

3. Variablen und Operatoren

3.1. Variablen

3.1.1. Motivation für Variablen

Bisher haben wir innerhalb eines Befehlsaufrufes stets konkrete Werte als Parameter übergeben:

Ein Rechteck mit (30, 50) als linker oberer Eckpunkt, einer Breite von 400px und Höhe von 200px wurde wie folgt geschrieben:

```
rect( 30, 50, 400, 200 );
```

Der Pacman Körper mit einem Durchmesser von 200px und Mittelpunkt (300, 300) wurde folgendermaßen gezeichnet:

```
arc( 300, 300, 200, 200, 1, 6 );
```

Was jedoch, wenn man später die Größe des Pacmans weiterverwenden möchte?

Man müsste in der Code-Zeile, welche den Befehl zum Zeichnen des Pacman beinhaltet, die Parameterwerte für die Pacman-Größe ablesen und diese Werte dann in den anderen Befehlsaufrufen, verändert oder unverändert, händisch eintragen.

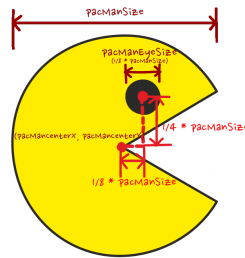
So könnte man z.B. einen zweiten, halb so großen Pacman zeichnen:

```
arc( 300, 300, 100, 100, 1, 6 );
```

Jedoch muss jedes Mal, wenn die Größe des ersten Pacmans geändert wird, auch beim zweiten Pacman die Größe händisch angepasst werden.

Dieses Beispiel zeigt, dass Werte voneinander abhängig sein können: Die Körpergröße eines zweiten Pacmans ist $\frac{1}{2}$ mal die Körpergröße des ersten Pacmans.

Ein anderes Beispiel ist die Abhängigkeit der Position und Größe des Auges von der Position und Größe seines Körpers:



Der Durchmesser des Auges (`pacManEyeSize`) ist $\frac{1}{8}$ mal der Durchmesser des Pacmans (`pacManSize`). Der Mittelpunkt des Auges ist relativ zum Zentrum des Pacman-Körpers um $\frac{1}{8}$ mal `pacManSize` nach rechts und $\frac{1}{4}$ mal `pacManSize` nach oben verschoben. (Hinweis: Grafik ist rein schematisch und nicht maßstabsgetreu)

Wird nun jedoch der Körper vergrößert oder verschoben, soll auch das Auge entsprechend mit vergrößert oder in seiner Position versetzt werden. Wenn für die Parameter zum Zeichnen des Pacman Körpers und zum Zeichnen des Auges konkrete Werte (z.B. ganze Zahlen oder Dezimalzahlen) eingesetzt werden, müssten auch alle betroffenen Parameterwerte beim Verschieben, Vergrößern oder Verkleinern händisch korrigiert werden. Besonders bei komplexeren Grafiken, Animationen bzw. Programmen ist diese Vorgehensweise aufwändig, fehleranfällig und ineffizient, manchmal sogar unmöglich.

Die Lösung dafür liegt in der Verwendung von *Variablen*.

3.1.2. Variablen verwenden

Variablen dienen zum Speichern von Werten. Jede Variable hat einen Namen, mit dem auf den gespeicherten Wert zugegriffen werden kann. Der in einer Variablen gespeicherte Inhalt kann durch bestimmte Programmbefehle abgefragt oder auch geändert werden. Üblicherweise kann in Programmiersprachen für jede Variable bestimmt werden welche Art von Inhalt gespeichert werden.

Eine Variable beinhaltet also drei Informationen:

- die Art von Inhalt, die sie speichern kann
- einen Wert bzw. den Inhalt selbst
- 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, voll füllen 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

Für einen Computer sind Variablen benannte Speicher. Damit eine Variable auffindbar ist, ist so ein Name unbedingt notwendig. Dabei muß der Name jeder Variablen eindeutig sein. *Eindeutig* bedeutet, dass verschiedene Variablen auch verschiedene Variablennamen haben müssen (Ausnahme: siehe 3.4 *Variablen und Operatoren: Sichtbarkeit von Variablen*). Den Variablennamen kann man bei der Erstellung der Variablen selbst festlegen. Dabei sind Gesetzmäßigkeiten zu beachten (siehe 3.2 *Variablen und Operatoren: Variablennamen und deren Regeln*).

Deklaration und Initialisierung von Variablen

So wie man ein Gefäß zunächst bereitstellt und beschriftet, bevor Inhalt darin abgelegt wird, wird eine Variable vor ihrer Verwendung deklariert. **Deklarieren** bedeutet also, dass die Variable vorgestellt wird, sodass der Computer bei der Ausführung weiß, dass eine Variable mit der gewählten Bezeichnung und Art existiert.

Beispiele:

```
int score;  
float jump_mark;
```

Die erste Zeile deklariert eine Variable mit dem Namen `score`, in der ein ganzzahliger Spielstand eines Computerspiels gespeichert werden soll.

In der zweiten Zeile wird eine Variable mit dem Namen `jump_mark` deklariert, welche die Sprungweite eines Athleten oder einer Athletin beim Weitsprung als Dezimalzahl (Kommazahl) beinhalten soll.

Eine Variable wird in Processing allgemein folgendermaßen deklariert:

`datentyp variablenname;`

Im ersten Teil *datentyp* wird dem Computer der Datentyp der Variable (vgl. Form und Größe des Gefäßes) mitgeteilt, das bedeutet, welche Art von Wert eine Variable beinhalten darf. Der Typ einer Variablen kann nach der Deklaration nicht mehr geändert werden.

Der zweite Teil *variablenname* gibt den Variablennamen (vgl. Beschriftung des Gefäßes) an, unter welchem die Variable ansprechbar sein soll. Abgeschlossen wird die Deklaration, wie jede Anweisung in Processing, mit einem `;` (Strichpunkt).



Deklaration von Variablen

Mehrere Variablen vom selben Datentyp können auch in einer Zeile gemeinsam deklariert werden:

```
int score, gameNumber;  
float jump_mark, athlete_weight, athlete_height;
```

Die Datentypen `int` und `float` sowie weitere Datentypen für Variablen sind in Kapitel 4 im Detail beschrieben.

Wird der Variablen das erste Mal ein Wert zugewiesen (vgl. wird das Gefäß zum ersten Mal befüllt), spricht man von der **Initialisierung** der Variablen. Für die Zuweisung eines Wertes an eine Variable verwendet man in Processing den Zuweisungsoperator `=`.

```
score = 0;
jump_mark = 0.0;
```

Achtung: Der Zuweisungsoperator `=` hat eine andere Bedeutung als das Gleichheitszeichen in der Mathematik! Es geht hier um das Ändern des Inhalts der Variablen. Auf der rechten Seite des Gleichheitszeichens steht der Wert, der nun als neuer Inhalt in der Variablen auf der linken Seite gespeichert werden soll.

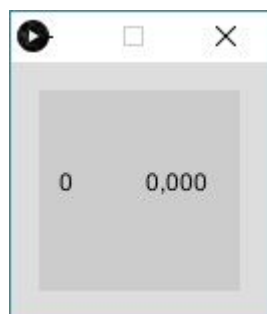


Initialisierte Variablen

Die Deklaration und die Initialisierung können auch in einem einzigen Befehl zusammengefasst werden:

```
int score = 0;
float jump_mark = 0.0;
```

Nachdem die Variable dem System nun bekannt ist, kann die Variable verwendet werden. Anstelle des Wertes kann nun der Variablenname geschrieben werden, z.B. als Parameter in einer Anweisung:

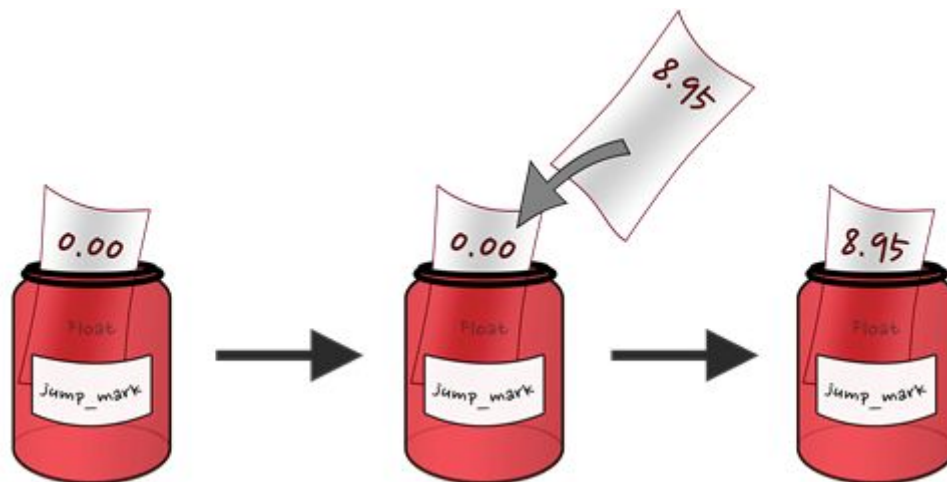


```
void setup() {
  fill( 0 );
  int score = 0;
  float jump_mark = 0.0;
  text( score, 10, 50 );
  text( jump_mark, 50, 50 );
}
```

Einer Variablen kann (nach ihrer Initialisierung) **ein neuer Wert zugewiesen** werden:

```
score = 100;  
jump_mark = 8.95;
```

Der bisher gespeicherte Wert einer Variablen, (hier 0 für score bzw. 0.0 für jump_mark), wird mit dem neuen Wert überschrieben, das heißt, die Variable "verliert" ihren "alten" Wert. Daher wird diese Art der Zuweisung auch *destruktive Zuweisung* genannt.



Zuweisung eines neuen Werts an die Variable jump_mark

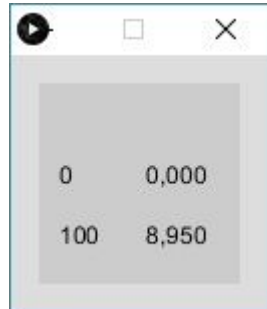
Im obigen Beispiel beinhalten die Variablen nach der Zuweisung folgende Werte:

Variablenname	Wert
score	100
jump_mark	8.95



Werte der Variablen nach der Zuweisung

Die erneute Ausgabe der Variablen in der zweiten Zeile des Sketch-Fensters zeigt, dass die neuen Werte übernommen wurden.



```
void setup() {
  fill( 0 );

  int score = 0;
  float jump_mark = 0.0;
  text(score, 10, 50);
  text(jump_mark, 50, 50);

  score = 100;
  jump_mark = 8.95;
  text(score, 10, 80);
  text(jump_mark, 50, 80);
}
```

Anstelle eines konkreten Wertes kann auch der Wert einer anderen Variablen unter Verwendung deren Variablennamens zugewiesen werden:

```
int highscore = 0;
int score = 0;
score = 10;
highscore = score;
```

Der Wert in der Variablen highscore wird mit dem Wert in der Variablen score überschrieben. Die Variable highscore beinhaltet daher nach der Zuweisung den Wert "10".



```
void setup() {
  fill( 0 );
  int highscore = 0;
  int score = 0;
  score = 10;
  highscore = score;
  text(score, 10, 50);
  text(highscore, 50, 50);
}
```

Wertveränderungen nach Initialisierung verhindern

Will man jedoch festlegen, dass sich der Wert einer Variablen nach der Initialisierung nicht mehr ändern darf, kann dies durch Angabe des Schlüsselworts `final` vor dem Datentyp bei der Deklaration erreicht werden.

```
final int pacmanSize = 200;
final float maxLength = 195.50;
```

3.2. Variablennamen und deren Regeln

Im Speicher eines Computers hat jede Variable eine eindeutige Adresse. Diese Adresse wird in der Regel als Hexadezimalzahl angegeben, zum Beispiel 727c1264. Mit Hilfe dieser Adresse kann der Computer direkt und sehr schnell zum Inhalt der Variablen zugreifen. Diese Speicheradressen sind für uns Menschen jedoch schwer zu merken. Müssten wir als Menschen Variablen über diese Adressbezeichnung direkt ansprechen, verlieren wir schnell den Überblick selbst in unseren eigenen Programmen.

Daher hat jede Variable einen Variablennamen - eine Bezeichnung, die wir Programmiererinnen und Programmierer ihr geben und unter welcher wir die Variable ansprechen können. Um Programme lesbar und wartbar zu halten, verwenden wir daher für Variablen sinnhafte und aussagekräftige Bezeichnungen, welche ihre Bedeutung bzw. Verwendung im Programm widerspiegeln.

Dies ist besonders ratsam, wenn Sie in einem Team entwickeln und damit auch andere Personen Ihren Code leichter lesen, verstehen und auch erweitern können.

Doch nach welchen Kriterien soll man einen Variablennamen vergeben und was genau sind aussagekräftige Bezeichnungen?

Zunächst einmal ist es, mit einer Ausnahme (siehe 3.4 *Variablen und Operatoren: Sichtbarkeit von Variablen*), nicht möglich, mehreren Variablen den gleichen Namen zu geben.

Vergeben Sie immer aussagekräftige und eindeutige Variablennamen.

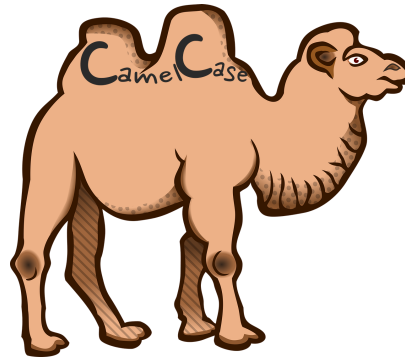
Folgende Regeln **müssen** Sie bei der Wahl der Variablennamen in Processing beachten:

- Variablennamen dürfen nur aus Buchstaben, Ziffern oder Unterstrichen `_` bestehen
- Sie müssen mit einem Buchstaben oder einem Unterstrich anfangen.
- Die Buchstaben `ü`, `ä`, `ö` und `ß` sind nicht erlaubt und führen zu einem Fehler.
- Groß- und Kleinschreibung werden unterschieden. Das bedeutet, dass etwa `radius`, `RADIUS`, `raDiUs` und `Radius` unterschiedliche Variablen sind.
- Sogenannte **“reservierte Wörter”** von Processing dürfen nicht verwenden. Diese haben in Processing eine bestimmte Wirkung. Beispiele für solche reservierten Wörter sind `int`, `float`, `if`, `while`, `for`, `null`, `true`, `false`, etc. Diese werden in Processing in oranger oder grüner Schriftfarbe dargestellt.

Zum guten Programmierstil gehört aber auch, dass folgende Punkte beachtet werden **sollen**:

- Verwenden Sie für Ihre Variablen englische Bezeichnungen
- Variablennamen werden standardmäßig in Kleinbuchstaben geschrieben, z.B. `radius`, `height`, `width`
- Bei Aneinanderreihung von Wörtern werden die Anfangsbuchstaben im Wortinneren groß geschrieben, z.B. `radiusEye`, `diameterPoint`, `colorPointYellow`. Diese Schreibweise nennt man auch **CamelCase**, da die Großbuchstaben wie Höcker

eines Kamels aussehen.



CamelCase Schreibweise für Variablennamen

Eine weitere geläufige Variante ist die Trennung mittels Unterstrich, z.B. `radius_eye`, `diameter_point`, `color_point_yellow`

- Falls sich der Wert der Variablen nach ihrer Initialisierung NIE mehr ändert, wird der gesamte Name üblicherweise in Großbuchstaben geschrieben. z.B. `RADIUS`, `WIDTH`, `COLOR_YELLOW`. Solche Variablen werden **Konstanten** genannt.

Man sieht also, dass es möglich ist, viele beliebige Variablennamen zu vergeben. Aussagekräftige Variablennamen erhöhen jedoch sehr die Lesbarkeit des Codes.

3.3. Operatoren

Mittels Operatoren vermitteln wir dem Computer, wie er mit Informationen rechnen kann bzw. wie er Ergebnisse verarbeiten soll. Wir unterteilen die Operatoren in folgende zwei Kategorien:

- Operationen mit zwei Operanden
- Operationen mit einem Operanden.

3.3.1. Operationen mit zwei Operanden

Operationen mit zwei Operanden benötigen zwei Werte und einen Operator. Folgende Operatoren stehen zur Verfügung:

- Addition (+)
- Subtraktion (-)
- Multiplikation (*)
- Division (/)
- Modulo (%)
- Zuweisung (=)

Die **Zuweisung** weist einer Variablen einen Wert zu und wird in Processing mit dem Zuweisungsoperator = geschrieben:

```
int b;  
b = 10;
```

Sprechweisen: "b bekommt den Wert 10" oder "b ergibt sich zu 10" oder "b wird zu 10", etc., aber bitte niemals: "b ist gleich 10".

Addition, Subtraktion, Multiplikation und Division kennen Sie bereits aus der Mathematik, auch die Notation in Processing ist identisch:

```
int x;  
x = 3 + 5;  
  
int y;  
y = 10 - 3;  
  
int z;  
z = 4 * x;  
  
int a;  
a = z / 16;
```

Bei der Division ist folgendes zu beachten: Werden ganzzahlige Werte für Divisor und Dividend verwendet, handelt es sich um eine ganzzahlige Divisionen. Das bedeutet, Sie erhalten als Ergebnis auch eine ganze Zahl, d.h. nur den ganzzahligen Anteil.

Um den Rest der Division zu erhalten, wird der **Modulo-Operator** verwendet. Die Modulo Operation wird mit dem % Zeichen geschrieben und liefert den Rest der ganzzahligen Division.

```
int c;  
c = 10 % 3;
```

Als Beispiel:

Division	Modulo
7/3 liefert 2	7%3 liefert 1
10/5 liefert 2	10%5 liefert 0
14/4 liefert 3	14%4 liefert 2

Wie oben erwähnt, ist die Zuweisung, die Sie bereits kennen gelernt haben, auch eine Operation. Sie speichert den Wert, der auf der rechten Seite des Zuweisungsoperators = steht, in die Variable auf der linken Seite.

Mit einer Zuweisung ist allerdings noch mehr möglich:

Was bedeutet denn die folgende Zuweisung bzw. welchen Wert hat die Variable x nach der Zuweisung?

```
int x = 3;  
x = x + 5;
```

Auf den ersten Blick sieht $x = x + 5$ wie eine mathematische Gleichung ohne sinnvoller Lösung aus. Nun, in der Programmierung bedeutet diese Zeile jedoch was ganz anderes. Am Ende der Zuweisung hat die Variable x den Wert 8.

Die Operation macht dabei folgendes:

- Der Computer beginnt auf der rechten Seite des Zuweisungsoperators und liest den Wert von x aus. Das ist 3.
- Er merkt sich temporär nur für diese Berechnung diesen Wert.
- Er addiert die Zahl 5 hinzu.
- Daraus resultiert der Wert 8.
- Dieser Wert 8 wird dann in die Variable x auf der linken Seite gespeichert.
- Der Wert 3 in der Variable x wird dabei überschrieben und ist dann nicht mehr vorhanden!

Generell wird bei einer Zuweisung immer zuerst der (arithmetische) Ausdruck auf der rechten Seite der Zuweisung ausgewertet und danach wird der Ergebniswert in der Variablen auf der linken Seite gespeichert (und dabei sein vorheriger Wert überschrieben).

Kurzschreibweisen

Um sich Schreibarbeit zu ersparen, ist es möglich, bestimmte Kombinationen von Zuweisung und Operation zu verkürzen. Die Kurzschreibweise kann angewendet werden, wenn die Variable der linken Seite auch auf der rechten Seite steht:

Normalschreibweise	Kurzschreibweise
<code>x = x + y;</code>	<code>x += y;</code>
<code>x = x - 3;</code>	<code>x -= 3;</code>
<code>x = x * 5;</code>	<code>x *= 5;</code>
<code>x = x / 7;</code>	<code>x /= 7;</code>
<code>x = x % 2;</code>	<code>x %= 2;</code>

Dabei sind `+=`, `-=`, `*=`, `/=` und `%=` spezielle Zuweisungsoperatoren.

Achtung: `x = 3 - x;` lässt sich nicht als `x -= 3;` abkürzen! Man beachte in der Tabelle die Reihenfolge der Operanden.

Hinweis: Rechenoperationen, welche in der Mathematik nicht möglich sind, führen auch beim Programmieren während der Durchführung des Programms zu einem Fehler. Zum Beispiel liefert eine ganzzahlige Division durch 0 (`x = x / 0`) oder Modulo 0 (`x = x % 0`) einen Fehler:

ArithmeticException: / by zero

Weitere Operationen

Andere Operationen, wie Quadratwurzel oder Potenz können in Processing nicht so angegeben werden, wie etwa am Taschenrechner - Processing bietet dafür kein eigenes Operationszeichen an. Diese Operationen werden nur über eigene Befehlsnamen in Processing aufgerufen: z.B. für Potenzen `pow()` oder für Quadratwurzel `sqrt()`.

Beispiele:

$$d = 4^2$$

```
float d;  
d = pow(4, 2);
```

$$e = \sqrt{25}$$

```
float e;  
e = sqrt(25);
```

Weitere Details und Befehlsnamen für weitere ähnliche Operatoren, zB. Logarithmus, Exponent, Minimum, Maximum, Runden, etc. finden Sie in der Processing API Reference im Unterpunkt *Calculation*.

3.3.2. Operationen mit einem Operanden

- Positives Vorzeichen (+)
- Negatives Vorzeichen (-)
- Inkrement (++)
- Dekrement (--)

Zwei der Operationen sind Ihnen bereits aus der Mathematik bekannt, nämlich die Vorzeichen. Genau wie in der Mathematik muss bei negativen Werten ein Minuszeichen vor die Zahl geschrieben werden. Das Positiv-Vorzeichen ist optional. Falls Sie kein Vorzeichen verwenden, dann interpretiert Processing die Zahl als positiven Wert.

Anspruchsvoller sind die Operationen **Inkrement** und **Dekrement**.

Inkrement steht für Vergrößerung und Dekrement für Verminderung, sie dienen also zur Vergrößerung bzw. Verminderung des Wertes einer ganzzahligen Variablen um 1.

Angenommen wir haben bereits eine Variable `x` deklariert und ihr einen Wert zugewiesen. In Zusammenhang mit Variablen gibt es zwei Möglichkeiten für die Inkrement Operation:

```
++x;  
x++;
```

In beiden Fällen wird der Wert von `x` um 1 erhöht und ist daher äquivalent zur Zuweisung

```
x = x + 1;
```

Allerdings werden die Unterschiede dann ersichtlich, wenn sie im Zusammenhang mit anderen Operationen angewendet werden. Bei folgendem Code werden Sie unterschiedliche Ergebnisse für `y` und `z` erhalten.

```
int x = 3;  
int y = x++;  
int w = 3;  
int z = ++w;
```

Am Ende dieses Programms hat `y` den Wert 3 und `z` den Wert 4.

Die Erklärung dafür ist: Wenn das Inkrement vor der Variable steht (`++w;`), dann wird die Variable um eins erhöht, bevor irgendeine andere Operation ausgeführt wird (in dem Fall die Zuweisung). Dies nennt man Präinkrement.

Falls das Inkrement nach der Variable steht (`x++;`), dann werden die anderen Operationen, hier also die Zuweisung, zuerst ausgeführt und erst danach die Variable um eins erhöht. Daher hat `y` den Wert 3, da zuerst der unveränderte Wert zugewiesen wurde und danach erst `x` erhöht wurde.

Für Dekrement gelten dieselben Regeln wie für Inkrement. Die zwei Minuszeichen können vor oder nach der Variable stehen (`--x`; `x--`;) und verringern die entsprechende Variable um 1. Stehen sie vor der Variable, dann wird die Variable zuerst um 1 verringert und danach die restlichen Operationen in der Zeile ausgeführt. Stehen sie nach der Variable, dann werden alle anderen Operationen zuerst ausgeführt und erst am Ende die Variable um 1 verringert.

3.3.3. Auswertungsreihenfolge der Operationen

Die Operationen können natürlich auch miteinander zu komplexeren Formeln und Berechnungen kombiniert werden. Berücksichtigen Sie dabei jedoch immer die Reihenfolge, in welcher die Operationen abgearbeitet werden:

Grundsätzlich gilt: Klammer- vor Punkt- vor Strichrechnung. Der Modulo-Operator zählt zur Punktrechnung. Vielleicht ist Ihnen die Regel noch als “KlaPuStri” - Regel bekannt.

Danach wird von links nach rechts abgearbeitet mit Ausnahme von der Zuweisung, die von rechts nach links abgearbeitet wird, d.h. erst wird der Wert auf der rechten Seite des Zuweisungsoperators berechnet und dann der linken Seite zugewiesen.

Beispiel:

```
int x = -5 * 7 - 14 % 2;
```

x hat am Ende den Wert -35.

(Punkt- vor Strichrechnung: $-5 * 7$ ergibt -35, 14 modulo 2 ergibt 0, $-35 - 0$ ergibt -35)

Beispiel:

```
int x = -5 * (7 - 14) % 2;
```

x hat am Ende den Wert 1.

(die beiden Punktrechnungen werden von links nach rechts ausgewertet)

Beispiel:

```
int a = 15;
int x = -5 * 7 - --a % 2;
```

x hat am Ende den Wert -35.

(a wird zu allererst um 1 verringert)

Beispiel:

```
int a = 15;
int x = -5 * 7 - a-- % 2;
```

x hat am Ende den Wert -36.
(a wird erst nach der Zuweisung um 1 verringert)

3.4. Sichtbarkeit von Variablen

3.4.1. Lokale und Globale Variablen

Bis jetzt haben wir unsere eigenen Variablen innerhalb des `draw()` Bereichs deklariert und verwendet. Damit ist aber noch nicht das ganze Potential von Variablen ausgeschöpft. Angenommen Sie wollen folgendes Bild erzeugen:



Intuitiv würden Sie vielleicht mit dem gelernten Wissen folgendes Programm schreiben:

```

void setup() {
  background( 200, 200, 0 );
  size( 500, 500 );
}

void draw() {
  int rColor = 200;
  int gColor = 200;
  int bColor = 0;
  int circleX= 250;
  int circleY = 250;
  fill( 0, 0, 0 );
  ellipse( circleX, circleY, 400, 400 );
  fill( rColor, gColor, bColor );
  ellipse( circleX, circleY, 300, 300 );
  fill( 0, 0, 0 );
  ellipse( circleX, circleY, 200, 200 );
  fill( rColor, gColor, bColor );
  ellipse( circleX, circleY, 100, 100 );
}
  
```

In `draw()` benötigt es hier nur die Änderung der drei Variablen `rColor`, `gColor` und `bColor` um die Farben für die zwei inneren Kreise anzupassen.

Aber man kann das Programm noch weiter vereinfachen. Momentan sind die Hintergrundfarbe und die Kreisfarben noch nicht abhängig voneinander. Jedesmal, wenn Sie die Hintergrundfarbe ändern möchten, müssen Sie in `draw()` auch die drei Farbwerte ändern. Der logische nächste Ansatz wäre, die drei Werte im Befehl `background()` durch die Variablen `rColor`, `gColor` und `bColor` zu ersetzen.

```
void setup() {
  background( rColor, gColor, bColor );
  size( 500, 500 );
}

void draw() {
  int rColor = 200;
  int gColor = 200;
  int bColor = 0;
  int circleX= 250;
  int circleY = 250;
  fill( 0, 0, 0 );
  ellipse( circleX, circleY, 400, 400 );
  fill( rColor, gColor, bColor );
  ellipse( circleX, circleY, 300, 300 );
  fill( 0, 0, 0 );
  ellipse( circleX, circleY, 200, 200 );
  fill( rColor, gColor, bColor );
  ellipse( circleX, circleY, 100, 100 );
}
```

Leider wird Processing folgenden Fehler melden:

“rColor cannot be resolved to a variable”

Damit will Processing darauf aufmerksam machen, dass die Variable nicht bekannt ist. Sie können es auch umgekehrt versuchen und die Variablen in `setup()` deklarieren und in `draw()` verwenden. Processing wird Ihnen die gleiche oder eine ähnliche Fehlermeldung zeigen.

Damit kommen wir zur Thematik der Sichtbarkeit von Variablen. Variablen sind je nachdem, wo man sie deklariert, nicht überall verwendbar. Wenn Variablen in `draw()` deklariert wurden, dann können sie nur in `draw()` verwendet werden. Wenn Variablen in `setup()` deklariert wurden, dann können sie nur in `setup()` verwendet werden. Sie sind somit nur innerhalb der geschweiften Klammern sichtbar. Man sagt auch, dass sie nur **lokal sichtbar** sind. Nach den schließenden geschweiften Klammern werden die lokalen Variablen von Processing aus dem Speicher entfernt und man kann nicht mehr auf den Wert der Variable zugreifen. Für Processing existiert dann diese Variable nicht mehr.

```

void setup() {
  background( 200, 200, 0 );
  size( 500, 500 );
}

void draw() {
  int rColor = 200;
  int gColor = 200;
  int bColor = 0;
  int circleX= 250;
  int circleY = 250;
  fill( 0, 0, 0 );
  ellipse( circleX, circleY, 400, 400 );
  fill( rColor, gColor, bColor );
  ellipse( circleX, circleY, 300, 300 );
  fill( 0, 0, 0 );
  ellipse( circleX, circleY, 200, 200 );
  fill( rColor, gColor, bColor );
  ellipse( circleX, circleY, 100, 100 );
}

```

Aber wie können wir sie nun anlegen, sodass bestimmte Variablen für beide, `setup()` und `draw()`, verwendbar sind? Die Lösung ist, die Variablen außerhalb von `setup()` und `draw()` anzulegen und sie dadurch zu **globalen Variablen** zu machen. Sobald Variablen außerhalb von irgendwelchen geschwungenen Klammer deklariert werden, sind sie globale Variablen in Processing. Sie sind damit überall verwendbar.

```

int rColor = 200;
int gColor = 200;
int bColor = 0;
int circleX = 250;
int circleY = 250;

void setup() {
  background( rColor, gColor, bColor );
  size( 500, 500 );
}

void draw() {
  fill( 0, 0, 0 );
  ellipse( circleX, circleY, 400, 400 );
  fill( rColor, gColor, bColor );
  ellipse( circleX, circleY, 300, 300 );
  fill( 0, 0, 0 );
  ellipse( circleX, circleY, 200, 200 );
  fill( rColor, gColor, bColor );
  ellipse( circleX, circleY, 100, 100 );
}

```

In diesem Programm sind nun die Hintergrundfarbe und die Kreisfarben voneinander abhängig und können an einer Stelle abgeändert werden.

Aber warum nicht gleich alle Variablen global deklarieren?

Stellen Sie sich vor, Sie hätten ein großes Programm, das in einem Team entwickelt wird. Hier kommen ein paar Hundert Variablen ins Spiel und Sie würden sehr schnell den Überblick verlieren wo welche Variablen verwendet und verändert werden. Daher ist es üblich, dass man Variablen nur dann global deklariert, wenn man sie tatsächlich global braucht oder wenn man weiß, dass sie nicht mehr geändert werden.

Processing hat neben den Befehlen auch bereits vordefinierte globale Variablen (z.B. `width`, `height`, `mouseX`, `mouseY`), die wir nach und nach kennen lernen werden.

3.4.2. Gleichnamige Variablennamen

Aufgrund der Trennung von globalen und lokalen Variablen ist es möglich zwei Variablen den gleichen Namen zu geben. Dabei müssen die gleich benannten Variablen in verschiedenen Blöcken (geschwungene Klammern Paare) deklariert sein oder auch eine der Variablen global sein und die andere lokal.

```
int rColor = 200;

void setup() {
  int rColor = 100;
  background( rColor, 0, 0 );
  size( 500, 500 );
}

void draw() {
  fill( rColor, 0, 0 );
  ellipse( 250, 250, 300, 300 );
}
```

Nach dem Ausführen dieses Programms ist erkennbar, dass die Ellipse und der Hintergrund unterschiedliche Rottöne verwenden. Der Befehl `background()` verwendet in diesem Fall die lokale Variable, die den Wert 100 besitzt, während der Befehl `fill()` im `draw()`-Bereich die globale Variable mit dem Wert 200, verwendet. Sobald eine lokale Variable deklariert wird, die den gleichen Namen besitzt wie eine globale Variable, wird die globale Variable **überschattet** von der lokalen Variable. Die globale Variable wird NICHT überschrieben bzw. erhält keinen neuen Wert. Das heißt, die lokale Variable wird statt der globalen verwendet, aber die globale existiert weiterhin.

Die Verwendung von unterschiedlichen Variablen mit gleichem Namen ist möglich, aber kann unübersichtlich sein, da man nicht unbedingt sofort erkennt, ob Variablen überschattet sind oder nicht. Es ist daher empfehlenswert globale Variablen nicht zu überschatten, um die Leserlichkeit zu verbessern und Fehler zu vermeiden.

3.5. Ausgabe von Werten

Um mehr Einsicht in die Werte von Variablen zu haben, kann man sie mit den Befehlen `print()` und `println()` ausgeben. Als Parameter kann man Werte, Variablen und viele andere Dinge angeben, um sie auszugeben, aber dazu kommen wir später.

Bis jetzt erfolgte die Ausgabe in grafischer Form auf dem Sketchfenster. Die Ausgabe der Befehle `print()` und `println()` erfolgt in der Konsole, ein rein textbasiertes Ausgabesystem. Die beiden Befehle sind in ihrer Funktionsweise sehr einfach, man gibt nur an, was man ausgeben möchte. Die Reihenfolge der Ausgabe hängt von der Programmausführung ab, also in welcher Reihenfolge die `print()`-Befehle im Programm stehen. Dieses Prinzip kennen Sie bereits von der Überlappung von geometrischen Formen.

```
void setup() {  
  int one = 1;  
  println(42);  
  println(one);  
}  
  
void draw() {  
  
}
```

Wenn Sie das oben stehende Programm ausführen wird das Sketchfenster leer bleiben, weil wir nichts zeichnen. Dafür sehen Sie in der Konsole folgende Ausgabe:

```
42  
1
```

Hier wird die 42 vor der 1 ausgegeben weil der Befehl `println(42)` vor dem Befehl `println(one)` im Programm kommt. Wenn man die Reihenfolge vertauscht dann wird sich auch die Ausgabe umdrehen. Sie können das gerne ausprobieren!

`print()` und `println()` unterscheiden sich dadurch dass `println()` nach der Ausgabe des Wertes einen Zeilenumbruch macht (`println()` wird daher als *println* ausgesprochen). Ersetzen Sie `println()` durch `print()` und vergleichen Sie den Unterschied.

3.6. Ausführungsreihenfolge Revisited

Bis jetzt hatten wir nur geometrische Formen, die sich nicht bewegt haben, aber mit Processing ist es möglich Animationen und Interaktionen zu gestalten.

Um geometrische Formen zu zeichnen, genügt es die Programmausführung von oben nach unten zu betrachten. Wenn wir Interaktionen gestalten wollen, muss der Computer auf unsere Eingaben reagieren können. Wir haben das Konstrukt dafür schon kennengelernt nur nicht voll ausgeschöpft. Nun verwenden wir zwei bereits erlernte Konzepte: die Globalen Variablen und die Ausgabe um dieses Konstrukt sichtbar zu machen:

```
int count;

void setup() {
    count = 0;
    println(count);
}

void draw() {
    ++count;
    println(count);
}
```

Dieses Programm erstellt eine globale Variable names `count`, im `setup()` Bereich wird die Variable `count` initialisiert und ausgegeben. Die Initialisierung ist so in Ordnung, denn die Variable bekommt so vor der ersten Verwendung einen Wert. Im `draw()` Bereich wird `count` inkrementiert und ausgegeben.

Sie werden jetzt vielleicht annehmen, dass die Ausgabe des Programms folgendermaßen aussieht:

```
0
1
```

Diese Annahme ist aber nicht ganz vollständig. Denn wenn Sie das Programm bei sich ausführen, werden Sie feststellen, dass die Ausgabe wie folgt aussieht:

```
0
1
2
3
4
5
6
7
8
...
```

Die Ausgabe kommt nicht zum Ende und die Ausführung des Befehls läuft immer weiter. Erst wenn Sie das Programm stoppen, stoppt auch die Ausgabe. Nun stellt sich die Frage, warum dieses Phänomen auftritt?

Wenn Sie das vorige Beispiel genauer betrachten, werden sie feststellen, dass die Befehle im `draw()` Bereich immer wieder aufs neue ausgeführt werden. Dies zeigt uns, dass Processing Programme einem Lebenszyklus folgen. Die Geburt ist der Programmstart, also das Drücken auf *Play* und sie leben so lange bis man sie durch Drücken auf *Stop* beendet, also terminiert. Dieses fortlaufende Leben wird den Processing Programmen durch ständige Wiederholung der Befehle im `draw()` Bereich eingehaucht. Dabei ist es unabhängig, wo sich der `draw()` Bereich im Code befindet. Folgendes Programm verhält sich genauso wie das vorhergehende Programm.

```
int count;

void draw() {
  ++count;
  println( count );
}

void setup() {
  count = 0;
  println( count );
}
```

Durch diese automatische Wiederholung der Befehle im `draw()` Bereich ist es möglich Interaktionen im Programm zu gestalten. Denn würde der `draw()` Bereich nur ein einziges Mal ausgeführt werden, wäre es nicht möglich kontinuierlich auf Benutzereingaben oder Änderungen von Variablenwerten zu reagieren.

3.7. Animation in Processing

Eine der wichtigsten Vorzüge der Programmiersprache Processing ist die einfache Erstellung von Animationen mit nur wenigen Programmzeilen. Eine Animation entsteht dadurch, dass mehrere Bilder schnell hintereinander abgespielt werden, zum Beispiel wie bei einem Daumenkino. Durch die hohe Geschwindigkeit nimmt das menschliche Auge die Änderung nicht als einzelne Bilder wahr, sondern als fließende Bewegung. Es werden mindestens 28 Bilder pro Sekunde (engl: Frames per Second (FPS)) benötigt, damit das Auge diese Bilder als eine fließende Bewegung sieht.

Grundlage für Animationen in Processing ist der `draw()`-Bereich. Der `draw()`-Bereich wird nach dem `setup()` immer wieder ausgeführt, solange bis er vom Benutzer gestoppt wird. Im `draw()` arbeitet Processing von oben nach unten. Wenn es das Ende des `draw()` Bereichs erreicht hat, wird das Bild im Sketchfenster angezeigt. Danach wird im `draw()` wieder von oben nach unten abgearbeitet und am Ende wieder ein neues Bild in das Sketchfenster gezeichnet. Das passiert, wenn man es nicht anders einstellt, 60 Mal in der Sekunde. Das heißt Processing zeichnet 60 Bilder in einer Sekunde in das Sketchfenster. Der **Programmfluss**, also in welcher Reihenfolge das Programm abgearbeitet wird, verläuft somit nicht nur von der ersten Zeile bis zur letzten Zeile, sondern wiederholt sich im `draw()`.



Damit sich etwas bewegt bzw. verändert, müssen im `draw()`-Bereich Änderungen vorgenommen werden, sodass sich die Ausgaben am Sketch-Fenster von Durchlauf zu Durchlauf unterscheiden. Zu diesem Zweck kommen globale Variablen zum Einsatz. Die Änderungen werden dann durch Veränderung der globalen Variablen innerhalb des `draw()`-Bereichs erreicht. Das kann zum Beispiel eine Erhöhung einer globalen Variable um 1 am Beginn oder Ende des `draw()`-Bereichs sein.

Lokale Variablen des `draw()` Bereichs haben das Problem, dass sie am Ende eines Durchlaufes in `draw()` aus Processing Sicht nicht mehr existieren und somit auch keine Werte für den nächsten Durchlauf speichern können. Beim nächsten Durchlauf wird die lokale Variable neu angelegt und immer wieder mit dem gleichen Wert initialisiert. Globale

Variablen bleiben hingegen am Ende eines Durchlaufs bestehen und können somit nach jedem Durchlauf Informationen für den nächsten Durchlauf speichern.

Lokale Variable (Keine Animation)	Globale Variable (Mit Animation)
<pre> void setup() { size(200, 100); } void draw() { int x = 0; ellipse(x, 50, 50, 50); x++; } </pre>	<pre> int x = 0; void setup() { size(200, 100); } void draw() { ellipse(x, 50, 50, 50); x++; } </pre>

Beispiel:

Der gelbe Kreis `ellipse(100, 50, 50, 50);` soll sich von links nach rechts bewegen. Dabei ist die Ausgangsposition des Kreises so zu wählen, dass sein Mittelpunkt den linken Rand des Sketch-Fensters berührt.

Lösungsansatz: Die Bewegung bzw. Animation liegt in der x-Koordinate der Kreisposition. Eine Bewegung nach rechts bedeutet im Koordinatensystem eine wiederholte Erhöhung der x-Koordinate. Daher muss der Wert der x-Koordinate des Kreises bei jedem Durchlauf des `draw()`-Bereichs um "1" höher sein als beim vorigen.

Umsetzung: Wir deklarieren eine globale Variable `int xPosition`, welche die veränderliche x-Position des Kreismittelpunktes speichert und initialisieren diese mit 0 (Zeile 1). Dieser Wert entspricht der Ausgangsposition, dem linken Rand des Sketch-Fensters.

Im `draw()`-Bereich wird nun der Kreis gezeichnet, wobei `xPosition` als Parameter für die x-Koordinate des Kreismittelpunktes angegeben wird (Zeile 14). Außerdem wird `xPosition` am Ende des `draw()`-Bereichs inkrementiert, d.h. um 1 erhöht (Zeile 16).

```

1  int xPosition = 0; //globale Variable; x-Position des Kreises
2
3  void setup() {
4
5      size( 200, 100 );
6
7  }
8
9  void draw() {
10
11      background( 255, 255, 180 );
12
13      fill( 250, 230, 0 );
14      ellipse( xPosition, 50, 50, 50 );
15
16      xPosition++;

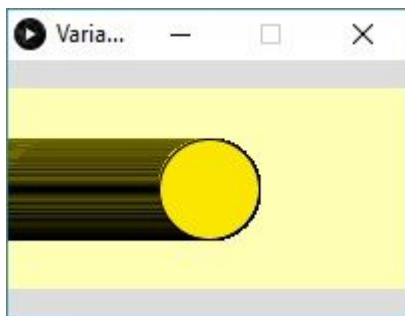
```

17 }

Für die ersten fünf Durchläufe des `draw()`-Bereichs sieht der Befehlsaufruf für den Kreis mit konkreten Werten also wie in der dritten Spalte aus:

Durchlauf	xPosition Zeile 14	Befehlsaufruf mit expliziten Werten Zeile 14	xPosition nach Zeile 16
1	0	<code>ellipse(0, 50, 50, 50);</code>	1
2	1	<code>ellipse(1, 50, 50, 50);</code>	2
3	2	<code>ellipse(2, 50, 50, 50);</code>	3
4	3	<code>ellipse(3, 50, 50, 50);</code>	4
5	4	<code>ellipse(4, 50, 50, 50);</code>	5

Da sich `setup()` und `draw()` unterschiedlich verhalten, macht es einen Unterschied, in welchem Bereich man Befehle schreibt. Insbesondere gilt das auch für den Befehl `background()`, der die Hintergrundfarbe setzt. Tatsächlich zeichnet dieser Befehl über das ganze Sketch-Fenster die gewünschte Farbe. Wenn dieser Befehl nur im `setup()` steht, dann wird nur zu Beginn das Sketch-Fenster eingefärbt. Wenn durch `draw()` eine Bewegung eines Objektes stattfinden soll, werden alle einzelnen Bilder übereinander gezeichnet. Man kann sich das ähnlich vorstellen, wie wenn man ein Blatt Papier verwendet und mehrere verschiedene Bilder mit Deckfarbe übereinander zeichnet. Das Resultat ist, dass das zuletzt gezeichnete Bild ganz sichtbar ist, während die anderen Bilder teils verdeckt sind.



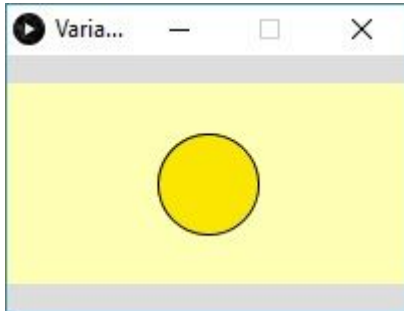
```
int xPosition = 0;

void setup() {
  background(255, 255, 180);
  size(200, 100);
}

void draw() {
  fill(250, 230, 0);
  ellipse(xPosition, 50, 50, 50);
  xPosition++;
}
```

Im Beispiel bewegt sich ein Kreis von links nach rechts. Da `background()` im `setup()` steht, wird bei jedem Durchlauf von `draw()` nur der Kreis neu gezeichnet und durch die Verschiebung sieht es so aus, dass der Kreis einen Schweif zieht.

Wird der Befehl jedoch in `draw()` geschrieben, sieht man diesen Schweif nicht mehr. Bevor irgendeine Formen gezeichnet werden, wird durch den Befehl `background()` das gesamte Sketch-Fenster mit der Hintergrundfarbe eingefärbt und somit das vorhergehende Bild komplett übermalt. Es ist wie, wenn man mit Deckfarbe, bevor man etwas zeichnet, zuerst das ganze Blatt einfärbt. Man muss natürlich dann auch darauf achten, dass dieser Befehl ganz am Anfang im `draw()` steht.



```
int xPosition = 0;



void setup() {
  size(200, 100);
}

void draw() {
  background(255, 255, 180);
  fill( 250, 230, 0 );
  ellipse(xPosition, 50, 50, 50);
  xPosition++;
}
```

3.8. Debugger

Die Programme werden mit der Zeit immer komplexer. Syntaxfehler sind leicht zu erkennen, da sie von Processing mittels roter Wellenlinie oder unterhalb des Programmierbereichs rot angezeigt werden.

Semantische “Fehler” hingegen sind nicht so leicht auffindbar. Das Programm sieht syntaktisch korrekt aus, tut aber nicht das, was man eigentlich erwartet hätte. Mit dem Debugger ist es möglich, Schritt für Schritt (oder auch in größeren Schritten) das Programm abzuarbeiten und zu analysieren, wie sich Variablenwerte und Ausgaben verändern. Dadurch können Fehlerquellen von ungewöhnlichem oder falschem Programmverhalten systematisch ermittelt werden

Um debuggen zu können, muss vorerst der Debugger Modus aktiviert werden. Der Debugger wird in Processing über den  Button oben rechts aktiviert bzw. deaktiviert. Ist der Modus aktiviert, erscheint ein weiteres Fenster, in dem während das Programm läuft die Variablenwerte angezeigt werden. Zwischen dem *Start-Button* (*Run-Button*) und dem *Stop-Button* erscheinen zwei weitere Buttons/Schalter. Um den Debugger Modus zu deaktivieren, reicht ein weiterer Klick auf .

Fenster in dem die aktuellen Variablenwerte angezeigt werden

Buttons zum Debuggen:

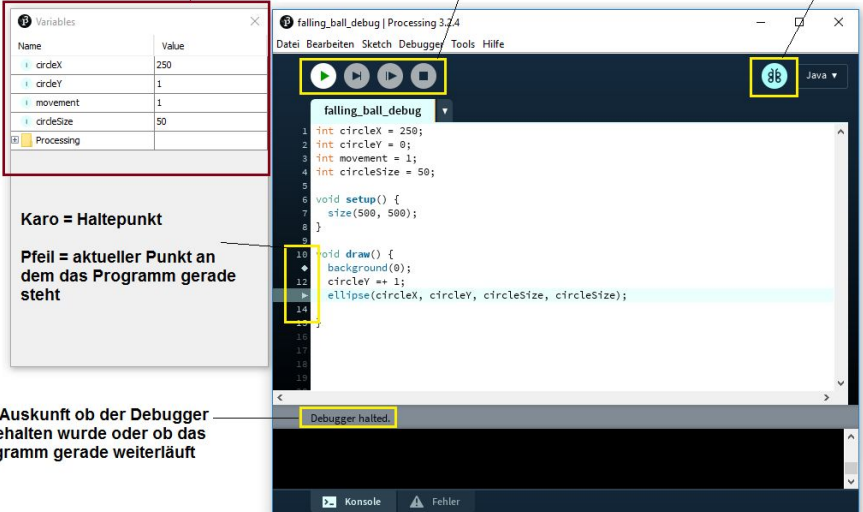
1. Run: startet den Debugger
2. Schritt: führt den aktuellen Befehl aus und springt in die nächste Codezeile
3. Weiter: Führt das Programm bis zum nächsten Haltepunkt aus
4. Stoppen: Beendet den Durchlauf

aktivierter Debug-Modus - sobald auf diesen Button geklickt wird, verändert sich dessen Farbe und die Buttons auf der linken Seite erscheinen

Karo = Haltepunkt

Pfeil = aktueller Punkt an dem das Programm gerade steht

gibt Auskunft ob der Debugger angehalten wurde oder ob das Programm gerade weiterläuft



Will man die Variablenwerte ab einem bestimmten Punkt im Programm analysieren, kann man in der entsprechenden Zeile durch Klicken auf eine Zeilennummer (welche Programmcode enthält) einen Haltepunkt setzen. An dieser Stelle wird dann ein Rauten-Symbol angezeigt. Klickt man auf die Raute, wird der Haltepunkt wieder entfernt.

Startet man nun den Debugger, wird das Programm bis zum gesetzten Haltepunkt ausgeführt und bleibt dann “stehen”. Ein Pfeil zeigt an, wo gerade das Programm steht. Im Variablen-Fenster werden dann die aktuellen Werte aller momentan existierender Variablen angezeigt. Ab dann lässt sich über die einzelnen Buttons “Run/Schritt/Weiter/Stoppen” (siehe Abbildung oben) der weitere Debug-Vorgang steuern, indem man eine Programmzeile weiter springt oder gar zum nächsten Haltepunkt.