FunctionScope

Hardwarenahe Programmierung

Elektronik und Technische Informatik

Version: 1. März 2021

Wolfgang Uriel Kuran

Processing:

Zuerst sei auf die Seite: https://processing.org/reference/ hingewiesen ! Dort findet sich eine Übersicht über die Befehle von Processing mit genauer Beschreibung und Beispielen.

Processing ist eine C sehr ähnliche Sprache! Oft hat es mehr Ähnlichkeit mit Java, denn mit C. Das liegt auch an den vielen Objekten die bereits von Anfang an zur Verfügung stehen.

Der Einstieg sehr leicht. Aber das Skriptum nützt erst dann etwas, wenn du Processing auch installierst und es gleich beim Durcharbeiten dieser Seiten ausprobierst.

Ausdrücklich möchte ich darauf hinweisen, dass C und C++ in vielen Punkten etwas andere Syntax verlangt. Genau auf diese Unterschiede werde ich hier aufmerksam machen.
Meist bietet Processing einfachere Möglichkeiten an.

a) Wo befindet sich, welcher Teil eines Programms bei Processing:

Der Programmablauf hat viel Ähnlichkeit zu Arduino – Programmen. Das ist kein Zufall, denn Arduino wurde auf Basis von Processing entwickelt.
Es gibt auch hie eine Funktion **void setup()** - die ganz zu Beginn des Programmlaufes einmal durchlaufen wird, und eine Funktion **void draw()** die immer wieder durchlaufen wird. (die entsprocht somit der loop – Funktion von Arduino).

Beide Funktionen erlauben keine Parameterübergaben! Deshalb ist es unumgänglich, mit globalen Variablen zu arbeiten. Außerdem gibt es in Processing kein **#defines**.

Es ist wichtig immer die Übersicht zu behalten – wie sieht ein typischer Programmentwurf aus?

So zum Beispiel!

```
// Programmkopf (Name des Programms, Autor Versionsnummer …

// globale Variablen

void setup()
{
   // lokale Variablen der Funktion setup
}

void draw()
{  // lokale Variablen der Funktion draw
}
```

Dazu kommen noch einige sehr interessante Ereignisfunktionen:

```
void mousePressed()
{
}
```

Solche Ereignisfunktionen füge ich stets am Ende hinzu. Und natürlich: nicht innerhalb draw ….!

Programmablauf: zuerst wird setup einmal durchlaufen und dann immer wieder draw.
Ereignet sich ein Maus-Click so wird mousePressed aufgerufen und dann wieder draw.

Es gibt noch mehr solcher Ereignisfunktionen. Die Eingangs erwähnte Web-Site gibt Auskunft darüber.

Ein Besuch der Web-Site lohnt sich, denn dort finden sich auch viele andere nützliche hinweise.

Darüber hinaus auch jede Menge vordefinierter Varaiblen, wie zB mouseX für die x-Position der Mauskoordinaten.

Auch für die Tastatur gibt es viele interessante vordefinierte Befehle.
key zum Beispiel, für das zuletzt gedrückte ASCII-Zeichen der Tastatur, keyPressed usw. usw.

b) Konstante:

Es gibt eine Liste nützlicher vordefinierter Konstanten:

PI
TWO_PI
HALF_PI zum Beispiel

Aber auch width oder hight sind zwei Variable, die vordefiniert wurden.
Diese Variablen weisen von Anfang an den Wert 100 auf.

Wird aber die Funktion size(300,400); aufgerufen, so ändert sich der Wert der Konstanten width auf 300 und der Wert der Konstanten hight auf 400.

c) Variable:

Es gibt die Datentypen int, float, long, double, char alles gut…

Allerdings wird sich gleich herausstellen, dass diese Typen ein Eigenleben haben – nicht so wie bei C. Dazu ein Beispiel: Casten. Beim Casten von einer Type auf eine andere muss etwas anders vorgegangen werden als bisher:

Beispiel:

```
int a , b ;
float x;


x = 2.5;
b = 10;
```

schreibe ich nun: a = b / x; so gibt es schon einen Error!

Processing wandelt hier nicht selbstständig, automatisch zuerst alles auf float – wie das in C geschieht! Jede Anweisung zum Umwandeln muss explicit vom Programmierer angegeben werden, so kann sie auch nicht übersehen werden.

Hier muss der Programmierer also genau sagen, was er will:
In diesem Beispiel: zuerst will er die Variable b in ein float umwandeln.

In Cc ginge das so: (float)b - das wird als „casten“ bezeichnet.
In Processing geht das so: float(b)

Das würde jetzt so aussehen:

a = flaot(b) / x;

Gibt aber immer noch einen Error! - Weil: das Ergebnis rechts vom Ist-Gleich eine float Zahl ist und links eine Variable vom Typ int steht.

Abhilfe: wir wandeln alles nach int:

a = int(float(b) / x);

Das klappt.

Bei genauer Betrachtung erkennen wir hier: dass wir es viel mehr mit Objekten zu tun haben, als mit einfachen Variablen.

Noch offensichtlicher wird das, wenn wir zB ein Array anlegen:

Hier zB: ein int Array:

int[] ArrayName = new int[128];

Es kann später mit dem Array ganz normal gearbeitet werden:

Zb ArrayName[7] = 3;

Die Objekteigenschaften des Arrays bieten allerdings auch viele Vorteile:
So gibt es zB eine Methode, mit der die Länge des Arrays ermittelt werden kann.

A = ArryName.length;

Spätestens hier wird klar: Processing nutzt Objektorientierung.

TEXTE:

Noch deutlicher wird das bei den Strings:

```
String output
int a;

output = „Hi!“;

println(output);

a = output.length();

println(a);
```

Die Variable output ist ein Objekt! Operationen wie ist-gleich (=) sind überlagert!
Alles sieht so einfach aus. Das soll auch so einfach sein. Doch es steckt hier viel dahinter.

Übrigens:
Mit println können einzelne Ergebnisse in der Konsole unterhalb des Processing-Editors ausgegeben werden.

Wenn es also ohnehin schon von Objekten nur so wimmelt, dann ist es sicherlich eine gute Idee, die Taster unseres Funktionsgenerators bzw. unseres Oszis ebenfalls als Objekte anzulegen.

Das wird uns später viel Arbeit sparen und zur Übersichtlichkeit einen großen Beitrag leisten.

Doch zunächst brauchen wir hier natürlich eine Klassendefinition unserer Taster (buttons):

Klassendefinitionen, wie wir sie von C++ her kennen, sehen etwas anders aus als wir es hier in Processing machen werden. Vieles ist in Processing einfacher.

Bitte nicht mit C++ verwechseln!

Das nächste Beispiel soll das kurz demonstrieren:

Klassendefinition und Verwendung für zwei Taster:

Wir werden für unseren Funktionsgenerator einige Tasten brauchen um die Signalform auswählen zu können. So brauchen wir später einen Taster für Sin-Signale und einen für Dreieck-Signale. Hier ein einfaches kleines Besipiel, dass die Vorgangsweise verdeutlichen soll:

```
button sinus;     // globale Variable, die später auf ein Button-Objekt zeigen wird.
button triangle;

class button
{
  int xLeftTop;
  int yLeftTop;

  button(int x, int y)
  {
    xLeftTop = x;
    yLeftTop = y;
    fill (255, 255, 255);
    rect(x, y, 200, 30);
  }

  void moveOver(int x, int y)
  {
    if ((x > xLeftTop) && (x < (xLeftTop + 200)) &&
        (y > yLeftTop) && (y < (yLeftTop +  30))    )
    {
        fill (255, 0, 0);
    }
    else
    {
        fill (255, 255, 255);
    }
    rect(xLeftTop, yLeftTop, 200, 30);
  }
};

void setup()
{
  size(1000, 500);
  background(20,80,100);

  sinus = new button(50, 50);
  triangle = new button(50, 150);

}

void draw()
{
  sinus.moveOver(mouseX,mouseY);
  triangle.moveOver(mouseX,mouseY);
}
```

Wenn dieses kleine Beispielprogramm in ein Sketch mit dem Namen button abgespeichert wird

– kommt es zum Error: the nested type cannot hide an enclosing typ
Einfache Abhilfe: den Sketch anders benennen – zB buttontest

Dieses Beispiel zeigt bisher lediglich zwei Button an.

Bewegt sich die Maus über eine dieser Tasten, so ändert der Taster seine Farbe.

Klcikt man auf diese Schaltfläche passiert aber noch gar nichts.

Das bauen wir jetzt ein:

Ziel: wenn auf den Taster geklickt wird soll kurz der Text: click-Sinus bzw. click-Dreieck in der Console ausgegeben werden.

Vorbereitung:

Zunächst brauchen wir eine Ereignisfunktion für den Maus-Click. Die finden wir auf der Web-Site:

```
void mousePressed()
{
}
```

Wird also die Maus geklickt – so wird diese Funktion aufgerufen.

Jetzt kommt es natürlich darauf an, wo hin geklickt wurde. Und das erfahren wir über die mouseX und die mouseY Variablen.

Diese beiden Werte können nun über eine neue Methode an die Tasten weitergegeben werden. (diese Methode müssen wir natürlich erst schreiben)

Sinnvollerweise gibt man an alle Taster diese beiden Variablen weiter. Jede Taste soll für sich nachsehen, ob sie gemeint ist.

Könnte so aussehen:

```
void mousePressed()
{
  if (sinus.click(mouseX, mouseY))
  {
    println("click! - Sinus");
  }
  if (triangle.click(mouseX, mouseY))
  {
    println("click! - Dreieck");
  }
}
```

Wir brauchen also zunächst Methoden, die click heißen soll, zwei Parameter bekommt (x und y)

und die ein true oder false zurückliefert.

Die Methode könnte so aussehen:

```
boolean click(int x, int y)
{
  boolean ret = false;
  if ((x > xLeftTop) && (x < (xLeftTop + 200)) &&
      (y > yLeftTop) && (y < (yLeftTop + 30))        )
  {
      ret = true;
  }
  return ret;
}
```

Wo muss diese Methode abgespeichert werden? In der Klassendefinition des Tasteres – button.

Gleich nach moveOver

Hinweis: in dem Entwurf hier wurde durchgängig die Größe der Taste mit 200 breit und 30 hoch festgelegt. Leider gibt es keine Defines… so tauchen diese Zahlen immer wieder im Programm auf und es bleibt leider nichts anderes über, als einzeln diese Zahlen zu verändern.

Auch hier in der Methode click wird das verwendet. Die Taste muss ja auch „wissen“ wie breit und wie hoch ihre Fläche ist.

Nächster Arbeitsschritt:

Es wäre schön, wenn die Taste eine Beschriftung bekäme:

Auf der sinus-Taste soll sinus stehen und auf der Dreieck-Taste soll triangle stehen.

Wo kann dieser Text an das Objekt übergeben werden?

Richtig, bei der Erstellung – also im construktor:

Statt:
sinus = new button(50, 50);

solll

sinus = new button(50, 50, “sinus“); aufgerufen werden, und dieser Text soll als String im Objekt

gespeichert werden. Daher werden wir die Klasse erweitern.

Es kommt die Variable: String label dazu.

(warum nicht String text ? - ganz einfach: text ist eines der Befehle, die es in Processing gibt. Wieder zu finden auf der eingangs erwähnten Web-Site)

Der Construktor muss daher auch erweitert werden:

```
button(int x, int y, String n)
{
  xLeftTop = x;
  yLeftTop = y;
  label = n;
  fill (255, 255, 255);
  rect(x, y, 200, 30);
}
```

Natürlich muss es diese label – Variable auch geben! (zu Beginn der Klassenbeschreibung)

Allerdings müssen wir nun an jeder Stelle, die die Taste zeichnet diesen Text ausgeben.

Zuerst im Construktor selbst:

```
button(int x, int y, String n)
{
  xLeftTop = x;
  yLeftTop = y;
  label = n;
  fill (255, 255, 255);
  rect(x, y, 200, 30);
  text(label, x + 10, y + 25);
}
```

Ausprobiert – nichts zu sehen – warum?
Die Farbe der Schrift ist durch das fill – Kommando vor dem rect Befehl auf weiß umgeschalten. Darum wird mit weißer Schrift auf weiße Fläche geschrieben.

Abhilfe: die Farbe auf schwarz umschalten:

```
button(int x, int y, String n)
{
  xLeftTop = x;
  yLeftTop = y;
  label = n;
  fill(255, 255, 255);
  rect(x, y, 200, 30);
  fill(0);
  text(label, x + 10, y + 25);
}
```

Übrigens kann man fill auch mit nur einem Parameter verwenden. Fill ist Überlagert !

Ausprobiert:

Man sieht noch immer nichts von den Texten.

Warum?

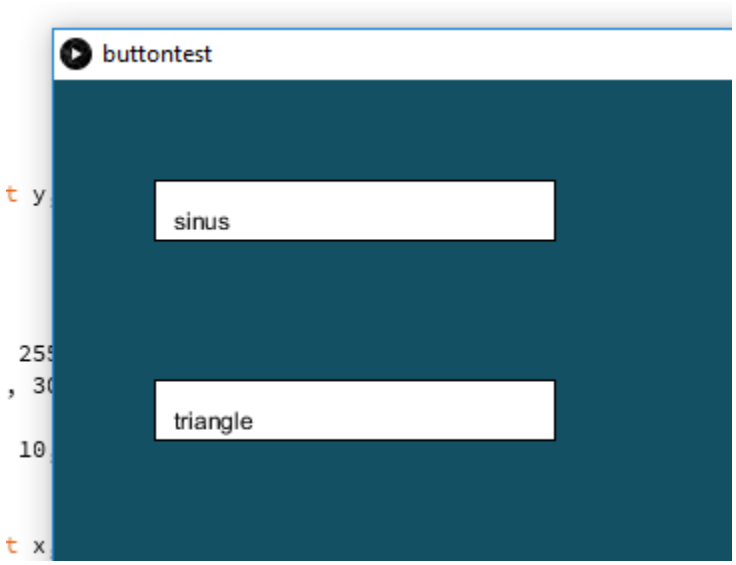
Richtig: die Methode moveOver überpinselt unsere Taster mit jedem Durchlauf von draw!

Daher muss auch dort zunächst der Text mit eingebaut werden:

```
void moveOver(int x, int y)
{
  if ((x > xLeftTop) && (x < (xLeftTop  + 200)) &&
      (y > yLeftTop) && (y < (yLeftTop + 30)))
  {
      fill (255, 0, 0);
  }
  else
  {
      fill (255, 255, 255);
  }
  rect(xLeftTop, yLeftTop, 200, 30);
  fill(0);
  text(label, x + 10, y + 25);

}
```

Funktioniert noch immer nicht - findest Du den Fehler?



Teil 2:

Lösung des Rätsels:

In der Methode :

```
void moveOver(int x, int y)
{
  if ((x > xLeftTop) && (x < (xLeftTop  + 200)) &&
      (y > yLeftTop) && (y < (yLeftTop + 30)))
  {
      fill (255, 0, 0);
  }
  else
  {
      fill (255, 255, 255);
  }
  rect(xLeftTop, yLeftTop, 200, 30);
  fill(0);
  text(label, x + 10, y + 25);
}
```

wird der label-Text zuletzt einfach an die Stelle x + 10 und y + 25 geschrieben. Das ist aber die Maus-Position.

Richtig wäre es hier von xLeftTop und yLeftTop auszugehen:
Also so:

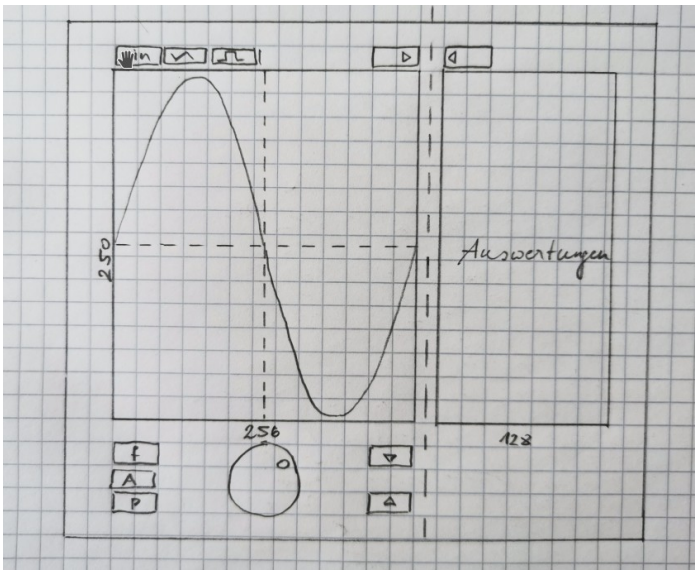
```
void moveOver(int x, int y)
{
  if ((x > xLeftTop) && (x < (xLeftTop  + 200)) &&
      (y > yLeftTop) && (y < (yLeftTop + 30)))
  {
      fill (255, 0, 0);
  }
  else
  {
      fill (255, 255, 255);
  }
  rect(xLeftTop, yLeftTop, 200, 30);
  fill(0);
  text(label, xLeftTop + 10, yLeftTop + 25);
}
```

Alles klar?

Scope-Signal-Ausgabe:

Nun wird es Zeit das erste Signal auszugeben. Was wir also brauchen, ist großes schwarzes Rechteck, dass uns als Scope dienen wird.

Mit einer einfachen Skizze könnte unsere Oberfläche entworfen werden. Zb so:

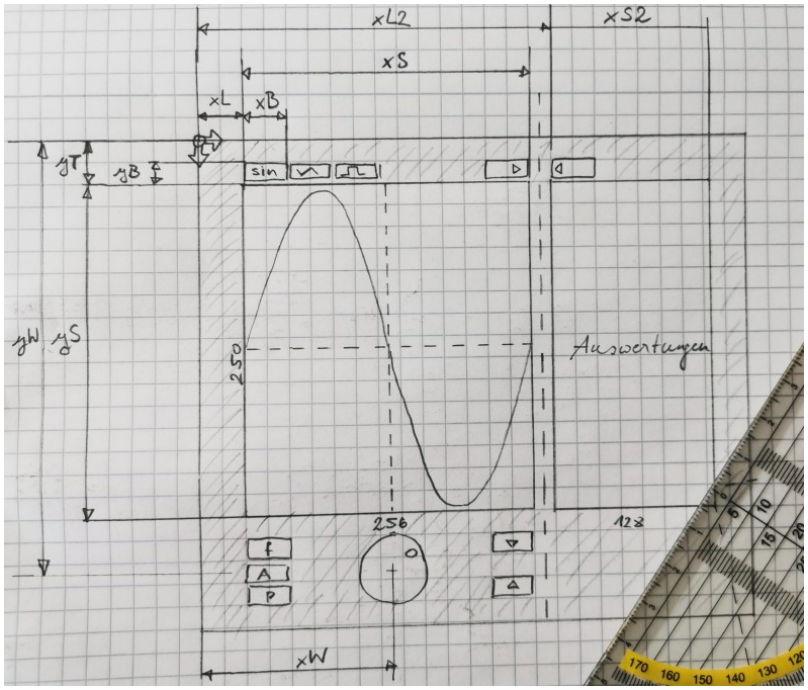


Dabei ist der rechte Bereich (Auswertung) für später vorgesehen. Sehen wir uns zunächst den linken Teil an: Oben sind einige Taster vorgesehen (Sinus, Dreieck, Rechteck)
In der Mitte links soll das Signal ausgegeben werden. Es soll eine rechteckige schwarze Fläche als Hintergrund gezeichnet werden. Das Signal soll als gelbe Linie sichtbar sein.
Und unterhalb des „Scopes“ soll ein Stellrad (wheel) dazu verwendet werden können, die Amplitude, die Frequenz oder die Phase zu verändern) Was von diesen drei, soll durch die linken drei Tasten ausgesucht werden können. Mit den Tasten oben – Mitte soll der rechte Teil ein bzw ausgeschalten werden können.

In einem C – Programm würde man mit Defines die einzelnen Abstände definieren.
Hier, in Processing, gibt es keine Defines. Aber es gibt immerhin die Möglichkeit, gleich zu Beginn des Programms einzelne Variable mit Werte zu belegen. Diese Werte können im weiteren Programm als Konstante Zahlen gesehen werden.

Dieser Weg soll hier zunächst beschritten werden.
Deshalb werden zunächst alle wichtigen Maße aus der Skizze mit Namen festgelegt und am Programm Anfang werden diese Werte angelegt.

Hier also die gleiche Skizze noch einmal – mit Bemaßung:



Diese Längen können nun als Werte am Anfang des Programms festgelegt werden.
Die Fläche für die Scope-Asugabe kann auch in setup bereits ein erstes Mal gezeichnet werden.
Weiters sei gleich auch ein dritter Button angelegt – den wir für Rechtecksignale brauchen.

Ausgemessen wird alles von der linken oberen Ecke aus. So könnte der Wert für xL etwa 20 Pixel groß sein. Am besten misst Du alles mit einem Lineal aus.

```
// zB könnten diese Werte verwendet werden

int xL = 20;
int yT = 20;
int xB = 45;
int yB = 12;
int xS = 512;
int yS = 500;
. . .
```

Damit Du dieses Skriptum von hier ab gut verwenden kannst um alle weiteren Arbeitsschritte selbst auszuführen, sei an dieser Stelle, alles, dass bisher programmiert wurde noch eimal abgedruckt:

```
//*************************************************************
// PROCESSING - FunctionScope
// Kuran 2021
//*************************************************************

// const:
int xL = 20;
int yT = 20;
int xB = 45;
int yB = 12;
int xS = 512;
int yS = 500;

button sinus;
button triangle;
button rectangle;

class button
{
  int xLeftTop;
  int yLeftTop;
  String label;

  button(int x, int y, String n)
  {
    xLeftTop = x;
    yLeftTop = y;
    label = n;
    fill(255, 255, 255);
    rect(x, y, xB, yB);
    fill(0);
    text(label, x + 5, y + 10);
  }

  void moveOver(int x, int y)
  {
    if ((x > xLeftTop) && (x < (xLeftTop  + xB)) && (y > yLeftTop) && (y < (yLeftTop + yB)))
    {
        fill (255, 0, 0);
    }
    else
    {
        fill (255, 255, 255);
    }
    rect(xLeftTop, yLeftTop, xB, yB);
    fill(0);
    text(label, xLeftTop + 5, yLeftTop + 10);

  }

  boolean click(int x, int y)
  {
    boolean ret = false;
    if ((x > xLeftTop) && (x < (xLeftTop  + xB)) && (y > yLeftTop) && (y < (yLeftTop + yB)))
    {
        ret = true;
    }
    return ret;
  }

};

void setup()
{
  size(1000, 700);
  background(20,80,100);

  fill(0);
  rect(xL, yT, xS, yS);

  sinus = new button(xL, yT - yB, "sinus");
  triangle = new button(xL + xB, yT - yB, "triangle");
  rectangle = new button(xL + xB + xB, yT - yB, "rect");
}

void draw()
{
   sinus.moveOver(mouseX,mouseY);
   triangle.moveOver(mouseX,mouseY);
   rectangle.moveOver(mouseX,mouseY);
}

void mousePressed()
{
  if (sinus.click(mouseX, mouseY))
  {
    println("click! - Sinus");
  }
  if (triangle.click(mouseX, mouseY))
  {
    println("click! - Dreieck");
  }
  if (rectangle.click(mouseX, mouseY))
  {
    println("click! - Dreieck");
  }
}
```

Noch geht alles auf eine einzige Seite …

Auch das Scope ist bereits als schwarze Fläche vorbereitet.

Der nächste Schritt ist nun, dass wir ein Sinus-Signal in einem Array vorbereiten um dieses anzuzeigen.

Dazu brauchen wir ein Array:

```
int[] signal = new int[128];
```

Wohin damit: klar! - zu den globale Variablen.

Jetzt wäre es natürlich das einfachste, einfach Werte in das Array einzutragen und das dann im Scope anzuzeigen.

Doch denken wir an dieser Stelle erst ein bisschen weiter:

Wir wollen später die Frequenz mit dem Stellrad verändern können.
Wir wollen die Amplitude und auch die Phase verändern können.

Jedesmal müssen wir dann die Kurve löschen und wieder neu zeichnen.

Daher ist es klug, von Anfang an eine Funktion für dieses Vorbereiten des Signals und eine weitere für das Zeichnen des Signals vor zu bereiten.

Wir schreiben also eine Funktion:

```
void prepareNewSinusSignal(float amplitude, float frequenz, float phase)
{
}
```

und eine weitere:

```
void drawNewSignal()
{
}
```

Warum haben wir in der Funktion drawNewSignal nichts von „sinus“ im Namen aufgenommen: ganz einfach: es wird einfach nur das Array dargestellt.
Später, wenn wir zB ein Rechtecksignal anzeigen wollen, verwenden wir die gleiche Funktion.

Wir beginnen zuerst genau mit dieser Ausgabe.
Wo kommt die Funktion vorerst hin?
Ganz ans Ende.

Wo soll sie aufgerufen werden?
Vorerst am Ende der Draw-Funktion.

Sieh dir diese Zeilen an:

```
void drawNewSignal()
{
    int i;

    fill(0);
    rect(xL, yT, xS, yS);

    stroke (255, 255, 0);         // gelbe Linien
    for (i = 1; i < 128; i++)
    {
        line(xL + (i-1) * 4, yT + yS - (signal[i-1]) * 2,
             xL +  i    * 4, yT + yS - (signal[i]    * 2));
    }
}
```

Kann das funktionieren?

Zuerst wird das Scope „gelöscht“ – stimmt nicht : eigentlich wird es komplett schwarz übermalt.
Dann wird die Linienfarbe (stroke) auf gelb umgeschalten.

Und schließlich werden lauter kleine Linienelemente gezeichnet, die die einzelnen ArrayPunkte miteinander verbinden.

Ok, dann brauchen wir eine Funktion für das SinusSignal:

Zuerst legen wir die globalen float Variablen Amplitude, Frequenz und Phase an.

float amplitude = 100.0 , frequency = 1000, phase = 0; Wir speichern hier gleich Werte.

Jetzt entwerfen wir die Funktion, die ein Sinus-Signal in das Array speichert:, zB so:

```
void prepareNewSinusSignal(float a, float f, float p)
{
    int i;

    for (i = 0; i < 128; i++)
    {

        signal[i] = 128 + (int)(a * sin((i * f/20371) + p));  // 20371 = 2pi / 128

    }
    newSignal = true;
}
```

Interessant hier: am Ende setzen wir die Variable newSignal auf true !

Diese Variable (newSignal) ist vom Typ boolean, auch global angelegt – Anfangs auf false gesetzt.

Immer wenn sich nun etwas am Signal ändert, wird dieses Bit gesetzt.

In der Funktion draw fragen wir dieses Bit nun ab und schon wird das Signal sichtbar:

```
  if (newSignal)
   {
       drawNewSignal();

       newSignal = false;
   }
```

Und wo bereiten wir das Signal vor ? - ganz einfach: immer wenn die Taste „sinus“ gedrückt wird.

```
void mousePressed()
{
int i;

  if (sinus.click(mouseX, mouseY))
  {
       prepareNewSinusSignal(amplitude, frequency, phase);
  }
}
```

Genauso kann natürlich mit den anderen Signalen vorgegangen werden…

Jetzt zum Drehknopf:

Zuerst brauchen wir einen Winkel – ebenfalls einfach als globale Variable.

Dann müssen wir den Drehknopf zeichnen:

zB so:

```
void drawWheel()
{
  int xp, yp;

  fill(255);
  ellipse(xW, yW, 100, 100);
  fill(0);
  xp = xW + int(40 * sin(angle));
  yp = yW - int(40 * cos(angle));
  ellipse(xp, yp, 10, 10);
}
```

später mehr …