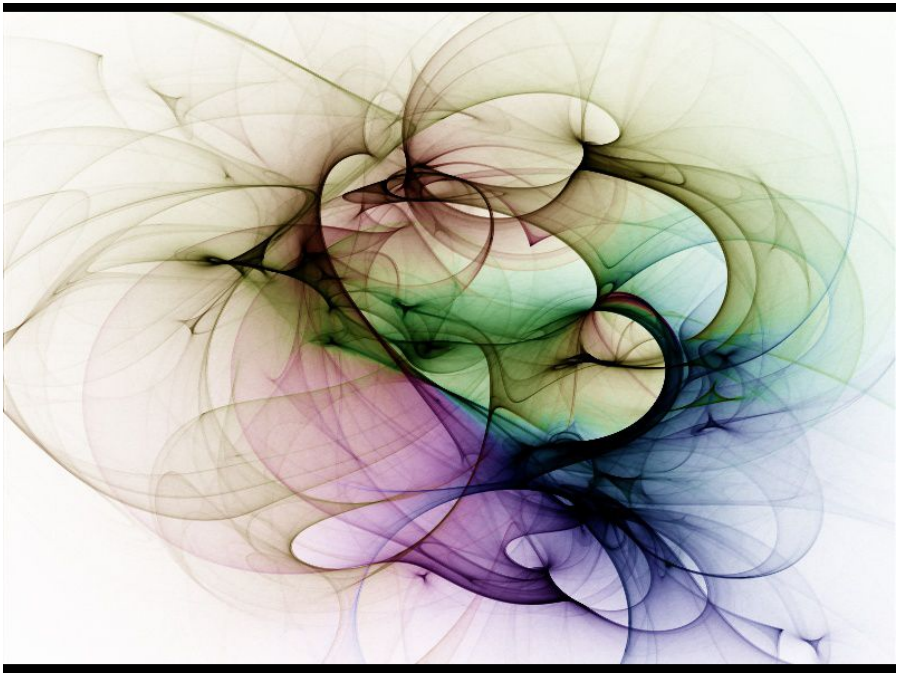
Programmieren oder Zeichnen?



Mit der auf Java basierenden Skriptsprache: Processing.

Inhaltsverzeichnis

1. Processing Fundamentals
   - 1.1 Was ist Processing
   - 1.2 Was kann Processing tun
   - 1.3 Das Processing Format
   - 1.4 Processing IDE

2. Kleine Einleitung für Nicht-Programmierer
   - 2.1 Kommentare
   - 2.2 Sag „Hallo"
   - 2.3 Ausdrücke
   - 2.4 Bedingungen
   - 2.5 Schleifen
   - 2.6 Funktionen
   - 2.7 Das Display Fenster
   - 2.8 Processing Programmfluss

3. Variablen
   - 3.1 Variablen Deklarieren
   - 3.2 Primitive Variablen
   - 3.3 Konvertieren von Variablen
   - 3.4 Gemischte Variablen

4. Kontrollstrukturen
   - 4.1 if-Bedingungen
   - 4.2 switch / case
   - 4.3 Operatoren

5. Funktionien
   - 5.1 Funktionen
   - 5.2 Funktionen definieren
   - 5.3 Parameter übergeben
   - 5.4 Terminierung und Rückgabe
   - 5.2 Rekursive Funktionen

6. Objekt - Orientiertes - Programmieren
7.1 Was ist OOP
7.2 Klassen und Objekte
7.3 Verdeckung
7.4 Konstanten
7.5 Zerstörung von Objekten
7.6 Mehrere Konstruktoren
7.7 Vererbung
7.8 Super
7.9 OOP Dokumentation

7. xxx
8.1 xxx
8.2 xxx
8. xxx
9.1 xxx
9.2 xxx
9. xxx
10.1 xxx
10.2 xxx
10. xxx
11.1 xxx
12.2 xxx
11. xxx
12.1 xxx
12.2 xxx
12. xxx
13.1 xxx
13.2 xxx

1 Processing Fundamentals

° Was ist Processing
° Was kann Processing tun
° Das Processing Format
° Processing IDE

1.1 Was ist Processing

Processing ist eine Programmiersprache basierend auf Java (Sun). Ausgeliefert wird Processing in einer einfach zu verstehenden Entwicklungsumgebung (IDE).

Entwicket wurde Processing am MIT in Boston, Interaction Design Institute Ivrea, von Ben Fry und Casey Reas.

Besonders visuell denkende Menschen werden daran eine Freude haben, denn durch das einfach zu verstehende Konzept erleichtert es den Einstieg in die interaktive Welt. Somit eignet sie sich sehr gut für Studierende, Interaction-Screendesigner, und Menschen, die einen leichten Einstieg in die höhere Programmiersprache Java suchen.

Processing wird in der Punkt.Syntax (dot syntax) geschrieben. Dadurch liegt es dem gleichen Konzept wie Javascript, Php oder Flash Actionscript zu Grunde.

Jedoch bietet Processing dem ambitionierten Programmierer die Möglichkeit - OOP Konzepten nachzugehen, in schon bekannter Java Syntax. Der Übergang der Scriptsprache zu Java ist fliessend. Die Lernkurve steigt schnell, denn erste sichtbare Erfolge stellen sich nach wenigen Minuten ein.

Dazu ist Processing Freeware!
Da braucht man gar nicht lange überlegen und es für sein jeweiliges OS bei http://processing.org runterladen (Windows, MacOSX, Linux.)

Da stellt sich doch die Frage:

1.2 Was kann Processing tun?

Processing wird benutzt um alle Arten von interaktiven Inhalten darzustellen, typische Web basierende Anwendungen.
Hier ein paar Möglichkeiten: digitale Kunst, MP3-Player, ein Datenbank basiertes Interface, Multiuser Zeichenbretter, Video-Chats, 2D Spiele, 3D-Echtzeit Simulationen oder auch Agentensyteme etc.

Jede dieser Applikationen benutzt eine Kombination der Möglichkeiten die Processing bietet. Also fangen wir an darüber nachzudenken, wie wir die Dinge kombiniert bekommen, um unsere Applikationen damit zu bauen.

° Interaktion
Processing kann auf Benutzereingaben reagieren, sie Auswerten, speichern oder gegebenfalls weiterversenden. Wie z.B. das Drücken der Maustaste.

° Bild und Ton Kontrolle
Processing kann verschiedene Parameter von Bild und Ton verändern. Wie z.B. verschiedenen Pixeln eines Bildes andere Farbwerte zuweisen.

° Zeichenwerkzeuge
Processing kann unterschiedlichste Formen und Linien darstellen (Ellipse, Linie, Punkt, Quardat, Rechteck, Trinangle, Bezierkurve).

° 3D Werkzeuge

Processing bietet die Möglichkeit 3D Primitive zur Laufzeit zu erstellen. Durch die Erweiterung der Java 3D API Bibliothek ist man in der Lage 3D Modelle von Programmen wie 3DMax, Alias Wavefront, Wings3D(frei) im .obj Format zu improtieren.

° Server Kommunikation

Mit Processing kann man Verbindungen zu Servern erstellen und so Informationen austauschen. Es gibt eine Reihe von Werkzeugen um Informationen zu senden oder auch zu empfangen.

Somit kann jede Serverseitige Sprache von Processing aus aufgerufen werden, z.B. PHP, ASP, JAVA, etc.

Diese Macht bekommt Processing mit der Möglichkeit der Erweiterbarkeit. Es lassen sich andere Java Bibliothekenimportieren, anpassen und benutzen. Schöne Beispiele und Resourcen können von der offizellen Website runtergeladen werden. Im weiteren Verlauf werde ich auf einige, mit Beispielen eingehen.

Es gibt natürlich auch noch viel mehr dass Processing bietet, um es heraus-zufinden, schmöker doch ein wenig in der Dokumentation. Im Processing Forum findest auch viele Beispiele und Ideen anderer User. Wenn du nach ausgiebiger Suche nichts zu deinem Problem gefunden hast dann poste doch deine Frage. Es wird dir bestimmt geholfen!

1.3 Das Processing Format

Processing wird als standart Java .jar kompiliert und mit Sun's Java-Virtual-Machine ausgeführt. Zu finden ist der Kompiler im Processing Hauptverzeichnis namens „jikes.exe". Das Programm kann auch aus der Konsole gestartet werden, um z.B. einen anderen Editor, wie Eclipse zu benutzen.Die entwickelten Programme laufen auf unterschiedlichen Systemen, wie Windows OS, Mac OSX und Linux.

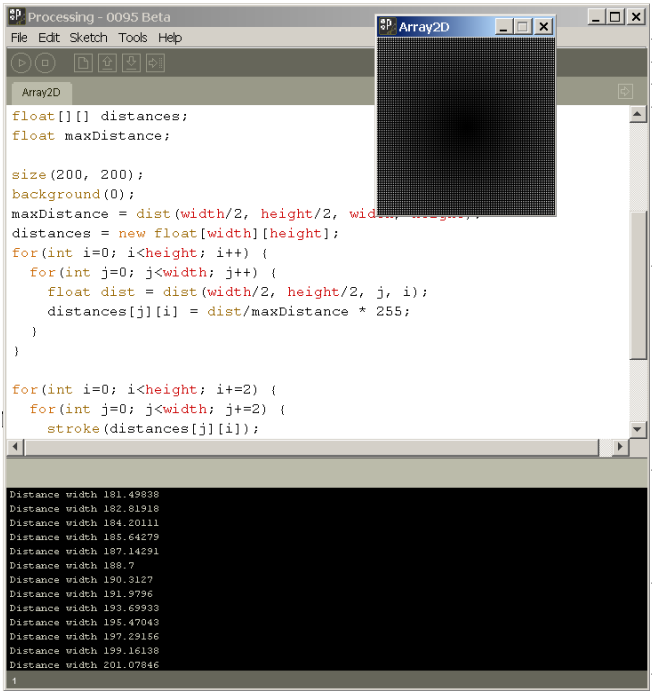
Sämtliche mit Processing hergestellte Software, kann aufgrund von ActiveX im Webbrowser dargestellt werden. Es lassen sich aber auch „standalone" Programme damit entwickeln.

Mittlerweile lässt sich Processing auf mobilen Endgeräten,
wie z.B. Handy, PDA und Smartphone anzeigen.
Besuch doch dafür die Website http://mobile.processing.org

Es lässt sich auch externe Hardware über die „serielle" Schnittstelle oder per USB ansteuern.
Besuch doch dafür die Website http://hardware.processing.org

1.4 IDE

http://www.processing.org/reference/environment/index.html



<- Hauptenü (Dokument)
<- Submenü (Dokument)
<-Shortcuts
<- Dokumentname

<-Skripteditor

<- Processing Feedback Messages

<- Output

Aktuelle <- Zeilennumer

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.

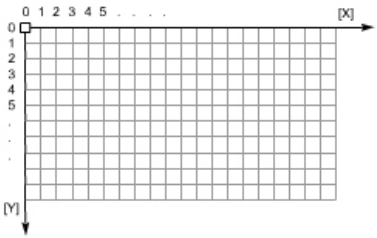
Code:

```
/*
  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 zugeweisen

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 :-)

3 Variablen

Container zum Füllen von Informationen

- Variablen Deklarieren
- Primitive Variablen
- Konvertieren von Variablen
- Gemischte Variablen

3.1 Variablen deklarieren

Processing bietet uns verschieden Variablentypen an, die unterschiedliche Inhalte aufnehmen können. Wie man eine Variable definiert (deklariert) ist ganz einfach. Man überlegt sich den Typ und einen Namen dafür.
Für eine Ganzzahl z.B. `int ganzZahl;`
Jetzt existiert die Variable, wie eine leere Seite eines Buches. D.h. Sie hat noch keinen Wert zugewiesen bekommen. Den Wert einer Variable kann man auch aus anderen Programmabschnitten ändern, sofern sie ihr Gültigkeitsbereich nicht verletzt wird. Denn es gibt lokale und globale Variablen. Den Unterschied werden wir etwas später kennenlernen. Wenn die Variabel einmal deklariert ist muss man ihren Typen (z.B int) bei anderer Wertzuweisung nicht nochmal schreiben, denn dann denkt Processing , dass es hier um eine neue Dekleration handelt. Das sollte dann so aussehen.

```
int ganzZahl;          // Dekleration
....
ganzZahl = 20;         // Zuweisen eines Wertes
....
int ganzZahl = 20;     // ober beides zusammen
```

Um dies in einem kleinen Skript zu verdeutlichen möchte ich gern zwei Ganzzahlen-Variablen definieren, addieren und hinterher ihren Wert in eine Ergebniss-Variable schreiben und auslesen.

Code:

```
// Definieren der Variablen
int ergebnis;              // Ergebniss der Addition
int zahl_1 = 100;          // var 1
int zahl_2 = 100;          // var 2

// Var mit neuen Wert überschreiben
zahl_1 = 2 ;

// Jetzt zusammen addieren und in Ergebnis schreiben
ergebnis = zahl_1 + zahl_2;

// Ergebnis Wert auslesen und im Output darstellen
println("Das Ergebnis: "+ergebnis );
```

Noch etwas zu Namensgebung von Variablen

Bevor du gleich verschiedenste Variablen definierst – Vorsicht!
Variablen Namen müssen:

° Ausschliesslich aus Buchstaben, Nummer oder Unterstrichen generiert werden. (Keine Leerzeichen, Sonderzeichen oder Punkte)

° Mit einem Buchstaben oder Unterstrich anfangen.

° Dürfen nicht länger als 255 Zeichen lang sein. (Okay, wer das diese Länge ausreizt hat irgendwas nicht richtig verstanden:)

° Als case-sensitive betrachtet werden. D.h. für den Kompiler sind die Variablennamen „ganzZahl" oder „ganzzahl" zwei verschieden paar Schuhe!

Legale Variablennamen können so aussehen:

```
int vor_name;
int zaehler;
int sehrLangerVariablenName;
```

Diese Variablennamen würden dich in Probleme rennen lassen:

```
int 1vor_name;                  // Startet mit einer Nummer
int variable mit leerzeichen;  // Enhält Leerzeichen
int auch-illegal;               // Enthält Sonderzeichen
```

Es ist eine gute Übung Variablen gleich am Anfang des Skriptes zu schreiben und zu Kommentieren. So sieht ein Skript dann gleich gut organisiert und schnell verständlich aus:

```
//----------------------------------------
// Initialisierung aller globalen Variablen
//----------------------------------------
float  ballGeschwindigkeit;
int    punktestand;
String spielerName
```

Ok. lassen wir das Gelesene nochmals vor unserem geistigen Auge ablaufen, so fällt uns gleich die generische Struktur auf.

VariablenTyp VariablenName = Wert;
Dies gilt für alle primitiven Variablentypen.

Vorher habe ich zwischen globalen und lokalen Variablen unterschieden. Globale Variablen sind aus jeder stelle des Codes lesbar und setzbar. Sie sollten daher im Kopf des Codes stehen, damit sie sofort gesammelt und übersichtlich zu sehen sind.

Lokale Variablen haben nur eine bestimmte Lebensdauer und werden nach ihrem Benutzen vom Speicher entfernt. Sie stehen auch nur einem begrenzten Programm Abschnitt zur Verfügung. Ein Beispiel wäre hier die for-Schleife:

```
for(int i = 0; i < 10; i++)
{
    ... Anweisung
}
```

Nachdem die Schleife abgearbeitet ist verfällt „i". Es kann dann nicht mehr darauf zugegriffen werden.

Ein anderes Beispiel für lokale Variablen kann man auch anhand von Funktionen sehen. In dem nächsten Beispiel werde ich zwei Variablen gleichen Namens definieren und durch eine Funktion auslesen. Man merkt schnell welchen Gültigkeitsbereich sie haben.
Spiel doch einfach ein bisschen mit dem Code damit es später richtig flutscht :-)

Code:

```
// Definiere global Variable
int zahl = 99;

// Definiere lokale Variable gleichen Namens
// in einer Funktion
void meineFunktion()
{
   int zahl = 11;

   // Zeigt auf die lokale Variable
   println("lokal: "+zahl);
}

void setup()
{
  // Zeigt auf die globale Variable
  println("global: "+zahl);

  // Aufruf der Funtion "meineFunktion"
  meineFunktion();
}
```

3.2 Primitive Variablen

Ich möchte nun hier die verschiedenen primitiven Variablentypen aufführen und ihnen gleich einen Wert zuweisen. Da Processing typisiert ist, dürfen nur die richtigen Werte in eine bestimmte Variabel und nicht einfach so gemischt werden, sonst gibt uns der Kompiler eine Fehlermeldung aus.

° Ganzzahlen `(int)`:
Sie haben einen Gültigkeitsbereich von
2,147,483,647 bis -2,147,483,648.

```
int ganzZahl = 10;
```

° Kommazahlen `(float)`:
Der Datentyp float hat einen Dezimalpunkt - also eine grössere Auflösung als die Ganzzahl. Anderst als bei z.B Java verbrauchen beide Typen 32 bit. Dies liegt daran dass Processing keine andern Typen wie z.B. long, short benutzt, der Einfachheitshalber. Sie haben einen Gültigkeitsbereich von 3.40282347E+38 bis -3.40282347E+38.

```
float dezimalZahl = 10.4;
```

° Boolean (boolean):

Boolsche Werte haben nur zwei Zustände. Nämlich wahr oder falsch. Sie werden oft für Bedingungen oder Programmflüsse gebraucht, um den Code übersichtlicher zu machen.

```
boolean bool = true; // beide möglichen Werte true/false
```

° Characters (char):

Dieser primitive Datentyp steht für typografische Symbole. Damit sind alle Symbole oder Zeichen die der Computer bereithält gemeint. Z.B. das 'm' oder '@'. Jeder char ist 2 Byte lang (16bit) und enthält jeweils ein Zeichen des Unicode-Satzes. Er kann durch ein Zeichen in einzelnen Anführungszeichen beschrieben werden.

```
char myChar = 'a';
```

oder auch durch den alphanummerischen Wert.

```
char myChar = 97; // steht für 'a'
```

° Byte (byte):

In Bytes können alphanummerische Werte von 127 bis -128 gespeichert werden. Sie verbrauchen nur 8bit (1 Byte) an Steicherplatz. Sie können auch zur Repräsentation von chars verwendet werden.

```
byte myByte = 97; // kann für 'a' stehen
```

° Color (color):

In dem Datentyp color werden Farbwerte abgelegt und mit 32bit representiert. Dazu gibt es mehrere Möglichkeiten. Ein Farbwert setzt sich aus R,G,B zusammen (rot, grün, blau). Mit diesem Wert können alle Farben dargestellt werden, die der Bildschirm anzeigen kann. Die Mischung machts! Z.B `color(0,0,0)` steht für schwarz und `color(255,255,255)` steht für weiss. Daraus ergibt sich ein Wertebereich von 0-255. Ich möchte es gern an einem kleinen Beispiel aufzeigen, welches den Hintergrund eines Applets verfärbt.

Code:

```
color bgColor = color(255,0,0); // ergibt rot
background(bgColor);            // setzte BG Farbe
```

Es gibt auch die Möglichkeit den Farbwert in #HEX anzugeben.
Manche kennen das auch aus HTML.

Code:

```
color bgColor = #00FF00; // ergibt grün
background(bgColor);     // setzte BG Farbe
```

Auch den umgekehrten Weg gibt es! Man kann die Farbe eines Pixels auslesen! `get(int xPosition,int yPostition)` Stell dir vor, du möchtest einen Farbwert eines Bildes an einer bestimmten Stelle auslesen, um mit diesem Wert z.B weiter zeichnen zu lassen. Dazu malen wir zwei Kreise an verschiedenen Stellen und färben den Zweiten mit der Farbe des Ersten.

Code:

```
// Zeichne ersten Kreis
color kreisFarbe = #00FF00; // ergibt grün
fill(kreisFarbe);
ellipse(20,width/2,30,30);

// Zeichne zweiten Kreis mit der Farbe von einem
// bestimmten Pixel
color kreisFarbe_2 = get(20,40); // Pixel x:20 y:40
fill(kreisFarbe_2);
ellipse(80,width/2,30,30);
```

3.3 Konvertieren von primitiven Variablen

Es gibt Situationen im Leben eines Programmierers, da möchte man gern z.B. zwei verschieden Variablen miteinander kombinieren und sie dann als ein Ergebnis in einer Dritten speichern. Was passiert aber wenn ich integer (int) mit einem float addieren möchte, um es dann als float zu speichern? Probieren wir es doch aus!

Code:

```
int   a = 10;
float b = 9.9;
float   c;
c = a + b;
println(c); // Ergebniss passt 19.9!
```

Soweit so gut.
Jetzt versuchen wir mal das Gleiche, nur mit dem Unterschied dass die Ergebnisvariable c ein integer.

Code:

```
int   a = 10;
float b = 9.9;
int   c;
c = a + b;
println(c); // FEHLER!!!
```

Dieser Fehler entsteht dadurch, dass das Ergebnis wenn ich a und b addiere ein float ist. Wir wissen dass ein float hochauflösender, da er eine Dezimalstelle besitzt. Da ist es doch ganz klar dass das Resultat nicht in den int c reinpasst.

Auch für solch' spannende Konvertierungen gibt es eine Möglichkeit – namens **Casting**. `(typ)wert` Es ist nichts anderes als ob ich es vom Programm erzwinge ein float in ein int zu stecken. Denn immerhin habe ich die Kontrolle über den Computer und mein Programm! Jedoch fällt dabei leider die Dezimalstelle heraus. Sie hat einfach keinen Platz und wird übersehen.

Code:

```
int   c;
c = 10 + (int)9.9; // ich caste auf (int)9.9 -> 9
println(c);
```

Solche Zahlen sollte man vorher noch zu runden, um bessere Ergebnisse zu bekommen. Aber das besprechen wir im Math Kapitel. Versuche doch zuvor andere „cast" Möglichkeiten von Primitiven zu Programmieren.
Viel Spass!

3.3 Gemischte Variablen

Der Unterschied zu den Primitiven Variablen ist, dass man eine von Processing vorgegebene Klasse instanziert und deren Methoden erbt. Um jetzt nicht so weit auszuschweifen, will ich damit sagen, es verlangt eine andere Herangehensweise und Umgang mit ihnen .
Man beachte: Da der Typ String eine Instanz der Klasse String ist, wird er zu Anfang gross geschrieben - im Gegensatz zu den primitiven Variablen.

Wir wollen ein Zeichenkette wie „Hallo Welt" abbilden. Diese Zeichenkette besteht eigentlich aus zehn verschiedenen chars (die einzelnen Buchstaben). Es wäre doch mühsam alle zehn verschiedenen chars händisch zu einer Zeichenkette zu addieren! Dafür gibt es doch die String Klasse- sie macht alles viel einfacher.

° Strings (Zeichenketten):

Um einen String darzustellen gibt es mehrere Möglichkeiten.

```
String zeichenKette_1 = "Hallo Welt";

String zeichenKette_2 = new String("Hallo Welt 2");
```

oder auf die mühsame Art über chars.

```
char[] zeichenKette_3 = {'h','a','l','l','o'};
String zusammenGesetzt= new String(zeichenKette_3);
```

Man kann auch Zeichenketten aneinader hängen

```
String zusammenGesetzt= new String(zeichenKette_3);
```

Code:

```
String zeichenKette_1 = "Hallo Welt";
String zeichenKette_2 = new String("Hallo Welt 2");

char[] zeichenKette_3 = {'h','a','l','l','o'};
String zusammenGesetzt= new String(zeichenKette_3);

// Ausgabe
println(zeichenKette_1);
println(zeichenKette_2);
println(zusammenGesetzt);
```

Die Methoden der String Klasse

`length()` Gibt die Anzahl der Buchstaben im verwendeten String.
`charAt(int)` Gibt anhand dem Index einen Buchstaben zurück.
`indexOf(char)` Gibt den Index des Buchstaben im String zurück.
`equals(String)` Vergleicht zwei Strings miteinander. Gibt bool.
`toLowerCase(String)` Konvertiert String nach Kleinbuchstaben.
`toUpperCase(String)` Konvertiert String nach Grossbuchstaben.
`substring(int)` Liefert einen Teil des Strings als neuen String.

Ich möchte hier näher auf die einzelnen Methoden eingehen. Sie sind desweiteren alle in der Processing Dokumentation enthalten und beschrieben.

`- length()`
Anders als bei den Primitven gibt uns die String Klasse Methoden vor um bestimmte Eigenschaften auszulesen. Man möchte also z.B. wissen, aus wie vielen Zeichen die Zeichenkette besteht, nutzt man die Methode `.length()`. Wenn sie aufgerufen wird, liefert sie einen Integer zurück, der weiss wieviele Buchstaben der String enthält. Schauen wir uns das in der Praxis an:

Code:

```
String zeichenKette_1 = "Hallo Welt";
println("Zeichenkette enhält: ");
println( zeichenKette_1.length() );    //- > 10
println("Buchstaben");
```

Wie wir in diesem Beispiel sehen können, ist es uns erlaubt, da der String ein Objekt ist - eine Methode darauf anzuwenden. Dies geschieht mit einem Punkt zwischen des Objekts und der Methode. -> `zeichenKette_1.length()`
Dieses Wissen können wir jetzt versuchen auf alle weiteren Methoden zu applizieren.

`- charAt(int)`
Diese Methode soll uns ein Buchstaben einer bestimmten Position des Strings zurück liefern. Wenn ich also den zweiten Buchstaben haben möchte benutze ich `String.charAt(1)`. In der Praxis:

Code:

```
String zeichenKette_1 = "Hallo Welt";
println( zeichenKette_1.charAt(1) ); // 'a'
```

- `indexOf(char)`

Wenn ich wissen möchte ob und an welcher Stelle ein bestimmtes oder mehrere Zeichen in einem String vorkommen oder nicht, bietet mir diese Methode das richtige Werkzeug. Falls das Gesuchte nicht im String gefunden werden kann liefert es -1 zurück. So kann man recht komfortabel überprüfen, ob z.B bestimmte Namen in Listen vorhanden sind.

Code:

```
// Prüfe ob der gesuchte Vorname enthalten ist
String myStr = "Wartmann : Christoph ";
println( myStr.indexOf("Christoph") ); // gibt 11
```

- `equals(String)`

Prüft auf Gleichheit zweier Strings und gibt einen boolschen Wert zurück.

Code:

```
// Prüfe ob der gesuchte Vorname enthalten ist
String myStr   = "Katze";
String suchStr = "Hund";
boolean pruefergebnis = myStr.equals(suchStr);
println( pruefergebnis ); // ergibt false
```

- `toLowerCase()`

Um einen String in Kleinbuchstaben zu konvertieren. Das Ergebnis kann einfach die Variable überschreiben.

Code:

```
// Konvertiere in Kleinbuchstaben
String myStr = "ChRiStoP WaRtmAnN";
myStr        = myStr.toLowerCase();
println( myStr ); // ergibt "christoph wartmann"
```

- `toUpperCase()`

Funktioniert nach dem gleichen Prinzip wie `toLowerCase()`. Mit dem Unterschied dass der String in Grossbuchstaben gewandelt wird.

Code:

```
// Konvertiere in Grossbuchstaben
String myStr = "ChRiStoP WaRtmAnN";
myStr        = myStr.toUpperCase();
println( myStr ); // ergibt "CHRISTOPH WARTMANN"
```

- substring(int)

Mit dieser Methode können wir einen Teil eines Strings kopieren. Dabei sollten wir aufpassen, denn diese Methode liefert einen neuen String zurück. Als Argument können wir den Anfangs- und Endpunkt festlegen.

Code:

```
// Schneide die letzten 5 Buchstaben aus dem String
String teilStr;     // Hier wird das Ergebnis gespeichert
String myStr = "Gedanken Mobil"; // Orginal String

// Scheidprozess
teilStr =
myStr.substring(myStr.length()-5,myStr.length());

// Ausgabe
println( teilStr ); // ergibt "Mobil"
```

° Arrays (Listen):

Es bleibt noch ein wenig trocken. Aber ich versuche den Code möglichst klein zu halten damit man das Kernproblem besser sehen und verstehen kann. Kommen wir zu den Arrays. Ein Array ist eine Art Schrank welcher mehrere Schubladen hat um Informationen rein zu stecken. Es kann aber auch so gross sein wie ein Hochhaus mit vielen Stockwerken. In jedem Stockwerk sind mehrere Zimmer in denen Schränke stehen usw. Da haben viele, viele Informationen Platz. Das Array ist also eine Art Container. Die Konstruktor des Arrays sieht wie folgt aus.

```
Typ[] name = new Typ[ Anzahl der Felder ];
```

Wie wir schon bei den „Strings" gesehen haben, brauchen wir eine Instanz von Array, die man durch „new" bekommt.

Ein Array in Processing kann immer nur einen Typ von Variable aufnehmen. Bei einer Namensliste wäre es der Typ String. Die Deklaration des Arrays sieht wie folgt aus.

```
String[] meinStrArray = new String[10];
```

Wir versuchen uns erstmal an einer Namensliste. Da Namen als Zeichenketten abgespeichert werden, ist unser Array vom Typ String. Eine Möglichkeit ein Array zu definieren, ist es zuerst zu deklarieren, um später Werte hinein zu schreiben. Die Anzahl der Felder des Arrays müssen schon am Anfang festlegt werden. Diese Anzahl kann in Processing im Nachhinein leider nicht mehr verändert werden. Der Index des ersten Felds fängt immer bei 0 an und nicht bei 1, wie man es ja eigentlich gewohnt ist. Das könne dann so aussehen:

Code:

```
// Deklariere Array vom Typ String
// mit 5 Feldern (0-4)
String[] namensListe = new String[5];

// Weise einen Namen einer Position zu
namensListe[0] = "Christoph"; // erste position
namensListe[1] = "Maria";     // zweite position
namensListe[2] = "Julia";     // etc...
namensListe[3] = "Martin";
namensListe[4] = "Josef";

// Zeige alle Elemente des Arrays
println( namensListe );

// Zeige ein Element des Arrays, z.B. das 2. Feld
println( namensListe[1] ); // zeigt 'maria'
```

Um an einzelne Werte des Array zu kommen schreibt man einfach den Index in eckige Klammern. So lassen sich auch Werte überschreiben.

```
array[index] = neuerWert.
```

Das war jetzt ein ein-dimensionales Array. Als nächstes möchte ich auch gern auf ein zwei-dimensionales Array zu sprechen kommen. Im letzten Beispiel hatten wir nur verschieden Spalten. Jetzt kommen noch Zeilen hinzu, wie in einem Exel Sheet oder einer Tabelle. In der Ersten Spalte steht immer noch der Vorname. In der Zweiten steht der Nachname und in einer dritten Spalte der Beruf.

Tabellenstruktur:

Christoph	Maria	Julia	Martin
Wartmann	Seiler	Kromer	Kazula
Assistent	Hilfsassistent	Administrator	Student

Versuchen wir das mal ins Array zu übertragen.
Der Konstuktor eines 2D-Arrays enthält lediglich eine zweite eckige Klammer.
`String[][] namensListe2D .`
Um später an den jeweilen Wert zu kommen, müssen wir den Index der Spalte und der Zeile angeben. Schauen wir uns das an einem Beispiel an:

Code:

```
// Deklariere 2D Array vom Typ String
// mit 4 Feldern (0-4)
// In Zeile 0 steht der Vorname in den Spalten
// In Zeile 1 steht der Nachname in den Spalten
// In Zeile 3 steht der Beruf in den Spalten
String[][] namensListe2D = new String[3][4];

//Weise der 1 Zeile die Vornamen zu
namensListe2D[0][0] = "Christoph";
namensListe2D[0][1] = "Maria";
namensListe2D[0][2] = "Julia";
namensListe2D[0][3] = "Martin";

//Weise der 2 Zeile den Nachnamen zu
namensListe2D[1][0] = "Wartmann";
namensListe2D[1][1] = "Seiler";
namensListe2D[1][2] = "Kromer";
namensListe2D[1][3] = "Kazula";

//Weise der 3 Zeile den Beruf zu
namensListe2D[2][0] = "Assistent";
namensListe2D[2][1] = "Hilfsassistent";
namensListe2D[2][2] = "Administrator";
namensListe2D[2][3] = "Student";

// Output des ersten Eintrags der Tabelle
println("Erster Vor-Nachname: "+namensListe2D[0][0]+" "
+namensListe2D[1][0]+" "+namensListe2D[2][0]);
```

Das Array hat eine `length` Eigenschaft. Sie gibt die Anzahl der Felder zurück.

Für die Anzahl der Zeilen:

```
println("Anzahl der Zeilen: "+namensListe2D.length);
```

Für die Anzahl der Spalten:

```
println(" Anzahl der Zellen: "+namensListe2D[0].length);
```

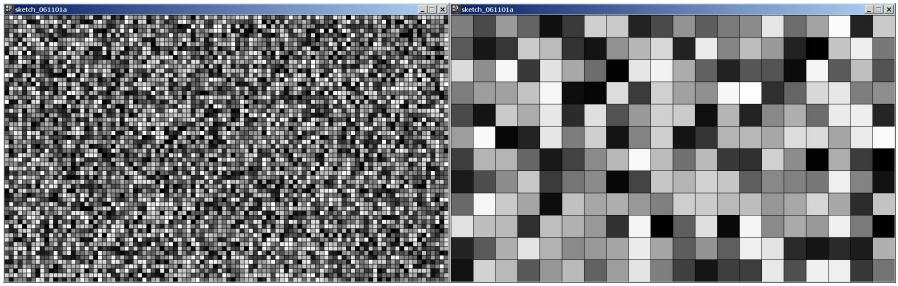
Um eine sinnvolle Einsatzmöglichkeit zu zeigen, möchte ich das gesamte Array, mit allen Feldern, Auslesen und Anzeigen lassen. Dies lässt sich am besten mit einer for-Scheife realisieren, eigentlich zwei. Man müsste sonst das Array händisch, Zeile für Zeile auslesen; wäre mühsam und nicht besonders dynamisch, da man bei Grössenänderungen des Arrays, Grossteile des Codes immer wieder schreiben würde.

Code:

```
// Schleift durch die Zeilen
for(int i = 0; i < namensListe2D.length; i++)
{
  // Output Zeile
  println("Zeile: "+i);
  println("-----------------------------------------------");

  // Schleift durch die einzelnen Zellen
  for(int j = 0; j < namensListe2D[0].length; j++)
  {
    // Schreibt die Inhalte der Zellen in den Output
    println(" Zelle:"+j+" Inhalt: "+namensListe2D[i][j] );
  }

  println("-----------------------------------------------");
}
```

Die erste Schleife geht durch alle Zeilen des Arrays bis „length" erreicht ist. Bei jedem Durchgang 0..1..2.. usw. führt es eine zweite Scheife aus, die wiederum alle weiteren Felder ausliesst bis lenght erreicht ist. Dann fängt es wieder von vorne an. Diese Technik werden wir im weiteren Verlauf immer wieder antreffen, deshalb lohnt es sich jetzt schon sie einer genaueren Studie zu unterziehen.

Übung zum Kapitel:
Also, gibt es hier noch eine nette Übung mit grafischer Ausgabe dazu um das Wissen zu vertiefen. Und so könnte es aussehen:



Die Übung besteht darin, Quadrate einer varablen Grösse und Farbe auf einer variablen Appletfläche, ganzflächig zeichnen zu lassen.

Vom Konzept her könnte es so gehen:
Wir definieren als erstes die Bühnengrösse, dann bestimmen wir die Grösse des Quadrats. Daraus ergibt sich die Anzahl der Quadrate, die wir zum Füllen der Bühne brauchen *(Bühnengrösse / Quadratgrösse)*. Die erste for-Schleife regelt den vertikalen Vorschub *(i * Höhe)* solange die Maximalhöhe nicht erreicht ist. Die Zweite den Horizontalen (*j* Breite*). Vor dem Zeichnen, generieren wir uns eine Zufallsfarbe, hier einen Grauwert über *random(int).* Das Zeichnen von Quadraten folgt den gleichen Regeln wie das Zeichenen von Ellipsen.

Kopier doch den Code - spiel mit den Variablen der Bühnengrösse (size) und der Quadratgrösse. Du wirst sehen, die Rechtecke werden sich automatisch an den Rand des Applets anpassen. Falls dir das Malen eines Rechtecks noch ein Rätsel ist, schau doch geschwind in die Dokumentation und mache ein eigenes Beispiel dazu.

Code:

```
size(800,600);                  // Appletgrösse
int quadgroesse = 50;           // Grösse des einzelen Quadrats
int zeilen = height/quadgroesse;// Zeilen Berechnung
int zellen = width/quadgroesse; // Zellen Bereichnung

for(int i = 0; i < zeilen; i++)
{
  for(int j = 0; j < zellen; j++)
  {
     fill(random(0,255));        // Setzen einer Zufallsfarbe
     //Und Loszeichenen
     rect((j*quadgroesse),(i*quadgroesse),quadgroesse,quadgroesse);
  }
}
```

4 Kontrollstrukturen

- if-Bedingung
- while-Loop
- for-Schleife
- Operatoren

4.0 Kontrollstrukturen

Ich möchte nun gern zuerst auf die „Bedingungen" und Schleifen, die ich im zweiten Kapitel angesprochen habe zu sprechen kommen und das Wissen vertiefen. Wir erinnern uns, dass die Bedingung (z.B. if-Anweisung) den Programmfluss steuern kann. Die Bedingung steht zwischen den runden Klammern. Die weiteren Anweisungen zwischen den Geschweiften. Anfänger machen hier gern den Fehler, dass sie zu Anfang ein Semikolon hinter der runden Klammer schreiben. Das führt zu einem semantischen Fehler. D.h. Es gibt keine Fehlerausgabe im Output Fenster. Für den Kompiler verhält es sich so als ob hinter der Bedingung keine Anweisung erfolgt und die eigentliche Anweisung immer ausführt. Denn sie wird ja durch keine Bedingung gesteuert.

Sehr oft benutzte Bedingungen sind sind z.B. Prüfen von:

Gleichheit	– *(x == y)*
Nicht Gleich	- *(x != y)*
Kleiner als	- *(x < y)*
Grösser als	- *(x < y)*
Kleiner oder gleich	- *(x <= y)*
Kleiner oder gleich	- *(x <= y)*

4.1 if – Bedingung

Die if-Anweisung haben wir auch schon bereits im Kapitel zwei kennen gelernt. Deshalb gehe ich hier gleich etwas weiter.
Ihre generisch Struktur der if-Bedingung:

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

Bisher waren unsere Abfragen jedoch sehr einfach. Wir haben immer nur eine Bedingung abgefragt. Es können aber auch Situationen entstehen an denen der Programmfluss von zwei oder sogar drei Bedingungen die gleichzeitig erfüllt sein müssen gesteuert wird. Wenn ich z.B. feststellen möchte, ob eine Variable in einem bestimmten Wertebereich liegt, trifft dies genau zu. Dazu haben wir die logischen Operatoren zur Hand. Damit lassen sich logisch verknüpfte Ausdrücke schreiben. Processing besitzt zwei: Das logische ODER || . Das logische UND &&. Der Code der prüfen soll, ob eine Variable zwischen zwei Wertebereichen liegt könnte so aussehen:

Code:

```
int x = 10;

if(x >= 5 && x <= 10 )
{
  println("x liegt im Bereich von 5 - 10");
}
```

Wenn ich prüfen möchte ob einer oder mehrere von drei verschiedenen Zuständen wahr sind, damit das Programm weiter machen kann könnte ich folgenden Code schreiben.

Code:

```
boolean x = true;
boolean y = false;
boolean z = false;

if(x == true || y == true || z == true)
{
  println("Logische oder Bedingung triff zu");
}
```

Diese Beispiele bieten jeweils nur eine Alternative. Nämlich wenn die Bedingung zutrifft. Das optionale Schlüsselwort *else* veranlasst eine zweiseitige Alternative. Da es auch passieren kann, dass eine Bedingung mal nicht zutrifft und man das auch wissen möchte. Z.B. bei der Prüfung von Geschlechtern. Wenn ich eine Frau bin dann werde ich in Zukunft mit „Frau" angesprochen ansonsten mit „Herr". Hierzu schreiben wir ein kleines Programm welches zwei Alternativen anbietet.

Code:

```
String name       = "Hollig";
String geschlecht = "Frau";

if(geschlecht == "Frau")
{
  println("Hallo Frau "+name);
}
else
{
   println("Hallo Herr "+name);
}
```

Hier sehen wir, dass die zwei unabhängigen Anweisungsblöcke durch das *else* verbunden werden können.

```
if( Bedingung )
{
   Anweisungen...
}
else
{
   Anweisungen...
}
```

Oder noch etwas komplexer und verschachtelter

```
if( Bedingung )
{
   if( Bedingung )
  {
    Anweisungen...
  }
}
else
{
   if( Bedingung )
  {
    Anweisungen...
  }
  else
  {
    Anweisungen...
  }
}
```

Da die if-Anweisung zur Programmführung häufig benutzt wird um einen bestimmten Wert zu überprüfen, kann man sie mit else */ if* schachteln.
Wenn eine Variable einem bestimmten Wert entspricht, wird eine Anweisung ausgeführt,sonst wird eine andere Variable mit einem Wert getestet und so weiter. So können komplexe Abfrage erstellt werden.
Dazu der generische Code:

```
if( Bedingung )
{
   Anweisungen...
}
else if( Bedingung )
{
    Anweisungen...
}
else if( Bedingung )
{
    Anweisungen...
}
else if( Bedingung )
{
    Anweisungen...
} etc...
```

Dazu ein keines Codebeispiel. Es Wert einer Variable soll geprüft in einer Aufzählung werden . Je nach bestimmten Wert soll das Programm immer andere Kommandos ausführen.
Der Code dazu könnte so aussehen:

Code:

```
int zahl = 3;

if(zahl == 0)
{
  println("Zahl ist: 0");
}else if(zahl == 1)
{
  println("Zahl ist: 1");
}else if(zahl == 2)
{
  println("Zahl ist: 2");
}else if(zahl == 3)
{
  println("Zahl ist: 3");
}else if(zahl == 4)
{
  println("Zahl ist: 4");
}else if(zahl == 5)
{
  println("Zahl ist: 5");
}else if(zahl < 5)
{
  println("Zahl ist grösser als 5");
}
```

4.2 while – Schleife

Eine weitere Kontrollanweisung ist die while Scheife. Im Kapitel 2 haben wir auch diese schon mal ein wenig kennen gelernt. Von der Struktur her ist die while-Schleife der if-Bedingung ähnlich. Nur dass die while-Schleife so oft ausgeführt wird wie ihre Bedingung zutrifft. Im Gegensatz dazu wird das if nur einmal ausgeführt. D.h. beim ersten Durchlauf wird in der Schleife die Bedingung geprüft. Wenn diese zutrifft, werden die Anweisungen (Schleifen-Körper) zwischen den geschweiften Klammern ausgeführt. Danach beginnt, anders wie beim if, das Programm wieder die Bedingung zu Prüfen. Wenn diese wieder zutrifft werden die Anweisungen, oder Schleifenkörper ausgeführt. Das passiert so lange bis die Bedingung eben nicht mehr zutrifft. Erst dann geht das Programm aus der Schleife raus und bearbeitet das nächste Kommando. Bei Programmieren von while-Schleifen sollte man unbedingt darauf achten dass keine Endlosschleifen produziert werden.
Die generische Struktur der while-Schleife:

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

Loop Terminologie:

Initialisierung
Die Angabe oder Ausdruck, die eine Variable oder mehrere definiert, welche in der Loop-Bedingung getestet werden.

Test des Ausdrucks
Die Bedingung, die zutreffen muss, damit die Subanweisungen im Loopkörper ausgeführt werden.

Update
Die Anweisung die die Variable modifiziert, welche in der Bedingung getestet wird. Eine typische Anwendung ist z.B. ein Zähler der die Schleifendurchgänge hoch oder runter zählt.

Verschachtelte Loops
Ein Loop der einen zweiten Loop beinhaltet. Das Erlaubt eine Iteration durch z.B. Zwei- oder Mehr- dimensionale Daten. Bei einer Tabelle ist der erste Loop für die Zeilen, der Andere für Iterierung der Zellen zuständig.

Iterator oder Index-Variable
Eine Variable die sich bei jedem Durchlauf des Lopps erhöht oder vermindert. Oft benutzt bei Zählern die Sequenziell durch Loops laufen. Herkömmlich heissen diese Zähler i, j, k ....

Loop body (Schleifen-Körper)
Beinhaltet den Block von Anweisungen die ausgeführt werden, wenn die Loop-Bedingung zutrifft. Der Schleifen-Körper kann gar nicht oder mehrere tausend Mal ausgefürt werden.

Loop Heder (Schleifen-Kopf)
Der Teil des Loops, der das Loop Schlüsselwort beinhaltet(*while, for*) und auf die dazugehörige Bedingung testet.

Endlosschleifen
Ein Loop der unendlich oft ausgeführt wird, weil der Wert seines Ausdrucks niemals falsch ist. D.h. Die Bedingung trifft immer zu.

Als kleine Übung zu diesem Thema wollen wir lauter Dreiecke der Breite nach zeichnen, bis das Applet in der komplette Breite ausgefüllt ist. Um ein Dreieck zeichnen zu können, verwenden wir das Freiform-Zeichenwerkzeug *beginnShape(TRIANGLES)*. Dazu iterieren wir jede x-Koordinate der drei Punkte , die es braucht ein Dreieck zu zeichnen. Die verschieden Zeichenwerkzeuge werde ich in einem gesonderten Kapitel erklären.
Und hier der dazugehörige Code:

Code:

```
size(600,80);             // Appletgrösse
int a = 0;                // Zähler oder x-Koordinate
while(a < width)          // Schleifen-Kopf und Bedingung
{
  beginShape(TRIANGLES); // Beginne Triangle
  vertex(a, 75);          // Erster  Vertexpunkt (x,y)
  a+=10;                  // Zähler wird um 10 pixel iteriert
  vertex(a, 20);          // Zweiter Vertexpunkt (x,y)
  a+=10;                  // Zähler wird um 10 pixel iteriert
  vertex(a, 75);          // Dritter Vertexpunkt (x,y)
  endShape();             // Schliesse Triangle
}
```

4.3 for-Schleife

Die for-Schleife ist mit der while-Schleife sinnverwandt. Der Unterschied ist, dass die for-Schleife den Iterator im Schleifenkopf definiert und in- oder dekrementiert. Die Syntax der Bedingung bleibt dieselbe. for-Schleifen werden oft genutzt um Aufzählungen abzuarbeiten.

Die generische Struktur der for-Schleife:

```
for(Initialisierung; Bedingung; Iteration)
{
   Anweisungen...
}
```

Die for-Schleife platziert also die Schlüssel-Komponenten, die es für eine Schleife braucht ordentlich in den Schleifen-Kopf. Sie werden durch ein Semikolon getrennt. Vor der ersten Iteration der Schleife, wird die Initialisierung des Zählers nur einmal vorgenommen! Wenn die Bedingung zutifft wird der Zähler iteriert und der Schleifen-Körper ausgeführt. Die schleife stoppt sobald ihre Bedingung falsch ist. Auch hier ist es möglich mehrere Schleifen ineinander zu verschachteln.

Damit man den Loop besser versteht und unterscheiden kann, hier zwei equivalente Beispiele mit der for- und while-Schleife.

Code:

```
// --- for-Schleife ---
for(int i = 0; i < 10; i++)
{
  println("Durchlauf for-Schleife: "+i);
}

// --- while-Schleife ---
int i = 0;
while(i < 10)
{
  println("Durchlauf while-Schleife: "+i);
  i++;
}
```

Um Bedingungen und Schleifen kontrollieren zu können brauchen wir zu unseren Operanden auch Operatoren. Die ich gerne auf der nächsten Seite aufführen möchte.

4.4 Operatoren

Ein Operator ist ein Symbol oder Schlüsselwort, dass Daten manipuliert, kombiniert und transformiert. Einige der Operatoren sind auch dem Nicht-Programmierer bestens bekannt, wie z.B:

- Das Multiplizieren (4*3). Der Operand(4) wird mit dem (*) Operator und einem anderen Datenwert (Operand 3) im Ausdruck multipliziert.
- Wir finden auch den = Operator bei jeder Zuweisung von Variablen-Werten.
- Um Bedingungen zu schreiben brauchen wir die Vergleichsoperatoren (<, >, ==,...) somit können Werte und Variablen miteinander verglichen werden.
- Und nicht zu Letzt die logischen Operatoren(&&, ||), die Verknüpfungen von Variablen oder Bedingungen ermöglichen.

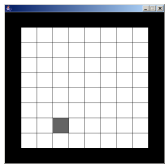
Viele haben wir in den letzten Skripten schon benutzt.

Der Vollständigkeitshalber möchte ich hier auf die verfügbaren Operatoren eingehen, die uns Processing bietet.

Operator	Beschreibung
.	Zugang zur Objekt Eigenschaft
[]	Zugang zum Array Element
()	Parenthese
function()	Funktionsaufruf
x++	Postfix increment (zählt x um 1 rauf)
y--	Postfix decrement(zählt y um 1 runter)
!	Logisches NICHT (NOT)
new	Neue Instanz
void	Keine bestimmte Rückgabe
*	Multiplikator
/	Division
%	Modulo
+	Addition
-	Subtraktion
<	Kleiner als
<=	Kleiner oder gleich als
>	Grösser als
>=	Grösser oder gleich als
==	Gleichheit
!=	Nicht-Gleichheit
&&	Logisches UND
||	Logisches ODER
=	Zuweisung
+=	Addition und Zuweisung
-=	Substraktion und Zuweisung
*=	Multiplikation und Zuweisung
/=	Division und Zuweisung

Übung zum Kapitel:

Als Trainingsübung habe ich die letzte Übung zum Kapitel 3 vom Konzept her erweitert. Es soll nun ein Raster gezeichnet werden das am Rand schwarz eingefärbt wird. Ein Quadrat soll sich per Zufall über das Raster bewegen ohne jemals über den Rand gezeichnet zu werden.



Code:

```
int quadrat = 50;                // Grösse des einzelen Quadrats
int zeilen;                      // Anzahl der Quadrate in x Richtung
int zellen;                      // Anzahl der Quadrate in y Richtung

float xpos;                      // x Postion des Quadrats
float ypos;                      // y Postion des Quadrats

void setup()                     // Aufruf bei Initialisierung
{
   framerate(5);                 // Setzen der Abspielgeschwindigkeit
   size(500,500);                // Setzten der Appletgroesse
   zeilen = width/quadrat;       // Zeilen Berechnung
   zellen = height/quadrat;      // Zellen Bereichnung
}


void draw()                      // Aufruf bei jeder Frame
{
  background(255);               // Setzten des Hintergrunds

  /************************************************************
  Malen des Rasters
  ************************************************************/
  for(int i = 0; i < zeilen; i++)
  {
    for(int j = 0; j < zellen; j++)
    {
      // Bedingung für den schwarzen Rahmen
      if(i == 0 || i == zellen-1 || j == 0 || j == zeilen-1 )
      {
         fill(0);
         rect((j*quadrat),(i*quadrat),quadrat,quadrat);
      }else{
         fill(255);
         rect((j*quadrat),(i*quadrat),quadrat,quadrat);
      }
    }
  }
  /************************************************************
  Setzten und Zeichnen der Zufallskoordinaten des Quadrats
  ************************************************************/
  xpos = (int)random(1,zellen-1);  // Wird passend konvertiert(int)
  ypos = (int)random(1,zeilen-1);  // sonst passt's nicht ins Raster

  fill(100);                       // Füllfarbe des Quadrats
  rect((quadrat*xpos),(quadrat*ypos),quadrat,quadrat);

}
```

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.

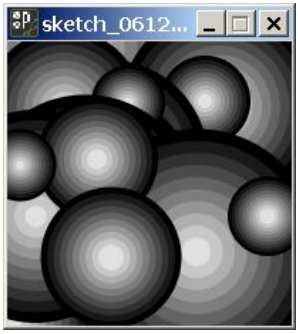
Code:

```
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);
  }
}
```

6 OOP – Objekt orientierte Programmierung

- Was ist OOP
- Klassen und Objekte
- Verdeckung
- Konstanten
- Zerstörung von Objekten
- Mehrere Konstruktoren
- Vererbung
- Super
- OOP Dokumentation

6.0 OOP – Objekt orientierte Programmierung

OOP – ist ein Programmierkonzept, dass von vielen höheren Programmier-sprachen (Java, C++, Actionscript, PHP etc...) unterstützt wird. Es ist also keine neue Programmiersprache, sondern eine andere Sichtweise auf Programmierung und Problemlösung. Oder anders gesagt, es ist ein Konzept, wie man über Programmierung nachdenkt und herangeht. Grundsätzlich kann man sagen, dass OOP – Programme nicht unbedingt effizienter, schneller oder schlanker sind. Jedoch ist ein Programm, dass in OOP programmiert worden ist, übersichtlicher und kann schneller von anderen Programmieren verstanden werden. Der Code ist Nachhaltiger, da er in grossen Teilen immer wieder von anderen Programmen verwendet werden kann. Im Mittelpunkt von OOP steht das Objekt. Statt wie bisher prozedural oder mit Elementen zu arbeiten, bietet das OOP eine Kapselung von Elementen und Methoden. So kann das Objekt als Einheit betrachtet werden und ein hohes Mass an Komplexität aufweisen.

Wenn du neu im Objekt orientierten Programmieren bist, hast du sicherlich schon davon gehört, dass OOP für manche ein grosses Mysterium ist, sehr schwierig zu verstehen oder zu lernen. Das stimmt nicht! Es ist eine sehr intuitive Art zu programmieren. OOP ist leichter als du glaubst. In OOP begreifst du einfach Abschnitte des Codes als abgeschlossene Objekte. Abgeschlossen heisst hier aber nicht von einander isoliert, sondern unabhängig voneinander. Man kann Objekte benutzen ohne sich Gedanken um ihre internen Details machen. Objekte sind einfach zu verstehen, wenn man überlegt, dass die ganze Welt aus Objekten besteht und man jeden Tag damit konfrontiert wird. Ein Auto, ein Hund, der Computer, ein Schuh, ein Haus – alle sind Objekte. Ich meine, alle diese Objekte können unabhängig von einander oder miteinander interagieren. Sie können auch Dinge auf Anfrage machen. Der Hund z.B. kann auf Zuruf eintrainierte Kommandos ausführen, selbst wenn man die internen Details des Hundes (Knochen, Muskeln, Gewebe usw.) nicht kennt. D.h. Man muss kein Biologe sein um einem Hund beizubringen wie man einen Stock zurück bringt, man muss kein Ingenieur sein um ein Auto zu fahren, man muss auch kein Psychologe sein um mit seinen Eltern ein Gespräch zu führen oder ein Computerwissenschaftler um E-mails abzurufen. Alles was man braucht, sind die Kommandos eines Objekts (sie heissen Methoden), die es befolgen soll und die Resultate daraus. Zum Beispiel, wenn ich das Gaspedal meines Autos durch-drücke, dann erwarte ich, dass es beschleunigt. Wenn ich meinem Hund sage „sitzt", dann erwarte ich, dass er sich hinsetzt.
Gerüstet mit dieser weltlichen Anschauung was ein Objekt in der realen Welt ist und tut, werde ich versuchen dies auf die Programmierung und Processing zu übertragen.

Das klassische Beispiel ein Objekt zu programmieren ist ein springender Ball. Wie ein echter Ball hat sein programmierter Objekt-Repräsent auch gewisse Eigenschaften, die von Attributen dargestellt werden, wie z.B. Radius, Farbe, Masse, Position, und Material (Elastizität). Um einen Ball in unserer Programmierung zu representieren, erzeugen wir ein Objekt welches die Eigenschaft *radius* hat, eine andere Eigenschaft enthält die *Farbe* und so weiter... Die Eigenschaften des Objekts repräsentieren einen Zustand zu einer gewissen Zeit. Manche Eigenschaften können sich aber auch zu Zeiten ändern.
Im Gegensatz zum echten Leben kann ich als Programmieren jeder Zeit, wenn ich will, meinem abstrakten Ball-Objekt ein anderes Material zuweisen. Bei einem reellen Ball wird das schwierig nachdem er erzeugt worden ist :-) Oder noch besser, dem Objekt-Hund einfach sieben Beine hinzufügen...

Um unseren Ball zu bewegen brauchen wir nicht unbedingt Newton Gesetzte anzuwenden, sondern wir beschreiben es als Computerausdruck wie sich der Ball in einer bestimmten Zeitspanne bewegen soll. Die simpelste Methode wäre da, die Fahrt multipliziert mit der Zeit geteilt durch die Distanz.

```
Position = Geschwindigkeit * Zeit
```

Wir können nun diese Gleichung nach Processing übersetzten, die die neue Ball Position berechnet:

```
ball.position += ball.xGeschwindigkeit * vergangeneZeit;
```

Diese Objekt-Verhalten sind einfach Dinge oder Dienste die das Objekt ausführen kann. Einer unserer Ball-Verhalten hat die Fähigkeit sich zu Bewegen. Um ein Objekt-Verhalten in OOP zu beschreiben verwenden wir die so genannten *Methoden*. Wir kennen Methoden schon von dem letzten Kapitel. Da hiessen sie nur ein bisschen anders, nämlich Funktionen. Im Objekt-Kontext heissen sie Methoden, sind aber immer noch Anweisungsblöcke. So kann ein Objekt mehrere Methoden haben, z.B. `bewegen()`, `spring()`, `stop()` usw...
Wir können also eine eine `spring()` Methode erzeugen, die die Richtung des Balls umdreht und die Geschwindigkeit abbremst. Die Gleichung der Methode könnte so aussehen:

```
ball.xGeschwindigkeit = -(ball.xGeschwindigkeitt) * 0,95;
```

O.K. um zum Thema zu kommen, lass uns doch ein paar Definitionen formalisieren. Ein Objekt ist eine abstrakte (Daten)Struktur, die verschiedene, in Beziehung stehende Eigenschaften (Variablen) und Methoden (Funktionen) gruppiert, d.h. ein Objekt kapselt seine Verhalten. Die internen Details, wie die Verhalten ausgeführt werden sind nicht unbedingt ausserhalb des Objekts sichtbar. So interagiert das Programm mit einem Objekt durch seine *interfaces.* (d.h. Objekt-Methoden die öffentlich von Ausserhalb erreichbar sind)
Der Rest des Programms macht sich keine Gedanken darüber, was das Objekt tut und warum es das tut. Andersherum, das Programm bietet dem Objekt I*nputs* an und checkt die *Outputs* wenn sie geeignet sind. Natürlich können auch Objekte untereinander kommunizieren. Denn wir erinnern uns, dass z.B. der Hund mit anderen Hunden interagieren kann.

Du hast vielleicht in der Beziehung OOP schon mal von Klassen (*class*) oder Instanzen (*instance*) gehört. Eine Klasse ist zunächst ein abstraktes Modell von einem Objekt, mit all seinen Eigenschaften und Methoden. Erst wenn man eine Klasse instanziert, erzeugt man ein Objekt (Kopie) von dieser Klasse. Dieses Objekt heisst dann Instanz. Diese Instanz hat immer einen bestimmten Namen, den man natürlich selbst vergeben muss.

Um es an einem Beispiel zu verdeutlichen:
Dein Hund zu Hause ist eine Instanz von einer abstrakten *Hund-Klasse.* Sowie jeder Hund, der der *Hund-Klasse* abstammt bellt, hat vier Beine und Pfoten. Aber dein Hund hat eigene bestimmte Werte, wie z.B die Grösse, das Gewicht, die Fellfarbe. Wenn man sich dieses Konzept verdeutlicht hat, versteht man auch das Konzept der Objekt-orientierten-Programmierung.

Egal ob wir eigene Objekte erschaffen oder die Objekte die uns Processing bietet benutzen (z.B. *array.length()*), OOP bewahrt die verschiedenen Teile des Programms separat und sauber von einander getrennt (Kapselung) auf und lässt sie untereinander operieren ohne die internen Details zu kennen.
Zurück zu unserem Ball Beispiel. Das Programm kümmert es wenig, ob das newtonsche Gesetz unseren Ball bewegt, es ruft nur die Ball-Objekt Methode `bewegeDich()` auf und lässt das Objekt sich um die Bewegung kümmern.

Ein anderes nettes Feature welches uns OOP bietet ist, dass verschiedene Objekte gleiche Methoden haben können. Solange ihre Namen gleich sind. Nehmen wir zum Beispiel einen Kreis und ein Rechteck. Solange beide Objekte die Methode `getArea()` haben, die den Wert der Fläche zurück gibt, brauchen wir uns beim Aufruf keine Gedanken machen wie das Objekt seine Fläche berechnet. Wir erwarten einfach den Wert der Fläche. Na klar, beim schreiben der abstrakten Klasse müssen wir es natürlich wissen, denn wir programmieren es ja immerhin:-)

Des weiteren werden wir in diesem Kapitel lernen, wie man ein Objekt erzeugt und wie man eine Klasse schreibt. Wenn wir damit komfortabel geworden sind werde ich auf die Vererbung zu sprechen kommen,d.h. wie Klassen gemeinsame Charakteristiken teilen können.

Die Anatomie eines Objekts

Wie ein Array ist auch ein Objekt ein Container dass verschiedene Container enthält. Das typische Array hält multiple Werte in individuell nummerierten Elementen; ein Objekt analog dazu, hält multiple Werte in individuellen namentlich benannten Eigenschaften. Eine Eigenschaft ist eine Variable, die auf einem Objekt definiert ist. Eigenschaften in der Programmiersprache Java heissen, Instanzvariablen, denn sie gehören immer zu einem bestimmten Objekt. Eine Methode, im Vergleich ist eine Funktion, die auf einem Objekt definiert ist. Praktisch gesprochen ist ein Objekt nichts anderes als in Beziehung stehende Variablen und Funktionen. Die Eigenschaften (Variablen) eines Objekts werden nie wieder nachdem sie erzeugt worden sind „angefasst". Um ihre Werte später einmal zu verändern, schreiben wir eine Methode dazu, die sie überschreibt oder ausliest. So bleiben wir dem Konzept der Kapselung treu.

```
int eigenschaft = 10;

int getEigenschauft()      // um die Variable auszulesen
{
   return  eigenschaft;
}

void setEigenschauft();      // um die Variable auszulesen
{
   eigenschaft = neuerWert;
}
```

Mit dieser Programmierpolitik (getter, setter) bleibt unser Code immer schön sauber.

6.1 Klassen

Bist du bereit dir die Finger schmutzig zu machen? Jetzt kommen wir nämlich zum Spass Teil:-) Wir kreieren unsere erste Klasse anhand des vorhergehenden Ball Beispiels. Der erste Schritt eine Klasse zu designen ist es die Eigenschaften und Methoden der Klasse die sie anbietet aufzuschreiben. Unser Ball hat also:

Eigenschaften	*Methoden*
Grösse | `radius`	Zeichne | `draw()`
Farbe | `colour`	Bewegung | `move()`
Position X | `xpos`	Wechsle Farbe | `changeColor()`
Position Y | `ypos`	
Geschwindigkeit X | `velX`	
Geschwindigkeit Y | `velY`	

Es hilft ungemein sich seine Basisklasse erst einmal vorzustellen, dann fällt das Programmieren leichter, denn man hat das Objekt durchdacht und kann sich auf den Code konzentrieren. Falls man doch noch eine Methode oder Eigenschaft vergessen hat, ist es dank Kapselung kein Problem sie hinter her zu implementieren.

Herstellung einer Klasse

Es gibt eine spezifische Klassen Dekleration in Processing. Sie folgt den selben Regeln wie Java und ihre generische Struktur sieht so aus:

```
class Name                                        // ... Klassen Name
{

   ...Platz für Dekleration der globalen Klassenvariablen

   Name(Argument,Argument...)                     // ... Konstruktor
   {
        ... Initialisierung der Variablen
        ... wird automatisch einmal beim Instanzieren
        ... ausgeführt (optinal)
   }

   Typ MethodenName(Argument,Argument...)    // ... Methode
   {
        ... Anweisungen
   }

   ... weiter Methoden des Objekts

}
```

Bitte beachte, dass der Klassenname und der Name der Konstruktors gleich sein müssen. So weiss der Kompiler, dass diese *„Funktion"* bei der späteren Instanzierung automatisch aufgerufen wird. Im Konstruktor werden alle globalen Variablen (Eigenschaften), die das Objekt braucht gesetzt. Hier wird kein Typ angegeben. Die Konstruktor-Methode folgt den selben Regeln, in Definition und Aufruf,wie die bereits im letzten Kapitel besprochene Funktionen.

Instanzierung einer Klasse

Um ein Objekt einer Klasse abzuleiten, brauchen wir das *new* Schlüsselwort. Die generelle Syntax ist:

```
new KonstruktorFunktion()
```

in dem obigen Beispiel würde es dann so aussehen:

```
Name InstanzName = new Name(Parameter, Parameter...);
```

Wir kennen dieses Prinzip schon von vorhergehenden Kapiteln, zum Beispiel den Arrays. Da haben wir die selbe Technik angewandt um ein Array zu generieren. Erinnern wir uns daran:

```
Typ[] Array = new Array[index];
```

Der einzige Unterschied der Klasse ist, dass die Klasse einen eigenen Typ hat. Dh der Typ der Klasse ist immer der Klassenname. Übertragen wir unser Wissen jetzt in die Ball-Klasse und lassen mal ganz einfach einen Ball zeichnen. Wir benutzten hier nur das Nötigste um dies zu tun.

Code:

```
class Ball                              // Klassendefinition
{
  float radius;                         // Variablen Dekleration
  float xPos;                           // Variablen Dekleration
  float yPos;                           // Variablen Dekleration

  Ball(float r, float xp, float yp)     // Konstruktor mit Parametern
  {
       radius = r;                      // Wertübergabe von lokalen
       xPos = xp ;                      // zu globalen Variablen
       yPos = yp;                       // übergeben aus Instanzierung
  }

  void draw()                           // Zeichenmethode
  {
    ellipseMode(CENTER);                // setzt Kreismittelpunkt
    ellipse(xPos,yPos,radius*2,radius*2);  // zeichne Ellipse
  }
}                                       // Ende der Klasse

// ---------------------------------------------------------------------

void setup()                            // Initialisierung des
{                                       // Programms
  Ball myball = new Ball(20,50,50);     // Instanzierung der Klasse
  myball.draw();                        // Aufruf der Objektmethode
}
```

So, das sollte es gewesen sein. Eigentlich ganz einfach, wenn man das Prinzip versteht. Als Nächstes fügen wir die weiteren Methoden und Eigenschaften nacheinander der Basisklasse hinzu, die wir uns am Anfang des Kapitels erarbeitet haben. Wir fangen mir der Bewegung an, das wäre die Methode `move()`.

Das impliziert zwei weitere Eigenschaften in unserer Basisklasse , die Geschwindigkeit in X und Y Richtung:

```
velX und velY
```

Um den Ball zu bewegen brauchen wir die Gleichung, die wir uns auch schon erarbeitet haben - für die X und Y Achse:

```
ball.position += ball.velX;
```
und
```
ball.position += ball.velY;
```

Und hier das vollständige Skript (roten Zeilen sind zum Letzten dazugekommen)

Code :

```
class Ball                              // Klassendefinition
{
  float radius;                         // Radius Variable
  float xPos;                           // Position in X
  float yPos;                           // Position in Y
  float velX;                           // Geschwindigkeit in X
  float velY;                           // Geschwindigkeit in Y

  Ball(float r, float xp, float yp ,float vx, float vy)
  {
       radius = r;                      // Wertübergabe von lokalen
       xPos = xp ;                      // zu globalen Variablen
       yPos = yp;                       // übergeben aus Instanzierung
       velX = vx;                       // Dieses Mal auch mit der
       velY = vy;                       // Geschwindigkeit
  }

  void move()
  {
    draw();                         // Aufruf den Ball zu zeichnen
    xPos += velX;                   // Berechnung der neuen X Position
    yPos += velY;                   // Berechnung der neuen Y Position
  }

  void draw()                               // Zeichenmethode
  {
    ellipseMode(CENTER);                    // setzt Kreismittelpunkt
    ellipse(xPos,yPos,radius*2,radius*2); // zeichne Ellipse
  }
}

//---------------------------------------------------------------------
Ball myball;                          // Globale Definition Ball
void setup()                          // Initialisierung des Programms
{
  size(500,500);                      // Definition Applet Grösse
  myball = new Ball(20,50,50,2,2); // Instanzierung der Klasse Ball
}

void draw()                           // Wird an jeder Frame aufgerufen
{
  background(30);                     // Zeichne Hintergrund
  myball.move();                      // Aufruf der Ball Objekt-Methode
}
```

Anhand dieses Beispiels kann man die Kapselung eines Objekts sehr gut erkennen. Die Methode `move()` ist nur dazu da den Ball zu bewegen, die Methode `draw()` nur den Kreis zu zeichnen und der Konstruktor bereitet alle globalen Variablen des Objekts zur weiteren Verarbeitung vor.

Nach dem Kompilieren und Starten des Applets, sehen wir, wie der Ball sich diagonal mit einer Geschwindigkeit von zwei Pixeln per Frame fortbewegt und aus dem Applet verschwindet. Das sollte geändert werden. D.h. wenn der Ball an den Rand des Applets stösst, soll er in die entgegengesetzte Richtung laufen. Dieses Verhalten können wir mir if-Bedingungen abfragen und die Geschwindigkeit dementsprechend umkehren. Schön wäre es auch wenn wir nicht nur einen Ball hätten, sondern z.B. 50 verschiedene Bälle, die alle eine andere Farbe und Geschwindigkeit besitzen würden. Dank OOP ist das eine leichte Übung, denn ein Ball gleicht dem anderen und wird immer durch eine for-Schleife von der Ball Klasse abgeleitet (es ändern sich nur die Parameter) und in einem Array abgespeichert. So stellen wir sicher, dass man später jeden einzelnen Ball ansprechen kann. Die Grösse, Farbe und Geschwindigkeit werden per random bestimmt, sonst folgt jeder Ball den gleichen Regeln. Wir können daher ohne Problem auf dem bereits bestehenden Script aufbauen. Die `move()` Methode wird durch eine Abprallbedingungen erweitert und jedes Mal wenn ein Ball an den Rand stösst, dann soll er seine Farbe wechseln. Dafür brauchen wir eine weiter Methode namens `changeColor()`. In dieser Methode wird ein Grauwert per Random erstellt. Die verschiedenen Grössen und Farben werden beim Instanzieren statt wie bisher mit festen Werten auch durch Random Werte ersetzt. Also Ärmel hoch krempeln und los geht es! Der Code in rot ist neu oder überarbeitet.

Code:

```
class Ball                                  // Klassendefinition
{
  float radius;                             // Radius Variable
  float xPos;                               // Position in X
  float yPos;                               // Position in Y
  float velX;                               // Geschwindigkeit in X
  float velY;                               // Geschwindigkeit in Y
  color col = color(255);                   // Ball Farbe

  Ball(float r, float xp, float yp ,float vx, float vy)
  {
       radius = r;                          // Wertübergabe von lokalen
       xPos = xp ;                          // zu globalen Variablen
       yPos = yp;                           // übergeben aus Instanzierung
       velX = vx;                           // Dieses Mal auch mit der
       velY = vy;                           // Geschwindigkeit
  }

  void move()
  {
    draw();                           // Aufruf den Ball zu zeichnen
    if(xPos + radius > width)         // Abfrage ob der Ball rechts
    {                                 // angestossen ist - dann
      changeColor();                  // wechsel die Farbe und
      velX = -velX;                   // kehre die Geschwindigkeit um
    }else if(xPos - radius < 0){      // wenn Ball links angestossen ist
      changeColor();                  // wechsel die Farbe und
      velX = -velX;                   // kehre Geschwindigkeit um
    }
```

Code:

```
    if(yPos + radius > height)    // Dasselbe mit der Y Achse
    {                             // D.h. oben und unten
      changeColor();
      velY = -velY;

    }else if(yPos -radius < 0){
      changeColor();
      velY = -velY;
    }

    xPos += velX;                 // Berechnung der neuen X Position
    yPos += velY;                 // Berechnung der neuen Y Position
  }

  void changeColor()              // Methode für Farbwechsel
  {
    col = (int)random(255);
  }

  void draw()                                 // Zeichenmethode
  {
    ellipseMode(CENTER);                      // setzt Kreismittelpunkt
    fill(col);
    ellipse(xPos,yPos,radius*2,radius*2); // zeichne Ellipse
  }
}

//-----------------------------------------------------------------------
int ballnumber = 50;                      // Anzahl der Bälle
Ball[] myBalls = new Ball[ballnumber]; // Definition Ball Array
                                          // somit kann jeder Ball
                                          // später durch seinen Index
                                          // direkt angesprochen
                                          // werden

void setup()                           // Initialisierung des Programms
{
  size(500,500);                          // Definition Applet Grösse
  for(int i = 0; i < ballnumber; i++)  // Schleife zur Generierung
  {                                       // von den 50 Bällen
    // Instanzierung der Klasse Ball mir Random Werten
    myBalls[i] = new Ball(random(5,20),50,50,random(5),random(5));
  }
}

void draw()                            // Wird an jeder Frame aufgerufen
{
  background(30);                            // Zeichne Hintergrund
  for(int i = 0; i < ballnumber;i++)      // Schleife durch das
  {                                       // myBall Array um alle
    myBalls[i].move();                    // Bälle zu bewegen
  }
}
```

Ich hoffe, ich konnte mit diesem Beispiel die grosse Kraft, die hinter OOP steht aufzeigen. Es werden hier weitere Beispiele folgen, denn wann immer es sich wieder anbietet Objekt-orientiert zu Programmieren werde ich es aufgreifen.

6.2 Klassen in der Processing IDE aufteilen

Wenn das Programm mehr als zwei oder drei Klassen besitzt gibt, es die Möglichkeit, sie in verschiedene geteilte Klassenskripte zu dividieren und in der Processing IDE als Tabs nebeneinander anzuzeigen. Der Vorteil der dadurch entsteht liegt auf der Hand. Jede Klasse bekommt sein eigenes File. Somit teilen wir das Hauptprogramm in die verschiedenen Klassen und bekommen eine bessere Struktur. Die einzelnen Skripte werden dadurch kürzer, lesbarer und wir behalten das Konzept der Kapselung bis zuletzt ein.

Du fragst dich wie das geht? Ganz einfach! Schneide aus dem letztem Beispiel nur die Klasse aus. Erstelle ein neues File über das Dropdown Menü „FILE" -> „NEW" und setze es ein. Dannach speicher es ab. Lade jetzt wieder das Hauptprogram. In diesem Hauptprogramm befinden sich nur noch die *setup* und *draw* Funktionen plus die globalen Variablen. Stelle sicher, dass alle Klassen entfernt wurden. Um die Klasse dem Hauptprogramm wieder hinzuzufügen, einfach über das Processing IDE dropdown Menue „SKETCH" -> „ADD FILE" die vorher abgespeicherte Klasse einladen. Sie erscheint dann als weiteres Tab neben dem Hauptprogramm. Bitte speicher und starte jetzt das Hauptprogramm um zu sehen ob es funktioniert hat. Besser noch - man speichert seine Klassen einem Unterordner des Projekts, mit dem Namen „classes" und lädt sie erst danach ein. So befinden sich alle an einem Platz und gehen nie wieder verloren.

6.3 Verdeckung

Eine Eigenart von Processing ist das Verdecken von globalen Variablen durch Instanzvariablen. Da lauert der Fehlerteufel! Wenn ein globale Variable im Hauptprogramm den gleichen Namen hat wie eine Instanzvariable eines Objekts, dann verdeckt die Instanzvariable die Globale, wenn man versucht die globale Variable aus dem Objekt zu ändern. Abhilfe schafft da nur, alle globalen Variablen mit einzigartigen Namen zu versehen. Das folgende Skript illustriert das.

Code:

```
class Ueberdeckung
{
    int position ;             // Lokale Variable

    void setPosition(int pos) // Methode setzte Position
    {
      position = pos;        // setzte vermeidlich globale variable
      this.position = pos;  // setzte Instanz Variable
    }

    int getPosition()       // Methode gebe Position zurück
    {
      return this.position;
    }

}

// ---------------------------------------------------------------
```

Code:

```
int position = 30;      // Globale Variable

 void setup()            // Initialisierung des Programms
 {
   Ueberdeckung obj = new Ueberdeckung();    // Erzeuge Instanz
   obj.setPosition(40);  // Setzte obj Instanzvariable auf 40
   println("obj: "+obj.getPosition() ); // gibt überschrieben 40
   println("Global: "+position ); // Gibt unverändert 30 zurück
 }
```

Wir sehen das neue Schlüsselwort `this`, dass auf die Instanz Variable `position` zeigt. Man benutzt dies um explizit auf die Instanzvariable einer Klasse zu verweisen. Wenn eine globale Variable den gleichen Namen wie ein Instanzvariable trägt wird die globale nicht überschrieben, denn Processing schreibt der Einfachheit halber immer wenn man kein `this` in einer Klasse benutzt automatisch das eins dazu. Hier gilt es aufzupassen.

6.4 Konstanten

Instanzvariablen die nur einmal gesetzt und später nicht mehr verändert werden dürfen, kann man mit einem `final` vor der eigentlichen Variablendefinition versehen. Die dient später dazu wichtige Variablen vor nicht gewollten Zugriffen zu schützen.

```
final int variable = 10;
```

6.5 Zerstörung von Objekten

Sowie man Objekte erzeugen kann, ist es möglich sie auch wieder zu zerstören, dh. aus dem Speicher zu löschen. Dafür gibt es da Schlüsselwort `null`. Wenn man das einem Objekt zuweist dann ist es zerstört. Schauen wir uns das anhand eines Beispiels an.

Code:

```
class Obj
{
  int a = 10;
}

void setup()
{
  Obj obj = new Obj();
  println(obj.a); // Gibt 10 zurück
  obj = null;     // Zerstöre Objekt
  println(obj.a); // Gibt einen FEHLER aus da o nicht existiert
}
```

6.6 Mehrere Konstruktoren

Eine Klasse kann mehrere Konstruktoren für unterschiedliche Zwecke besitzen, aber nur ein Konstruktor initialisiert das Objekt. Nehmen wir an, wir wollen ein Objekt mit zwei Eigenschaften, einmal normal konstruieren und eine anderes mal durch ein Objekt.

Code:

```
class Hochschule
{
  int Studierende;
  int Dozenten;

  Hochschule(int a ,int b)
  {
    Studierende = a;
    Dozenten    = b;
  }

  Hochschule(Hochschule obj)
  {
    Studierende = obj.Studierende;
    Dozenten    = obj.Dozenten;
  }
}

void setup()
{
  Hochschule epfl = new Hochschule(9000,800);   // Übergabe int,int
  println(epfl.Studierende+":"+epfl.Dozenten);  // Gibt Anzahl
  Hochschule eth = new Hochschule(epfl);    // Übergabe Objekt
  println(eth.Studierende+":"+eth.Dozenten);// Ergibt gleiche Anzahl
}
```

Beide Ergebnisse liefern die gleichen Werte zurück, denn das zweite Objekt `eth` wurde durch das bereits existierende `epfl` erzeugt.

6.7 Vererbung

Für eine Wiederverwendbarkeit von Klassen gibt es die Vererbung. Somit kann eine Kind-Klasse von einer Mutter-Klasse Eigenschaften und Methoden erben. Man stelle sich die Klasse „Säugetier" vor. Jedes Säugetier hat ein Herz und eine Lunge. Das verbindet alle Säugetiere miteinander. D.h. alle Säugetiere erben dies von dieser Mutter-Klasse. Jedes Säugetier kann ganz verschieden aussehen. Es kann verschieden viele Beine haben, oder auch aus verschiedenen Materialien bestehen (Haut, Fell). Wenn wir also z.B einen Hund beschreiben wollen, erbt die Hund-Klasse von der Säugetier-Klasse das Herz und die Lunge.

Eine neue definierte Klasse kann durch das Schlüsselwort `extends` eine Klasse erweitern. Sie wird dann zur *Unter-* oder *Sub*-Klasse bzw. *Kind*-Klasse.

Die *Kind*-Klasse erbt also von der *Eltern*-Klasse. Sie kann auch *Ober*-Klasse oder *Super*-Klasse heissen.
Durch dieses Prinzip werden alle Eigenschaften und Methoden der *Ober*-Klasse in die *Kind*-Klasse übertragen. Eine *Ober*-Klasse vererbt also Eigenschaften an die *Unter*-Klasse.

```
class UnterKlasse extends Oberklasse
{
}
```

Hiermit werden die Eigenschaften und Methoden der Oberklasse vererbt und die UnterKlasse kann diese für sich nutzen. Ich möchte das anhand eines Beispiels aufzeigen. Nehmen wir an die Klasse Hochschule erbt von der Klasse Gebäude.

Code:

```
class Gebaeude
{
  int raeume = 100;
}

class Hochschule extends Gebaeude
{
  int getAnzahlRaeume()
  {
    return raeume;
  }
}

void setup()
{
    Hochschule eth = new Hochschule();
    println(eth.getAnzahlRaeume());       // Ergibt 100
}
```

Ganz typisch für die Modellierung von Klassen-hierarchieren; die *Ober*-Klasse weiss gar nichts von der *Unter*-Klasse.

6.8 Super

Man kann durch das Schlüsselwort `super` explizit Eigenschaften oder Methoden der Mutter-Klasse aufrufen, wenn Diese in der Kind-Klasse den z.B. den gleichen Namen haben. Schauen wir uns dieses Beispiel dazu an. Hier werden einmal die Eigenschaften der Kind-Klasse aufgerufen und danach die geerbten Eigenschaften der Eltern-Klasse.

Code:

```
class Gebaeude
{
  int raeume = 100;

  void getAnzahlRaeume()
  {
    println("Mutterklasse: "+raeume );
  }
}

class Hochschule extends Gebaeude
{
  int raeume = 200;

  void getAnzahlRaeume()
  {
    println( "Kindklasse: "+raeume );
  }

  void getAnzahlSuperRaeume()
  {
    super.getAnzahlRaeume();     // Methode der Mutterklasse auf
    print( super.raeume );       // Instanzvariable von Mutterklasse
  }
}

void setup()
{
    Hochschule eth = new Hochschule();
    eth.getAnzahlRaeume();       // 200
    eth.getAnzahlSuperRaeume(); // 100 und 100 der Mutter Klasse
}
```

Zu Information:
Damit man auf Instanzvariablen oder Methoden der Mutterklasse zugreifen kann, muss die Mutterklasse ein *super()* im Konstruktor haben.
Processing fügt dies jeder Klasse automatisch hinzu. Somit man kann getrost darauf verzichten. Sonst sähe es so aus:

```
Konstruktor()
{
       super();
}
```

6.9 OOP Dokumentation

Wir haben einiges an Grundsätzlichem über OOP gelernt. In diesem Abschnitt werde ich versuchen noch ein paar nützliche praxisnahe Tips zu geben.

Definiere alle Variablen und Methoden in einer Klasse, die logisch zusammen gehören. Das verhindert Kappselungsprobleme.

Speicher jede Klasse in einem eigenen File. In grösseren Programmen hilft es bei der Organisation deines Sktipts. Ausserdem können sie so in verschiedenen Projekten genutzt werden. Verwende beim Abspeichern den gleichen Namen wie den der Klasse. Das dient der besseren Übersicht.

Dokumentiere immer jede Klasse, alle Methoden und Instanzvariablen der Klasse. Beschreibe in dieser Dokumentation alle globale und lokale Variablen. Das kann wie folgt aussehen.

Für die Ober-Klasse:

```
/*
 * Deine Klasse Klasse
 *   Version: 1.0.0
 *   Desc: Beschreibung der Klasse kommt hier rein.
 *
 * Konstruktor Parameter:
 *   param1          -Kurze Beschreibung
 *   param2          -Kurze Beschreibung
 *
 * Methoden:
 *   deineMethode( )    -Kurze Beschreibung
 *
 * Statische Eigenschaften
 *   staticProp1       -Kurze Beschreibung
 *   staticProp2       -Kurze Beschreibung
 *
 * Statische Methoden
 *   staticMeth( )      -Kurze Beschreibung
 */

class Name
{
  /*
   * Class Constructor
   */
  Name()
  {

  }

/*
 * Methode: deineKlasse.deineMethode( )
 *    Desc: Kurze Beschreibung.
 *
 * Params:
 *   param1          -Kurze Beschreibung
 */
  void methode()
  {

  }

}
```

Für die Unter-Klasse:

```
/*
 *   Deine Sub -Klasse  extends Ober-Klasse
 *   Version: 1.0.0
 *   Desc: Beschreibung der Sub-Klasse kommt hier rein.
 *
 * Konstruktor Parameter:
 *   param1            -Kurze Beschreibung
 *   param2            -Kurze Beschreibung
 *
 * Methoden:
 *   deineMethode( )   -Kurze Beschreibung
 *
 * Statische Eigenschaften
 *   staticProp1       -Kurze Beschreibung
 *   staticProp2       -Kurze Beschreibung
 *
 * Statische Methoden
 *   staticMeth1( )    -Kurze Beschreibung
 *   staticMeth2( )    -Kurze Beschreibung
 */

class Name
{
  /*
   * Class Constructor
   */
  Name()
  {

  }

/*
 * Methode: deineSubKlasse.deineMethode( )
 *    Desc: Kurze Beschreibung.
 *
 * Params:
 *   param1            -Kurze Beschreibung
 */
  void methode()
  {

  }

}
```