten (prozedural) abgearbeitet.

Der Aufruf:

`strokeWeight(2)` setzt die Linienstärke auf 2 Pixel

`stroke(255, 255, 255)` setzt die LinienFarbe in R,G,B

`fill(255, 0, 0)` setzt die Farbe der Füllung in R,G,B

`ellipse(56, 46, 55, 55)` zeichnet den Kreis mit der
horizontalen Bildschirmposition X von 56, vertikale Bild-schirmposition Y von 46, sowie einer Höhe und Breite von
55 Pixel. Schema: *ellipse(x, y, höhe, breite)*.

Dieses kleine Beispiel zeigt, dass egal welche Sprache wir benutzen, das Schema des Kreiszeichnens den selben Mustern unterliegt.

Wenn es dich interessiert wie das als Processing Programm aussieht
dann kopiere doch den folgenden Code in den Editor und drücke anschiessend die „Run" Taste, um den Code zu Kompilieren / Auszuführen.

Code:

```
void draw()
{
  strokeWeight(2);           // Definiert Liniendicke
  stroke(255, 255, 255);     // Linienfarbe
  fill(255, 0, 0);           // Füllfarbe
  ellipse(56, 46, 55, 55);   // Ellipse zeichnen
}
```

Solange du noch unsicher bist, kannst du, bevor du etwas in der Computersprache ausdrückst, lieber einmal die ganze Prozedur in der eigenen Muttersprache aufschreiben. Das hilft oft enorm.

Viel Zeit des Programmierens wird nicht zum schreiben von Code genutzt - denn, bevor man eine einzige Zeile schreibt, werden sämtliche Programm-Logiken und Zusammenhänge vorgedacht und konzipiert. Meistens auf einem Flowchart oder Blueprint. Das setzt breite Kenntinsse der Programmiersprache und Programmier-erfahrung voraus.

Die Essenz des „coding" ist eigentlich, Anweisungen zu formulieren, sowie diese in verschieden sinnvolle Einheiten gut zu gliedern.

Es passiert schnell, dass sich ein „Bug" (Fehler) in den Code einschleicht.
Daher sollte man sehr explizit, korrekt „reden" und darauf Obacht geben, dass es Processing richtig versteht und ausführt. Zu vieles Unstrukturiertes - und es endet in einer Kakofonie der Anweisungen. Da wird die Fehlersuche ab und zu zur einer leider langweiligen „Mission Impossible".

2.1 Kommentare – Dokumentation des Skripts

Um das Chaos unter Kontrolle zu halten, benutzen wir Kommentare. Sie werden vom Kompiler übergangen und dienen nur zur Berschreibung des geschriebenen Codes.

Ein einzeiliger Kommentar wird durch einen doppeltem Schrägstrich definiert.

Code:

```
// Das ist ein einzeiliger Kommentar

// Und noch eine anderer Kommentar

// Weil es so schön hier noch mehr...
```

Es ist auch möglich Kommentare über mehrere Zeilen zu schreiben:

Code:

```
/*
   Und hier ein Kommentar
   ueber mehrere
   Zeilen
*/
```

Tip:
Benutze Kommentare häufig und dokumentiere so deine Gedanken - Es hilft enorm zu wissen, was man sich gedacht hat, als man den Code geschrieben hat, besonders wenn es schon eine kleine Weile zurückliegt.

2.2 Sag „Hallo"

Um mal ein bisschen in die Welt des Programmierens einzutauchen, schreiben wir ein kleines Programm, was uns ein „Hallo – Willkommen bei Processing" in das Output Fenster ausgibt. Dazu öffnen wir die IDE „processing.exe" und schreiben folgendes Skript:

```
println("Hallo – Willkommen bei Processing");
```

Um das Skript zu kompilieren, dücken wir die „run" Taste. Es sollte sich kurze Zeit später ein neues Fenster öffnen, wichtiger aber, es sollte „Hallo…" im Output (unterer schwarzer Bereich der IDE) zu lesen sein.

Was passiert hier?
Es wird das Kommando println() (kurz: print line) ausgeführt. Dieses Kommando (Werkzeug) wird von Processing besreitgestellt. (Alle Processing Kommandos können in der Hilfe->Referenz nachgeschaut werden.) Jetzt weiss der Computer, dass er etwas ins Outputfenster schreiben soll. Um den Text dafür anzugeben, braucht es addtionelle Parameter, normalerweise „Argumente" genannt. Sie werden innerhalb der Klammer geschrieben.

Zeichenketten (Strings) müssen innerhalb von Anführungszeichen ("text") geschrieben sein, damit der Kompiler es versteht. Ich werde diesem Thema ein eigenes Kapitel widmen.

Das Semikolon ";" am ende der Zeile, beschreibt das Ende des Befehls und muss hinter jeder Aktion gesetzt werden, um Kopilierungsfehler zu vermeiden.

Der Kompiler wandelt also das Skript in Bytecode um (Maschiniensprache). Dafür liesst er den Code Buchstabe für Buchstabe und versucht den Sinn zu verstehen. Es ist wie einen Satz aus einem Buch zu lesen und versuchen den Inhalt zu verstehen. Es herrschen strenge Syntax Regeln, denn schon bei einer Kleinigkeit, wie das Vergessen nur eines Buchstabens, versteht der Computer nichts mehr, gibt einen Kompilierungsfehler aus und bricht ab. *nerv*

Das Output Fenster ist das wichtigste Tool, das dem Programmierer ermöglicht, in die Welt der Vorgänge und Werte des Programms in Echtzeit zu blicken. Damit kann man jeden Zustand eines Programms prüfen (Debug).
Ohne den „Output" ist der Programmierer quasi blind.

Tip:
Mehrere Zeichenketten oder Variablen können mit dem „+" Operator miteinander verbunden werden. z.B. so:

Code:

```
println("String 1" + "String2" + "String3");
```

Nach dem Kompilieren sollte „String 1String 2String 3" im Output stehen. Im weiteren Verlauf werden wir uns die Möglichkeiten des Outputs intensiv aneignen. Die folgenden Unterkapitel beschreiben kurz und knapp die Grundpfeiler der Programmierung und sollen das spätere Verständinss fördern.

2.3 Ausdrucke (Expressions)

Wir lernten vorher, dass Instruktionen wie println sinnlos sind ohne weitere Daten zu enthalten. Wenn diese „Argumente" miteinander druch Operatoren kombiniert werden, nennt man es „Expression". Es gibt literarische Ausdücke, wie unsere vorhergehende „String" Verkettung. Dem literarischen Ausdruck folgt der komplexe Ausdruck. Um das zu veranschaulichen, nehmen wir uns ein paar Variablen. Diese kann man sich wie Platzhalter (Container) für irgendwelche Werte vorstellen. Eine Variable hat einen bestimmten Typ, der nicht vermischt oder gewechselt werden kann (typisiert). Dazu später mehr.

Code:

```
int zahl_1 = 11;
int zahl_2 = 22;
println(zahl_1 + zahl_2);
```

Hier sehen wir nach dem Kompilierung eine „33" im Output stehen. Da der Kompiler weiss, dass es sich um Ganzzahlen handelt, wird eine Berrechung angestellt, anstatt „1122" auszugeben. Das würde nur passieren wenn man die Werte "11" und "12" als Zeichenkette definiert hätte.

Es ist auch möglich den literarischen Ausdruck mit einem variablen Ausdruck zu mischen.

```
println("Das ist eine Zahl " + zahl_1);
```

Oder gleichzeitig, eine Multiplikation durchführen.

```
println("Das Ergebniss" + (zahl_1 * zahl_2));
```

Aritmetische Berechungen kurz und knapp.

```
println( (2 + 8) * (50 / 2) - 50 );
```

Es tut gut daran, sich schon am Anfang der Programmierkarriere mit Ausdrücken auseinander zu setzen und vertraut zu machen. Denn der Term „Ausdruck" oder „Expression" wird häufig bei der Beschreibung eines Programmkonzepts genutzt und voraus gesetzt.

2.4 Bedingungen (Conditionals)

Fast jedes Programm besitzt Bedingungen. Logiken und Strukturen werden damit hinzugefügt, so dass das Programm schlauer auf Zustände reagieren kann. Um dies zu verdeutlichen, ein ganz weltliches Beispiel.
Ein Mädchen namens „Julia" hat keine Lust darauf nass zu werden, wenn sie jeden Morgen zur Schule geht. Deshalb schaut sie immer bevor sie aus dem Haus geht durchs Fenster und entscheidet ob sie einen Regenschirm braucht oder nicht. Sie ist schlau und verwendet grundlegende Logik. Sie trifft ihre Entscheidung anhand einer Bedingung.

Dieselbe Möglichkeit, nach einer Reihe von Optionen zu schauen, um darauf Entscheidungen in verschieden Zuständen zu erreichen, benutzen wir auch in der Programmierung – hier einige Beispiele.

- Wenn sich ein Ball auf dem Bildschirm bewegt und wir wollen, dass er an der Bildschirmkannte zurückprallt, setzten wir die Kantenposition in eine Variable und fragen die momentane Ballposition ab.

- Wenn die momentane Position grösser ist als die Gesetzte, dann wird die Ballgeschwindigkeit umgekehrt (90 Grad) und der Ball fliegt in eine andere Richtung.

- Das Abfragen, ob eine bestimmt Taste gedrückt oder die Maus eine neue Position hat.

- Wenn wir ein Passwort auf einer Website eingeben, wird abgefragt ob das Eingegebene dem gespeicherten Passwort gleicht. Wenn ja, dann Weiterleitung, sonst Fehlerausgabe.

Hier ein Codebeispiel:

Code:

```
// Erzeuge Variabel passwort
String passwort = "julia";

// Wenn die Bedingung zutrifft dann
if(passwort == "julia")
{       // Anweisung
         println("Passwort richtig");
}else{  // wenn nicht andere Anweisung
         println("Passwort falsch");
}
```

Die generische Struktur des Beispiels:

Wenn das eingegebene Passwort Julia ist, gebe aus:
-> „Passwort richtig",
-> ansonsten gebe aus:
-> „Passwort falsch"

```
In Pseudocode:

if(condition is met)
{
        then execute this line of code
}else{
        then execute this line of code instead
}
```

Die verschieden Operatoren, welche in Bedingungen benutzt werden können, beschreibe ich im Kapitel 4 - Kontrollstrukturen.

2.5 Schleifen (loops)

Damit unsere Programme ein weiters „Power-up" bekommen, werden wir die „Schleifen" besprechen. Sie sind da um „sich-wiederholende Aufgaben", mit wenig Schreibaufwand für uns auszuführen.

Nehmen wir mal an, du möchtest eine Zahlensequenz, von 1-5 in den Output ausgeben. Du könntest folgenden Code schreiben:

```
println(1);
println(2);
println(3);
println(4);
println(5);
```

Wenn sich aber die Zahlen oder die Längen der Sequenzen erhöhen, wird damit der Schreibaufwand nur unnötig länger. Wie man an diesem Beispiel gut erkennen kann.

```
println(10000000);
println(10000001);
println(10000002);
println(10000003);
println(10000004);
println(10000005);
println(10000006);
println(10000007);
println(10000008);
println(10000009);
println(10000010);
println(10000011);
println(10000012);
println(10000013);
println(10000014);
println(10000015);
… usw.
println(10002000);
```

O.K. – um sich das immerwährende Schreiben der Zahlen zu ersparen, hätte ich da noch einen Einfall. Man könnte ja die Zahl in einer Variable speichern und sie dann immer um die Zahl 1 erhöhen. Das könnte so aussehen:

```
int zahl = 10000000;
println(zahl);
zahl = zahl + 1;
println(zahl);
zahl = zahl + 1;
println(zahl);
zahl = zahl + 1;
println(zahl);
zahl = zahl + 1;
println(zahl);
… usw.
zahl = zahl + 1;
println(zahl);
```

Das erspart zwar schon merklich Zeit, ist uns aber immer noch viel zu viel Getippe. Deshalb kommen wir zur while – Schleife. Sie ist genauso üblich und wichtig, wie das Mehl, zum Backen von schmackhaften Brötchen.

Ich werde hier nur ein Beispiel zum Verständnis zeigen. Im weiteren Verlauf komme ich dann explizit noch einmal auf die verschieden Schleifen-Arten zu sprechen.

Die generische Struktur:

```
while(Bedingung)
{
   Anweisungen
   ...
}
```

Und so wird es programmiert:

Code:

```
int i = 0;      // Deklatiere Gabzzahl namens i
while(i < 100) // Scheife mit Bedingung(solang i kleiner als 100)
{
  println(i);  // Ausgabe von Wert i
  i = i + 1;   // Erhöhe i um 1
}
```

Was passiert mit dem obigen Code:
Erstmal definieren wir eine Variable als Ganzzahl (int) namens i. Durch das Schlüssel-wort "while" geben wir an:
Solange die Bedingung i kleiner als 100 ist wird der Code zwischen den geschweiften Klammern ausgeführt. Der da wäre: Schreibe die Variable i in den Output und erhöhe die Zahl um eins und überprüfe ob „i" immer noch kleiner ist. Und das solange bis die while Bedingung nicht mehr zutifft. Das spart massiv Zeit und Nerven – oder?

Wenn wir uns das Ausgegebene genauer anschauen, stellen wir fest, dass das Programm nur bis einschliesslich 99 gezählt hat. Es kann auch nicht höher, denn wenn i gleich 100 ist, kann es ja nicht gleichzeitig kleiner sein. Wir können aber den Operator tunen, indem wir „<=" (kleiner gleich) benutzen, anstatt „<". Somit trifft die Bedingung bis 100 zu. Die verschieden Operatoren werden zu einem späteren Zeitpunkt einzeln behandelt.

Um den Computer mal so richtig schön ins Nirvana ab zu schiessen, eignet sich die „while" Scheife hervorragend. Man lösche nur die letzte Anweisung i = i + 1; und starte das Programm aufs Neue.
(Vorsicht Gefählich! – Bitte keine Computer-Ersatzklagen an mich :)
Ohne diese Zeile gerät der Computer in eine endlos Schleife, denn i ist ab nun, für immer 0. Das bringt ihn so ins Schwitzen, dass er sich einfach aufhängt, oder je nach Kompiler, bei 250 000 Durchgängen abbricht um dies zu Verhindern.
(gilt nicht für Processing)

Die zweite Möglichkeit eine Schleife zu bilden ist, die for-Schleife zu benutzen. Anders als bei der while-Schleife, zählt man i schon als Argument im Kopf hoch. Hier ein Beispiel in hunderter Schritten:

Code:

```
for(int i = 0; i < 1000; i = i + 100)
{
      println(i);
}
```

Die generische Struktur der for-Schleife:

for (Vaiablen Definiton; Bedingung; Zählen)
{
   Anweisungen
   …
}

Die for-Schleife braucht drei Parameter, die durch ein Semikolon getrennt werden.
- Dekleration der Variable i (hier: i ist 0)
- Die Bedingung (hier: i kleiner 1000)
- Das Inkrementieren (Hochzählen von i + 100)

Zwischen den geschweiften Klammern steht der Code der ausgeführt werden soll.

2.6 Funktionen (Modularer Code)

Bis jetzt bestanden unsere Programme höchstens aus sechs Zeilen Code. Das ist nicht viel. Aber binnen kurzer Zeit werden unsere Programme komplexer und dadurch länger. Aus den sechs Zeilen werden ganz schnell 100, 500, ja selbst 5000 Zeilen sind nach einiger Zeit nicht unüblich. Darum brauchen wir Funktionen; um Befehle zusammen zu fassen und je nach Bedarf, beliebig ausführen zu können.

Funktionen halten das Skript modular, reduzieren den Code, lassen sich schnell an verschiedene Szenarieren anpassen und bringen mehr Effizienz beim Managen.

Man kann sich Funktionen als einen Art Block mit einem Namen vorstellen. Ein Block an Code, der mit verschieden Befehlen gefüllt ist. Den man immer wieder ausführen kann, optional sogar mit verschiedenen Argumenten (Parametern) um das Skript noch weiter zu generalisieren.

Die generische Struktur der Funktion:

void Name(Argumente)
{
  Anweisungen
  …
}

Ein kleines Beispiel dazu. Wir wollen, dass unser Programm uns begrüsst, mit „Hallo – Programm gestartet".Der Name der Funktion soll „startApplication" sein.

Exemplarisch:

```
void startApplication()
{
     println("Hallo – Programm gestartet");
}
```

Nun muss die Funktion nur noch aufgerufen werden:

```
startApplication();
```

Wenn wir es jetzt in Processing schreiben, müsste der Code so aussehen:

Code:

```
void startApplication()
{
    println("Hallo – Programm gestartet");
}


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

Hm, unser Funktionaufruf wird von einer anderen Funktion
namens „setup" aufgerufen – wieso das denn ?
„setup" ist eine von Processing vordefinierte Methode, die beim starten des Applets das ganze Programm initialisiert. Was heisst, sobald man nicht mehr prozeduralen Code schreibt (code wird von oben nach unten ausgeführt), sondern Modularen, braucht man „setup" um, eine Art von „Startschuss" zu geben. Andere Programmiersprachen benutzen anstatt „setup" oft auch „main".
Tip:
Es gibt noch mehr vordefinierte Funktionen, deren Namen sollte man nicht durch die eigene Namensgebung überschreiben. Sie sind in der Processing Dokumentation nachzulesen.

Jetzt ist es uns möglich, unsere Funktion, von jedem beliebigen Ort unseres Programms aus aufrufen zu können. Egal ob der User interagiert oder eine beliebig andere Bedingung eintritt.

Um seinen Code weiter zu generalisieren, kann man der Funktion verschiedene Parameter gleich beim Aufruf mitgeben.

Also schreiben wir eine Methode, der beim Aufruf, zwei Argumente übergeben (Ganzzahlen), miteinander multipliziert und einfach ins Ausgabefenster (Output) geschrieben werden.

Dazu der Processing Code:

Code:

```
void calculate(int number1, int number2)
{
  println("Ergebnis von ");
  println( number1 +" * "+ number2);
  println("ist: "+(number1 * number2));
}

void setup()
{
  calculate(99, 33);
}
```

Da alle Variablen in Processing strikter Typisierung unterliegen, (Variablentypen müssen vordefiniert und können im Nachhinein nicht mehr geändert werden) sind sie in der Definition (Konstruktor) der Funktion auch zu Typisieren, und durch Komma getrennt zwischen die runden Klammern geschrieben.

Damit es nicht so Langweilig bleibt, programmieren wir ein Skript dass nicht nur Text in den Output schreibt, sondern ins Applet selbst zeichnet. D.h. in das eigentliche Programm Fenster.

Im Demonstrationsbeispiel dazu, sollen beliebig viele horizontale Linien, in einem beliebigen Abstand zueinander gezeichnet werden. Klingt erstmal schwierig.

Bevor wir in die Tasten hauen, sollten wir also ein kleines Konzept dafür erarbeiten. Die Funktion soll Linien zeichnen, die sich nur in der vertikalen Position Y unterscheiden.

Hm, da würde doch eine Scheife den Job am besten erledingen. In jedem Durchlauf könnte man auch die vertikale Position der Linie errechnen.

Um ein Linie zu zeichnen, gibt uns Processing die Methode `line(x1,y1,x2,y2)` vor. Die Argument dafür sind:
- x1 steht für die X Koordinate des Anfangspunkts der Line
- y1 steht für die Y Koordinate des Anfangspunkts der Line
- x2 steht für die X Koordinate des Endpunkts der Line
- x2 steht für die X Koordinate des Endpunkts der Line

```
line(10,10,50,50);//zeichnet eine diagonale Linie
```

Das soll uns jetzt aber nicht weiter kümmern denn die Zeichenwerkzeuge werden auf einem anderen Blatt beschrieben. Hier gilt es nur Konzepte der Progammierung zu besprechen.

Sodann, scheiben wir eine Funktion, die mit zwei Ganzzahlen aufgerufen wird und diese dann in einer for-Schleife zeichnet.

Der Processing Code dazu könnte so aussehen:

Code:

```
void drawLines(int number, int space)
{

  for(int i = 0 ; i < number; i++)
  {
    line(0, space * i ,200, space * i);
  }

}


void setup()
{
  size(200,200);
  drawLines(20, 10);
}
```

Zur besseren Übersicht schreibe ich den nächsten Codeblock auf die nächste Seite.

Die kommentierte Version beschreibt das ganze etwas besser.

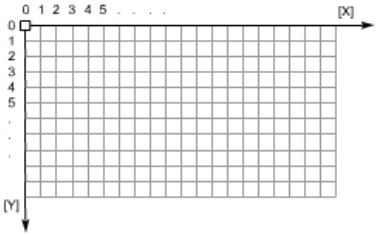
Code:

```
/*
  drawLines wird definiert und mit zwei
  Parametern vom Typ Ganztahl (int) ausgestattet.
*/

void drawLines(int number, int space)
{
  /*
    Die for-Schleife wiederholt sich bedingt durch
    den erhaltenen Parameter "number".
    Die Horizontale Position des Strichs errechnet
    sich durch den Parameter "space", multipizert
    mit der variable i, die sich bei jedem
    Durchlauf um 1 erhöht.
  */

  for(int i = 0 ; i < number; i++)
  {

   /*
   Linie wird mit ihren Parametern gezeichnet
   (Xpos_Line1, Ypos_Line1, Xpos_Line2, Ypos_Line1)
   */

   line(0, space * i ,200, space * i);

  }

}


void setup()
{

  // Definiert die Bühnengrösse in Höhe und Breite
  size(200,200);

  /*
    Ausführen der Funktion drawLines
    mit Linienanzahl 20 und einem Abstand von 10
    als Argumente -
    Spiel doch ein bisschen mit den Zahlen.
  */
  drawLines(20, 10);

}
```

Um es nochmal zu verdeutlichen. Das Konzept des Skriptes besteht darin, eine Funktion zu bauen, die zwei Ganzzahlen (int) empfängt, um in einer Schleife die gewünschten Linien zu zeichnen. Die Bedingung liegt auf der Anzahl der zuzeichnenden Linien. Die beiden mitgelieferten Argument (Parameter) „number" und „space" stehen für Anzahl und Abstand der Linien.

2.7 Das Display Fenster

Das Display Fenster unterliegt einem 2D kartesischen Koordinaten-system. Der Nullpunkt befindet sich oben links. Höhere Werte verschieben sich nach unten und rechts.



Um die Fenstergrösse zu definieren hält uns Processing das Kommando: `size(Breite, Höhe)` bereit. Damit können wir nun unser Fenster beliebig skalieren.
Im umgekehrten Sinn können wir auch die Grösse des Fensters auslesen. Dafür gibt es die Systemvariablen `width` und `height`. Diese werden bei Aufruf von `size()` gesetzt.

Hierzu ein kleines Beispiel zur Verdeutlichung:

Code:

```
size(600,800);
println("Breite des Fensters: " + width);
println("Höhe des Fensters: " + height);
```

Nach Ausführung des Programms, sollte sich ein leeres Fenster der Grösse 800x600 öffnen und der Output muss das bestätigen.

Einen sinnvollen Einsatz von `width` und `height` möchte ich gern als nächstes aufzeigen. Im Kapitel 2.0 zeigte ich Anfangs, wie man eine Ellipse zeichnet. Die kommende Aufgabe besteht darin sie immer in der Mitte des Fensters zeichnen zu lassen.

Code:

```
size(600,800);
int breite = 50;
int hoehe = 50
ellipse(width/2,height/2,breite,hoehe);
```

Als kleinen Tip:
Um ein Fenster in der selben Grösse wie die Bildschirmgrösse zu öffnen kann man folgendes schreiben.

Code:

```
size(screen.width, screen.height);
```

`screen.width` und `screen.height` liest die momentane Bildschirmauflösung aus und gibt diese zurück.

2.8 Processing Programmfluss

Bisher haben wir unsere Programme meistens prozedural, d.h. statisch geschrieben. Hier wird der Code von oben nach unten abgearbeitet. Jedoch ist eine Interaktion des Programms unmöglich.
Um dies erreichen zu können bietet Processing auch eine modulare Form an. Das Dokument teilt sich dann in zwei westentliche Teile.
`void setup(){.....};` und `void draw(){.....};`

° `void setup(){Anweisungen}` wird zu Beginn des Programms nur einmal aufgerufen. Es kann zur Initialisierung dienen,
wie z.B die Fenstergrösse.

° `void draw(){Anweisungen}` wird kontinuierlich ausgeführt bis das Programm beendet wird. Es gibt aber auch die Möglichkeit es per Code auszuschalten `noLoop()` oder wieder einzuschalten mit `loop()`.

Wagen wir uns doch gleich an ein Beispiel. Ich möchte, dass ein Kreis in der Mitte unseres Applets immer von links nach rechts wandert. Wenn es am rechten Rand angekommen ist sollte es wieder nach links springen... und immer so weiter.

Code:

```
// Definiere Kommazahl xPosition mit einem Wert von 0.0
// Diese Position wird später der Elispse zugeweiesen

float xPosition = 0.0;

// Programm Initialisierung
void setup()
{
  size(200, 200);
}


// Programm Logik
void draw()
{

  background(204);               // Zeichne Hintergrund grau
  xPosition = xPosition + 0.1;   // Erhöhe x um 0.1
  if (xPosition > width)         // Bedingung wenn xPosition
  {                              // grösser ist als die Breite
    xPosition = 0;               // dann zurück an Anfang
  }
  ellipse(xPosition, height/2, 20, 20);  // Zeichne Kreis
}
```

Um ein bisschen Interaktion in unsere neuen Erkenntnisse zu bringen, werden wir die Maustaste abfragen. Wenn sie gedrückt worden ist, sollte der Kreis anfangen zu laufen. Beim loslassen solle der Kreis stoppen. Dazu müssen wir wissen, wie wir die Maustaste abfragen. Ein Blick in die Processing Referenz verrät uns das schnell! Es bietet uns gleich eine boolsche Systemvariable an in der der Mauswert (true-false) gespeichert ist und ausgelesen werden kann.
So brauchen wir also nun eine Bedingung in der abgefragt wird ob die Maustaste gerade gedrückt worden ist. Wenn das zutrifft wird dann unsere „xposition" aufaddiert, wie eben im letzten Skript.

```
if( mousePressed == true )
{
   xPosition = xPosition + 0.1; // Erhöhe x um 0.1
}
```

Versuche es mal selbst die Zeilen in das vorhergehende Skript zu Implementieren. Wenn Du aber noch etwas Hilfe brauchst kannst Du auch gern hier mal schauen.

Code:

```
float xPosition = 0.0;

// Programm Initialisierung
void setup()
{
  size(200, 200);
}


// Programm Logik
void draw()
{

  background(204);          // Zeichne Hintergrund grau

  if( mousePressed == true )// Bedingung ob Maustaste
  {                         // gedrückt worden ist
     xPosition = xPosition + 0.1;// Erhöhe x um 0.1
  }

  if (xPosition > width)    // Bedingung wenn xPosition
  {                         // grösser ist als die Breite
    xPosition = 0;          // dann zurück an Anfang
  }

  // Zeichne Kreis
  ellipse(xPosition, height/2, 20, 20);
}
```

Na schon am Schwitzen? Nö, dann geht sicher noch mehr!
Doch für eine kurze Einleitung ist es erstmal genug. Jetzt heisst es Durchatmen und Wiederholen. Dazu sollten die vorhergehenden Kapitel einfach mit eigenen Ideen aufgemotzt und kombiniert werden. Anfänger können dafür ruhig eine regnerisches Wochenende investieren :-)