Inhalt

1. Einführung in Processing

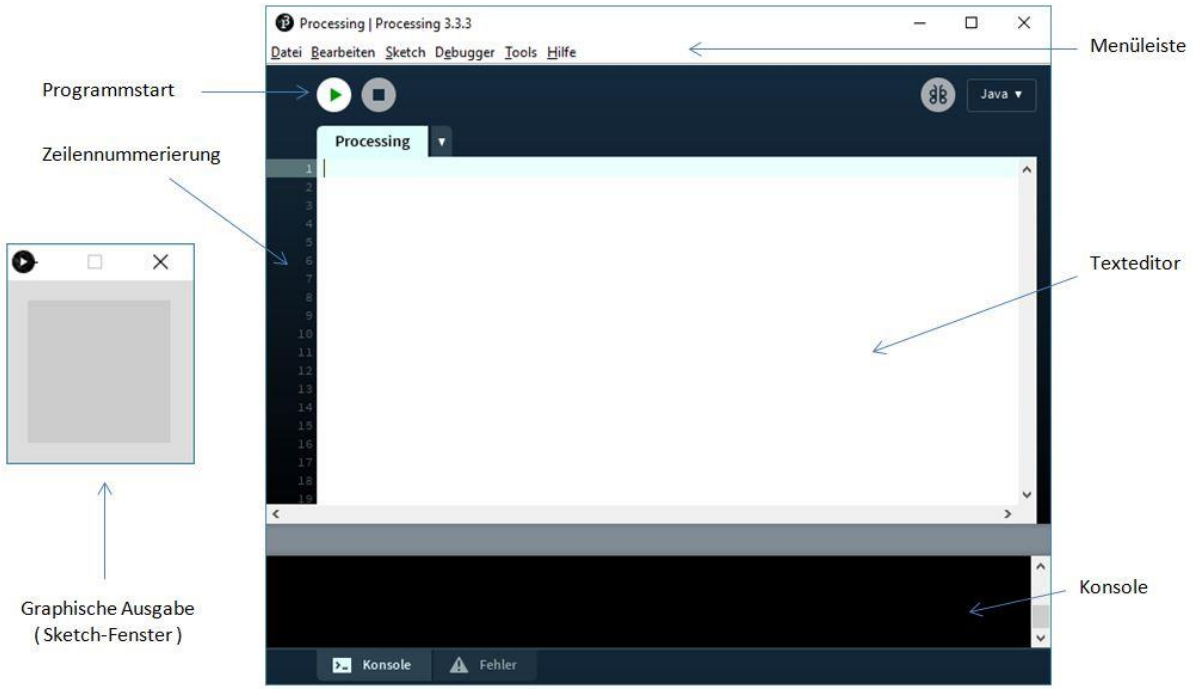
1.1. Processing

Processing basiert auf der bekannten Programmiersprache Java und ist eine Programmiersprache, mit der sehr schnell grafische Formen, Animationen und Interaktionen erstellt werden können.

1.1.1. Processing Entwicklungsumgebung

Klassische Künstlerinnen und Künstler arbeiten in ihrem Atelier und verwenden Pinsel und Farben als Werkzeuge, um Gemälde zu erschaffen. In der Programmierung ist der Computer dieses Atelier und das wichtigste Werkzeug beim Erstellen von Programmen ist die **Entwicklungsumgebung**.

Die folgende Abbildung zeigt die Hauptansicht der Processing IDE:



Im Texteditor wird der Programmcode geschrieben und über die Play-/Stop-Buttons ausgeführt bzw. gestoppt. In der Konsole werden bei Ausführung des Programms die Textausgaben sowie bestimmte Fehlermeldungen angezeigt. Wird ein Programm gestartet, öffnet sich ein Sketch-Fenster, in welchem die grafische Ausgabe in Processing erfolgt.
Der Debugger dient zur systematischen und erleichterten Fehlersuche und wird über den Button rechts oben gestartet.

1.1.2. Syntax und Semantik

Im Kontext des Programmierens verstehen Computer nur bestimmte Programmiersprachen. Das bedeutet, dass beim Programmieren ***Regeln*** (**= *Syntax***) der Programmiersprache exakt eingehalten werden muss.

Die **Syntax** ist vergleichbar mit der Grammatik in einer natürlichen Sprache. Wird die Syntax nicht eingehalten, wird in Processing ein Fehler gemeldet. Im Idealfall wird der Fehler auf der Konsole ausgegeben und im Texteditor rot unterstrichen. Es gibt Fälle, in denen ein Fehler andere Fehler verursacht, sodass der ursprüngliche zuerst gefunden werden muss.

Die **Semantik** meint die Bedeutung des Programmcodes. Liegt ein Semantikfehler vor, kann ein Programm zwar syntaktisch korrekt sein, aber sich anders verhalten, als erwünscht. Ein Beispiel für einen Semantikfehler wäre, z.B.: $a + b * c$ statt $(a + b) * c$ Da liegt kein Syntaxfehler vor, denn das Programm kann problemlos ausgeführt werden. Das Ergebnis des Programms ist jedoch anders als "erwartet".

Oder als Vergleich mit der deutschen Sprache: Der Satz "Zwei Joghurts sitzen im Keller und stricken Benzin" ist rein grammatikalisch korrekt, ergibt aber inhaltlich keinen Sinn.

1.1.3. Kommentare

In Processing gibt es zwei Möglichkeiten Kommentare einzuführen.

- Einfache Kommentare sind einzeilig und beginnen mit `//`.
  Beispiel: `//Kommentartext`
- Blockkommentare können einzeilig oder mehrzeilig sein. Sie werden mit einem`/*` eingeleitet und mit `*/` geschlossen.
  Beispiel: `/* Kommentartext (mehrere Zeilen können dazwischen sein) */`

Korrekt erkannte **Kommentare** werden in Processing in ***grauer Schriftfarbe*** dargestellt.

1.1.4. Programmaufbau und Programmfluss

Alle Programme in Processing bestehen aus den zwei Teilen `setup()` und `draw()`. Ihr Programm sollte also wie folgt aufgebaut sein:

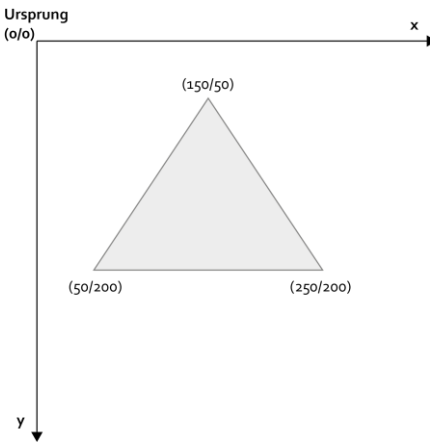
```
void setup() {

  //hier können Voreinstellungen gemacht werden

}

void draw() {

  // hier kommt der Code für das Zeichnen

}
```

Das Programm wird innerhalb dieser zwei Bereichen immer von ***oben nach unten*** abgearbeitet. Die Reihenfolge der Befehle ist daher wichtig.

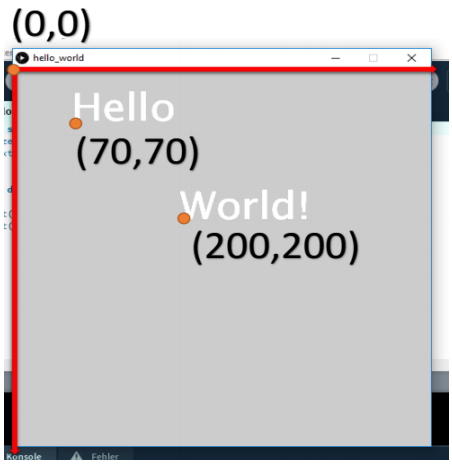
- `void setup() {...}` In diesem Bereich werden alle Voreinstellungen gemacht, z.B. Festlegen der Fenstergröße. Dieser Bereich wird nur **EINMAL ausgeführt**. Das Zeichnen auf das Sketch-Fenster ist hier auch möglich.
- `void draw() {...}` wird im Anschluss ausgeführt und beinhaltet alle Code-Teile, die auf das Sketch-Fenster gezeichnet werden sollen. Dieser Bereich wird solange ausgeführt, bis das Programm beendet wird.

1.1.5. Koordinatensystem

Der Ursprung des Koordinatensystems in Processing liegt in der linken oberen Ecke. Die x-Achse bleibt, wie im Mathematik-Unterricht, erhalten. Die positive y-Achse hingegen verläuft nach unten.



Koordinatensystem in Processing



Koordinatensystem Visualisierung im Sketch-Fenster

1.2. Grundformen

Das Zeichnen am Bildschirm ist vergleichbar mit dem Zeichnen am Papier. Das Sketch-Fenster ist die Zeichenfläche, auf welcher die grafischen Ausgaben gemacht werden. Das Sketch-Fenster ist in ein Pixelraster unterteilt. Breite und Höhe werden in Pixel angegeben, z.B. Punkt $(50|200)$ hat die $Breite = 50\,Pixel$ und $Höhe = 200\,Pixel$.

1.2.1. Befehlsaufruf

Einfache Befehle können in Processing wie im nachfolgenden Beispiel aussehen:

```
// Fenstergröße: Breite= 100 Pixel; Höhe= 150 Pixel
   size(100, 150);
```

1. Ein Befehl beginnt immer mit dem Befehlsnamen, hier: `size`.
2. Dann folgen runde Klammern. In diesen runden Klammern eingeschlossen stehen sogenannte **Argumente** oder **Parameter**. Parameter sind notwendige Informationen für den Befehl. Die Parameter hier geben die Fensterbreite 100px und Fensterhöhe 150px an.
3. Der Befehl wird mit einem Strichpunkt (;) beendet.

Da man sich nicht alle Befehle merken kann, findet ihr eine Befehlsliste unter dem Link https://processing.org/reference/.

Nun haben wir die Größe des Fensters festgelegt. Im Texteditor sieht es dann folgendermaßen aus:

```
void setup() {

// Fenstergröße: Breite= 100 Pixel; Höhe= 150 Pixel
   size(100, 150);

}
```

1.2.2. Geometrische Grundformen in Processing

Nun werden die häufigsten geometrischen Grundformen Rechteck, Viereck, Dreieck, Ellipse (inkl. Kreis) und Bogen beschrieben.

Rechteck:

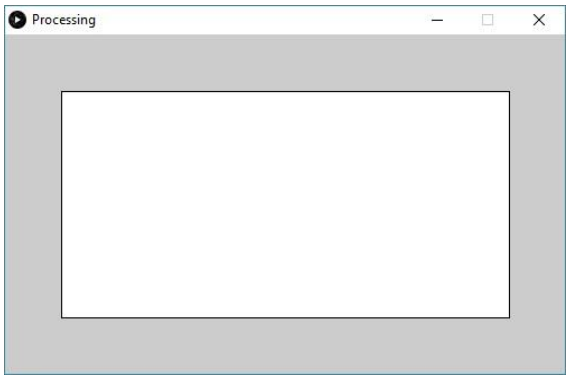
Ein Rechteck wird mit dem Befehl `rect()` gezeichnet. Dieser benötigt als Parameter die x- und y-Koordinate der Startposition sowie die Breite und Höhe des zu zeichnenden Rechtecks.

Struktur von rect():

```
// Rechteck: linker oberer Eckpunkt(x,y); Breite= b Pixel; Höhe= h Pixel
   rect(x, y, b, h);
```

Beispiel:

Der Beispielcode zeichnet ein Rechteck mit dem linken oberen Eckpunkt bei $(50, 50)$, $400px\ Breite\ und\ 200px\ Höhe$.




```
void setup() {
  size(500, 300);
}

void draw() {
  rect(50, 50, 400, 200);
}
```

Hinweis:
Wenn die Position oder Größe des geometrischen Objekts so gewählt wurden, dass sie sich (teilweise) außerhalb der Leinwand befinden, wird nur der sichtbare Bereich gezeichnet.

Viereck:

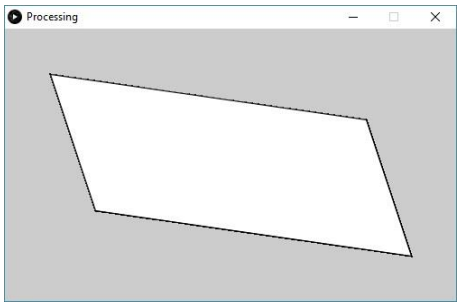
Ein Viereck hat vier Eckpunkte und wird mit dem Befehlsnamen `quad()` gezeichnet. `quad()` hat als Parameter die Koordinaten jedes einzelnen Eckpunktes. (***Achtung:*** *Entweder werden die Punkte im oder gegen den Uhrzeigersinn angegeben. Die Reihenfolge muss erhalten bleiben.)*

Struktur von quad():

```
// Vieleck: Punkt1(x1,y1); Punkt2(x2,y2); Punkt3(x3,y3); Punkt4(x4,y4)
   quad(x1, y1, x2, y2, x3, y3, x4, y4);
```

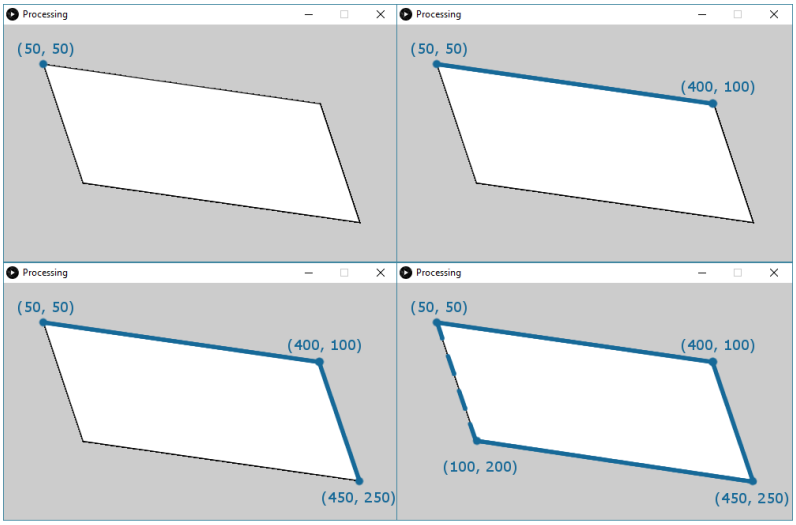
Beispiel:

Dieses Beispiel zeichnet ein Viereck mit den Eckpunkten $(50, 50)$, $(400, 100)$, $(450, 250)$ sowie $(100, 200)$.




```
void setup() {
  size(500, 300);
}

void draw() {
  quad(50, 50, 400, 100, 450, 250, 100, 200);
}
```

Vorgangsweise von Processing beim Zeichnen von `quad()`



Dreieck:

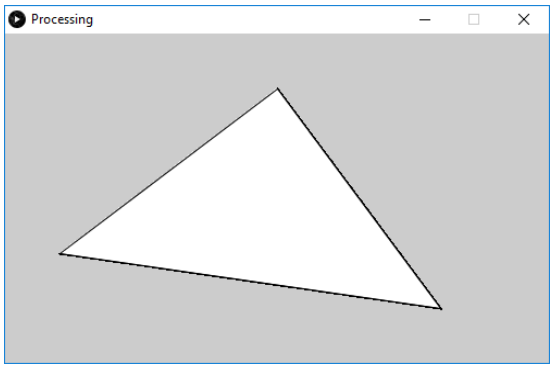
Ein Dreieck wird über seine drei Eckpunkte definiert und mit dem Befehl `triangle()` gezeichnet.
Jeder Eckpunkt wird mit seinen Koordinaten einzeln angegeben. Die Reihenfolge der Punkte kann im Uhrzeigersinn oder gegen den Uhrzeigersinn verlaufen.

Struktur von triangle():

```
// Dreieck: Punkt1(x1,y1); Punkt2(x2,y2); Punkt3(x3,y3)
   triangle(x1, y1, x2, y2, x3, y3);
```

Beispiel:

Dieses Beispiel zeichnet ein Dreieck mit den Eckpunkten (250, 50), (50, 200) und (400, 250).




```
void setup() {
  size(500, 300);
}

void draw() {
  triangle(250, 50, 50, 200, 400, 250);
}
```

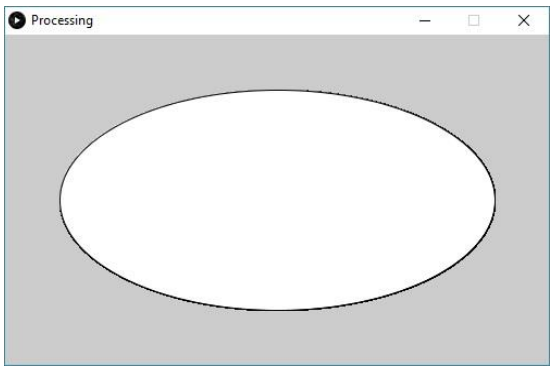
Ellipse:

Eine Ellipse wird mit den Befehlsnamen `ellipse()` gezeichnet. Die Parameter der Ellipse sind deren Mittelpunkt (x- sowie y-Koordinate) sowie deren Breite und Höhe.

Struktur von ellipse():

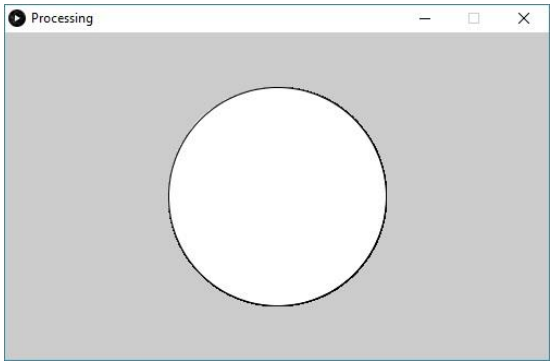
```
// Ellipse: Mittelpunkt(x1,y1); Breite= b Pixel; Höhe= h Pixel
   ellipse(x, y, b, h);
```

Beispiel: Dieses Beispiel zeichnet eine Ellipse mit Breite 400 und Höhe 200 in die Mitte der Leinwand.




```
void setup() {
  size(500, 300);
}

void draw() {
  ellipse(250, 150, 400, 200);
}
```

Das nachfolgende Beispiel zeichnet anstelle der Ellipse einen Kreis mit einem Durchmesser von 200px.




```
void setup() {
  size(500, 300);
}
```

```
void draw() {
  ellipse(250, 150, 200, 200);
}
```

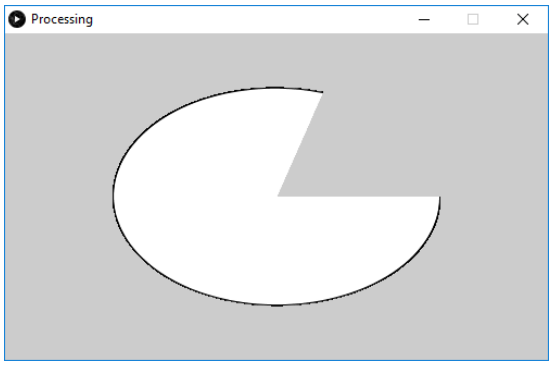
Bogen:

Ein Bogen wird mit dem Befehl `arc()` gezeichnet. Der Bogen basiert auf einer Ellipse. Parameter sind der $Mittelpunkt\ des\ Bogens\ (x, y)$, die $Breite\ (b)$ und $Höhe\ (h)$ sowie jene zwei Bogenwinkel (in Radiant), an denen der Bogen beginnt (alpha) bzw. endet (beta).
(Anmerkung: Die Winkel sind **NICHT** in Grad angegeben. Die Winkel in Radiant gehen von 0 bis 2π. 2π entspricht $360\ Grad$. In Processing kann man eine sogenannte Konstante **PI** für den Winkel von $180\ Grad$ verwenden.)

Struktur von arc():

```
/* Bogen: Mittelpunkt(x,y); Breite= b Pixel; Höhe= h Pixel;
   Anfangsradiant= alpha; Endradiant= beta */
   arc(x, y, b, h, alpha, beta);
```

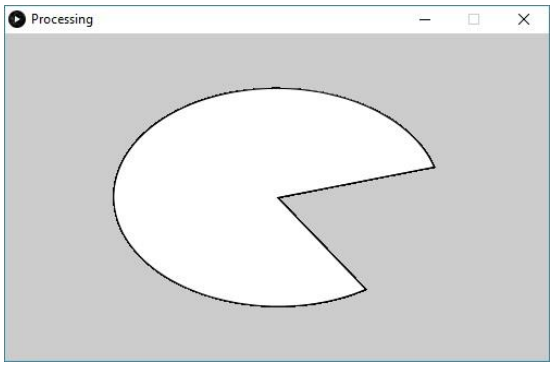
Beispiel: Bogen




```
void setup() {
  size(500, 300);
}

void draw() {
  arc(250, 150, 300, 200, 0, 5);
}
```

(***Hinweis*:** *Der Winkel wird nicht wie möglicherweise gewohnt von oben im Kreis gemessen sondern von rechts anfangend gemessen*.)

Beispiel: Kreissektor




```
  void setup() {
    size(500, 300);
  }

  void draw() {
    arc(250, 150, 300, 200, 1, 6, PIE);
}
```

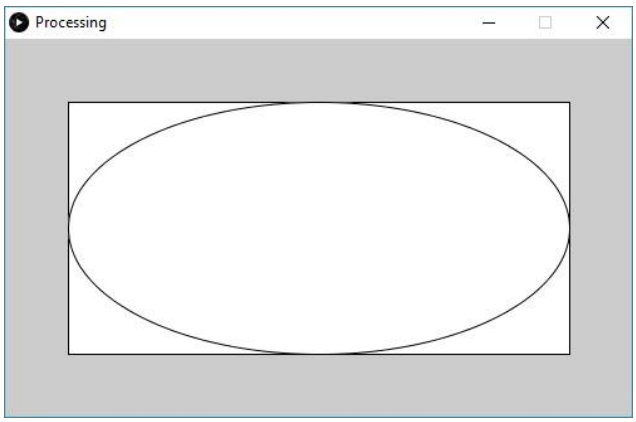
Für den Befehl `arc()` existieren mehrere Modi. Der entsprechende Modus wird optional als zusätzlicher Parameter am Ende angegeben.
Modus:

- PIE: Kreissektor (d. h. Bogen mit den zwei Radien zum Mittelpunkt)
- CHORD: Kreissegment
- OPEN: Kreissegment ohne Kreissehne

1.2.3. Ausführungsreihenfolge von Anweisungen

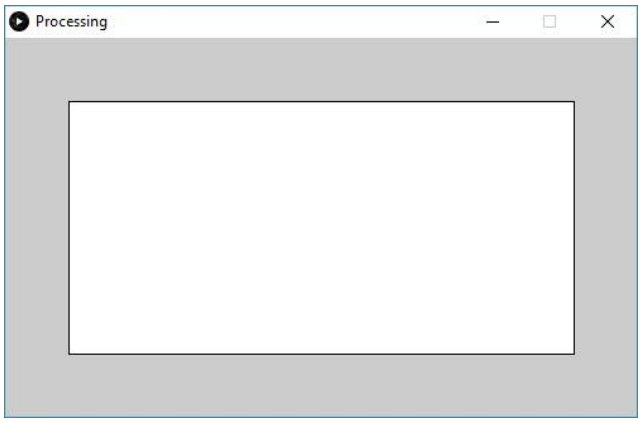
Anweisungen in Programmen werden Zeile für Zeile von oben nach unten ausgeführt. Beim Zeichnen von geometrischen Formen spielt das eine wichtige Rolle, denn wie beim Zeichnen auf einer Leinwand können bereits gezeichnete Formen von anderen überdeckt werden.

Das nachfolgende Beispiel zeigt eine Ellipse in einem Quadrat.




```
void setup() {
  size(500, 300);
}

void draw() {
  rect(50, 50, 400, 200);
  ellipse(250, 150, 400, 200);
}
```

Wird die Reihenfolge der Anweisungen für die Ellipse und das Rechteck umgedreht, dann ist nur das Rechteck sichtbar. Die Ellipse wird vom Rechteck verdeckt.




```
void setup() {
  size(500, 300);
}

void draw() {
  ellipse(250, 150, 400, 200);
  rect(50, 50, 400, 200);
}
```

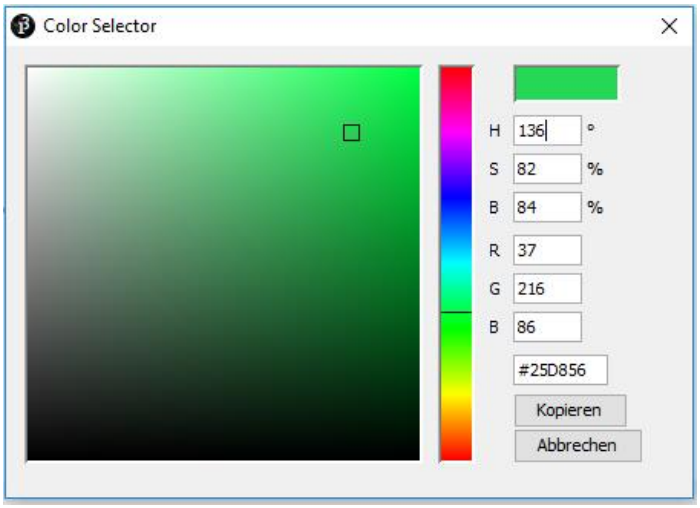
1.3. Farben & Graustufen und Konturen

In Processing kann in Farben oder Graustufen gezeichnet werden. In Processing wird dazu `fill()` verwendet.

- Graustufen werden mit einem Wert zwischen $0\ (Schwarz)\ und\ 255\ (Weiß)$ angegeben. Je kleiner der Wert desto dunkler ist also die Farbe. Eine Hexadezimaldarstellung ist auch möglich.

- Mit drei Parametern kann in Farbe gezeichnet werden, wobei die drei Werte für die Farbkomponenten Rot, Grün und Blau (in dieser Reihenfolge) stehen. Jeder Wert von $0\ bis\ 255$ ist zulässig und gibt die Intensität der entsprechenden Farbe an. Das bedeutet, je kleiner der Wert, desto dunkler die Farbkomponente. Werte $kleiner\ als\ 0$ werden mit 0 angenommen und Werte $größer\ als\ 255$ mit 255.

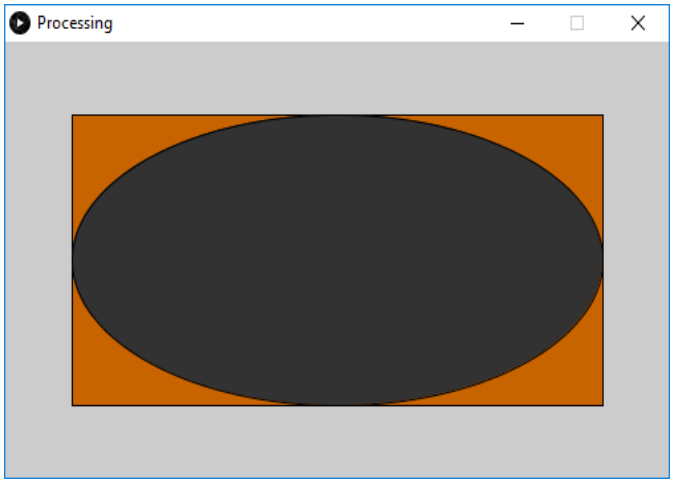
Um einfach den richtigen Farbmix für eine gewünschte Farbe zu finden, wählt man in der Menüleiste unter *Tool/Farbauswahl* den *Color Selector*. Mit ihm können Sie leicht die RGB- und die Hexadezimal-Werte für Ihre Wunschfarbe auslesen.



Durch unterschiedliche Intensitäten der Farbkomponente können unterschiedliche Farbmischungen entstehen. Beispielsweise entsteht die Farbe Gelb, wenn Rot und Grün gemischt wird. Hier sind ein paar Farbmischungen, die du austesten kannst

```
fill(255, 255, 0);                  // grelles Gelb
fill(200, 200, 0);                  // dunkleres Gelb
fill(0, 0, 255);                    // intensives Blau
fill(100, 100, 255);                // helleres Blau
fill(0, 255, 255);                  // Türkis
fill(255, 0, 255);                  // Magenta
```

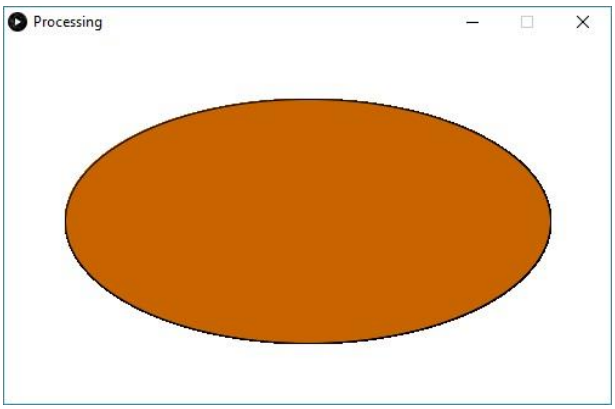
Beispiel: Oranges Rechteck und die dunkelgraue Ellipse




```
void setup () {
  size(500, 300);
}

void draw() {
  // Füllfarbe: orange
  fill(200, 100, 0);
  rect(50, 50, 400, 200);
  // Füllfarbe: grau
  fill(50);
  ellipse(250, 150, 400, 200);
}
```

Wichtig ist, dass sobald eine Farbe gesetzt wurde, diese solange verwendet wird, bis eine andere gesetzt wird. Hält man sich nicht daran, führt es zu Programmverhalten, das wir im Folgekapitel genauer behandeln.

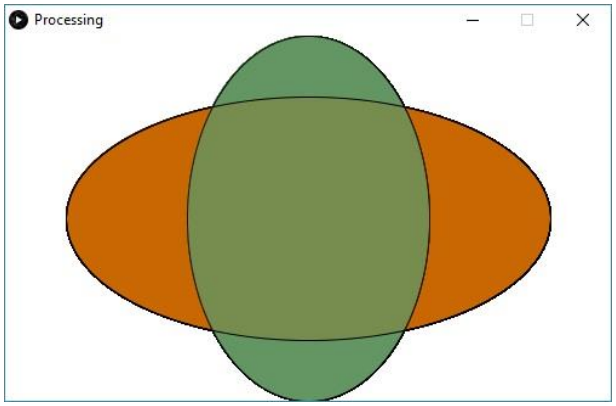
Der Hintergrund des Sketch-Fensters, also die Leinwandfarbe, kann mit `background()` eingefärbt werden. Wie das folgende Beispiel zeigt, lassen sich hier auch wieder Farben und Graustufen verwenden.




```
void setup() {
  background(255);
  size(500, 300);
}

void draw() {
  fill(200, 100, 0);
  ellipse(250, 150, 400, 200);
}
```

Die beiden Möglichkeiten zur Farbwahl können in einem Programm beliebig verwendet werden.

Soll eine Form bzw. ein geometrisches Objekt nicht gefüllt werden, sondern transparent sein, wird `noFill()` verwendet. Auch diese Anweisung muss vor der entsprechenden geometrischen Form gesetzt werden. Zusätzlich zum Farbwert kann optional bei `fill()` als weiterer Parameter ein Alpha-Wert für die Deckkraft (opacity) angegeben werden.



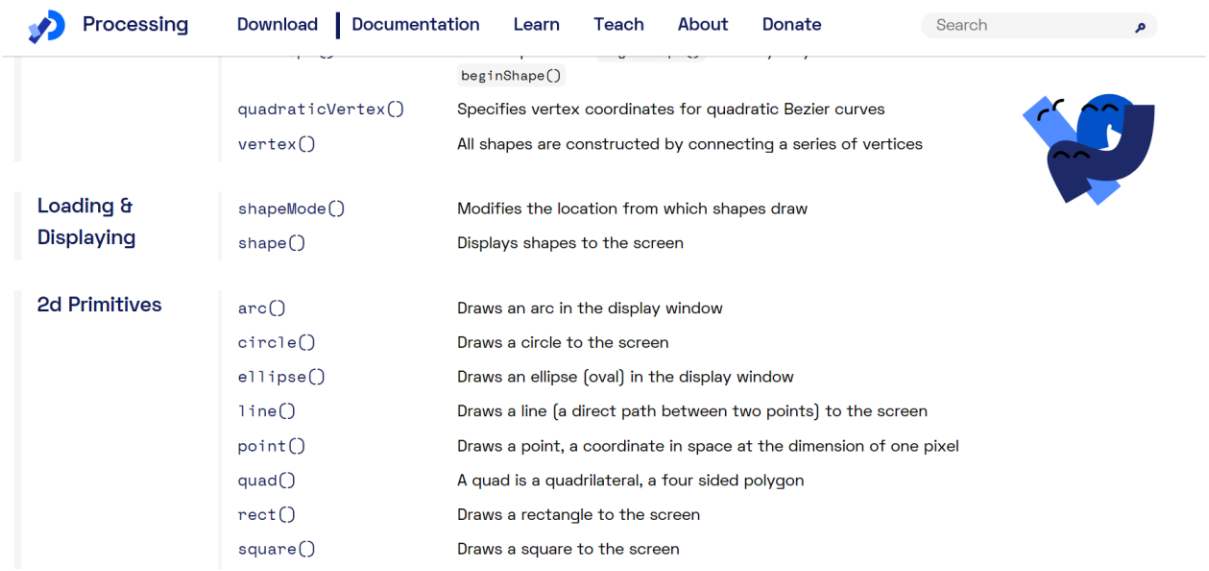

```
void setup() {
  background(255);
  size(500, 300);
}

void draw() {
  fill(100, 150, 100);
  ellipse(250, 150, 200, 300);
  fill(200, 100, 0, 50);
  ellipse(250, 150, 400, 200);
}
```

Der obige Code zeichnet eine leicht transparente Ellipse. Um die Transparenz zu verdeutlichen wird eine zweite Ellipse mit höherer Deckkraft gezeichnet, sodass die halbtransparente Ellipse diese halb überdeckt.

2. Application Programming Interface (API)

Aufgrund der Fülle an verschiedenen Befehlsnamen, kann der Überblick über die zur Verfügung stehenden Befehle bzw. deren Wirkung, Anwendung und genaue Funktionsweise verloren gehen.
Ein nützliches Hilfsmittel ist hier die Referenz (auch Dokumentation) bzw. API (kurz für Application Programming Interface) einer Programmiersprache. Sie ist eine Sammlung an Dokumentationen für die vorhandenen Befehlsnamen bzw. Anweisungen und dient Programmiererinnen und Programmierern als Lexikon bzw. Nachschlagewerk.

Die Processing API ist unter https://processing.org/reference/ zu finden. Hier werden alle Befehle dokumentiert, ihre Syntax, Parameter und Funktionsweise beschrieben und deren Verwendung anhand von Beispielen visualisiert. Alle bekannten Befehle sind in der Kategorie *Shape → 2D Primitives* zu finden.



Quelle: Processing Referenz (https://www.processing.org/reference)

Zu jedem Befehlsnamen gibt es zuerst eine Kurzbeschreibung. Beim Klicken auf den Befehl eine ausführliche Erklärung mit Beispiel (*Examples*), welches dessen Anwendung demonstriert. Es kann direkt in den Texteditor kopiert und angepasst werden.

Name

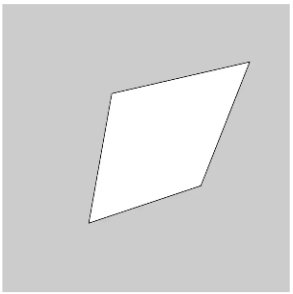
quad()

Description

A quad is a quadrilateral, a four sided polygon. It is similar to a rectangle, but the angles between its edges are not constrained to ninety degrees. The first pair of parameters (x1,y1) sets the first vertex and the subsequent pairs should proceed clockwise or counter-clockwise around the defined shape.

Examples

```
size(400, 400);
quad(152, 124, 344, 80, 276, 252, 120, 304);
```



Syntax

```
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```

Parameters

- **x1** (float) x-coordinate of the first corner
- **y1** (float) y-coordinate of the first corner
- **x2** (float) x-coordinate of the second corner
- **y2** (float) y-coordinate of the second corner
- **x3** (float) x-coordinate of the third corner
- **y3** (float) y-coordinate of the third corner
- **x4** (float) x-coordinate of the fourth corner
- **y4** (float) y-coordinate of the fourth corner

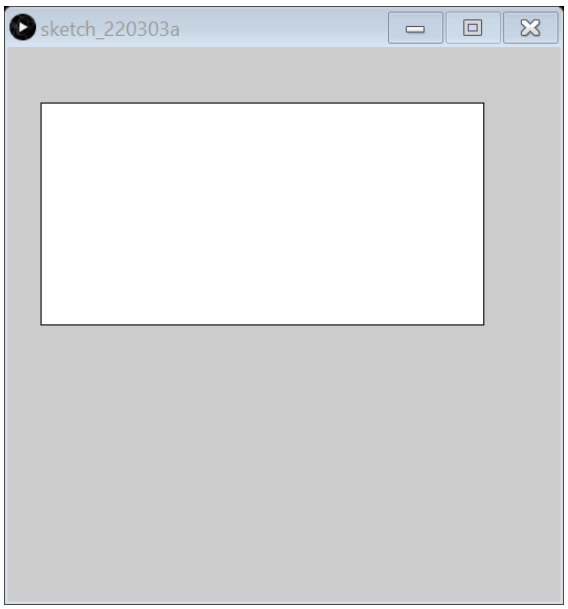
Quelle: Processing Referenz (https://www.processing.org/reference/quad_.html)

3. Variablen und Verzweigungen

3.1. Motivation für Variablen

Wir haben immer konkrete Werte als Parameter verwendet:

Ein Rechteck mit $(\mathbf{30}, \mathbf{50})$ $als\ linker\ oberer\ Eckpunkt$, einer $Breite\ von\ \mathbf{400}\ px$ und $Höhe\ von\ \mathbf{200}\ px$ bei einer Fenstergröße von 500×500 wurde wie folgt geschrieben:




```
void setup() {
  size(500, 500);
}

void draw() {
  rect(30, 50, 400, 200);
}
```

Was jedoch, wenn man später die Fenstergröße ändern, aber das Rechteck seine Position unabhängig der Fenstergröße beibehalten möchte? Man müsste in der Code-Zeile stets die Werte händisch neu eintragen und das ist sehr umständlich.

So müsste man bei einer Fenstergröße `size(1000, 1000)` folgendes schreiben:

```
rect(60, 100, 800, 400);
```

Wenn wir also die Fenstergröße ändern, müssen wir im schlimmsten Fall alle Parameter ändern. Dieses Beispiel zeigt, dass Werte voneinander abhängig sein können. Dazu wäre es hilfreich, wenn der Computer uns die Werte automatisch berechnet. In diesem Beispiel fällt auf das bei einer Verdoppelung der Fenstergröße auch die Parameterwerte verdoppelt werden müssen. Platzhalter oder sogenannte Variablen wären wichtig.

3.2. Variablen

Eine Variable beinhaltet stets drei Informationen: (1) ***die Art von Inhalt***, die sie speichern kann, (2) einen ***Wert bzw***. den ***Inhalt*** selbst und (3) einen ***eindeutigen Namen***.
Variablen kann man sich vorstellen, wie Gefäße. Diese Gefäße können verschiedene Formen und Größen haben, die anzeigen, welche Art und Menge von Inhalten sie aufnehmen können.

Zum Beispiel ist das Bierglas (in den meisten Fällen) zum Befüllen mit Bier gedacht. Biergläser gibt es auch in unterschiedlichen Größen. In einem Pfefferstreuer wird üblicherweise gemahlener Pfeffer aufbewahrt.

Bei Variablen bestimmt der **Datentyp** die Art des erlaubten Inhalts. Das können beispielsweise ganze Zahlen, Dezimalzahlen, Zeichen, Wahrheitswerte oder Zeichenketten sein. Jeder Datentyp hat auch einen eigenen Wertebereich, vergleichbar mit der Größe eines Gefäßes. Dieser beschreibt die zulässigen Werte, die eine Variable von einem bestimmten Datentyp aufnehmen kann.

Die Größe von Gefäßen ist fix, aber man kann sie leer lassen, vollfüllen oder beispielsweise zur Hälfte füllen. Ein Gefäß hält zu einem bestimmten Zeitpunkt eine bestimmte Menge an Inhalt. Bei Variablen ist dies der Wert der Variablen. Wichtig ist, dass eine Variable immer nur **einen** diskreten Wert aus dem entsprechenden Wertebereich beinhalten kann.

Zu guter Letzt können Gefäße mit einer Beschriftung versehen werden, um zu erkennen, wofür bzw. für welchen Inhalt das Gefäß gedacht ist. Bei Variablen ist dies der Variablenname und dieser muss vergeben werden. Damit der Zweck der Variablen auf den ersten Blick erkennbar ist, sind aussagekräftige Variablennamen zu verwenden.

Variablennamen, vgl. Beschriftung der Gefäße

Dabei muss der Name jeder Variablen eindeutig sein. *Eindeutig* bedeutet, dass verschiedene Variablen auch verschiedene Variablennamen haben müssen. Den Variablennamen kann man bei der Erstellung der Variablen selbst festlegen. Dabei sind Gesetzmäßigkeiten zu beachten.

3.3. Vordefinierte Variablen

In Processing haben wir den Vorteil, dass wir manchmal nicht eigene Variablen erstellen müssen. Oft werden sogenannte vordefinierte Variablen zur Verfügung gestellt, dessen Wert aber nicht geändert werden kann. Eine Liste der wichtigsten vordefinierten Variablen:

vordefinierte Variablen	Beschreibung
`height`	Höhe des Fensters
`width`	Breite des Fensters
`mouseX`	x-Position des Mauszeigers
`mouseY`	y-Position des Mauszeigers
`key`	Wert einer Taste auf der Tastatur
`keyPressed`	liefert den Wert `true` bei Tastendruck

Beispiel: Erhaltung der Proportionen eines Rechtecks bei Änderung der Fenstergröße von `size(500, 500)`

```
rect(30 * height/500, 50 * width/500 , 400 * height/500, 200 * width/500);
```

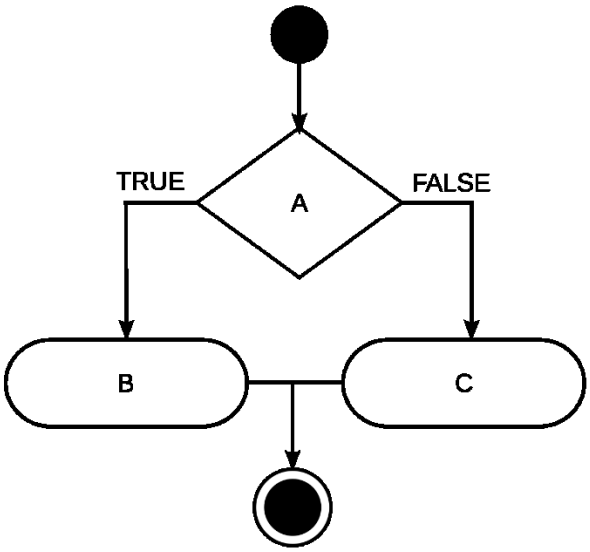
An dieser Stelle sind die Parameter von der Breite und Höhe der Fenstergröße abhängig, sodass keine händischen Änderungen mehr vorgenommen werden müssen.

3.4. Verzweigungen

3.4.1. Bedingung

Entscheidungen spielen in der Informatik eine wichtige Rolle, im konkreten Fall etwa bei selbstfahrenden Autos. Wenn keine Personen am Übergang stehen, soll weitergefahren werden, sonst angehalten. Solche Entscheidungen werden in Programmen "***Verzweigungen***" genannt, da sie alternative Programmabläufe bieten, ähnlich wie bei einer Weggabelung.

Mit einer Verzweigung hat man also die Möglichkeit den Ablauf eines Programms zu steuern. Man muss dafür einen Ausdruck angeben. Dieser kann richtig oder falsch sein. Ein möglicher Verlauf einer Verzweigung, könnte so aussehen:



Wikipedia: Verzweigung

Hier läuft das Programm normal ab, bis eine sogenannte Bedingung gestellt wird, z.B. $A = \text{„}Sind\ Leute\ am\ Übergang?\text{“}$. Wenn ja, dann soll $B = \text{„}angehalten\ werden\text{“}$. Wenn nein, dann soll $C = \text{„}weitergefahren\ werden\text{“}$.

Eine **Bedingung ist** also **eine Abfrage** oder auch eine Aussage, anhand derer eine Entscheidung getroffen wird: Trifft die Bedingung zu (die Frage wird mit "ja" beantwortet oder die Aussage ist wahr), dann werden bestimmte Aktionen ausgeführt. Trifft die Bedingung jedoch nicht zu, dann wird entweder nichts unternommen oder alternative Aktionen werden ausgeführt.

Hinweis: Bedingungen können trotz unterschiedlich formuliert werden, aber das Gleiche bedeuten. Zum Beispiel kann die Bedingung: "*Wenn Alice maximal so alt ist*

wie Bob" auch umformuliert werden zu "*Wenn Alice nicht älter ist als Bob*". Beide Bedingungen haben eine andere Formulierung, aber die gleiche Bedeutung.

3.4.2. Vergleichsoperatoren

Um **Bedingungen** in Processing auszudrücken, verwenden wir Vergleichsoperatoren. Wahrscheinlich kennt ihr die Vergleichsoperatoren bereits aus der Mathematik.

Ihre Schreibweise in Processing ist ein wenig anders und in der folgenden Tabelle zusammengefasst:

Bezeichnung	Vergleichsoperator **Schreibweise**
kleiner	<
kleiner gleich	<=
größer	>
größer gleich	>=
gleich	==
ungleich	!=

Das Ergebnis eines Vergleichs ist in Processing und vielen weiteren Programmiersprachen vom sogenannten Datentyp `boolean` und hat daher entweder den Wert **true** (wahr) oder den Wert **false** (falsch). Wir wissen, dass Datentypen Zahlen, Zeichen, aber auch Wahrheitswerte speichern können. Hier haben wir nur zwei Werte.

3.4.3. if - Anweisungen

Mithilfe von Vergleichsoperatoren ist es möglich Aussagen zu formulieren. Diese Aussagen können nun als Bedingungen verwendet werden, um in einem Computerprogramm optionale Code-Ausführungen zu ermöglichen. Das heißt: Nur falls die Bedingung zutrifft, sollen bestimmte Anweisungen ausgeführt werden.

Dafür gibt es die **`if`-Anweisung**. Mit der `if`-Anweisung können Probleme der Form "Wenn … dann … " dargestellt werden. Beispiel: "Wenn vor dir kein Hindernis ist, dann fahre geradeaus".

Aufbau der `if`-Anweisung:

```
if (Bedingung) {
      // bestimmte Anweisung(en)
}
```

```
if (keine Person am Übergang) {
// fahre weiter
}
```

Aufbau der `if`- `else` - Anweisung:

```
if (Bedingung) {
      // bestimmte Anweisung(en)
}
else {
      // andere Anweisung(en)
}
```

```
if (keine Person am Übergang) {
// fahre weiter
}
else {
      // bleib stehen
}
```

Die `if` - Anweisung wird mit dem Schlüsselwort `if` eingeleitet. In runden Klammern folgt die Bedingung. Danach werden in geschwungene Klammern der optionale Code hingeschrieben. Dieser wird nur ausgeführt, wenn die Bedingung zutrifft, also die Aussage wahr ist. Ist die Bedingung nicht erfüllt, wird der Code in den geschwungenen Klammern nicht ausgeführt.

Möchte man eine Alternative anbieten, steht uns `else` zur Verfügung. Hier gibt es keine Runden klammern. `else` entspricht dem Wort „sonst“.

Beispiel: Sobald mit dem Mauszeiger in x-Position über die Fensterhälfte gegangen wird, soll ein Rechteck gezeichnet werden. Sonst ein Kreis.

```
void setup() {
 size(500, 500);
}

void draw() {
 if (mouseX > width/2) {
   rect(height/2, width/2, 100, 50);
}

else {
   ellipse(height/2, width/2, 50, 50);
}
```

Teste das Programm aus!