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()
  {

  }

}
```