Einstieg in die Programmierung mit Processing[1]

Mit Processing können Probleme aus verschiedenen Bereichen gelöst werden. Beispielsweise lassen sich Bilder sehr effizient und leicht bearbeiten. Auch die Programmierung von Robotern oder Mikrocontrollern ist möglich. Um die Programmierung mit Processing kennenzulernen, wollen wir zunächst die Zeichenfunktionen von Processing nutzen, um Grafiken zu erstellen.

Erste Schritte

Ein Processing Programm gliedert sich in mehrere Methoden. In die Methode *void setup()* schreiben wir alle Befehle, die beim Starten des Programms einmalig ausgeführt werden sollen. Hier können z. B. die Größe des Fensters und die Hintergrundfarbe festgelegt werden.

Aufgabe 1: Erstellen Sie Ihr erstes Programm, indem Sie folgende Zeilen eingeben und das Programm mit Klick auf den grünen Pfeil starten:

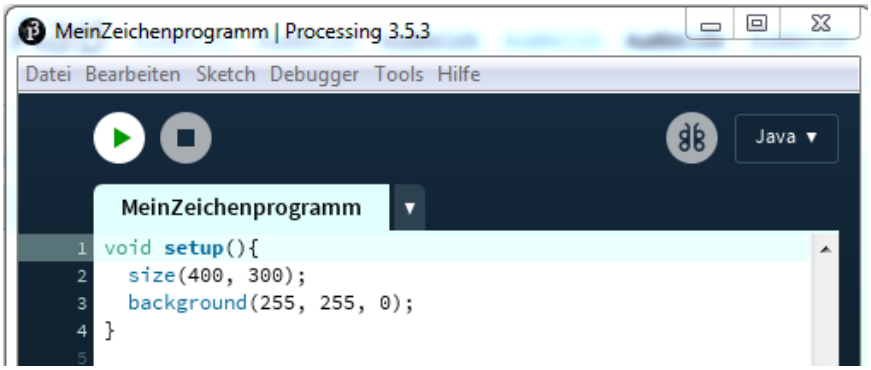

Abbildung 1: Ein erstes Programm in Processing

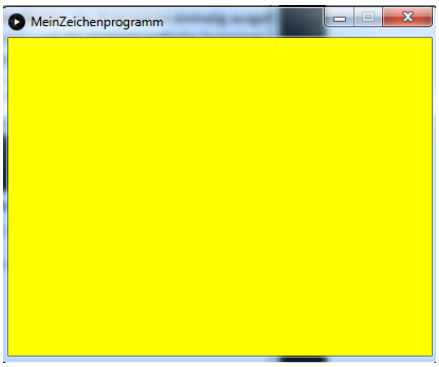

Abbildung 2: Programmfenster zu dem Beispiel aus Abbildung 1

Es sollte ein 400 Pixel breites und 300 Pixel hohes Fenster mit gelbem Hintergrund angezeigt werden. Die Farbwerte werden als RGB-Werte eingegeben, so steht (255, 255, 0) z. B. für gelb. Die folgende Tabelle enthält einige Beispiele für RGB-Werte. Dazu lernen wir später mehr.

Farbe	RGB-Wert
weiß	255, 255, 255
schwarz	0, 0, 0
rot	255, 0, 0
grün	0, 255, 0
blau	0, 0, 255
gelb	255, 255, 0
cyan	0, 255, 255
magenta	255, 0, 255

Tabelle 1: Beispiele für RGB-Werte

Aufgabe 2: Probieren Sie unterschiedliche Größen und Farben für das Fenster aus.

[1] Die Programmierumgebung Processing wurde 2001 von Ben Fry und Casey Reas initiiert. Nähere Informationen finden Sie unter https://processing.org/. Für die Beispiele wurde Processing in der Version 3.5.3 verwendet.

Figuren zeichnen

In das Fenster können wir nun verschiedene Figuren und Linien zeichnen.

Aufgabe 3: Ergänzen Sie den folgenden Code in Ihrem Programm. Beobachten Sie, was beim Starten des Programms gezeichnet wird. Überlegen Sie, welche Bedeutung die Methoden (so werden in Processing die Anweisungen genannt) und die Parameter, die in Klammern stehen, haben könnten?

```
1  void setup(){
2    size(400, 300);
3    background(255, 255, 0);
4    fill(255, 0, 0);
5    rect(30, 60, 50, 50);
6    fill(0, 255, 0);
7    rect(180, 100, 20, 40);
8    noFill();
9    circle(200, 250, 60);
10   strokeWeight(5);
11   ellipse(250, 150, 30, 60);
12 }
```

Beispiel 1: Figuren zeichnen

Die folgende Tabelle zeigt einige Beispiele für Methoden, die zum Zeichnen zur Verfügung stehen. Eine vollständige Übersicht erhalten Sie, wenn Sie im Menü *Hilfe* den Punkt *Referenz* aufrufen.

Methode	Bedeutung
rect(x, y, width, height)	zeichnet ein Rechteck mit der linken oberen Ecke an der Position x|y und der angegeben Breite und Länge
square(x, y, extent)	zeichnet ein Quadrat mit der linken oberen Ecke an der Position x|y und der Seitenlänge *extent*
ellipse(x, y, width, height)	zeichnet eine Ellipse mit dem Mittelpunkt an der Position x|y und der angegeben Breite und Länge
circle(x, y, extent)	zeichnet einen Kreis mit dem Mittelpunkt an der Position x|y und dem dritten Parameter als Durchmesser
line(x1, y1, x2, y2)	zeichnet eine Linie von Position x1|y1 zu Position x2|y2
point(x, y)	zeichnet einen Punkt an Position x|y
stroke(r, g, b)	legt die Farbe der (Umrandungs-)Linien fest, die nachfolgend gezeichnet werden
strokeWeight(weight)	legt die Dicke der (Umrandungs-)Linien fest, die nachfolgend gezeichnet werden
fill(r, g, b)	legt fest, mit welcher Farbe die nachfolgend gezeichneten Figuren gefüllt werden
noFill()	legt fest, dass die nachfolgend gezeichneten Figuren nicht gefüllt werden

Tabelle 2: Befehle zum Zeichnen

Das Koordinatensystem in Processing

Die Position der Figuren wird mithilfe der x- und y-Koordinate im Zeichenfenster angegeben. Dabei befindet sich die Position 0|0 in der linken oberen Ecke des Fensters. Die positive x-Achse verläuft nach rechts und die positive y-Achse (anders als in der Mathematik!) nach unten.

Aufgabe 4: Erkunden Sie das Koordinatensystem von Processing, indem Sie das Programm *KoordinatenTest* starten und verschiedene Positionen im Fenster mit der Maus anklicken. Das Programm zeigt Ihnen dann die Koordinaten der angeklickten Position.

Aufgabe 5: Testen Sie die Befehle zum Zeichen von Figuren, indem Sie z. B. einen Schneemann aus verschiedenen Formen zeichnen.

Zufallsgrafiken

Schauen wir uns nun an, wie wir Zufallsgrafiken erzeugen können. Eine Zufallszahl erhalten wir mithilfe der Methode *random()*. Als Parameter können wir entweder nur eine obere oder eine untere und eine obere Grenze angeben.

`random(200);` liefert uns eine zufällige Zahl z zwischen 0 und 200 ($0 \leq z < 200$).

`random(50, 200);` liefert uns eine zufällige Zahl z zwischen 50 und 200 ($50 \leq z < 200$).

Die Zufallszahl ist eine Gleitkommazahl vom Datentyp *float*. Möchte man diese in eine Ganzzahl umwandeln, verwendet man den Befehl *int()* und übergibt die Zufallszahl als Parameter:
`int(random(5, 20))` liefert eine zufällige Ganzzahl zwischen 5 und 20.

Möchten wir eine Figur an einer zufälligen Position zeichnen, sollte die x- Koordinate zwischen 0 und der Breite des Fensters und die y- Koordinate zwischen 0 und der Höhe des Fensters liegen. Dabei kann es hilfreich sein, die Systemvariablen `width` und `height` zu verwenden, die uns die Breite und Höhe des Fensters liefern. Wenn wir die Größe des Fensters ändern, werden die Werte so automatisch angepasst.

Wenn das Fenster 400 Pixel breit ist, würde `random(width);` also eine zufällige Zahl zwischen 0 und 400 liefern.

Aufgabe 6: Erstellen Sie ein Processing Programm, das ein Quadrat an einer zufälligen Position mit einer zufälligen Seitenlänge zwischen 2 und 100 Pixeln zeichnet. Achten Sie bei den Zufallszahlen für die x- und y- Koordinate darauf, dass sich das Quadrat innerhalb des Fensters befinden sollte.

Eine einzelne zufällig gezeichnete Figur ist noch nicht besonders spannend. Hübsch wird es, wenn wir z. B. 100 solcher Zufallsfiguren zeichnen lassen. Nun wäre es aber ziemlich umständlich 100-mal die Anweisungen für das zufällige Zeichnen eines Quadrats einzugeben. Deshalb verwenden wir dafür eine Zählschleife, die einen Befehl sooft wiederholt, wie wir es möchten. Eine Zählschleife lässt sich in Processing auf zwei Arten realisieren:

While-Schleife als Zählschleife

Die Anweisungen in einer *while*-Schleife werden wiederholt, bis die Schleifenbedingung nicht mehr erfüllt ist. Für eine Zählschleife muss daher eine Variable definiert werden, die z. B. mit dem Wert 0 startet und deren Wert innerhalb der Schleife fortlaufend erhöht wird. Die Schleifenbedingung kann dann so formuliert werden, dass die Schleife nur solange ausgeführt wird, bis die Variable einen bestimmten Wert erreicht hat. Im folgenden Beispiel startet die Variable `i` mit dem Wert 0 (siehe Zeile 1). Bei jedem Schleifendurchlauf wird der Wert um 1 erhöht. Die Anweisung `i++` in Zeile 6 ist

dabei eine Kurzschreibweise für `i = i + 1`. Wenn die Variable `i` den Wert 99 erreicht hat, bricht die Schleife ab, da die Schleifenbedingung `i < 100` in Zeile 2 nicht mehr erfüllt ist. Innerhalb der Schleife wird in Zeile 3 zunächst eine zufällige Füllfarbe ausgewählt. In Zeile 4 wird eine zufällige Seitenlänge zwischen 5 und 20 Pixeln erzeugt. In Zeile 5 wird das Quadrat dann an eine zufällige Position gezeichnet, wobei die Größe des Quadrates berücksichtigt wird. Insgesamt werden so 100 zufällige Quadrate gezeichnet.

```
1  int i = 0;
2  while(i < 100){
3    fill(random(255), random(255), random(255));
4    float seitenlaenge = random(5, 20);
5    square(random(width-seitenlaenge), random(height-seitenlaenge),
               seitenlaenge);
6    i++;
7  }
```

Beispiel 2: While-Schleife als Zählschleife

For-Schleife

Die *for*-Schleife arbeitet im Prinzip genauso wie die oben definierte *while*-Schleife. Es stehen jedoch alle Angaben, die Definition der Variablen, die Initialisierung mit einem Startwert, das Erhöhen des Variablenwertes und die Schleifenbedingung, innerhalb des Schleifenkopfes:

```
1  for(int i = 0; i < 100; i++){
2    fill(random(255), random(255), random(255));
3    float seitenlaenge = random(5, 20);
4    square(random(width-seitenlaenge), random(height-seitenlaenge),
               seitenlaenge);
5  }
```

Beispiel 3: For-Schleife

In Zeile 1 finden wir somit im Schleifenkopf zunächst die Anweisung zum Erstellen der Variablen `i` mit dem Startwert 0, dann die Bedingung, die erfüllt sein muss, damit die Schleife ein weiteres Mal durchlaufen wird (`i < 100`) und zum Schluss die Anweisung `i++`, die am Ende von jedem Schleifendurchlauf ausgeführt wird, um die Variable um 1 zu erhöhen.

Noch ein wenig abwechslungsreicher können wir das Bild gestalten, indem wir die Zählvariable `i` beim Erzeugen der Zufallszahlen mit einbeziehen. So können wir zum Beispiel in Zeile 3 für die Seitenlänge eine Zufallszahl zwischen 5 und i+6 erzeugen lassen. Dadurch werden die Quadrate tendenziell immer größer.

```
1  for(int i = 0; i < 100; i++){
2    fill(random(255), random(255), random(255));
3    float seitenlaenge = random(5, i+6);
4    square(random(width-seitenlaenge), random(height-seitenlaenge),
               seitenlaenge);
5  }
```

Beispiel 4: Verwendung der Zählvariablen der for-Schleife

Aufgabe 7: Verändern Sie das Programm aus Beispiel 4, indem Sie die Zählvariable auch an anderer Stelle miteinbeziehen und die Anzahl der erzeugten Quadrate, den Startwert oder die Schrittweite beim Hochzählen verändern. Sie können natürlich auch noch andere Formen hinzunehmen.

Verzweigungen

Noch ein bisschen mehr Abwechslung können wir in unser Zufallsbild bringen, wenn wir nicht alle Quadrate füllen, sondern einige nur als Umriss zeichnen. Zum Beispiel könnten wir das Quadrat füllen, wenn unsere Zählvariable `i` gerade ist und nicht füllen, wenn sie ungerade ist. Dazu benötigen wir eine Verzweigung, die die Fälle `i` gerade und `i` ungerade bzw. *sonst* unterscheidet.

Ob eine Zahl gerade ist, erkennen wir daran, dass der Rest beim Teilen durch 2 Null ist. Im folgenden Beispiel verwenden wir daher die Bedingung, dass `i` modulo 2 gleich Null sein soll. Der Operator Modulo ist in Processing das %-Zeichen. Für einen Vergleich wird ein doppeltes Gleichheitszeichen verwendet, da das einfache Gleichheitszeichen schon für die Wertzuweisung bei einer Variablen belegt ist. Wenn die Bedingung erfüllt ist, wird die Anweisung *fill* in Zeile 2 ausgeführt, ansonsten die Anweisung *noFill* in Zeile 4.

```
1  if((i%2) == 0){
2    fill(random(255), random(255), random(255));
3  }else{
4    noFill();
5  }
```

Beispiel 5: Verzweigung, die zwischen Zeile 1 und 2 in Beispiel 4 eingefügt werden kann

Aufgabe 8: Erweitern Sie Ihr Programm aus Aufgabe 7 um verschiedene Verzweigungen. Beispielsweise können abhängig von der Zählvariablen oder einem Zufallswert, die Formen oder die Farben variiert werden.

Aufgabe 9: Halten Sie Ihren Quelltext für die Lösung dieser Aufgabe durch die Verwendung von Schleifen und Verzweigungen möglichst kurz. Im Idealfall werden die benötigten Zeichenbefehle nur einmal aufgeschrieben.

a) Erstellen Sie ein Programm, das einen Turm aus Quadraten zeichnet. Die Farben der Quadrate sollen dabei wechseln (s. Abbildung 3).

b) Erstellen Sie ein Programm, das eine dreifarbige Raupe zeichnet (S. Abbildung 4). Ergänzen sie gerne weitere Details wie z. B. Fühler und Augen.

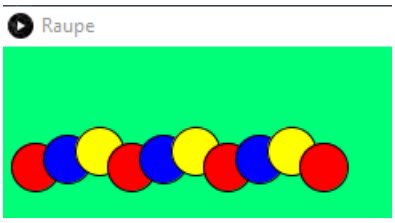

Abbildung 4: Dreifarbige Raupe, Ausgabe zu Aufgabe 9b

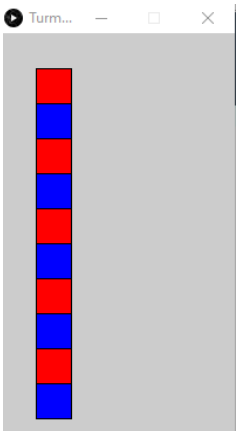

Abbildung 3: Turm aus roten und blauen Quadraten, Ausgabe zu Aufgabe 9a.

Interaktion mit dem Anwender

Bislang wird unser gesamtes Programm direkt beim Start ausgeführt und danach erhalten wir das fertige Bild, das wir nur durch einen erneuten Start des Programms neu zeichnen können. Schön wäre es, wenn der Anwender das Programm beeinflussen kann, während es läuft.
Dazu müssen wir neben der Methode *void setup()* zunächst die Methode *void draw()* in unserem Programm ergänzen. Diese Methode wird nicht nur einmal beim Starten des Programms ausgeführt, sondern immer wieder, solange das Programm läuft, so dass die Anzeige ständig aktualisiert wird. Hier könnten wir also Anweisungen platzieren, die in einer Endlosschleife immer neu ausgeführt werden sollen. Auch wenn wir hier nichts reinschreiben, sorgt diese Methode dafür, dass das Fenster

immer wieder neu gezeichnet wird und damit Veränderungen, die wir in weiteren Methoden vornehmen, sichtbar werden. Deshalb ist es wichtig, dass unser Programm diese Methode enthält, auch wenn wir sie im Moment noch nicht mit Inhalt füllen.

Reaktion auf die Maus

Eine Methode, die für die Interaktion mit dem Anwender interessant ist, ist *void mouseClicked()*. Diese wird immer dann aktiviert, wenn der Anwender mit einer Maustaste einen Mausklick innerhalb des Programmfensters ausführt. Wenn wir in diese Methode schreiben, dass ein Quadrat gezeichnet werden soll, dann wird das Quadrat nur dann gezeichnet, wenn der Anwender eine Maustaste betätigt hat. Um noch besser mit dem Anwender interagieren zu können, stellt Processing uns noch weitere Systemvariablen zur Verfügung.

Systemvariable	Bedeutung
mouseX	enthält die aktuelle x-Koordinate der Maus
mouseY	enthält die aktuelle y-Koordinate der Maus
pmouseX	enthält die vorherige x-Koordinate der Maus
pmouseY	enthält die vorherige y-Koordinate der Maus
mouseButton	enthält die Werte LEFT, RIGHT oder CENTER, je nachdem welche Maustaste gedrückt wurde

Tabelle 3: Systemvariablen zum Zustand der Maus

Das folgende Programm zeichnet bei jedem Mausklick ein rotes Quadrat an die aktuelle Position der Maus.

```
1  void setup(){  //diese Methode wird nur einmalig beim Programmstart ausgeführt
2    size(400, 300);
3    background(255, 255, 0);
4    fill(255, 0, 0);
5  }
6  void draw(){}  //diese Methode wird immer wieder von vorne durchlaufen
7  void mouseClicked(){ //diese Methode wird jedes Mal ausgeführt, wenn eine Maustaste gedrückt
                        //wurde
8    square(mouseX, mouseY, 30); //mouseX und mouseY liefern die aktuelle Position der Maus
9  }
```

Beispiel 6: Reaktion auf Mausklick

Neben *mouseClicked()* gibt es noch weitere Methoden, die auf Mausereignisse reagieren. Einen Überblick gibt Tabelle 4. Ausführliche Erläuterungen finden Sie in der Referenz[2].

Methode	wird ausgeführt, …
mousePressed()	sobald eine Maustaste gedrückt wird, unabhängig vom Zeitpunkt des Loslassens.
mouseReleased()	wenn eine Maustaste wieder losgelassen wird.
mouseDragged()	wenn die Maus mit gedrückter Maustaste bewegt wird.
mouseMoved()	wenn die Maus ohne gedrückte Maustaste bewegt wird.

Tabelle 4: Methoden für die Reaktion auf Ereignisse, die von der Maus ausgelöst werden

[2] Ben Fry & Casey Reas. Processing – Reference. https://processing.org/reference/ [letzter Zugriff: 18.07.2019]

Aufgabe 10: Implementieren Sie ein Programm, das eine Spur des Mauszeigers auf dem Bildschirm hinterlässt, wenn der Anwender die Maus mit gedrückter Maustaste über den Bildschirm bewegt.

Betrachten wir nun als weiteres Beispiel ein Programm, das es dem Anwender erlaubt, Rechtecke mit der Maus auf den Bildschirm zu zeichnen bzw. diese aufzuziehen. Die Maustaste wird dazu an der Position der linken, oberen Ecke gedrückt und die Maus dann mit weiterhin gedrückter Maustaste in die rechte, untere Ecke des Rechtecks gezogen und dort losgelassen. Wir benötigen dazu zwei Variablen, in denen wir uns die Startposition für das Rechteck merken. Daher werden in Beispiel 7 in Zeile 1 und Zeile 2 jeweils eine Variable für die x- und für die y-Koordinate der Startposition angelegt. Sobald die Maustaste gedrückt wird, sorgt die Methode *mousePressed()* in Zeile 13 bis 16 dafür, dass in den Variablen `x1` und `y1` die aktuelle Position der Maus gespeichert wird. Da die Methode nach dem Drücken nur einmal ausgeführt wird, bleiben die Werte bis zum Loslassen und erneuten Drücken einer Maustaste unverändert. Das Zeichnen eines Rechtecks zwischen Startposition als linker, oberer Ecke und aktueller Mausposition als rechter, unterer Ecke, geschieht nun innerhalb der Methode *draw()*. Da diese Methode immer wieder ausgeführt wird, passt sich das Rechteck kontinuierlich an die aktuelle Position der Maus an. Damit nur Rechtecke gezeichnet werden, solange der Anwender die Maustaste gedrückt hält, fragen wir in Zeile 9 mithilfe der Systemvariablen `mousePressed` noch ab, ob die Maustaste gerade gedrückt wird. Nur dann wird der Befehl zum Zeichnen des Rechtecks in Zeile 10 ausgeführt. Die Breite und Höhe des Rechtecks ergeben sich dabei aus dem Betrag der Differenz zwischen der x-Koordinate der Startposition (`x1`) und der x-Koordinate der aktuellen Position (`mouseX`) bzw. der y-Koordinate der Startposition (`y1`) und der y-Koordinate der aktuellen Position (`mouseY`). Abbildung 5 veranschaulicht diese Rechnung an einem Beispiel.

```
1  int x1;
2  int y1;
3  void setup(){
4    size(400, 300);
5    background(255, 255, 0);
6    fill(255, 0, 0);
7  }
8  void draw(){
9    if(mousePressed){
10     rect(x1, y1, abs(mouseX - x1),
       abs(mouseY-y1));
11   }
12 }
13 void mousePressed(){
14   x1 = mouseX;
15   y1 = mouseY;
16 }
```

Beispiel 7: Aufziehen von Rechtecken

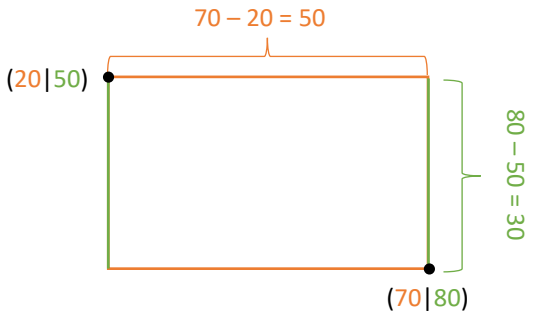

Abbildung 5: Berechnung der Höhe und Breite eines Rechtecks anhand der Koordinaten zweier gegenüberliegender Ecken

Aufgabe 11: Testen Sie, wie sich das Verhalten des Programms verändert, wenn Sie das Zeichnen des Rechtecks in die Methode *mouseReleased()* verlagern und die Methode *draw()* leer bleibt.

Aufgabe 12: Ändern Sie das Programm aus Beispiel 7 so, dass der Anwender einen Kreis aufziehen kann. Die Startposition legt dabei den Mittelpunkt des Kreises fest und die aktuelle Position der Maus, soll sich immer am Kreisrand befinden.

Lokale und globale Variablen

Vielleicht ist Ihnen aufgefallen, dass wir die Variablen in Beispiel 7 außerhalb der Methoden in den ersten beiden Zeilen des Programms definiert haben, während wir die Variablen in den vorherigen Beispielen innerhalb der Methoden, dort wo wir sie benötigen, angelegt haben. Es stellt sich somit die Frage nach dem Unterschied.

Die Werte der Variablen x1 und x2 in Beispiel 2 werden in der Methode *mousePressed()* verändert und in der Methode *draw()* verwendet. Die Werte der Variablen werden also nicht nur innerhalb eines Methodendruchlaufs, sondern während der gesamten Laufzeit des Programms benötigt. Solche Variablen bezeichnet man als **globale Variablen**. Globale Variablen müssen außerhalb der Methoden definiert werden. Ob sie oben, unten oder zwischen den Methoden stehen, ist dabei eigentlich egal. Damit wir den Überblick leichter behalten, sollten wir uns aber angewöhnen sie immer an den Anfang des Programms zu setzen.

Eine Variable, die innerhalb einer Methode definiert wird, bezeichnet man hingegen als **lokale Variable**. Diese steht nur solange zur Verfügung, bis die Ausführung der Methode beendet ist. Danach wird der Speicher für andere Zwecke wieder freigegeben. Eine solche lokale Variable haben wir in den Beispielen 2 und 3 als Zählvariable für die Schleifen verwendet. Wenn wir den Quellcode aus Beispiel 2 in die setup-Methode einfügen, steht die Variable `i` nach dem Beenden der setup-Methode nicht mehr zur Verfügung. In Beispiel 3 steht die Variable `i` bereits nach dem Beenden der for-Schleife nicht mehr zur Verfügung, da sie speziell für die Schleife definiert wurde.

Da Variablen, die nicht mehr benötigt werden, unnötig Speicherplatz blockieren, sollte man beim Entwerfen eines Programms für jede Variable genau überlegen, ob eine Definition als globale oder als lokale Variable sinnvoller ist.

Aufgabe 13: Im Ordner zu Aufgabe 13 finden Sie fünf kurze Programmbeispiele. Beispiel 1a und 1b sowie 2a und 2b unterscheiden sich jeweils in der Definition der verwendeten Variable als globale bzw. lokale Variable.

a) Erläutern Sie, wie sich die Verwendung einer lokalen bzw. globalen Variablen auf das Verhalten des Programms auswirkt.
b) Begründen Sie für Beispiel 1 und 2 jeweils, ob die Verwendung einer lokalen oder einer globalen Variablen sinnvoller ist.
c) Erläutern Sie, weshalb in Beispiel 3 alle gezeichneten Quadrate die gleiche Größe haben. Verändern Sie das Programm so, dass die Quadrate jeweils um 10 Einheiten größer werden.

Simulation von Buttons

In Aufgabe 10, 11 und 12 müssen die Farbe und die Dicke der gezeichneten Linie vom Programmierer festgelegt werden. Da wir auf Mausklicks reagieren können, können wir aber auch die Funktion von Buttons simulieren, um dem Anwender weitere Eingaben zu ermöglichen.
Um einen Button zu simulieren, können wir einfach ein Rechteck zeichnen und mithilfe der Koordinaten des Rechtecks und der Maus überprüfen, ob ein Mausklick innerhalb des Button-Rechtecks erfolgt ist. Um zwischen den Linienfarben Schwarz, Rot und Blau zu wählen, könnten wir beispielsweise einfach ein schwarzes, ein rotes und ein blaues Rechteck zeichnen, das der Anwender zur Auswahl der jeweiligen Farbe anklicken kann.
Um die Stiftdicke über einen Button zu verringern oder zu erhöhen, wäre aber vielleicht eine Beschriftung der Rechtecke hilfreich. Dazu steht uns die Methode *text()* zur Verfügung. Dieser

Methode übergeben wir einen Text und eine Position, an der dieser Text erscheinen soll. Als Schriftfarbe wird die Farbe, die mit der Methode *fill()* festgelegt wurde, verwendet. Die Schriftgröße kann mit der Methode *textSize()* festgelegt werden. In Beispiel 8 wird in der Methode *setup()* ein graues Rechteck mit der Beschriftung „dicker“ erzeugt.

```
1  void setup(){
2    size(400, 300);
3    background(255, 255, 0);
4    fill(200, 200, 200);
5    rect(10, 10, 50, 30);
6    fill(0,0,0);
7    text("dicker", 15, 30);
8  }
```

Beispiel 8: Zeichnen eines Buttons

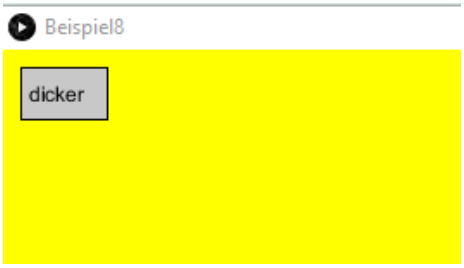

Abbildung 6: Programmfenster zu Beispiel 8

Aufgabe 14*: Erweitern Sie Ihr Programm aus Aufgabe 10 um zwei Buttons, mit denen man die Dicke der gezeichneten Linie regulieren kann. Erzeugen Sie dazu zwei Rechtecke mit geeigneter Beschriftung. Beim Anklicken des einen Rechtecks soll die Dicke der Linie erhöht, beim Anklicken des anderen Rechtecks die Dicke der Linie verringert werden.
Vorsicht: Achten Sie darauf, dass die Dicke nicht negativ wird.

Eingabe über die Tastatur

Die Eingabe über Buttons ist für den Anwender sehr komfortabel für den Programmierer jedoch relativ aufwändig. Manchmal reicht es daher aus, wenn der Anwender Eingaben über die Tastatur vornehmen kann. Im Folgenden schauen wir uns daher an, wie wir auf das Drücken einer Taste auf der Tastatur reagieren können.
Wir haben in den Aufgaben 11 und 12 zwei verschiedene Programme erstellt, eins zum Zeichen von Rechtecken und eins zum Zeichnen von Kreisen. Schöner wäre es, wenn wir nur ein Programm hätten, in dem der Anwender wählen kann, was er gerade Zeichnen möchte. Die Auswahl soll über die Tasten 'r' für Rechteck bzw. 'k' für Kreis erfolgen. Dazu benötigen wir die Methode *keyPressed()*, die immer dann einmalig ausgeführt wird, wenn eine Taste auf der Tastatur gedrückt wurde. Alternativ können wir die Methode *keyTyped()* verwenden. Anders als keyPressed() wird die Methode keyTyped() nicht ausgeführt, wenn Tasten gedrückt werden, die Steuerungsfunktionen übernehmen, also keine Zeichen codieren. Außerdem benötigen wir die Systemvariable `key`, die das Zeichen der zuletzt gedrückten Taste enthält.
Das Programm in Beispiel 9 speichert die zuletzt gedrückte Taste in der Variablen `taste`, indem in Zeile 23 innerhalb der Methode *keyPressed()* der Variablen `taste` der Wert der Systemvariablen `key` zugewiesen wird. Innerhalb der Methode *draw()* wird nun ein Rechteck gezeichnet, wenn die Bedingung in Zeile 10 erfüllt ist, also der Anwender vorher ein 'r' für Rechteck gedrückt hat, oder ein Kreis, wenn die Bedingung in Zeile 13 erfüllt ist, weil der Anwender vorher ein 'k' für Kreis gedrückt hat.

```
1  int x1, y1;
2  char taste = 'x';
3  void setup(){
4   size(400, 300);
5   background(255, 255, 0);
6   fill(255, 0, 0);
7  }
8  void draw(){
9   if(mousePressed){
10   if(taste == 'r'){
11    rect(x1, y1, abs(mouseX - x1), abs(mouseY-y1));
12   }
13   if(taste == 'k'){
14    circle(x1, y1, 2 * sqrt(((mouseY-y1)*(mouseY-y1)) +
                              ((mouseX-x1)*(mouseX-x1))));
15   }
16  }
17 }
18 void mousePressed(){
19  x1 = mouseX;
20  y1 = mouseY;
21 }
22 void keyPressed(){
23  taste = key;
24 }
```

Beispiel 9: Reaktion auf Tastatureingaben

Das Programm in Beispiel 9 würde auch funktionieren, wenn wir in den Bedingungen innerhalb der Methode *draw()* direkt die Systemvariable `key` abfragen würden. Das Vorgehen, in der Methode *keyPressed()* den Wert zunächst in einer Variablen zu speichern, ist jedoch hilfreich, wenn wir dem Anwender noch andere Eingaben über die Tastatur erlauben wollen. So könnten wir dem Anwender zusätzlich die Steuerung der Dicke der Umrandung über die Tastatur anbieten.
Hierfür gibt es verschiedene Möglichkeiten. Wir könnten z. B. fünf Werte für die Dicke zur Verfügung stellen, die mithilfe der Tasten 1 bis 5 direkt ausgewählt werden. Wir könnten die Dicke aber auch ähnlich wie bei den beiden Buttons bei Drücken der Plus-Taste um 1 erhöhen und bei Drücken der Minus-Taste um 1 verringern, sofern sie aktuell größer als 1 ist. Wir schauen uns erst einmal die Umsetzung der zweiten Variante an.
Wir benötigen eine globale Variable, in der wir uns die aktuelle Dicke merken. Diese ist zu Beginn 1, deshalb weisen wir der Variablen `dicke` in Zeile 1, den Wert 1 zu. Schauen wir uns zunächst die linke Version der Methode *keyPressed()* in Beispiel 10 an. Wir ergänzen drei Bedingungen in Zeile 4, Zeile 7 und Zeile 11. Die gedrückte Taste speichern wir nur noch in der Variablen `taste`, wenn es sich um die Taste ‚k' oder ‚r' handelt. Wenn die Plus-Taste gedrückt wurde, wird die Variable `dicke` um 1 erhöht und die Methode *strokeWeight()* zum Setzen der Dicke der Umrandung mit dem neuen Wert als Parameter ausgeführt. Wenn die Minus-Taste gedrückt wurde und die Variable `dicke` nicht schon beim Wert 1 angelangt ist, wird der Wert um 1 verringert und ebenfalls die Methode *strokeWeight()* mit dem aktuellen Wert ausgeführt.

```
1   int dicke = 1;
2   void keyPressed(){
3
4     if(key == 'k' || key == 'r'){
5       taste = key;
6     }
7     if(key == '+'){
8       dicke++;
9       strokeWeight(dicke);
10    }
11    if(key == '-' && dicke > 1){
12      dicke --;
13      strokeWeight(dicke);
14    }
15  }
```

Beispiel 10: Unterscheiden der Tastatureingaben mithilfe von if-Bedingungen

```
1   int dicke = 1;
2   void keyPressed(){
3    switch(key){
4     case 'k':
5     case 'r': taste = key;
6               break;
7     case '+': dicke++;
8               strokeWeight(dicke);
9               break;
10
11    case '-': if(dicke > 1){
12               dicke --;
13               strokeWeight(dicke);
14              }
15              break;
16   }
17  }
```

Beispiel 11: Unterscheiden der Tastatureingaben mithilfe eines switch-case Blocks

Je mehr Tasten wir in der Methode *keyPressed()* berücksichtigen wollen, desto mehr Bedingungen kommen zusammen. Deshalb lohnt es sich, dafür eine etwas einfachere Schreibweise einzuführen, die in der Variante rechts in Beispiel 11 zu sehen ist. Hinter dem Befehl *switch* in Zeile 3 steht in Klammern die Variable, die in jeder Bedingung überprüft werden soll. Jeder Wert, mit dem die Variable verglichen werden soll, steht hinter dem Befehl *case*. Wir haben also den Fall, dass die Systemvariable `key` die Werte 'k', 'r', '+' oder '-' annimmt. Hinter dem Doppelpunkt stehen dann die Anweisungen, die in dem jeweiligen Fall ausgeführt werden sollen. Es fällt auf, dass in Zeile 4 für den Fall 'k' kein Befehl angegeben ist. Das liegt daran, dass ab dem Fall, der zutrifft, auch die Anweisungen, die bei allen nachfolgenden Fällen stehen, ausgeführt werden, bis die Anweisung `break` gefunden wird. Da für den Fall 'k' und 'r' die gleiche Aktion ausgeführt werden soll, muss diese nur einmal hinter dem zweiten Fall angegeben werden. Da bei '+' und '-' unterschiedliche Anweisungen ausgeführt werden sollen, muss in Zeile 9 und Zeile 15 die Anweisung `break` stehen.

Aufgabe 15: Erweitern Sie das Programm aus Beispiel 6, in dem der Anwender Quadrate per Mausklick zeichnen kann, so, dass er über Tasten z. B. die folgenden Optionen auswählen kann:

- Die Form, die gezeichnet werden soll
- Die Dicke der Umrandung
- Ob die Form gefüllt sein soll oder nicht
- Die Farbe der Füllung (hier ist es sinnvoll, z. B. fünf Farben zur Auswahl zu stellen und mit Tasten zu belegen)
- Die Größe der Figur

Eingabe über ein Eingabefeld

Spätestens bei der Wahl der Farbe oder Größe in Aufgabe 15 fällt auf, dass es relativ umständlich ist, komplexere Tastatureingaben, die aus mehr als einem Zeichen bestehen, mithilfe der Methode *keyPressed()* zu erfassen. Wie wir aus einzelnen Zeichen eine Zeichenkette aufbauen können, müssen wir uns erst noch anschauen. Aber man kann sich vorstellen, dass dies zum Erfassen einer Tastatureingabe, die aus mehr als einem Zeichen besteht, relativ mühselig ist. Wesentlich einfacher ist es, dem Anwender für längere Eingaben ein Eingabefeld anzubieten.

Schauen wir uns anhand von Beispiel 12 an, wie der Anwender über ein Eingabefeld die Größe des zu zeichnenden Quadrats eingeben kann. Um die entsprechende Methode zum Anzeigen des Eingabefeldes verwenden zu können, müssen wir in die erste Zeile unseres Programms den folgenden Befehl einfügen: `import static javax.swing.JOptionPane.*;`

Nun können wir in einer unserer Methoden z. B. *draw()*, *mouseClicked()* oder *keyPressed()* die Methode *showInputDialog()* aufrufen (s. Zeile 10). Als Parameter wird der Text übergeben, der dem Anwender als Erklärung oder Aufforderung zu dem Eingabefeld angezeigt werden soll. Der Rückgabewert der Methode ist die Eingabe des Anwenders als Zeichenkette. Wenn der Anwender eine Zahl eingeben soll, kann diese ggf. auch in eine Ganzzahl oder Gleitkommazahl umgewandelt werden. Dies geschieht in Zeile 10 mithilfe der Methode *Integer.parseInt()*. Die Breite, die der Anwender eingegeben hat, wird dann in Zeile 11 in der Methode zum Zeichnen des Quadrates verwendet.

```
1  import static javax.swing.JOptionPane.*;
2  void setup(){
3    size(400, 300);
4    background(255, 255, 0);
5    fill(255, 0, 0);
6  }
7  void draw(){
8  }
9  void mouseClicked(){
10    int breite = Integer.parseInt(showInputDialog("Bitte gib die Breite
                    des Quadrates ein!"));
11   square(mouseX, mouseY, breite);
12 }
```

Beispiel 12: Benutzereingaben über ein Textfeld

Aufgabe 16: Erweitern Sie eines Ihrer Programme aus Aufgabe 9 so, dass der Anwender über ein Eingabefeld die Anzahl der Quadrate bzw. Kreise bestimmen kann, aus denen der Turm bzw. die Raupe bestehen sollen.

Projekte

Sie haben nun verschiedene Konzepte kennengelernt. Verwenden Sie diese, um ein etwas umfangreicheres Programm selbständig zu erstellen.

Projektidee 1: Kombinieren Sie das Gelernte zu einem etwas komplexeren Zeichenprogramm, das weitere Funktionen enthält. Folgende Funktionen könnten beispielsweise noch ergänzt werden:

- Ein Radiergummi, mit dem Teile der Zeichenfläche wieder gelöscht werden können.
- Vollständiges Löschen der bisherigen Zeichnung
- Weitere Buttons zur Auswahl der Farben, Formen oder anderer Optionen
- Mit der Maus angeklickte Positionen werden mit Linien verbunden, so dass ein Polygon entsteht.

Projektidee 2: Verwenden Sie die Zeichenfunktionen von Processing, um ein Pong-Spiel zu erstellen. Zwei schwarze Rechtecke, dienen als Schläger. Der linke Schläger kann z. B. mithilfe zweier Tasten auf der Tastatur hoch und runter bewegt werden, der linke Schläger mithilfe der linken und der rechten Maustaste. Die Spieler müssen die Schläger so positionieren, dass der Ball vom Schläger abprallen kann. Fliegt der Ball aus dem linken oder rechten Spielfeldrand, hat der jeweilige Spieler, der den Ball nicht aufhalten konnte, verloren. Am oberen und unteren Rand prallt der Ball ab.

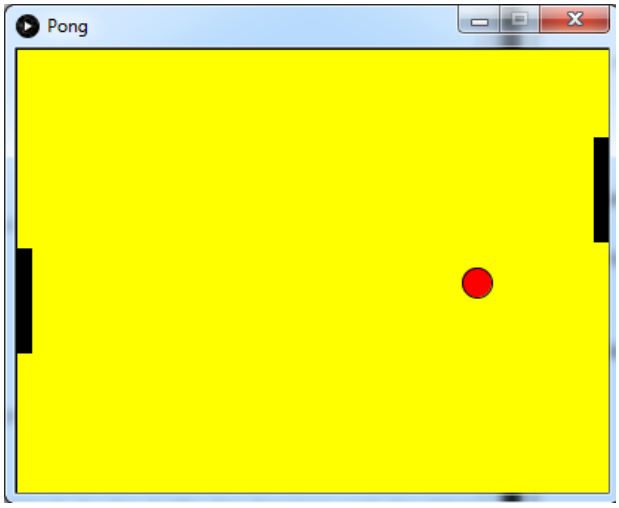

Abbildung 7: Exemplarisches Programmfenster zur Projektidee Pong-Spiel

Ergänzen Sie Ihr Spiel um weitere Funktionalitäten. Wünschenswert wäre beispielsweise eine Anzeige des Punktestands.
Hinweis: Damit der Ball nicht flackert und sich nicht zu schnell bewegt, kann es hilfreich sein mithilfe der Methode *frameRate()* die Anzahl der pro Sekunde gezeichneten Bilder herunterzusetzen. Probieren Sie z. B. `frameRate(4);` Diese Anweisung muss einmalig zu Beginn in der Methode *setup()* ausgeführt werden.

Projektidee 3: Gestalten Sie ein Quiz mit mehreren Fragen nach dem Prinzip von „Wer wird Millionär?“. Dem Anwender wird jeweils eine Frage mit 4 Antwortmöglichkeiten angezeigt. Die Auswahl der Antwort kann über Buttons oder über die Tastatur erfolgen.

Lizenz