SEKTOR 03
PLATZ 0388

Brückenkurs Informatik

höhere Programmiersprache
[Java / Processing]

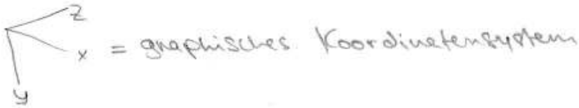
Übersetzung

Maschinensprache
[0110010011]

Manche Programmiersprachen sind der Maschinensprache näher als andere (Abstraktions-Niveau)

Processing:

IDE = Integrated developement environment



= graphisches Koordinatensystem

Funktion

kleines Programm

size (300, 450); -> Syntax

Name der Funktion — Argumente

Man kann nicht alle Funktionen auswendig lernen und liest alles deshalb immer in der Dokumentation nach. (Reference)

Variable

benannte Speicherstelle -> teilweise festgesetzt wie PI, HALF_PIE oder width und height

Datentyp

Festkomma (Ganze Zahlen)

Gleitkomma (Kommazahlen)

Zeichen

Wahrheits / Boolsche Werte

-> die Deklaration:
int x;

legt Wertebereich fest
Repräsentationen
Welche Operationen erlaubt sind (mit true/false kann man nicht multiplizieren)

-> Umwandlung float -> int (casting / Abschneiden)

float x = 10,25;
int y;

y = int (x);

y = (int) x;

die Initialisierung:
Zuweisung x = 10;

Operatoren

+, -, %, /, *, =

Vergleichsoperatoren:

a < b
a > b
a ≤ b - a <= b
a ≥ b - a >= b
a = b - a == b
a ≠ b - a != b

AND	a && b	Sowohl a als auch b wahr
OR	a || b	Entweder a oder b wahr
XOR	a ^ b	Entweder a oder b unwahr
	!a	a unwahr

Verzweigungen (If this then that)

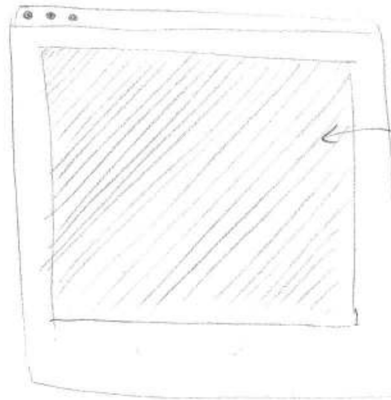
(Bei nur einer Anweisung können {} ausgelassen werden)

```
if (this is true) {
    then do that
}
```

→ Beispiel:

```
size (200, 200);
int zufall = int (random (10));

if (zufall > 5) {
    fill (100);
}

rect (20, 20, 160, 160);
```

random liefert floats bis 10



bei zufall = 9 wird es grau

bei zufall = 5 wird es weiß

Bei If-Verzweigungen

```
if
if
if
```

} nur das Zutreffende wird ausgeführt

Else Verzweigung

```
if (a > b) {
    statement 1
} else {
    statement 2
}
```

If - im If

```
if (a>b) {
    if (a>c) {
        statement } }
```

Ausführung gleich wie

```
if ((a>b) && if (a>c)) {
    Statement
}
```

Else-if-Verzweigung

```
if (test 1) {
    statement 1            -> Wenn Test = true dann statement 1
} else if (test 2) {       -> Ansonsten...
    statement 2
} else if (test 3) {       -> Ansonsten...
    statement 3
} else (test 4) {          -> Ansonsten...
    Statement 4 }
```

Schau ob Test 1 stimmt, ~~sonst schau ob test 2 stimmt~~

wenn stimmt → statement

sonst → schau ob test 2 stimmt

If Verzweigungen:

```
if (test) {
    Statement
}
```

wenn Test stimmt, führe Statement aus

Nur wenn if (test 1) {
und auch if (test) {
dann Statement
}
}

wenn Test 1 und Test 2 stimmen, führe Statement aus.

```
=> if ((test 1 == true) && (test 2 == true)) {
       Statement
   }
```

Entweder if (test 1) {
Statement }
Oder } else {
alternative Statement
}

wenn Test 1 stimmt führe Statement aus.
wenn alles andere als Test 1 stimmt führe nicht ~~test 1 sonst~~ Statement sondern alternative statement aus.

Entweder if (test 1) {
Statement 1
Oder } else if { (test 2) {
statement 2
oder } else {
alles andere }

wenn Test 1 stimmt führe statement1 aus.
wenn alles andere als Test 1 stimmt dann führe nicht statement 1 aus sondern schau ob test 2 stimmt.
wenn test 2 stimmt (das heißt test 1 wurde davor festgestellt stimmt nicht), dann führe statement 2 aus.

```
if (test 1) { statement }
if (test 2) { statement 2 }
```

← beide unabhängig voneinander

Wichtig zu beachten:
Es gibt keinen Unterschied zwischen und ...

```
if (test) {
    Statement
} else {
    Statement2
```

⟶

```
{ if (test) {
    Statement }
  if (! test {
    Statement }
```

Schleifen

Wiederholung mit kleinen Änderungen

```
for (init; test; update) {
    Statements
}
```

init = Initialisierung vor Start → Start
Wo beginnt die Schleife

Test = Abbruchkriterium → Ende
Bis wohin?

Update = Schleifeninkrement | Update → Update
Veränderung

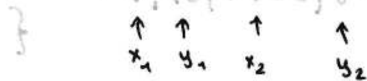
Beispiel: Zahlen von 0 bis 9 ausgeben

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

Ausgabefenster:

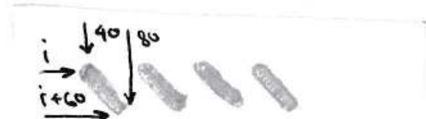
Beispiel: Animation

```
size (480, 120);
strokeWeight (8);
for (int i=20; i<400; i+=60) {
  line (i, 40, i+60, 80);
}
```



Start i = 20
Ende i = 399
Update i = +60

Sketch-Fenster:



Mapping Befehl

```
int m = int(map(i, 0, degree-1, 0, 255));
```

INPUT
OUTPUT

Formen von Endlosschleifen

```
while (1) {        oder while (true)
}
=
void draw () {
}
=
for ( ; true ; ) {   oder for( ;1; )
}
=
for ( ; ; ) {
}
```

For Schleifen:

```
for (init ; test ; update) {
    statements
}
```

initialisieren → int variable = 0

kontrollieren → variable ≤ 100
[0; 100]

verändern → variable ++

iteration → loop (~~veränderung~~) durch wiederholung

Beispiel:

```
int i=0;
for (i=0 ; i < 10 ; ~~i++~~){
    print (i + " ");
    i++; ← es macht keinen unterschied
}
println(i);
```

---

0 1 2 3 4 5 6 7 8 9 10

While Schleifen:

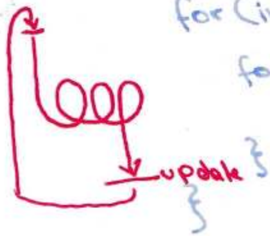
```
while (test) {
    statements
}
```

←

```
while (test){
    init
    ~~update~~ update

}
```

Bei doppelten For-Schleifen:

```
size (480,120);
background (0);
noStroke ();
fill (255, 140);
for (int [y=0] ①; y <= height ; [y += 40] ⑤) {
    for (int x=0; x <= width; x += 40) {
        ellipse (x, y, 40, 40);
    } ← update
}
```



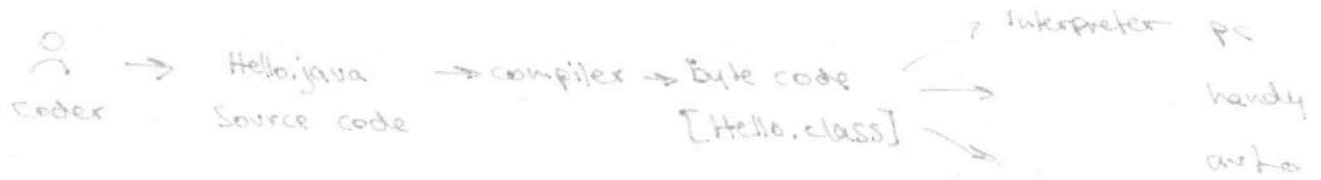
Step by step explaination -
Siehe debugger:
Update wird erst ausgeführt (+40) nachdem loop abgearbeitet wurde:

x	y
0	0
40	0
80	0
160	0
⋮	⋮
480	0
↻	1

Auch mehrere updates möglich:

```
int counter : 0;
for (int peter = 0; true ; peter ++, counter ++){
    statement;
}
```

Java von Oracle



Variablen

deklaration Datentyp ankündigen
initialisierung Wert festlegen

final int number = 100 → Konstante, darf nicht geändert werden

Konstanten

PI π
width Breite
height Höhe

→ Eine Konvention von Oracle:
Variablen: klein schreiben
Klassen: GROß SCHREIBEN

Kompatibilitätsbeziehungen → implizites Casting

- byte → short → int → long → float → double
- char → int

[Daten verschwinden]

Explizites Casting

(int) variable
int (variable)

Operatoren

Postinkrement a++
Präinkrement ++a

Postdekrement a--
Prädekrement --a

→ noch 1x normal verwenden, dann verändern

Beispiel:

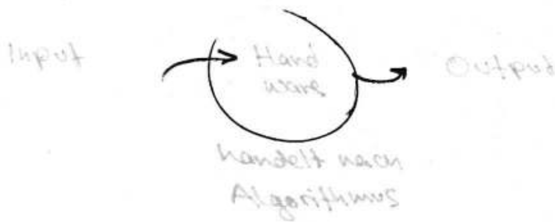
a = 5
b und c = 4

```
a = 3;
b = ++a;
c = a++;
println(a,b,c);
```

Algorithmen

Nöllenburg & Raidl

Algorithmen benötigen:



Analogie:
~~Ziegelst.~~
Kuchenrezept

Welche Eigenschaften muss ein Algorithmus haben?

- finite (endliche), präzise, ausführbare Schritte
- die nach endlicher Zeit zu einer Lösung führen
- kann nicht immer beliebige Eingaben verarbeiten

  Analogie: Wenn man aus Beton und Ziegelsteinen Kuchen backen will

- muss bei gleicher Eingabe ~~nicht~~ nicht immer zum gleichen Ergebnis führen und die gleichen Schrittfolgen ausführen (?)
- muss nicht jeden Schritt nach dem anderen ausführen (parallel processing)

möglicherweise:
Seed: - gleiche Eingabe
- Schrittfolge von Zufallsfaktoren beeinflusst

Algorithmen sind abstrakte Beschreibungen von Handlungsabläufen.
Algorithmen müssen nicht in Programmcode geschrieben werden. → auch Pseudocode

Algorithmus → Programme
Lösungsrezept — konkrete Umsetzung (Implementierung)

Kriterien bei der Entwicklung von Algorithmen:

Korrektheit der Lösung
Effizienz, niedrige Komplexitätsklasse
(Speicher und Zeitkomplexität) → Es muss für große Eingaben skalieren!

Effiziente Algorithmus Entwicklung → Effiziente Implementierung (Programmierung)

Komplexität bestimmt durch worst-case-Abschätzung.

Stabile Paare - Problem (Matching) / Marriage Problem
umgeändert auf „Schüler - Gastfamilien - Problem"

Alle Schüler und Gastfamilien haben eine eigene Liste an bevorzugten Partnern.

Wieviele mögliche Zuordnungen muss man in Betracht ziehen?

$O(n!)$ beziehungsweise brute-force

Alternative: Bieten mit Rückweisung

1. Schüler → Gastfam.: Antrag
2. Familien ✓ bestätigen die derzeitig besten Anfragen
3. Schüler ohne Partner → nächste Familie auf der Liste
4. Familien korrigieren wenn nötig weil sie bessere Schüler gefunden haben ↘ ist es stabil?

↻

$O(2n^2)$ - Gale-Shapely-Algorithmus

Das Acht-Damen Problem

Schach-Problem von Bezzel

Aufgabe:
8 Damen so auf Schachbrett setzen, dass sie sich nicht bedrohen

---

Das Rucksack-Problem

Dynamic-Programming

↑ Maximierung einer Summe (Wert)
↓ Minimierung einer anderen Summe (Volumen, Masse)

c_i = Wert

w_i = Volumen

Anweisung:

Maximiere $\sum_{i=1}^{n} c_i x_i$ unter der Bedingung, dass $\sum_{i=1}^{n} w_i x_i \leq K$

K: maximale Kapazität

Auf der Hauptplatine des Computers, dem Motherboard, befinden sich die wichtigsten Teile.

Diese kann mit der von-Neumann-Architektur modelliert werden:

Zentraleinheit
CPU – Control processing unit
führt Prozesse aus und besteht aus 2 Teilen:

- Steuerwerk
  CU – control unit
  steuert Ablauf der Programmbefehle

- Rechenwerk
  ALU – arithmetic logical unit
  elektronischer Rechner, arbeitet mit 1 und 0

Speicher
Memory
Beinhaltet alle Daten für CPU und Programme

- Arbeitsspeicher / Primärspeicher
  RAM – Random access memory
  schneller Speicher, flüchtig, ohne Stromversorgung gehen alle Daten verloren

- Sekundärspeicher / Externer Speicher
  hohe Speicherkapazität, langsamer, Daten bleiben ohne Stromanschluss erhalten

  Platz für Gespeichertes [zB.: CD, DVD, USB, SSD] und bereits installierte Programme.

Eingabe und Ausgabe
I/O Unit
Ermöglicht Interaktion mit Rechner
Interpretiert Tastatureingaben und Maus-Klicks

[Alles verbunden mit Bussystem]

Binäre Information

1 Spannung liegt an

0 Spannung liegt nicht an

Bit

Im binären Zahlensystem werden Bits hintereinander angereiht

n Bits ⇝ 2^n verschiedene Werte

2 Bits ⇝ 00, 01, 10, 11 } $2^2 = 4$

8 Bits werden zu einem Byte zusammengefasst. Damit gilt:

1 Byte = 8 Bits = 2^8 bzw 256 verschiedene Werte

Abstraktion

Das Abstrahieren = Reduzieren von redundanten Details

Man spricht von verschiedenen Abstraktionsebenen.

Hohe Abstraktionsebene : ↑ Übersicht ↓ Detail

Niedrige Abstraktionsebene : ↓ Übersicht ↑ Detail

Einfache Vorgänge wie das Kommunizieren im Alltag kann durch eine niedrige Abstraktion sehr komplex werden.

Gedanke formulieren
Neuronale Netze aktivieren

→ Entscheiden ob Tätigkeit ausgeführt wird

→ Person ansprechen

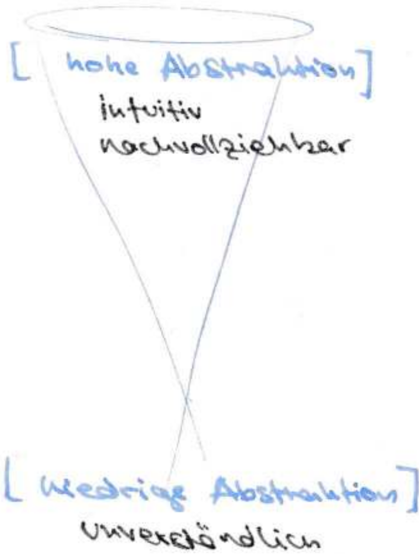
...

} = Frage stellen

Niedrige Abstraktionsebene
Geringe Abstraktion

Hohe Abstraktion

Abstraktionsebenen des Computers:



User-Anwendungen
Browser, Photoshop ...

Betriebssystem
Kommunikation zwischen Hardware-Komponenten

Hardware-Ebene
Alle Prozesse als 0, und 1.
Alle Spannungen

Algorithmen

Was ist ein Algorithmus?

Ein Algorithmus ist eine Handlungsvorschrift aus endlos vielen Schritten zur Lösung einer Aufgabe.

wie zB ein Kochrezept.
Im Kochrezept steht nicht wo man die Zutaten kaufen sollte und woher sie stammen sollten: Abstraktion

Ein Algorithmus ist die abstrakte Version eines Programmes.
Programme sind Ansammlungen von Algorithmen.

↳ Verschiedene Programmiersprachen = verschiedene Abstraktionsebenen, verschiedene Vor und Nachteile

Processing basiert auf JAVA, ist abstrakt und einfach erlernbar.

Processing

basiert auf JAVA, hohe Abstraktion
Umstieg auf Java, C++ und C# vereinfacht

Entwicklungsumgebung

Programme können in Texteditor geschrieben werden.
Die IDE (integrated development environment) vereinfacht das Programmieren.

zB.: Syntax Korrektur, bunt, Fehlersuche
schönes U/I

Syntax

Sprach-Grammatik → Syntax-Fehler = keine Ausführung möglich

Semantik

Bedeutung der Sprache → Wirkung des Programms
Semantikfehler → Programm reagiert anders als erwünscht

$a + b \cdot c \rightarrow (a+b) \cdot c$

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.

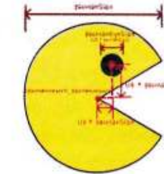
So könnte man z.B. einen zweiten, halb so großen Pacman zeichnen:

```
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 ½ 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 ⅛ mal der Durchmesser des Pacmans (pacManSize).
Der Mittelpunkt des Auges ist relativ zum Zentrum des Pacman-Körpers um ⅛ mal pacManSize nach rechts und ¼ 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*).

(cc) BY

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:

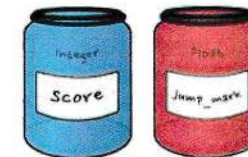
```
int score;
float jump_mark;
```

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, athlethe_height;
```

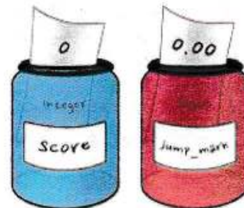
Die Datentypen `int` und `float` sowie weitere Datentypen für Variablen sind in Kapitel 4 im Detail beschrieben.

(cc) BY

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.



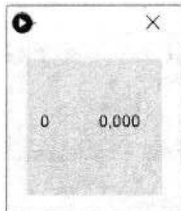
Initialisierte Variablen

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

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

Deklarierung Initialisierung

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:



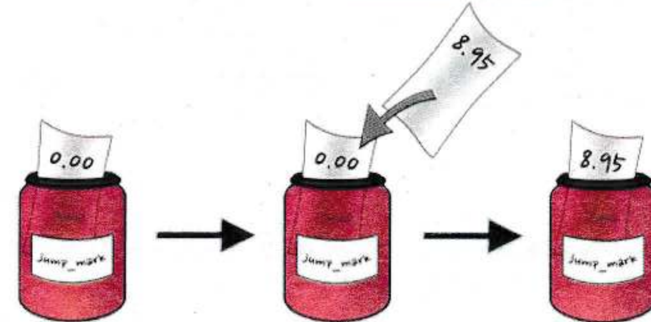

```
void setup() {
  fill( 0 );
  int score = 0;
  float jump_mark = 0.0;
  text( score, 10, 50 );
  text( jump_mark, 50, 50 );
}
```

(cc) BY TU Wien

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

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

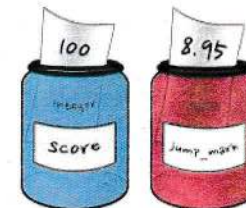
Der bisher gespeicherte Wert einer Variablen, (hier `0` für `score` bzw. `0.0` für `jump_mark`), wird mit dem neuen Wert überschrieben, das heißt, die Variable "verliert" ihren "alten" Wert. Daher wird diese Art der Zuweisung wird auch *destruktive Zuweisung* genannt.



Zuweisung eines neuen Werts an die Variable `jump_mark`

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

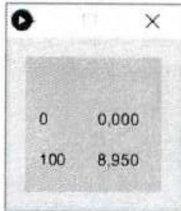
Variablenname	Wert
score	100
jump_mark	8.95



Werte der Variablen nach der Zuweisung

(cc) BY TU Wien

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

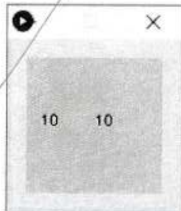



```
void setup() {
  fill( 0 );

  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;
```

(cc) BY TU Wien

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:

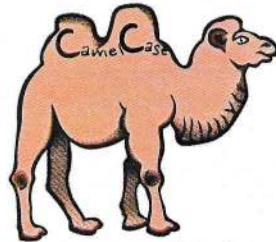
- Variablennamen dürfen nur aus Buchstaben, Ziffern oder Unterstrichen _ bestehen
- Sie müssen mit einem Buchstaben oder einem Unterstrich anfangen.
- Die Buchstaben ü, ä, ö und ß sind nicht erlaubt und führen zu einem Fehler.
- Groß- und Kleinschreibung werden unterschieden. Das bedeutet, dass etwa `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

(cc) BY TU Wien

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.

(cc) BY TU Wien

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;
```

(cc) BY TU Wien

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;
```

← Computer liest rückwärts!

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!

(cc) BY TU Wien

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
`x = x + y;`	`x += y;`
`x = x - 3;`	`x -= 3;`
`x = x * 5;`	`x *= 5;`
`x = x / 7;`	`x /= 7;`
`x = x % 2;`	`x %= 2;`

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:

$d = 4^2$

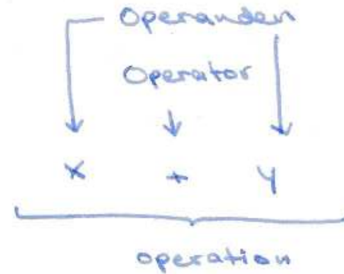
```
float d;
d = pow(4, 2);
```

$e = \sqrt{25}$

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

(cc) BY TU Wien

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;
```

(cc) BY TU Wien

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

(cc) BY TU Wien

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 );
}
```

(cc) BY TU Wien

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 geschwungenen Klammern sichtbar. Man sagt auch, dass sie nur **lokal sichtbar** sind. Nach den schließenden geschwungenen 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.

(cc) BY TU Wien

```
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.

(cc) BY TU Wien

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.

(cc) BY TU Wien

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 *printline* ausgesprochen). Ersetzen Sie `println()` durch `print()` und vergleichen Sie den Unterschied.

(cc) BY TU Wien

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, genügte 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
...
```

(cc) BY TU Wien

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 festellen, 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 );
}
```

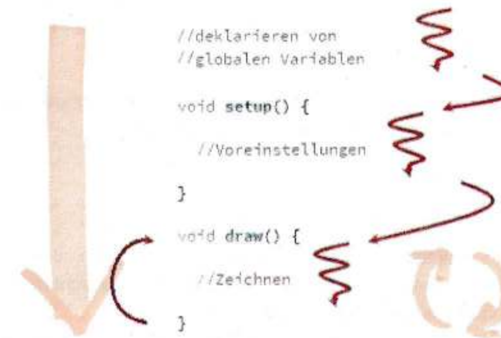
Durch diese automatische Wiederholung der Befehle im `draw()` Bereich ist es möglich Interaktionen im Programm zu gestalten. Denn würde der `draw()` Bereich nur ein einziges Mal ausgeführt werden, wäre es nicht möglich kontinuierlich auf Benutzereingaben oder Änderungen von Variablenwerten zu reagieren.

(cc) BY TU Wien

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 Bewegungen 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

(cc) BY TU Wien

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)

void setup() {
  size(200, 100);
}
 	
void draw() {
  int x = 0;
  ellipse(x, 50, 50, 50);
  x++;
}
	int x = 0;
	
	void setup() {
	  size(200, 100);
	}
	
	void draw() {
	  ellipse(x, 50, 50, 50);
	  x++;
	}

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++;
```

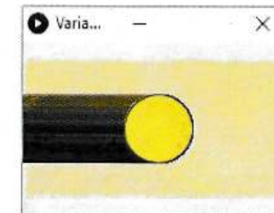
(cc) BY TU Wien


```
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	ellipse(0, 50, 50, 50);	1
2	1	ellipse(1, 50, 50, 50);	2
3	2	ellipse(2, 50, 50, 50);	3
4	3	ellipse(3, 50, 50, 50);	4
5	4	ellipse(4, 50, 50, 50);	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.

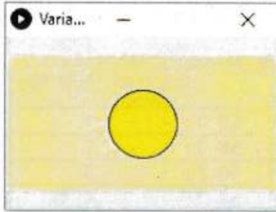



```
int xPosition = 0;

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 irgendwelche 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.

(cc) BY TU Wien




```
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 aktullen 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.

(cc) BY TU Wien

4. Datentypen und Operatoren

4.1. Datentypen Übersicht

Datentypen geben an, welche Art von Wert man in einer Variable abspeichern kann.
Eine Variable, die etwa mit `int` deklariert wurde, kann nur ganze Zahlen, aber niemals Kommazahlen oder anderweitige Zeichen halten.
Neben dem ganzzahligen Datentyp `int` und dem Datentyp `float` für Kommazahlen, die Sie bereits in Kapitel 3 kennengelernt haben, gibt es noch eine Reihe weiterer Datentypen. Im Folgenden sind die wichtigsten Typen in einer Tabelle zusammengefasst:

Punkt statt Komma

Datentyp	Deklaration und Initialisierung	Besonderheit
int	`int number;` `number = -42;`	Ganzzahl mit oder ohne Vorzeichen
float	`float decimal;` `decimal = -67.96f;`	Gleitkommazahl (mit Punkt), mit oder ohne Vorzeichen. Mit oder ohne f am Ende.
char	`char letter;` `letter = 'x';`	Ein Zeichen; Es wird begrenzt von Hochkommas geschrieben.
String	`String name;` `name = "Hans";`	Kein, ein oder mehrere Zeichen; Es wird bzw. sie werden begrenzt von Anführungszeichen (doppelten Hochkommas) geschrieben.
boolean	`boolean value;` `value = true;`	Wahrheitswert, es sind nur die beiden Werte `true` oder `false` zulässig.
color	`color yellow;` `yellow = color(127, 25, 0);`	Parameter des Befehls `color` müssen vom Typ `int` sein und zwischen 0 und 255 liegen.

Dies sind die in Processing gängigsten und am häufigsten verwendeten Datentypen. Daneben gibt es noch weitere Datentypen - diese werden in diesem Kurs allerdings nicht verwendet. Es wären dies unter anderem die Datentypen `byte`, `short`, `long` und `double`. Die ersten drei speichern, genauso wie `int`, ganzzahlige Werte, während der Datentyp `double` auch Kommazahlen speichert, genauso wie `float`. Diese Datentypen unterscheiden sich von `float` und `int` in ihrer Speicherkapazität, also wie groß die gespeicherten Zahlenwerte maximal bzw. wie klein sie minimal sein können.

(cc) BY TU Wien

Wenn Sie sich fragen, warum es so viele verschiedene Datentypen gibt und nicht nur ein einziger Datentyp verwendet wird: In manchen Programmiersprachen gibt es tatsächlich keine unterschiedlichen Datentypen, was sehr praktisch erscheint. Allerdings geht sehr leicht der Überblick verloren, welche Art von Daten in einer bestimmten Variable gespeichert ist, insbesondere auch wenn das Programm Fehler enthält. Unterschiedliche Datentypen für unterschiedliche Arten von Daten bietet uns Programmierern und Programmiererinnen zum einen Sicherheit und zum anderen eine verbesserte Lesbarkeit des Programms. Anhand der Deklaration können wir sicher sein, dass nicht zum Beispiel aus einer Zahl später plötzlich ein Buchstabe wird und dadurch ein Fehler im Programm hervorgerufen wird.

(cc) BY TU Wien

4.2. Binärsystem

Wie bereits erwähnt, "denkt" der Computer anders als wir Menschen. Wir Menschen lernen in unserer Kindheit das Dezimalsystem kennen. Wie der Name "Dezimal" bereits aussagt, handelt es sich um ein Zahlensystem mit 10 Zeichen, also den Ziffern 0 bis 9. Wir rechnen mit Zahlen, deren Ziffern aus 0 bis 9 bestehen. Sie zählen also 0, 1, 2..., 9, dann fügen wir die Zehnerstelle ein und die Einerstelle beginnt wieder von 0, etc.

Beim Computer sieht die Logik ganz ähnlich aus. Wie in Kapitel 1 bereits erwähnt, arbeitet der Computer im Hintergrund jedoch nur mit 0 und 1 (anstatt mit 0 bis 9). Alle Daten, die gespeichert werden, sind daher als eine Folge von 0 und 1 am Computer vorhanden und werden auch als 0 und 1 verarbeitet.

Man nennt dieses Zahlensystem des Computers, das nur aus den Ziffern 0 und 1 besteht, **das Binärsystem**. Auch hier steht der Teil "binär" für die Anzahl der Zeichen - nämlich für 2 Zeichen. Sie zählen 0, 1 und dann ist die Einerstelle voll und Sie müssen die nächste Stelle einfügen. Das heißt, die Zahl 3 im Dezimalsystem wäre 10 im Binärsystem, gesprochen Eins-Null.

Dezimal	Binär		Dezimal	Binär
0	0000		8	1000
1	0001		9	1001
2	0010		10	1010
3	0011		11	1011
4	0100		12	1100
5	0101		13	1101
6	0110		14	1110
7	0111		15	1111

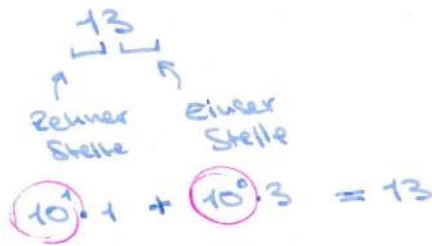
Anmerkung: Es gibt weitaus mehr Zahlensysteme als diese beiden - man muss nur eine andere Zahl als Basis verwenden. Das Dezimalsystem hat z.B. die Basis 10, weil es 10 Ziffern (0 - 9) gibt, und das Binärsystem hat Basis 2, weil es 2 Ziffern (0 und 1) gibt. Beispiele für weitere Zahlensysteme wären etwa das Oktalsystem (0 - 8) oder das Hexadezimalsystem (0 - 9 und zusätzlich A - F).

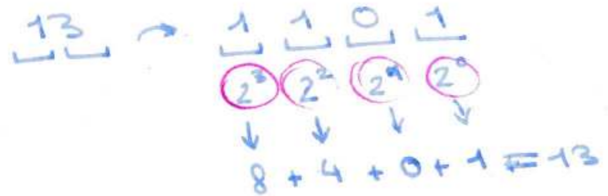
Zurück aber zum Binärsystem: Wie kann man nun mit 0 und 1 rechnen? Die Antwort ist simpel: Fast so, wie Sie es aus dem Dezimalsystem gewohnt sind. Um das Rechnen im Binärsystem und die interne Vorgehensweise des Computers besser nachvollziehen zu können, ist es nützlich, sich zunächst die Umrechnung von Zahlen zwischen Binär- und

(cc) BY TU Wien

Dezimal	→ Binär
13	1 1 0 1

Das Binärsystem besteht aus Potenzen von 2, während das Dezimalsystem aus Potenzen von 10 besteht





Um eine Dezimale Zahl in eine Binäre umzuwandeln könnte man hochzählen:

1. 001 4. 100 7. 111
2. 010 5. 101 8. 1000 ... 13. 1101
3. 011 6. 110 9. 1001

oder man könnte nachrechnen:

13 : 2 = 6 1R
6 : 2 = 3 0R
3 : 2 = 1 1R
1 : 2 = 0 1R

$13 = 2 \cdot 6$ 1R
$13 = 2 \cdot 2 \cdot 3$ 1R
$13 = 2 \cdot 2 \cdot 2$ 4R
$13 \neq 2 \cdot 2 \cdot 2 \cdot 2$

Man schaut eigentlich wie oft Zweierpotenzen in 13 hineinpassen würden

Dezimalsystem anzusehen. Denn dies führt der Computer auch erst intern aus, bevor er mit den Zahlen rechnen kann.

4.2.1. Vom Binärsystem ins Dezimalsystem umwandeln

Eine Zahl im Dezimalsystem kann durch ihre einzelnen Komponenten bzw. Stellen repräsentiert werden. Zum Beispiel können Sie die Zahl 735 wie folgt darstellen:

$$(735)_{10} = 700 + 30 + 5 = 7 * 100 + 3 * 10 + 5 * 1$$

bzw. mit Zehnerpotenzen:

$$(735)_{10} = 700 + 30 + 5 = 7 * 10^2 + 3 * 10^1 + 5 * 10^0$$

Jede Stelle kann also durch eine Potenz zur Basis 10 repräsentiert werden und die Summe ist die gewünschte Zahl. Die Einerstelle fängt dabei mit der Potenz 0 an (denn $10^0 = 1$) und für jede weitere Stelle, die links eingefügt wird, wird die Potenz um 1 erhöht.
Genauso lassen sich die Zahlen im Binärsystem auch durch Potenzen darstellen. Wenn Sie z.B. die Zahlen 1100 im Binärsystem gegeben haben, dann ist das nichts anderes als

$$(1100)_2 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 8 + 4 + 0 + 0 = 12$$

Statt der Basis 10 nehmen Sie beim Binärsystem die Basis 2 und gehen dann nach dem gleichen Schema wie im Dezimalsystem vor. Die Zahl 1100 im Binärsystem ist also im Dezimalsystem die Zahl 12.

4.2.2. Vom Dezimalsystem ins Binärsystem umwandeln

Die Umwandlung vom Binärsystem ins Dezimalsystem ist recht einfach. Umgekehrt ist es nicht viel komplizierter. Sie müssen mit einer Summe von Zweierpotenzen wieder auf die Dezimalzahl kommen.

Sehen wir uns aber zunächst das Ganze anhand des Dezimalsystems an. Die Zahl 735 besteht aus der Summe von drei Potenzen. Das können Sie natürlich sofort ablesen. Rechnerisch muss man es aber wie folgt angehen:

Rest kippen

735 / 10 = 73 + 5R
73 / 10 = 7 + 3R
7 / 10 = 0 + 7R

735

R steht hier für den Rest der ganzzahligen Division. Sie sehen, dass der Rest der ersten Division gleich der Einerstelle, der Rest der zweiten Division gleich die Zehnerstelle und der Rest der dritten Division gleich der Hunderterstelle ist. Das kann für größere Zahlen fortgesetzt werden. Das Ergebnis der Division wird für die nächste Division verwendet, bis das Ergebnis 0 ist. Wenn Sie die Reste von unten nach oben aneinanderreihen, dann erhalten Sie die ursprüngliche Zahl.

(cc) BY

Diese Division kann aber natürlich auch mit einer anderen Basis erfolgen um in ein anderes Zahlensystem umzuwandeln. Die Zahl 12 im Dezimalsystem kann man wie folgt ins Binärsystem umwandeln:

Rest kippen

12 / 2 = 6 + 0R
6 / 2 = 3 + 0R
3 / 2 = 1 + 1R
1 / 2 = 0 + 1R

1100

Statt durch 10, dividieren wir durch 2, da das Binärsystem die Basis 2 hat. Den Rest von unten nach oben aneinandergereiht ergibt die Zahl 1100 im Binärsystem und das Ergebnis stimmt mit dem vorigen Beispiel überein.

Online findet man auch einfache Umrechner. Z.B. unter dem Link
http://www.arndt-bruenner.de/mathe/scripts/Zahlensysteme.htm

4.2.3. Addieren im Binärsystem

Da Sie nun Zahlen zwischen Dezimalsystem und Binärsystem umwandeln können, können Sie nun auch im Binärsystem rechnen. Um Ihnen einen kleinen Einblick zu verschaffen, wie ein Computer rechnet, werden wir dies händisch an einem Beispiel ausführen.
Der Einfachheit halber werden Sie hier nur das Addieren lernen. Das Addieren ist genauso wie im Dezimalsystem. Addieren wir die Binärzahlen 1100 (12 in Dezimalsystem) und 100 (4 in Dezimalsystem):

1100 100 ------ 0	1100 100 ------ 00	1100 100 1 ------ 000	1100 100 11 ------ 0000	1100 100 11 ------ 10000

Wie gewohnt geht man bei der Addition von hinten nach vorne vor. Zuerst wird die erste Stelle addiert, dann die zweite Stelle. Bei der dritten Stelle haben wir einen Übertrag, denn 1 + 1 ist 10 im Binärsystem. Die dritte Stelle wird mit dem Übertrag addiert und wir erhalten einen weiteren Übertrag. An der fünften Stelle steht der Übertrag alleine und das Ergebnis dieser Addition ist 10000 im Binärsystem. Umgerechnet ins Dezimalsystem erhalten wir die Zahl 16.

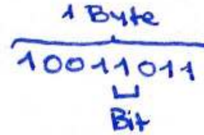
Online ist unter dem angeführten Link ein Binärrechner zu finden, der noch weitere Rechenoperationen als die einfache Addition enthält:
http://www.miniwebtool.com/binary-calculator/

4.2.4. Einheiten

Sowie es im Dezimalsystem Einheiten gibt, wie Tausend, Million, Milliarde, usw, gibt es im Binärsystem auch Einheiten, von denen Sie bestimmt schon gehört haben. In der

(cc) BY

Informatik redet man oft von Bits und bytes. Bit ist die Abkürzung für binary digit was nichts anderes heißt als Binärziffer. Ein Bit ist somit nichts anderes als eine Stelle im Binärsystem. Die Zahl 1100 im Binärsystem besteht also demnach aus 4 Bits. Mit 4 Bits kann man z.B. die Zahlen von 0 bis 15 darstellen. 8 Bits werden auch als 1 Byte bezeichnet.
Weitere Einheiten sind z.B. KiloByte (KB), MegaByte (MB), GigaByte (GB), etc.



Für Interessierte ist eine genauere Auflistung der Einheiten unter dem Link https://de.wikipedia.org/wiki/Byte#Vergleichstabelle zu finden.

4.2.5. Darstellung negativer ganzer Binärzahlen

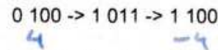
Mit dem bisher Gelernten können nur positive Zahlen dargestellt werden, aber keine negativen Zahlen. Im normalen Alltag würden Sie einfach ein Minuszeichen vor die Zahl schreiben und alle würden diese Zahl als eine negative Zahl interpretieren. Da aber der Computer nur mit 0 und 1 arbeitet und kein Minuszeichen kennt, muss dieses Minus durch 0 oder 1 ausgedrückt werden. Hierbei wird für jede Binärzahl ein zusätzliches Bit für das Vorzeichen gespeichert, eine 0 für ein Plus und eine 1 für ein Minus. Zum Beispiel für eine Maschine, die in 4 bit rechnet, stellen die Binärzahlen

0 100 und 1 100

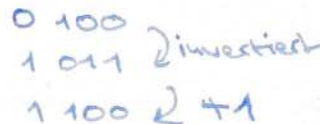
also die Zahlen 4 und -4 im Dezimalsystem, dar. Diese Darstellung wird auch **Vorzeichendarstellung** genannt. Problematisch ist diese Darstellung bei der Zahl 0, da diese dann zweideutig ist, nämlich einmal +0 und einmal -0. Außerdem lässt sich mit dieser Darstellung nicht so einfach rechnen wie vorhin angeführt. Daher wurde das Zweierkomplement eingeführt, das auch zur Darstellung negativer Zahlen in Processing verwendet wird.

Bei der **Zweierkomplementdarstellung** wird zwar wie bei der Vorzeichendarstellung das erste Bit als Vorzeichen verwendet, aber dafür die negativen Zahlen anders dargestellt. Für die positiven Zahlen ändert sich nichts, d.h. z.B. dass die Zahl 4 weiterhin als 0 100 dargestellt wird. Allerdings werden negative Zahlen nun etwas anders dargestellt. Die Zahl -4 wird im Zweierkomplement als 1 100 dargestellt. Um auf diese Zahl zu kommen, wird die positive Darstellung der Zahl invertiert, d.h. jede 1 wird zu 0 und jede 0 wird zu einer 1, und dann wird das Ergebnis um 1 erhöht. Die Zahl -4 folgt aus der Zahl 4 also wie folgt:

0 100 -> 1 011 -> 1 100



Durch diese Darstellung der negativen Zahlen kommt die 0 nur mehr einmal vor und wird mit 0 000 dargestellt. Negative Zahlen erkennt man an der 1 an der ersten Stelle. Mit 4 Bits lassen sich insgesamt $2^4 = 16$ Zahlen darstellen, nämlich von -8 bis 7, also -2^3 bis $+2^3 - 1$. Auch das Rechnen kann weiterhin fast wie vorhin gelernt durchgeführt werden.



(cc) BY TU Wien

4.3. Datentypen für Zahlen

4.3.1. Ganze Zahlen

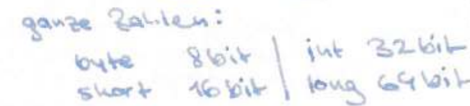
Jeder Datentyp, der Zahlen speichert, hat nur eine begrenzte Anzahl an Bits zum Speichern einer Zahl. Der ganzzahlige Datentyp `int` zum Beispiel hat 32 Bits. Das sind 2^{32} mögliche Zahlen, wenn man die Zweierkomplement Darstellung verwendet (siehe 4. Datentypen und Operatoren: Binärsystem). Der Wertebereich für ein `int` liegt im Bereich von

$$-2^{31} \ (-2.147.483.648) \text{ bis } 2^{31} - 1 \ (2.147.483.647).$$

Wird eine Zahl außerhalb des Bereichs zugewiesen, z.B. 2 147 483 648, erscheint eine Fehlermeldung:

The literal 2147483648 of type int is out of range

Auch die Datentypen `byte`, `short` und `long` speichern ganzzahlige Werte. `byte` besitzt 8 Bits, `short` verwendet 16 Bits und `long` hat 64 Bits.



4.3.2. Gleitkommazahlen

Der Datentyp `float` für Kommazahlen besitzt 32 Bits. Aber `float` verwendet nicht die Zweierkomplement Darstellung, sondern die Gleitkommadarstellung. Diese Darstellung ist etwas komplexer. Interessierte können die Funktionsweise der Darstellung auf https://de.wikipedia.org/wiki/Gleitkommazahl nachlesen. Wichtig ist, dass in dieser Darstellung, Zahlen nicht immer exakt dargestellt werden können. Ein `float` hat eine Darstellungsgenauigkeit von 7 signifikanten Dezimalstellen. Die achte Stelle ist üblicherweise nicht mehr genau.

Im folgenden Beispiel wird die Zahl Pi mit 15 Nachkommastellen einer `float` Variable zugewiesen.

```
void setup() {
  float piFloat = 3.141592653589793;
  print(piFloat);
}
```

Nach dem Ausführen des Programms erscheint in der Konsole nur die Zahl `3.1415927`. Die letzte Stelle wurde aufgerundet. Man beachte, dass auch die Vorkommastellen zu den signifikanten Stellen gezählt werden (sofern sie nicht 0 sind).

Es werden also immer nur die 7 signifikantesten (wichtigsten) Stellen einer Kommazahl dargestellt. Wenn eine Kommazahl mehr Stellen hat als darstellbar sind, werden die

(cc) BY TU Wien

hintersten Stellen nicht oder ungenau dargestellt. Denn es wird versucht, die Kommazahl mit den verfügbaren Bits noch so präzise wie möglich zu speichern.

Es gibt jedoch manche Zahlen, die unabhängig von der Anzahl der verfügbaren Bits, nicht genau dargestellt werden können und bereits nach den ersten 2 Dezimalstellen ungenau sind. Vergleichen Sie im folgenden Beispiel die Ausgabe der Variablen a und b:

```
void setup() {
  float a = 36.2;
  float b = 0.362 * 100;

  println(a);
  println(b);
}
```

Beide Male werden unterschiedliche Ergebnisse ausgegeben. Der Grund für dieses Verhalten liegt in der binären Darstellung der Zahlen. Nicht alle Zahlen sind im Binärsystem endlich darstellbar. Dadurch kommt es bei Operationen zu Rundungsfehlern. Das ist vergleichbar mit irrationalen oder periodischen Zahlen im Dezimalsystem, wie zum Beispiel der Zahl ⅓, die als Dezimalzahl 0.3 periodisch ist.

Ein float kann sowohl im negativen als auch im positiven Bereich die Zahlen

1.40239846E-45 ($1.40239846 * 10^{-45}$) bis 3.40282347E+38 ($3.40282347 * 10^{38}$)

darstellen (meistens bis zu 7 Dezimalzahlen genau). Wird versucht, eine Zahl außerhalb des Bereichs zuzuweisen, erscheint eine Fehlermeldung.

Ein weiterer Datentyp für Kommazahlen ist double. Der Datentyp double besitzt 64 Bit und hat eine Genauigkeit von etwa 15 Dezimalstellen.

Kommazahlen

float 32 bit 7 Dezimalstellen
double 64 bit 15 Dezimalstellen

int = 32 bit

↳ standard weil früher mehr 32 bit Systeme waren.
Java ist bisschen älter.

(cc) BY TU Wien

4.4. Datentypen char und String

4.4.1. Deklaration und Zuweisung

Neben Zahlen können auch Zeichen bzw. Zeichenketten gespeichert werden. Dafür werden die Datentypen char für einzelne Zeichen und String für ganze Zeichenketten verwendet. Hier sind ein paar Beispiele für die Datentypen und wie man ihnen Werte zuweist:

```
char m = 'M';
char questionMark = '?';
char four = '4';
char space = ' ';

String name = "Max";
String empty = "";
String a = "a";
String sentence = "Processing ist cool!";
```

In einem char Datentyp kann immer nur ein Zeichen gespeichert werden. Das Zeichen kann ein Buchstabe, Sonderzeichen, Zahl und auch sogenannte Steuerzeichen (siehe nächstes Unterkapitel) sein. Dieses Zeichen muss von einfachen Anführungszeichen (') umgeben sein.

Mit einem String können keine, eine oder auch mehrere Zeichen gespeichert werden. Intern in Processing sind sie als eine Verkettung von einzelnen chars dargestellt. Zeichenketten müssen von doppelten Anführungszeichen (") umgeben sein.

String und char Variablen können nach ihrer Initialisierung dann für Befehle verwendet werden, z.B. für den text() oder println() Befehl.

```
void setup() {
  size(170, 100);
  String sentence = "Processing ist cool!";
  fill(255, 100, 0);
  textSize(16);
  text(sentence, 10, 50);
}
```

(cc) BY TU Wien

4.4.2. Maskierung

Da Anführungszeichen für `String` und `char` eine bestimmte Bedeutung haben, kann man in einem `String` oder `char` nicht ohne weiteres ein Anführungszeichen selbst speichern. Zum Beispiel will man in einem Satz ein bestimmtes Wort in Anführungszeichen setzen:

```
String sentence = "Das nennt man "maskieren"! ";
```

Processing wird diese Zeile rot unterwellen, da das zweite doppelte Anführungszeichen das Ende des Strings (Zeichenkette) darstellt.

Damit man auch Anführungszeichen als eigenes Zeichen speichern kann, muss man das Zeichen zuerst **maskieren**, also ihre Bedeutung verschleiern. Das wird mit einem Backslash (\) bewerkstelligt. Dafür wird vor dem zu maskierenden Zeichen ein Backslash geschrieben.

```
String sentence = "Das nennt man \"maskieren\"! ";
```

Damit werden die beiden inneren Anführungszeichen als normale Zeichen ohne Bedeutung als Stringbegrenzer aufgefasst und können nun ohne Probleme in einem `String` gespeichert werden. Auch den Backslash selbst kann man maskieren, falls man ihn als ein normales Zeichen braucht.

```
String backslash = "\\";
```

In der Variable `backslash` ist nun ein Backslash gespeichert.

4.4.3. Steuerzeichen und Sonderzeichen

Der Backslash kann aber noch mehr. Mit einem Backslash können noch weitere sogenannte Steuerzeichen und Sonderzeichen eingeleitet werden.

Steuerzeichen sind keine am Bildschirm dargestellten Zeichen wie Buchstaben, Zahlen oder Satzzeichen, sondern besitzen bestimmte Bedeutungen. Das bekannteste Beispiel ist der Zeilenumbruch. Zeilenumbrüche wurden bis jetzt durch den Befehl `println()` erreicht. Nach jedem `println()` Befehl wird ein Zeilenumbruch eingefügt. Manchmal ist es aber wünschenswert, dass man auch mit nur einem `println()` mehrere Zeilenumbrüche erreicht. Oder in einem `text()` Befehl einen Zeilenumbruch hat. Zeilenumbrüche werden mit `"\n"` eingeleitet.

```
void setup() {
  size(170, 100);
  String sentence = "Processing \nist \ncool!";
  fill(255, 100, 0);
  textSize(16);
  text(sentence, 10, 30);
}
```

Wo ein Zeilenumbruch eingefügt werden soll, wird ein `\n` eingefügt. Alles direkt nach `\n` wird wieder als normaler Text interpretiert. Daher ist es kein Problem, wenn zwischen Text und dem `\n` kein Abstand ist.

Sonderzeichen sind alle Zeichen, die keine Buchstaben, Zahlen und Steuerzeichen sind. Das sind zum Beispiel Satzzeichen oder mathematische Symbole wie Plus und Minus. Aber auch chinesische Symbole, griechische Buchstaben oder das Copyright Symbol gehören dazu. Diese besonderen Zeichen können mit einem Backslash und ihrem Unicode dargestellt werden, falls sie nicht auf der Tastatur vorhanden sind. Unicode ist ein Standard, der jedem existierenden Zeichen oder Symbol einen eindeutigen Code zuweist. Eine Liste von Unicodes und ihre dazu gehörigen Symbole ist unter https://en.wikipedia.org/wiki/List_of_Unicode_characters zu finden.

```
void setup() {
  char smiley = '\u263A';
  String music = "\u266B";
  textSize(30);
  fill(255, 0, 0);
  text(music + " " + smiley, 20, 60);
}
```

In Processing sind die meisten Sonderzeichen nur über den Befehl `text()` anzeigbar. Falls man versucht mit dem `println()` Befehl diese Zeichen auszugeben, wird ein Fragezeichen angezeigt.

4.4.4. Konkatenation

Eine wichtige Operation mit den Datentypen `String` und `char` ist die **Konkatenation**, das Verbinden bzw. das Verketten von zwei Wörtern (bzw. Zeichen) zu einem Wort.

```
String name = "Hans" + " " + "GuckindieLuft";
```

Der Operator der Konkatenation ist das Plus (+), welches bereits aus der Addition bekannt ist. Die Variable `name` enthält also den String `"Hans GuckindieLuft"`. Der Plus-Operator

hat somit zwei Bedeutungen. Sind beide Operanden des Plus-Operators Zahlen, so werden die Zahlen addiert. Ist hingegen eines der Operanden ein String, dann wird daraus eine Konkatenation. Das Ergebnis einer Konkatenation ist immer vom Typ String.

Hier sind ein paar Möglichkeiten, wie und in welchen Kombinationen Konkatenation verwendet werden kann:

```
void setup() {
  String name = "Hans" + " " + "GuckindieLuft";
  int age = 23;
  char point = '.';
  String sentence = "Ich bin " + name + ".\n";
  sentence = sentence + "Ich bin " + age + " Jahre alt";
  sentence += point;
  println(sentence);
}
```

Beim Ausführen des Programmcodes erhält man in der Konsole die Ausgabe:

Ich bin Hans GuckindieLuft.
Ich bin 23 jahre alt.

Die Konkatenation mit Leerzeichen kann dafür genutzt werden, um eben Wörter voneinander zu trennen, aber auch, um einfache Einrückungen in der Konsolenausgabe umzusetzen.
Man sieht, dass nicht nur `String`- und `char`-Variablen mit dem Operator + konkateniert (bzw. verkettet) werden können, sondern auch Zahlen und andere Datentypen wie z.B. boolean. Diese Datentypen werden zuerst in ihre String-Repräsentation umgewandelt (hier z.B. bei der Verwendung der Variablen `point` und `age`). Jeder Datentyp besitzt eine String-Repräsentation, die Repräsentation muss aber nicht intuitiv sein, wie sie es bei Zahlen oder Zeichen ist. Das ist zum Beispiel beim Datentyp `color` der Fall (siehe Kapitel 4.6). Will man diesen Datentyp als String ausgeben, erhält man negative Zahlen im Bereich -16777216 und -1. Man kommt auf diesen Wertebereich, weil man pro Farbkanal 8 Bit verwendet. Insgesamt sind es also 24 Bit und 2^{24} ist 16777216. Die String-Repräsentation für den Datentyp `color` ist also eine Zahl, die aus den Werten der einzelnen Komponenten der RGB Farben berechnet wird.

Bei der Konkatenation mit Zahlen muss man jedoch aufpassen, wenn man zwei Zahlen konkatenieren will und nicht addieren. Wie schon bei den arithmetischen Operatoren werden auch Konkatenationen von links nach rechts betrachtet.

```
int a = 19;
int b = 70;
println("Das Jahr " + a + b);
```

Dieses Beispiel liefert die Ausgabe: `Das Jahr 1970`

(cc) BY TU Wien

Von links nach rechts betrachtet, ist das erste Plus eine Konkatenation (`"Das Jahr " + a`). Das Ergebnis davon ist wieder ein String und wird dann mit b konkateniert.
Das folgende Beispiel liefert jedoch die Ausgabe: `89er`

```
int a = 19;
int b = 70;
println(a + b + "er");
```

Das liegt daran, dass von links nach rechts das erste Plus als Addition aufgefasst wird, weil beide Operanden (jeweils links und rechts vom Operationszeichen) eine Zahl sind. Erst das zweite + Zeichen wird von Processing als Konkatenation aufgefasst, da einer der Operanden ein String ist. Processing entscheidet also automatisch anhand der Datentypen der Operanden, ob eine Addition oder eine Konkatenation durchgeführt wird.

Will man aber im ersten Beispiel nicht die Konkatenation beider Zahlen sondern mit Absicht die Summe, so reicht es, die gewünschte Addition zu klammern:

```
int a = 19;
int b = 70;
println("Das Jahr " + (a + b));
```

(cc) BY TU Wien

4.5. Datentyp boolean

Ein weiterer wichtiger Datentyp ist `boolean`. Das ist ein sogenannter **logischer Datentyp** und steht für einen Wahrheitswert. Er kann nur die Werte `true` oder `false` annehmen.

Variablen vom Datentyp boolean werden verwendet, wann immer eine Frage mit Ja oder Nein bzw. Richtig oder Falsch beantwortet werden soll. Dies ist insbesondere bei Entscheidungen oder bei Vergleichen der Fall.

Will man zum Beispiel zwei Werte vergleichen und wissen, ob der eine Wert größer ist als der andere, kann man das folgendermaßen schreiben

```
void setup() {
  int alice = 15;
  int bob = 17;
  boolean answer = alice > bob;
  println(answer);
}
```

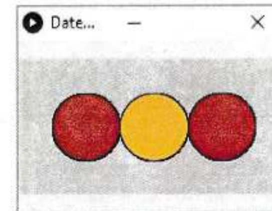
In diesem Beispiel werden zwei Integer-Werte angelegt, die das Alter der zwei Personen Alice und Bob repräsentieren sollen. Dann wird abgefragt, ob Alice älter als Bob ist bzw. ob 15 größer als 17 ist. Diese Antwort wird in der Variable `answer` gespeichert. In der Konsole wird dann diese Variable ausgegeben und `false` wird angezeigt, da Alice nicht älter als Bob ist. Wird das Größer-Zeichen durch ein Kleiner-Zeichen ersetzt, wird in der Konsole `true` ausgegeben, da die Frage mit "Ja" beantwortet werden kann.

Der Datentyp `boolean` wird bei den Verzweigungen (Kapitel 5) noch genauer behandelt, da er da eine ganz große Rolle spielt.

(cc) BY TU Wien

4.6. Datentyp color

Alle bisherigen Datentypen sind in vielen anderen Programmiersprachen auch vertreten. Der Datentyp `color` ist allerdings eine Eigenheit von Processing. Da Processing besonders auf die graphische Ausgabe spezialisiert ist, kommen meistens viele verschiedene Farben vor. Damit man nicht jedes Mal aufs Neue die RGB Werte (rot-grün-blau) eintippen muss (z.B. für den Befehl `fill()`) oder pro Farbkanal eine Variable angelegt werden muss, können die RGB Werte in einem Datentyp zusammengefasst werden und wiederverwendet werden.




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

void draw() {
              // color(R, G, B);
  color red = color(200, 0, 0);
  color yellow = color(250, 200, 0);
  int yCoord = 50;
  int diameter = 50;

  fill(red);
  ellipse(50, yCoord, diameter, diameter);
  fill(yellow);
  ellipse(100, yCoord, diameter, diameter);
  fill(red);
  ellipse(150, yCoord, diameter, diameter);
}
```

(cc) BY TU Wien

4.7. Casting

4.7.1. Zahlen-Datentypen nach Größe

Wenn man die Datentypen für Zahlen nach der Größe ihres Wertebereichs ordnet, kommt man zu der folgenden Beziehung:

```
int < float
```

Mit einem `int` können rund 4.2 Mrd. verschiedene ganze Zahlen dargestellt werden, die ca. im Bereich -2.1 Mrd. bis +2.1 Mrd. liegen. Der Wertebereich von `float` ist aber viel größer und enthält den von `int` zur Gänze, auch wenn die Genauigkeit der Zahlen oftmals darunter leidet. Daher kann jede Zahl, die in einer `int`-Variablen gespeichert werden kann, auch in einer `float`-Variablen abgespeichert werden, aber nicht umgekehrt. `float` ist also der größere Typ, `int` der kleinere.

4.7.2. Casting

Unter Casting versteht man die Umwandlung eines Datentyps in einen anderen Datentyp. Das Casting selbst lässt sich wieder in zwei Unterarten unterteilen: Einerseits das **implizite Casting** bei dem ein kleinerer Datentyp in einen größeren umgewandelt wird und andererseits **explizites Casting** bei dem ein größerer Datentyp in einen kleineren umgewandelt wird und dabei Information verloren gehen kann.

Implizites Casting

Implizites Casting ist eine Typumwandlung, die, falls es notwendig ist, während der Ausführung einer Operation automatisch gemacht wird, d.h. man muss es nicht extra hinschreiben, Processing macht es automatisch. Im vorigen Kapitel haben wir bereits die folgenden Operatoren mit zwei Operanden kennengelernt:

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

Jede Operation, außer dem Zuweisungs-Zeichen (=), hat ein Ergebnis. Betrachten wir zunächst alle Operanden außer der Zuweisung. Bei jedem Operanden kann es zu implizitem Casting kommen, deswegen ist es wichtig sich Gedanken darüber zu machen welchen Typ das Ergebnis hat. Es gibt zwei Fälle, die man unterscheiden muss:

(cc) BY

1) Beide Operanden sind vom gleichen Typ:
   Das Ergebnis hat auch den gleichen Typ
2) Die Operanden haben unterschiedliche Typen:
   Das Ergebnis hat den Typ des größeren Typs. → implizites Casting
   klein → groß

Beispiel: Betrachten wir nun folgenden Codeabschnitt

```
int height = 183;
float weight = 71.5;
result = weight / (height * height);
```

Welchen Datentyp hat nun der letzte Ausdruck? Welchen Typ sollte `result` haben?

Um das festzustellen werten wir den arithmetischen Ausdruck aus. Die erste Operation ist die Multiplikation (*), beide Operanden `height` haben den Typ `int`, daraus folgt nach 1), dass das Ergebnis den Typ `int` hat. Bei der zweiten Operation, der Division (/) hat der erste Operand den Typ `float` und der zweite den Typ `int`, nach 2) kommen wir zum Schluss, dass das Ergebnis des Ausdrucks den Typ `float` hat. Die Variable `result` sollte also den Typ `float` haben.

Bei der Zuweisung muss der Typ des Ausdrucks auf der rechten Seite entweder gleich oder kleiner sein, als der Typ der Variablen auf der linken Seite. Sind beide Typen gleich, ist Casting nicht erforderlich. Ist der Typ des Ausdrucks auf der rechten Seite nun kleiner, dann kommt es zu einem Impliziten Casting.

Beispiel: Betrachten wir folgende Zuweisungen:

```
int a = 3;
float b = a;
```

Bei der ersten Zuweisung kommt es zu keinem Impliziten Casting, da ein `int`-Wert (3) einer `int`-Variablen (`a`) zugewiesen wird. Aber bei der zweiten Zuweisung kommt es schon zum Impliziten Casting, denn `int` ist kleiner als `float`.

Warum ist es wichtig zu wissen, welchen Typ das Ergebnis eines Ausdrucks hat? Hier gibt es zwei Beispielfälle, die zu unerwartetem Verhalten führen.

Beispiel: Aus 1) folgt, dass auch Divisionen mit ganzzahligen Werten, also des Datentyps `int`, nur ein ganzzahliges Ergebnis liefern.

```
int a = 5;
int b = 2;
float c = a/b;
```

! Die Annahme, dass hier ein impliziter Cast passiert ist leider falsch, auch wenn es praktisch erscheint. Hier wird zuerst die Division ausgewertet. Da sowohl die Variable `a` als auch die

(cc) BY

Variable b vom Typ int sind, ist das Ergebnis ebenso vom Typ int und hat den Wert 2. Die Kommastellen werden abgeschnitten. Erst danach, bei der Zuweisung, geschieht ein impliziter Cast. Daher hat c den Wert 2.0 und nicht 2.5 .

Beispiel: Man kann das Ergebnis eines Ausdrucks nicht in einem kleineren Datentyp speichern. Folgendes Beispiel erzeugt einen Fehler:

```
float a = 5.0;
int b = 3;
int sum = a + b;
```

Das Ergebnis von a + b ist 8.0 und hat den Datentyp float. Es kann aber float keinem int zugewiesen werden, da int kleiner als ein float ist und es zu Informationsverlust kommen könnte. In unserem letzten Beispiel ist dieser Verlust aber praktisch nicht vorhanden.

Um diese Zuweisung in solchen Fällen trotzdem zu ermöglichen gibt es das Explizite Casting.

4.7.3. Explizites Casting

Bis jetzt haben wir gesehen wie Processing den Typ einer Variable oder Berechnung automatisch implizit auf den größeren Typ casted. Die Umwandlung erfolgte automatisch nur in diese Richtung. Explizites Casting ermöglicht uns Casting in die andere Richtung. Bei explizitem Casting wird eine Variable mit größerem Wertebereich in eine Variable mit kleinerem Wertebereich gesteckt. Dies kann, muss aber nicht zu einem Informationsverlust führen.
Wie das Wort "explizit" schon andeutet, muss man diesen Cast in der Anweisung explizit anführen, also ausprogrammieren. Bei der Typumwandlung von float zu int sieht das wie folgt aus:

```
float a = 42.0;
int b = (int) a;
```

Das (int) vor dem a gibt den Expliziten Cast an, dass der float Wert von a in den Datentyp int umgewandelt werden soll, bevor er der int-Variablen b zugewiesen wird. Der Datentyp in den runden Klammern gibt beim Expliziten Casting immer an zu welchem Datentyp der Ausdruck rechts davon umgewandelt werden soll.

In Processing gibt es noch eine zweite Möglichkeit einen Expliziten Cast durchzuführen. Statt den Datentypen in Klammern zu setzen kann man auch den Ausdruck rechts davon in Klammern setzen. Diese Form vom Expliziten Casten gleicht einem Befehlsaufruf:

```
float a = 42.0;
int b = int(a);
```


Beide Formen sind gleich und es macht keinen Unterschied welche Form man verwendet beim Casten. In anderen Programmiersprachen ist meistens die erste Variante (Datentyp in Klammern) gängiger. Der Vorteil der zweiten Variante ist, dass sie übersichtlicher ist. Es ist klarer, welche Variablen und Operationen vom Cast betroffen sind und ist daher leichter zu lesen bei langen Berechnungen. Im oben angeführten Beispiel sieht man, dass es in beiden Fällen zu keinem Informationsverlust kommt, da 42.0 ja keine Nachkommastellen hat. Würde die Variable a jedoch eine Zahl mit Nachkommastellen enthalten, ist in der Variable b nur der ganzzahlige Anteil gespeichert.

Explizites Casting kann nicht nur bei Zuweisungen eingesetzt werden. Es ermöglicht auch eine Umwandlung bei der Auswertung von Ausdrücken. Folgendes Beispiel, welches nur gerade Zufallszahlen von 0 bis 98 liefert, soll das Konzept erläutern:

```
float randomNumber = random(50);
println (( (int) randomNumber) * 2);
```

Die Funktion random(50) (siehe https://processing.org/reference/random_.html) liefert eine zufällige positive Gleitkommazahl vom Typ float, die kleiner als 50 ist. Wird diese zufällige float-Zahl mit Casting in eine int-Zahl umgewandelt, erhält man eine ganzzahlige Zufallszahl zwischen 0 und 49. Dabei werden die Nachkommastellen einfach weggelassen (beabsichtigter Informationsverlust!). Wenn wir diese Zahl dann mit 2 multiplizieren, erhalten wir natürlich eine gerade Zahl. Somit werden im obigen Beispiel nur gerade Zufallszahlen zwischen 0 und 98 ausgegeben.

4.8. Überlauf

In der Mathematik hat die Zahlen-Addition a + b immer ein exaktes Ergebnis. Leider ist das Ergebnis am Computer nicht immer korrekt bzw. präzise. Warum? Der Grund dafür ist, dass die Datentypen, z.B. `int` und `float`, nur einen eingeschränkten Wertebereich haben und daher nicht alle Zahlen, die es gibt, darstellen können.

Beispiel: Geben Sie den folgenden Code in Processing ein und führen Sie das Programm aus. Achten Sie auf die Konsolenausgabe:

```
void setup(){
  int a = 2000000000;  // 2 Milliarden
  int b = a;
  int c = a + b;

  println(c);
}
```

Dieses Programm liefert zur Laufzeit zwar keine Fehlermeldung, das Ergebnis der Addition - `-294967296` - ist aber offensichtlich falsch. Der Grund dafür ist, dass das Ergebnis außerhalb des Wertebereichs von `int` liegt. Der Datentyp `int` kann ja nur Werte zwischen -2.147.483.648 und 2.147.483.647 speichern. Da das Ergebnis (4.000.000.000) größer als die Obergrenze ist, entsteht hier ein sogenannter **Überlauf.** Processing beginnt beim Überschreiten der Obergrenze um 1 wieder bei der kleinstmöglichen darstellbaren Zahl. Das Unterschreiten der Untergrenze liefert analog dazu einen sogenannten **Unterlauf**.

Jede der arithmetischen Operationen +, -, *, / könnte (bei entsprechenden Werten der Operanden) einen Überlauf hervorrufen.

Überlauf bzw. Unterlauf gibt es auch beim Datentyp `float`. Wird hier jedoch der größte darstellbare positive Wert überschritten oder der kleinste darstellbare negative Wert unterschritten, erhält das Ergebnis den Wert `Infinity`.

Beim Datentyp `float` tritt auch eine weitere Form des Unterlaufs auf, wenn nämlich das Ergebnis einer Operation so nahe beim Wert 0 liegt, dass es aufgrund der Gleitkommadarstellung und dem verfügbaren Speicherplatz für einen `float`-Wert nicht mehr sinnvoll dargestellt werden kann. Hier gibt Processing eine Fehlermeldung aus:

```
The literal … of type float is out of range;
```

Der kleinste darstellbare positive `float`-Wert ist in Processing 0.000000000000000000000000000000000000000000001.

Um Überläufe zu vermeiden, sollte man sich schon beim Anlegen einer Variablen überlegen, wie groß und wie klein die Werte in dieser Variablen überhaupt werden können.

Dementsprechend muss dann für diese Variable ein entsprechend großer Datentyp gewählt werden. Für die meisten Anwendungen in Processing sind aber die Datentypen `int` und `float` ausreichend.

5. Verzweigung

5.1. Bedingung

Im Leben müssen oft Entscheidungen getroffen werden - manche Entscheidungen sind nicht weltbewegend, andere hingegen können lebenswichtig sein. Eine Entscheidung beim Autofahren könnte etwa sein: "wenn auf dem Zebrastreifen vorne kein Mensch über die Straße geht oder gehen möchte, kann ich Gas geben, ansonsten muss ich abbremsen und anhalten".

Solche Entscheidungen spielen auch in der Informatik eine wichtige Rolle, im konkreten Fall etwa bei selbstfahrenden Autos. Anhand bestimmter Bedingungen oder Einflüsse von außen wollen wir daher Computerprogrammen auch Entscheidungsmöglichkeiten geben, um verschiedenste Problemstellungen abbilden zu können. Solche Entscheidungen werden in Programmen "Verzweigungen" genannt, da sie alternative Programmabläufe bieten, ähnlich wie bei einer Weggabelung.

Eine Bedingung ist eine Abfrage oder auch eine Aussage, anhand derer danach 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. Im obigen Beispiel wäre etwa die Bedingung "Überquert jemand vorn am Zebrastreifen die Straße bzw. möchte sie überqueren?" Wenn dies zutrifft, dann führe ich die Aktion "abbremsen und anhalten" aus. Trifft diese Bedingung nicht zu, dann kann ich Gas geben und weiterfahren.

Hinweis: Bedingungen können trotz unterschiedlicher Formulierung 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,.

5.2. Vergleichsoperatoren

Um Bedingungen in Processing auszudrücken, verwenden wir Vergleichsoperatoren. Wie der Name bereits sagt, sind Vergleichsoperatoren dazu da, um Vergleiche zu formulieren. Wahrscheinlich kennen Sie Vergleichsoperatoren bereits aus der Mathematik. Die Bedeutung der Vergleichsoperatoren sind der aus der Mathematik gleich.
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 Datentyp `boolean` (siehe Kapitel 4) und hat daher entweder den Wert **true** (wahr) oder den Wert **false** (falsch).

Will man zum Beispiel herausfinden, ob Bob älter ist als Alice, könnte das Programm wie folgt aussehen:

```
void setup(){
  int ageAlice = 17;
  int ageBob = 19;

  boolean bobIsOlder = ageAlice < ageBob;
  println(bobIsOlder);
}
```

Nach dem Ausführen des Programms wird in der Konsole `true` ausgegeben, weil 19 größer ist als 17.

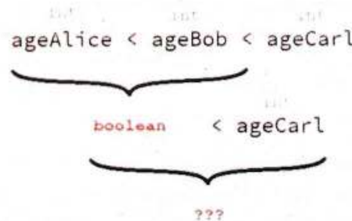
(cc) BY

5.3. Logische Operatoren

Will man nun das Beispiel um eine Person, Carl, erweitern und wissen, ob Bob vom Alter her zwischen Alice und Carl liegt, dann ist es (auf die Mathematik zurückgreifend) naheliegend, folgende Bedingung zu schreiben:

```
ageAlice < ageBob < ageCarl
```

Wären wir in der Mathematik, so wäre diese Bedingung korrekt. Leider ist das in Processing und so gut wie jeder anderen Programmiersprache nicht zulässig. Der Grund dafür liegt in einer "Mischung" von verschiedenen Datentypen, da diese Bedingung eigentlich aus zwei Teilbedingungen besteht, nämlich `ageAlice < ageBob` und `ageBob < ageCarl`.
Die untenstehende Grafik soll anhand der jeweiligen resultierenden Datentypen verdeutlichen, warum diese Bedingung für Programmiersprachen keinen Sinn ergibt.



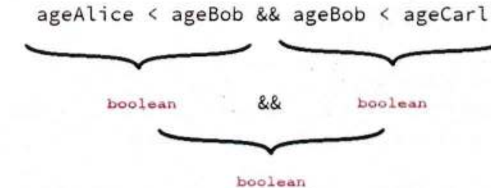
In Processing wird die Bedingung von links nach rechts abgearbeitet und daher zunächst die erste Teilbedingung (ageAlice < ageBob) evaluiert. Dabei werden zwei int-Zahlenwerte miteinander verglichen. Das Ergebnis dieses Vergleichs ist vom Datentyp `boolean`. Nun erfolgt die Evaluierung der zweiten Teilbedingung (Abbildung oben, zweite Zeile). Dabei wird das Ergebnis der ersten Teilbedingung mit dem Rest der Bedingung verglichen. Das bedeutet, es würde ein boolean Wert mit einem int Wert verglichen werden. Dies ist nicht möglich, weshalb es in Processing zu einem Fehler führt.

Die Lösung zu diesem Problem ist, die Bedingung auf zwei Bedingungen aufzuspalten und diese zwei Bedingungen miteinander zu verknüpfen. Für das konkrete Beispiel gilt: *Wenn `ageAlice < ageBob` wahr ist UND `ageBob < ageCarl` wahr ist,* dann ist die ganze Bedingung erfüllt. In Processing schreibt man dies folgendermaßen:

```
ageAlice < ageBob && ageBob < ageCarl
```

Der verwendete Operator (&&) nennt sich der UND-Operator. Logische Operatoren haben als Operanden immer Werte vom Datentyp `boolean`. Das Ergebnis einer solchen Operation ist wiederum ein `boolean`.

(cc) BY



Es stehen vier logische Operatoren zur Verfügung, die im weiteren im Detail erklärt werden:

Name	Logischer Operator
NOT	!
AND	&&
OR	||
XOR	^

5.3.1. Operator mit einem Operanden

Der einzige logische Operator mit nur einem Operanden ist die **Negation**. Sie wird verwendet, um eine Aussage zu negieren, also zu verneinen bzw. um abzufragen, ob etwas nicht zu trifft. Die Negation wird in Processing mit einem Rufzeichen (!) ausgedrückt und wird vor die Aussage gesetzt. Angenommen Sie wollen folgende Aussage formulieren: "Alice ist nicht älter als Bob". Dann müssten Sie zunächst die Teilaussage "Alice ist älter als Bob" formulieren und im zweiten Schritt negieren, was in Processing folgendermaßen aussehen würde:

```
boolean answer = !(alice > bob);
```

Im Grunde fügen wird also der normalen Aussage ein "nicht" vorne hinzu. Dadurch ergeben sich folgende Antworten für die Aussage `(alice > bob)`:

(alice > bob)	!(alice > bob)
true	false
false	true

War die Antwort `true`, dann bewirkt das Negieren, dass als neue Antwort `false` geliefert wird ("nicht wahr" entspricht "falsch"). War hingegen die Antwort `false`, dann bewirkt das Negieren, dass als neue Antwort `true` geliefert wird (d.h. "nicht falsch" entspricht "wahr").

(cc) BY

5.3.2. Operatoren mit zwei Operanden

AND-Operator (deutsch: UND-Operator)
Den Logischen AND-Operator schreibt man mit zwei Und-Zeichen (&&). Um den Operator zu verwenden, schreibt man ihn einfach zwischen zwei Aussagen

```
boolean answer = (alice > bob) && (alice > carl);
```

Mit dem Operator wollen Sie wissen, ob **sowohl** die linke Aussage **als auch** die rechte Aussage wahr ist. **Nur wenn beide Teilaussagen wahr sind, erhalten Sie als Antwort** `true`. Falls nur eine der beiden Aussagen oder gar beide Aussagen falsch sind, dann erhalten Sie als Antwort `false`.
Die Klammern sind hier nicht notwendig, sie verbessern aber die Lesbarkeit. Man kann natürlich auch noch mehr Aussagen anhängen z.B.

```
boolean answer = (alice > bob) && (alice > carl) && (bob > carl);
```

Hier erhalten Sie als Antwort nur dann `true`, wenn **jede** der Teilaussagen wahr ist.
Ansonsten erhalten sie `false`.

OR-Operator (deutsch: ODER-Operator)
Weiters gibt es den Logischen OR-Operator. Mit dem Operator setzen Sie die Abfrage um, ob zumindest eine der Aussagen wahr ist. Sie erhalten `true`, wenn mindestens eine der Aussagen wahr ist. Wenn gar keine der Aussagen wahr ist, dann erhalten Sie als Antwort `false`.

Der OR-Operator wird in Processing mit zwei senkrechten Strichen (||) dargestellt (auf einer deutschen Tastatur gibt man ihn mit der Tastenkombination "AltGr + <" (WIN, Linux) oder "Option/alt+ 7" (MAC) ein). Die Syntax ist die Gleiche wie für den AND-Operator - er wird zwischen die jeweiligen Aussagen geschrieben.

Zum Beispiel:
Wollen wir abfragen, ob Alice älter als Bob ist oder Alice älter als Carl ist, schreiben wir in Processing

```
boolean answer= (alice > bob) || (alice > carl);
```

XOR-Operator (deutsch: EXKLUSIVER ODER-Operator)
Es gibt vielleicht Situationen, wo Sie wissen wollen, ob nur genau eine der beiden Aussagen zutrifft. Ein Beispiel aus der Medizin: 2 Medikamente stehen für dieselbe Behandlung zur

(cc) BY TU Wien

Verfügung. Es darf nur eines davon, aber keinesfalls beide eingesetzt werden, da sich die beiden Medikamente nicht miteinander vertragen.

Solche **entweder...oder...** Operationen werden mit dem EXCLUSIVE-OR (XOR) Operator umgesetzt. Der XOR-Operator wird mit einem Zirkumflex - einem "Dacherl" - (^) dargestellt (auf einer deutschen Tastatur links neben der 1 zu finden). XOR wird auch zwischen zwei Aussagen geschrieben.

Das Beispiel mit den zwei Medikamenten, von welchem nur entweder das erste oder das zweite verwendet werden darf, um eine sichere Anwendung zu gewährleisten, wird in Processing wie folgt abgebildet:

```
boolean safeUse = useMedicineA ^ useMedicineB;
```

`useMedicineA` und `useMedicineB` sind dabei zwei Variablen vom Typ `boolean`, welche `true` beinhalten, wenn das Medikament verwendet wird bzw. `false`, wenn das Medikament nicht eingesetzt wird. Falls nur die linke Aussage wahr ist oder nur die rechte Aussage wahr ist, dann liefert die Operation `true` - nur eines der beiden Medikamente wird eingesetzt und es kommt zu keiner Wechselwirkung, d.h. der Einsatz ist sicher. Falls beide Aussagen falsch sind oder beide wahr sind dann erhalten Sie `false` als Antwort, d.h. der Einsatz ist nicht sicher, denn es besteht entweder eine Wechselwirkung aufgrund des Einsatzes beider Medikamente oder gar keine Wirkung, da keines der beiden Medikamente eingesetzt wird.

Wahrheitstabelle für Operatoren mit zwei Operanden
Die folgende Tabelle fasst das oben Erklärte nochmal kompakt zusammen indem sie die einzelnen Ausdrücke auswertet und deren Ergebnis, `true` oder `false`, aufschlüsselt. Diese Tabelle nennt man auch **Wahrheitstabelle**. Für die Aussagen `a` und `b` werden alle möglichen Kombinationen aufgelistet und die jeweiligen Resultate bei Verknüpfung mit bestimmten Operatoren in der Tabelle angezeigt.

Die ersten zwei Spalten repräsentieren dabei die Resultate für die einzelnen Aussagen `a` und `b`. Die Spalten drei bis fünf sind die Antworten, die Sie erhalten, wenn Sie die Logischen Operatoren anwenden auf `a` und `b`.

a	b	a && b AND	a || b OR	a ^ b XOR
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

(cc) BY TU Wien

zum Beispiel:

```
boolean a = true;
boolean b = false;
boolean answer= a || b;
println(answer);
```

Sie erhalten als Antwort `true`, denn `true || false` ist `true`. Die Variablen `a` und `b` können natürlich auch ausgewertete Ergebnisse sein und sinnvoller benannt werden:

```
void setup(){
  int ageAlice = 17;
  int ageBob = 19;
  int ageCarl = 25;

  boolean isYoungerB = ageAlice < ageBob;
  boolean isOlderC = ageAlice > ageCarl;
  boolean answer= isYoungerB || isOlderC;
  println(answer);
}
```

(cc) BY TU Wien

5.4. Optionale Codeausführung: If-Anweisung

Mithilfe von Vergleichsoperatoren und Logischen Operatoren ist es möglich komplexe 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. Ein Beispiel wäre:

"Wenn vor dir kein Hindernis ist, dann fahre gerade aus".

Die If-Anweisung ist folgendermaßen aufgebaut:

Allgemein:

```
if (Bedingung) {
  //optionaler Code
}
```

Bsp:

```
if (kein Hindernis vor dir) {
  //Anweisungen für 'fahre geradeaus'
}
```

Die `if`-Anweisung wird mit dem Schlüsselwort `if` eingeleitet. In runden Klammern folgt die Bedingung, unter der der optionale Code ausgeführt werden soll. 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 nicht ausgeführt.

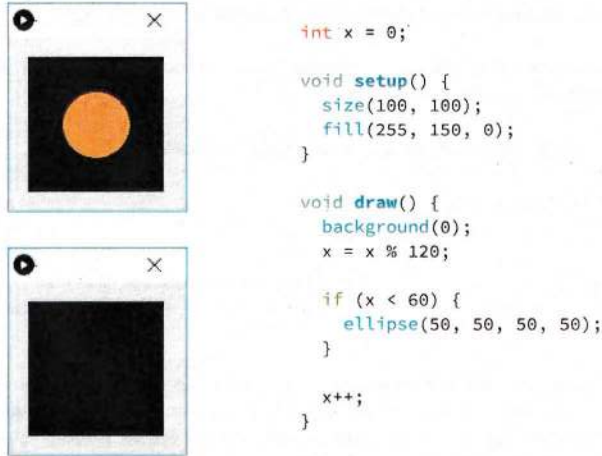
Beispiel: Auf der Suche nach der ältesten Person wurde Bob nach seinem Alter gefragt. Da Bob die erste Person ist, wird sein Alter als das älteste Alter gesetzt. Danach wird Alice nach ihrem Alter gefragt. Ist Alice älter als Bob? Falls ja, soll das Maximalalter auf das Alter von Alice gesetzt werden und die Ausgabe "Alice ist älter als Bob" erfolgen. Danach wird das Maximalalter ausgegeben.

```
void setup() {

  int bob = 25;
  int maximumAge = bob;
  int alice = 19;

  if (alice > bob) {
    maximumAge = alice;
    println("Alice ist älter als Bob.");
  }

  println("Maximales Alter: " + maximumAge);
}
```

(cc) BY TU Wien

Da in diesem Beispiel Alice nicht älter als Bob ist, folgt nur die Ausgabe des maximalen Alters. Wird das Alter von Bob auf zum Beispiel 18 gesetzt, dann würde der Code innerhalb des If ausgeführt werden.

Beispiel: Ein animiertes graphisches Beispiel zeigt einen orange blinkenden Kreis (oder Ampel).



```
int x = 0;

void setup() {
  size(100, 100);
  fill(255, 150, 0);
}

void draw() {
  background(0);
  x = x % 120;

  if (x < 60) {
    ellipse(50, 50, 50, 50);
  }

  x++;
}
```

Eine globale Variable `x` wird hoch gezählt im `draw()` und im Bereich zwischen 0 und 119 gehalten. Falls `x < 60` gilt, dann wird ein orangener Kreis gezeichnet. Da der `draw()` Bereich etwa 60 mal die Sekunde ausgeführt wird, entsprechen 120 Durchgänge etwa 2 Sekunden. Durch die Bedingung erscheint in der ersten Sekunde der Kreis und in der zweiten Sekunde wird er nicht gezeichnet.

5.4.1. Verschachtelte If-Anweisungen

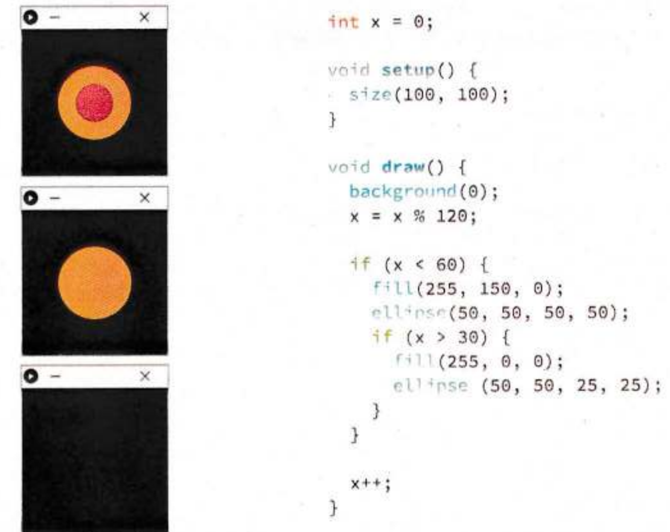
`if`-Anweisungen lassen sich natürlich auch verschachteln, das heißt, in einem `if`-Block kann noch ein `if`-Block stehen. Das macht dann Sinn, wenn man die zweite Anweisung nur dann ausführen möchte, wenn die erste bereits ausgeführt wurde. Im Beispiel mit dem Auto könnte das der folgende Fall sein:

"Wenn vor dir kein Hindernis ist, dann fahre gerade aus.
Wenn die Geschwindigkeit unter 10 km/h ist, beschleunige"

(cc) BY TU Wien

In dem Beispiel wollen wir nur dann beschleunigen, wenn wir bereits geradeaus fahren und zu langsam fahren. Damit verhindern wir, dass wir beschleunigen, wenn wir beispielsweise in der Kurve abbiegen.

Beispiel: Das animierte Ampel Beispiel wurde erweitert. Nun wird für eine kurze Zeit zusätzlich noch ein kleinerer rote Kreis gezeichnet. Der rote Kreis erscheint allerdings erst eine halbe Sekunde später auf als der orangene und kommt nie alleine vor.



```
int x = 0;

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

void draw() {
  background(0);
  x = x % 120;

  if (x < 60) {
    fill(255, 150, 0);
    ellipse(50, 50, 50, 50);
    if (x > 30) {
      fill(255, 0, 0);
      ellipse (50, 50, 25, 25);
    }
  }

  x++;
}
```

Verschachtelte `if`-Anweisungen sind sehr brauchbar. Manchmal fordern sie aber einen heraus zu verstehen, welche Bedingungen nun wirklich gelten. Sehen Sie sich das folgende Codebeispiel an, der unter bestimmten Bedingungen ein oder zwei Kreise zeichnen soll:

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

void draw() {
  int number = 8;

  if (number > 5) {
    ellipse(width/2, height/2, 100, 100);
    if (number < 3) {
      ellipse(width/2, height/2, 40, 40);
    }
  }
}
```

(cc) BY TU Wien

Ist es möglich, mit diesem Code einen zweiten Kreis zu zeichnen, wenn Ihnen nur erlaubt ist, den Initialisierungswert von `number` zu verändern? Es klingt für viele sehr einleuchtend, dass eine Zahl, die größer als 5 ist, nicht kleiner als 3 sein kann. Aber oft übersehen viele beim Programmieren diese Dinge. Sehen Sie sich das kleine Programm genauer an. Wenn die Zahl in `number` größer als 5 ist, dann wird ein Kreis gezeichnet. Danach wird überprüft, ob `number` kleiner als 3 ist. Dieser Fall kann nie eintreten, da `number` inzwischen nicht verändert wurde. Denn innerhalb vom ersten `if`-Block können wir garantieren, dass `number` größer als 5 ist, ansonsten hätten wir ja nie den `if`-Block betreten.

Diese Verschachtelung von `if` macht nur dann Sinn, wenn die Variable `number` verändert wird. Zum Beispiel so:

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

void draw() {
  int number = 8;

  if (number > 5) {
    ellipse(width/2, height/2, 100, 100);
    number = number / 4;
    if (number < 3) {
       ellipse(width/2, height/2, 40, 40);
    }
  }
}
```


5.5. Alternative Codeausführung: If-Else-Anweisung

Oft ist es auch sinnvoll nicht nur optionalen Code zu haben, sondern auch einen alternativen Code. Um alternativen Code auszuführen gibt es die **If-Else-Anweisung**. Sie ist eine Erweiterung der `if`-Anweisung. Mit ihr können Probleme der Form "Wenn ... dann ... sonst ..." dargestellt werden. Der "sonst"-Teil stellt die Alternative dar. Das Beispiel von vorhin sollte erweitert werden zu:

"Wenn vor dir keine Wand ist, dann fahre gerade aus, sonst bleibe stehen".

Die If-Else-Anweisung ist folgendermaßen aufgebaut:

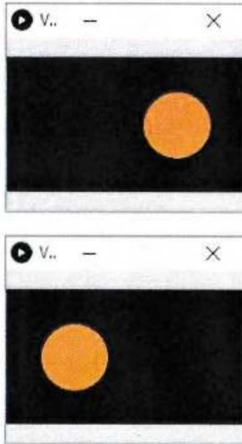
```
if (Bedingung) {
  // optionaler Code
} else {
  // Alternativer Code
}
```

Die `if`-Verzweigung wird um einen else-Teil, welcher mit dem Schlüsselwort `else` eingeleitet wird, erweitert. In geschwungener Klammer folgen dann die alternativen Anweisungen. Der `else`-Teil hat keine Bedingung, sondern wird immer dann ausgeführt, wenn die Bedingung im `if`-Teil nicht zutrifft. Durch die `if`-`else`-Anweisung wird somit auf jeden Fall einer der beiden Code-Blöcke ausgeführt. Im Beispiel oben fährt also das Auto entweder weiter oder es bleibt stehen.

Beispiel: Der Altersvergleich zwischen Alice und Bob kann um eine Ausgabe erweitert werden. Statt nichts auszugeben, falls Alice älter ist, kann alternativ ausgegeben werden, dass Alice nicht älter ist, um ein besseres Feedback ermöglichen.

```
void setup() {
  int bob = 25;
  int alice = 19;
  int maximumAge = bob;

  if (alice > bob) {
     maximumAge = alice;
     println("Alice ist älter als Bob");
  } else {
     println("Alice ist nicht älter als Bob.");
  }

  println("Maximales Alter: " + maximumAge);
}
```

Beispiel: Auch die blinkende Ampel kann erweitert werden. Statt eines Kreises sollen nun zwei Kreise abwechselnd für jeweils eine Sekunde angezeigt werden.



```
int x = 0;

void setup() {
  size(175, 100);
  fill(255, 150, 0);
}

void draw() {
  background(0);
  x = x % 120;

  if (x < 60) {
    ellipse(50, 50, 50, 50);
  } else {
    ellipse(125, 50, 50, 50);
  }
  x++;
}
```

5.5.1. Verschachtelung mehrerer If und Else Blöcke

In jedem `if`- oder `else`-Block können wiederum beliebig viele `if`-Blöcke, mit oder auch ohne `else`-Block, folgen. Aus der `if`- und `if-else`-Verzweigung können durch eine bestimmte Verschachtelung auch **mehrere Optionen/Alternativen** repräsentiert werden. Das Beispiel mit dem Auto kann erweitert werden:

"Wenn vor dir keine Wand ist, dann fahre gerade aus, sonst:
Wenn links von dir keine Wand ist, dann fahre nach links, sonst:
wenn rechts von dir keine Wand ist, dann fahre nach rechts, sonst bleib stehen."

Der Aufbau sieht folgendermaßen aus:

```
if (Bedingung 1) {
  // Option 1 Code
} else {
  if (Bedingung 2){
    // Option 2 Code
  } else {
    if (Bedingung 3) {
      // Option 3 Code
    } else {
      // Alternativer Code
    }
  }
}
```

(cc) BY TU Wien

Falls die erste Bedingung nicht zutrifft, wird der alternative Teil des ersten `if`s ausgeführt. Dieser wiederum besteht selbst nur aus einem Paar von `if-else`-Block. Falls die zweite Bedingung nicht erfüllt ist, wird wiederum die dritte Bedingung überprüft. Dieses Muster kann so oft wiederholt werden, wie man will. Treffen alle Bedingungen nicht zu, kann ein alternativer Code in einem letzten `else`-Block hinzugefügt werden. Da der Code bei großen Mengen von Optionen bzw. Alternativen sehr unübersichtlich wird, schreibt man diese Art von Verschachtelung gerne auch anders:

```
if (Bedingung 1) {
  // Option 1 Code
} else if (Bedingung 2){
  // Option 2 Code
} else if (Bedingung 3) {
  // Option 3 Code
}
...
else {
  // Alternativer Code
}
```

Um die Leserlichkeit zu bewahren, werden die geschwungene Klammern für die `else`-Blöcke weggelassen und der innere `if`-Block direkt an das `else` angehängt. Der restliche Code wird entsprechend eingerückt. Es handelt sich hier um keine neue Anweisung oder Konstrukt. Der Code ist aus dem vorigen verschachtelnden Code hergeleitet.

Beispiel: Das Beispiel mit dem Altersvergleich kann noch einmal erweitert werden. Denn "nicht älter" sein, kann zwei Dinge bedeuten. Entweder Alice ist jünger oder Alice und Bob sind gleich alt.

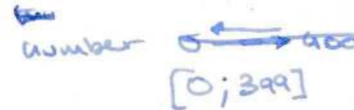
```
void setup() {
  int bob = 25;
  int alice = 19;
  int maximumAge = bob;

  if (alice > bob) {
    maximumAge = alice;
    println("Alice ist älter als Bob");
  } else if (alice == bob) {
    maximumAge = alice;
    println("Alice und Bob sind gleich alt");
  } else {
    maximumAge = bob;
    println("Alice ist jünger als Bob");
  }
  println("Maximales Alter: " + maximumAge);
}
```

(cc) BY TU Wien

5.5.2. Nie erreichter Programmcode

Mit Verzweigungen kann man also elegant steuern, welche Codeteile ausgeführt werden. Es muss aber auch die Reihenfolge, in welcher die Abfragen passieren, beachtet werden: Führen Sie folgenden Code in Processing aus. Idealerweise sollte das Programm die Farben und Positionen der gezeichneten Ellipsen ändern. Von schwarz auf rot über gelb zu grün und wieder von vorne. Allerdings wird nur ein grüner Kreis angezeigt. Versuchen Sie zu erklären, warum jenes Bild erzeugt wird.

```
int number = 0;

void setup() {
  size(100, 500);
}

void draw() {
  background(255);
  number = (number + 1) % 400;
  color red = color(255, 0, 0);
  color yellow = color(255, 255, 0);
  color green = color(0, 255, 0);
  color black = color(0);

  if (number < 400) {
    fill(green);
    ellipse(width/2, 400, 100, 100);
  } else if (number < 300) {
    fill( yellow );
    ellipse(width/2, 300, 100, 100);
  } else if (number < 200) {
    fill(red);
    ellipse(width/2, 200, 100, 100);
  } else if (number < 100) {
    fill(black);
    ellipse(width/2, 100, 100, 100);
  }
}
```



Erklärung:
Das Problem ist, dass in der ganzen `if`, `else if` Abfolge maximal nur ein Block ausgeführt wird, egal wie viele `else if` Blöcke angehängt werden. Entweder trifft einer der Aussagen zu, der jeweilige Block wird ausgeführt und der Rest der Blöcke wird übersprungen oder es trifft keine einzige Aussage zu und nichts wird ausgeführt. In diesem Fall wird zuerst die Aussage `number < 400` überprüft und die Aussage ist wahr. Wegen modulo 400 wird `number` immer kleiner 400 sein. Deshalb wird ein grüner Kreis gezeichnet und der Rest der Abfragen wird übersprungen.

(cc) BY TU Wien

Um dieses Problem zu lösen muss die Reihenfolge der Abfragen und Blöcke abgeändert werden.

```
int number = 0;

void setup() {
  size(100, 500);
}

void draw() {
  background(255);
  number = (number + 1) % 400;
  color red = color(255, 0, 0);
  color yellow = color(255, 255, 0);
  color green = color(0, 255, 0);
  color black = color(0);

  if (number < 100) {
    fill(black);
    ellipse(width/2, 100, 100, 100);
  } else if (number < 200) {
    fill(red);
    ellipse(width/2, 200, 100, 100);
  } else if (number < 300) {
    fill(yellow);
    ellipse(width/2, 300, 100, 100);
  } else if (number < 400) {
    fill(green);
    ellipse(width/2, 400, 100, 100);
  }
}
```

(cc) BY TU Wien

5.5.3. Mehrere unabhängige if vs. if- else if -else Anweisungen

Es gibt Situationen, in denen man nur eine Möglichkeit bzw. Alternative haben möchte (zum Beispiel beim Altersvergleich), manchmal ist es aber wünschenswert, dass mehrere Blöcke ausgeführt werden. In einer Abfolge von `if-else if-else` wird immer nur ein Block ausgeführt. Wenn mehrere Blöcke ausgeführt werden sollen, müssen die zusammenhängenden `if- else if-else` Blöcke getrennt werden. Folgender Code ist eine Möglichkeit, den Code aus dem vorigen Abschnitt zu trennen:

```
int number = 0;

void setup() {
  size(100, 500);
}

void draw() {
  background(255);
  number = (number + 1) % 400;
  color red = color(255, 0, 0);
  color yellow = color(255, 255, 0);
  color green = color(0, 255, 0);
  color black = color(0);

  if (number < 100) {
    fill(black);
    ellipse(width/2, 100, 100, 100);
  }
  if (number < 200) {
    fill(red);
    ellipse(width/2, 200, 100, 100);
  }
  if (number < 300) {
    fill(yellow);
    ellipse(width/2, 300, 100, 100);
  }
  if (number < 400) {
    fill(green);
    ellipse(width/2, 400, 100, 100);
  }
}
```

Nun werden - je nachdem welchen Wert `number` hat, mehrere Kreise gezeichnet. Durch die Aufteilung in mehrere `if`-Blöcke wird jede einzelne Aussage überprüft und keine Abfrage übersprungen. Jedes `if` ist daher unabhängig von den anderen `if`. Es können jetzt alle Blöcke ausgeführt werden, sofern die jeweilige Bedingung zutrifft. Testen Sie den Code und versuchen Sie die Unterschiede nachzuvollziehen!

(cc) BY

5.5.4. Sichtbarkeit von Variablen / Lokale Variablen Wiederholung

Die Sichtbarkeit der Variablen ist auf den Bereich der geschwungenen Klammern begrenzt. Innerhalb dieser ist die Variable nach ihrer Deklaration verwendbar. Außerhalb der geschwungenen Klammern existiert die Variable nicht. Auch in diesem Abschnitt spielt die Sichtbarkeit der Variable eine Rolle. Da die `if`-Anweisung (und die Alternative Anweisungen `else if` und `else`) geschwungene Klammern verwendet, können Variablen, die innerhalb der `if`-Anweisung deklariert wurden, außerhalb nicht verwendet werden. Folgenden Code z.B. lässt Processing nicht zu:

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

void draw() {
  int number = 10;

  if (number > 10) {
    int pos = 0;
  } else {
    int pos = 50;
  }

  ellipse(pos, pos, 100, 100);
}
```

Sie erhalten den Fehler "The variable 'pos' does not exist" bzw. "pos cannot be resolved to a variable" wenn Sie auf *Run* klicken. Um die Variable `pos` außerhalb der `if-else` Anweisung verwenden zu können (wie hier als Parameter in `ellipse(pos, pos, 100, 100);`), muss sie auch außerhalb der `if-else` Anweisung deklariert sein:

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

  void draw() {
    int number = 10;
    int pos;

if (number > 10) {
  pos = 0;
} else {
  pos = 50;
}
ellipse(pos, pos, 100, 100);
}
```

(cc) BY

5.6. Abfrage von Maus und Tastatur

Jetzt, da Sie wissen, wie Sie Processing dazu bringen Entscheidungen zu treffen, können wir noch mehr Interaktivität in das Spiel integrieren, wie etwa mittels Tastatur- und Mauseingaben.

In Processing kann abgefragt werden, ob eine Maustaste geklickt wurde und wenn ja, welche (links oder rechts). Auch die Abfrage von Tastatureingaben. Um diese Funktionalität zu nutzen, gibt es in Processing die folgenden internen (globalen) Variablen:

- `mousePressed`
- `mouseButton`
- `keyPressed`
- `key`

`mousePressed` und `keyPressed` sind Variablen vom Datentyp `boolean`. Sie haben also den Wert `true` oder `false`, je nachdem ob eine Taste gedrückt wurde oder nicht. Die Variablen `key` und `mouseButton` halten Informationen darüber, welche Tastaturtasten bzw. Maustasten gedrückt wurden.

Anstelle langwieriger Erklärungen - spielen Sie einfach mal mit dem vorgegebenen Code und finden Sie heraus, wie genau diese Variablen in Processing verwendet werden und wie mit ihnen eine Interaktion gestaltet werden kann. Laden Sie sich dazu die Datei *pacman5.pde* herunter. Starten Sie das Programm und versuchen Sie mit den Tasten `a` und `d` den PacMan zu steuern. (Achten Sie darauf, dass das Sketch Fenster fokussiert ist. Klicken ins Sketch Fenster, um das zu gewährleisten). Schauen Sie sich im Programm folgenden Codeabschnitt an:

```
if (keyPressed) {
  if (key == 'd') {
    pacManHorizontal = 1;
    pacManVertical = 0;
  }
  if (key == 'a') {
    pacManHorizontal = -1;
    pacManVertical = 0;
  }
}

pacManCenterX = (pacManCenterX + pacManHorizontal);
pacManCenterY = (pacManCenterY + pacManVertical);
```

Ändern Sie die Werte von `pacManHorizontal` und `pacManVertical` (erhöhen oder vertauschen Sie z.B. die Werte). Ändern Sie auch die Abfragen. Beobachten Sie, wie sich das Programm nach den Veränderungen verhält.

(cc) BY

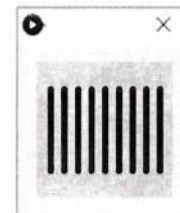
6. Schleifen

Mit den Fertigkeiten und dem Wissen aus den ersten fünf Kapiteln können bereits sehr interessante Programme geschrieben werden. Allerdings kann das Programmieren umständlich sein, wenn dieselben Befehle oftmals hintereinander ausgeführt werden sollen. Aktuell müssten nämlich Befehle so oft hingeschrieben werden, wie man sie eben zur Ausführung braucht. Das kann 5 mal sein, aber auch 100 mal. Und manchmal weiß man dies vor der Ausführung des Programms nicht einmal, da die Anzahl der Wiederholungen abhängig von anderen Parametern ist, welche vielleicht erst zur Laufzeit berechnet werden. Um sich Arbeit zu sparen, gleichzeitig aber auch flexibel zu sein, wird das Konzept der **Schleife** zur Umsetzung von Wiederholungen verwendet.

Die Wiederholung von Befehlen ist bereits aus Kapitel 3 bekannt: Durch die Wiederholung des `draw()` Bereichs ist es möglich, Animationen zu erstellen. Dazu wurde ein Bild gezeichnet und im nächsten Durchlauf von `draw()` mit einer geringfügigen Veränderung erneut gezeichnet. Auch mit Schleifen können Befehle einmal hingeschrieben werden und mehrmals wiederholt werden, ohne dabei mehr Schreibaufwand zu haben. Mit Hilfe von Variablen können innerhalb der Schleife außerdem kleine Änderungen an den Befehlen vorgenommen werden, um zum Beispiel interessante Muster zu erzeugen.

Starten wir mit einem einfachen Beispiel: Es sollen 10 Linien nebeneinander gezeichnet werden.

Eine Möglichkeit wäre es nun, für jede Linie den entsprechenden Befehl zu schreiben.



```
void setup() {
  strokeWeight( 5 );
  int num = 1;

  line( num * 10, 20, num * 10, 80 );
  num++;
  line( num * 10, 20, num * 10, 80 );
  num++;

  // weitere 8 mal
  // line( num * 10, 20, num * 10, 80 );
  // num++;

}
```

Das ist jedoch eine sehr mühsame Arbeit. Außerdem ist diese Lösung nicht sehr flexibel. Was ist, wenn man den Abstand zwischen den Linien ändern will? Oder die Anzahl der Linien? Oder die Länge der Linien? Jeder einzelne Befehl müsste schon bei kleinster Änderung angepasst oder erweitert werden. Mit einer Schleife lässt sich dieses Beispiel besser lösen.

(cc) BY

6.1. While-Schleife

Sachverhalte, die sich umgangssprachlich mit *"Solange (eine Bedingung zutrifft) wiederhole* ... " formulieren lassen, können elegant mit einer sogenannten `while`-Schleife abgebildet werden. Beim obigen Beispiel könnte dies vereinfacht formuliert werden mit: "Solange weniger als 10 Linien gezeichnet sind, zeichne eine weitere (vertikale) Linie (im selben Abstand, mit derselben Länge..)..."

Konkret wird das obige Beispiel dann in Processing mit einer `while`-Schleife folgendermaßen umgesetzt:

```
void setup() {
  strokeWeight( 5 );
  int num = 1;
  while( num < 10 ) {
    line( num * 10, 20, num * 10, 80 );
    num++;
  }
}
```

Sehen wir uns das Ganze nun nochmals Schritt für Schritt an:

Die `while`-Schleife wird mit dem Schlüsselwort `while` eingeführt. Danach folgt die Bedingung wie bei einer `if`-Verzweigung, gefolgt vom Code in geschwungenen Klammern, der wiederholt werden soll.

Der Aufbau der `while`-Schleife sieht allgemein wie folgt aus:

```
while (Bedingung) {
  // Code, der wiederholt werden soll
  // Anweisung 1;
  // Anweisung 2;
  // ...
}
```

Damit ist er dem Aufbau einer `if`-Anweisung sehr ähnlich. Genau wie bei der `if`-Verzweigung kann der Code einmal ausgeführt werden oder auch gar nicht. Aber mit einer Schleife kann der Code auch mehr als nur einmal ausgeführt werden, was mit einer `if`-Verzweigung nicht möglich war.

Bei der `while`-Schleife wird der Code solange ausgeführt, solange die Bedingung innerhalb der runden Klammern nach dem Schlüsselwort `while` wahr ist. Ist also die Bedingung wahr, dann werden die Anweisungen innerhalb der nachfolgenden geschwungenen Klammern ausgeführt. Bei der schließenden geschwungenen Klammer angekommen, wird dann

überprüft, ob die Bedingung noch immer wahr ist und der Code wird aufs Neue ausgeführt, wenn das der Fall ist. Dieser Vorgang wird solange wiederholt, bis die Bedingung nicht mehr wahr ist. Dann wird die Schleife beendet - die Anweisungen innerhalb der Schleife werden nicht mehr ausgeführt, und der Code nach der Schleife wird ausgeführt. Da das Beenden der Schleife von der Bedingung abhängt, die Schleife also abhängig von der Bedingung ausgeführt oder abgebrochen wird, wird die Bedingung auch als **Abbruchbedingung** bezeichnet.

Im Beispiel mit den senkrechten Linien wird eine sogenannte **Zählvariable** (wir nennen sie im Beispiel `num`) angelegt. Wir wählen den Namen so, weil die Variable `num` in der Schleife die Anzahl der Schleifendurchläufe mitzählt. Das bedeutet, diese Variable wird mit jedem Schleifendurchlauf um eins inkrementiert, d.h.hinauf gezählt (`num++;`).
Die Bedingung (`num < 10`) hängt nun von dieser Zählvariablen ab. Damit die Schleife irgendwann abgebrochen wird, muss gewährleistet sein, dass der Wert der Zählvariable so verändert wird, dass die Abbruchbedingung irgendwann nicht mehr zutrifft. In diesem Beispiel wird das mit dem Inkrementieren der Variable `num` erreicht (`num++;`). `num` wird bei jedem Schleifendurchlauf um 1 erhöht - so lange, bis `num` den Wert 10 enthält. Danach stimmt die Aussage `num < 10` nicht mehr und die Schleife wird abgebrochen.

Will man die Anzahl der Linien ändern, muss einfach die Abbruchbedingung entsprechend angepasst werden, z.B auf `num < 15` für 15 Linien.

Beispiel: Schleifen können auch innerhalb des `draw()` Bereichs verwendet werden. Die Schleife selbst zeichnet wiederum alle Linien. Mit einer kleinen Anpassung der Koordinaten können die Linien auch animiert werden. Testen Sie den Code aus!

```
float move = 0;

void setup() {
  strokeWeight( 5 );
}

void draw() {
  background(255);
  int num = 0;

  while( num < 5 ) {
    line( num * 20 + move, 20, num * 20 + move, 80 );
    num++;
  }

  move += 0.5;
  move = move % 20;
}
```

6.2. For-Schleife

Um jegliche Art von Wiederholungen in der Programmierung auszudrücken, reicht es grundsätzlich, nur die while-Schleife zu kennen und anwenden zu können. Es gibt jedoch noch einen weiteren Schleifentyp, welcher viel intuitiver anzuwenden ist, wenn die Anzahl der Wiederholungen bereits (direkt oder indirekt) bekannt ist - die for-Schleife.

Beispiel: Das Linien-Beispiel kann folgendermaßen mit einer for-Schleife umgesetzt werden:

```
void setup() {
  strokeWeight(5);
  for(int num = 1; num < 10; num++) {
    line(num * 10, 20, num * 10, 80);
  }
}
```

Die for-Schleife wird mit dem Schlüsselwort for eingeleitet. Die Initialisierung der Zählvariable (`int num = 1;`) wird im Vergleich zur Umsetzung mit einer while-Schleife jetzt in den ersten Teil innerhalb der runden Klammern der for-Schleife verschoben. Die Bedingung `num < 10;` steht im zweiten Teil (in der "Mitte"). Das Inkrementieren der Zählvariable (d.h. `num++`) wird nun nicht mehr (wie bei der while-Schleife) innerhalb des Anweisungsblockes geschrieben, sondern in den dritten Teil ("rechts") innerhalb der runden Klammern "vorgezogen". Die drei Teile werden voneinander mittels Strichpunkten getrennt.

Allgemein sieht der Aufbau einer for-Schleife damit wie folgt aus:

```
for(Initialisierung, Bedingung, Update){
  // Code, der wiederholt werden soll
  // Anweisung 1;
  // Anweisung 2;
  // ...
}
```

An erster Stelle steht die *Initialisierung*. Hier werden Variablen angelegt und initialisiert, welche für die Ausführung der Schleife relevant sind. In den meisten Fällen wird hier eine Zählvariable deklariert und initialisiert.
Dann folgt die *Bedingung*, wie sie von der while-Schleife bereits bekannt ist.
Als letztes wird im *Update*-Teil eine Änderung definiert, welche nach jedem Schleifendurchgang passieren soll, damit am Ende der Schleife die Abbruchbedingung nicht mehr erfüllt ist. Das ist in der Regel eine Beschreibung, wie die Zählervariable in der Initialisierung verändert wird, sodass dann die Bedingung irgendwann nicht mehr zutrifft.
In den geschwungenen Klammern folgt dann wieder der Code, der wiederholt werden soll.

Wenn man die for-Schleife und die while-Schleife einander gegenüberstellt, dann sieht man, dass die for-Schleife alle für die Schleife selbst relevanten Codeteile in einer Zeile zusammenfasst und in diesem Fall einen besseren Überblick verschafft.

6.3. For-Schleife vs. While-Schleife

Auch wenn es mehrere Arten von Schleifen gibt, kann für jedes Wiederholungsproblem sowohl eine for- als auch eine while-Schleife verwendet werden. Das Linien-Beispiel konnte sowohl mit der einen als auch mit der anderen Schleife gelöst werden

Es stellt sich dann die Frage, warum es mehrere Arten von Schleifen gibt?

Die for-Schleife wird gerne verwendet, wenn man weiß, wie oft die Schleife wiederholt wird, wenn Zählprobleme abgebildet werden sollen und wenn die Veränderung des Schleifenzählerwerts konstant ist. Auf einen Blick sieht man, welche Variablen angelegt werden, welche Bedingungen gelten müssen und wie die Variable verändert wird. Der Aufbau der Schleife hilft auch, das Hochzählen der Variable nicht zu vergessen und verhindert somit **Endlosschleifen** (siehe Unterkapitel Endlosschleife)

In der while-Schleife hingegen passiert schnell, dass auf die Zählvariable vergessen wird. Dafür sind while-Schleifen besonders geeignet, wenn die Anzahl der Wiederholungen nicht im vornherein bestimmt ist bzw. wenn keine Zählvariablen auftreten.

Ein abstraktes Beispiel ist: Solange der User das Programm nicht beendet, zeichne weiter. In diesem Fall gibt es keine Zählvariable. Die Bedingung ist abhängig von der Benutzerinteraktion.

Mit einer while-Schleife ließe sich das vereinfacht wie folgt darstellen:

```
boolean userPressed_ESC = false;

while( !userPressed_ESC ) {
  // draw things
}
```

Bei einer for-Schleife müsste man die Initialisierung und das Update auslassen:

```
boolean userPressed_ESC = false;

for( ; !userPressed_ESC; ) {
  // draw things
}
```

Beide Schleifen sind möglich, aber die while-Schleife ist für dieses Problem intuitiver.

(cc) BY

6.4. Endlosschleifen

Schleifen sind sehr praktisch und ersparen viel Schreibarbeit. Allerdings kann es zu sogenannten Endlosschleifen kommen. Das heißt, dass die Abbruchbedingung nie falsch wird und die Schleife somit ewig weiterläuft. In den meisten Fällen ist das unerwünscht und ungeplant. Typische Zeichen für eine unerwünschte Endlosschleife ist, wenn das Programm nicht auf Eingaben reagiert oder bestimmte Aktionen, die nach der Schleife ausgeführt werden sollten, nicht ausgeführt werden. In Processing könnte ein Zeichen für eine Endlosschleife sein, dass nur eine oder keine Form gezeichnet wird, statt mehreren oder die Animation nicht funktioniert.

Damit die Bedingung nach einigen Wiederholungen nicht mehr zutrifft, muss sichergestellt sein, dass es innerhalb der Schleife die Möglichkeit gibt, die für die Bedingung relevanten Variablen zu ändern. Im Linien-Beispiel wurde die Variable num in der Abbruchbedingung verwendet. In der Schleife wurde mit dem Inkrementieren sichergestellt, dass die Abbruchbedingung irgendwann nicht mehr zutrifft.

Die zwei häufigsten Quellen für Endlosschleifen sind

- falsche Abbruchbedingungen und,
- dass darauf vergessen wurde, die entsprechenden Variablen in der Schleife zu ändern.

(cc) BY

6.5. Verschachtelte Schleifen

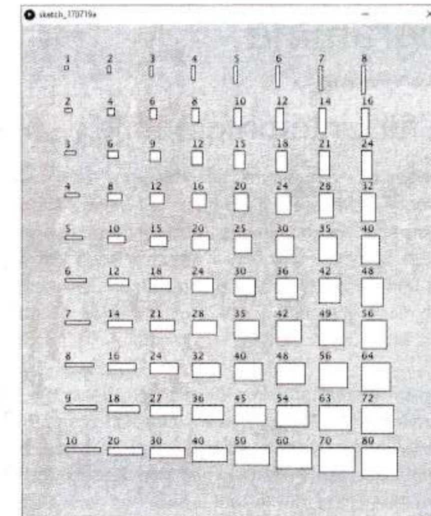
Einzelne Schleifen sind bereits sehr hilfreich und können sehr interessante Effekte erzeugen. Aber auch Schleifen können verschachtelt, d.h. Schleifen innerhalb von Schleifen, werden, um noch mehr interessante Muster zu zeichnen. Folgendes Beispiel ist eine Abwandlung des Linienbeispieles aus dem Video.

Kopieren Sie den Code in Processing und führen Sie ihn aus.

```
void setup() {
  size(600, 700);

  int xOffset = 60;
  int yOffset = 60;
  int scale = 5;
  for (int row = 1; row <= 10; row++) {
    for (int column = 1; column <= 8; column++) {
      fill(0);
      textSize(14);
      text(row*column, column*xOffset, row*yOffset-5);
      fill(255);
      rect(column*xOffset, row*yOffset, scale*row, scale*column);
    }
  }
}
```

Das Beispiel zeichnet im Sketch-Fenster eine Multiplikationstabelle, bei der die Ergebnisse nicht nur als Zahl, sondern auch als Fläche eines Rechtecks dargestellt sind.

(cc) BY TU Wien



Verändern Sie ein paar Werte (z.B. die Grenzen in den Bedingungen oder den Offset) und untersuchen Sie, wie sich die Änderungen auf das Ergebnis auswirken.

Um zu sehen, was die innere Schleife macht, ändern Sie die Obergrenze der äußeren Schleife in deren Bedingung von 10 auf 1. Dadurch hat die äußere Schleife nur einen Durchlauf und die innere Schleife wird genau einmal ausgeführt. Dann ändern Sie diese Obergrenze auf 2 usw. Dadurch kann man gut sehen, wie mit Hilfe der Schleifen die Tabelle zeilenweise von oben nach unten gezeichnet wird.

Überlegen Sie, ob die folgenden Aussagen stimmen oder nicht und versuchen Sie ihre Antwort zu begründen.

1. Die innere Schleife zeichnet eine Zeile in der Multiplikationstabelle
2. Die Obergrenze der inneren Schleife bestimmt die Anzahl der Spalten.
3. Das Ändern von `xOffset` bewirkt, dass der horizontale Abstand zwischen den Einträgen kleiner oder größer wird.
4. Die Obergrenze der äußeren Schleife bestimmt die Anzahl der Zeilen in der Tabelle
5. `scale` bestimmt die Größe der Rechtecke

(cc) BY TU Wien

7. Unterprogramme

(Funktionen)

7.1 Motivation für Unterprogramme

Im Alltag werden Tätigkeiten oft "vereinfacht" beschrieben: Reden wir zum Beispiel vom Kochen, Lesen, Laufen oder Telefonieren, dann bestehen diese Tätigkeiten zwar aus einer Abfolge von Einzelschritten, aber wir können uns trotzdem darunter etwas vorstellen. Mit nur einem Wort kann man sich so eine Abfolge von verschiedenen Einzelschritten vorstellen. Dieses Zusammenfassen von Einzelschritten - hier zu einer "Tätigkeit" - ist nichts anderes als zu abstrahieren. Durch diese Abstraktion verstehen wir - in der Regel ausreichend genau - was insgesamt geschieht, ohne die einzelnen Details genauer zu kennen.

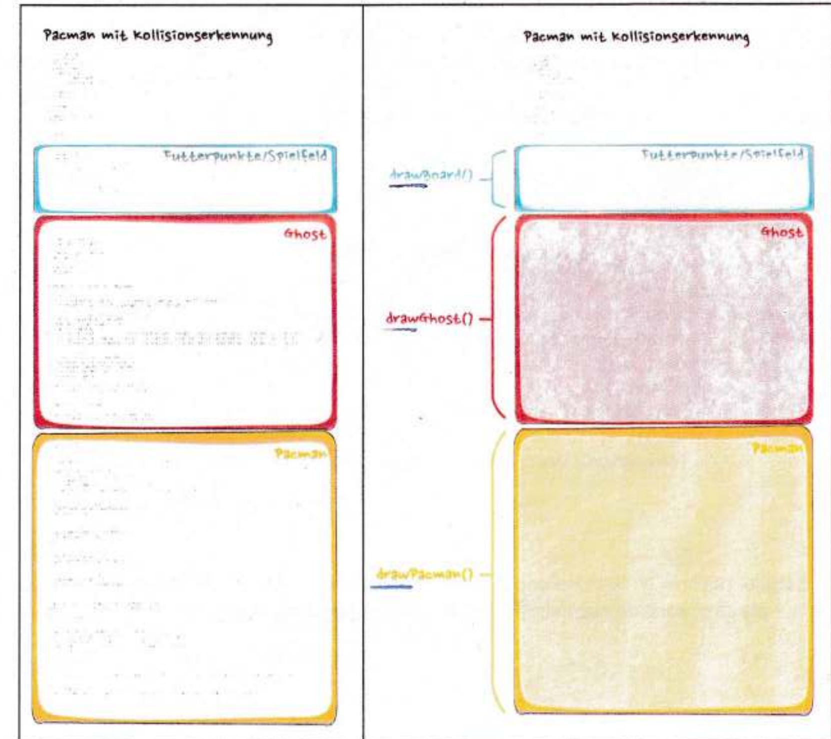
Im Pacman Beispiel ist dies ähnlich: Im draw-Bereich des Pacman Programms gehören jeweils einige Programmzeilen (vgl. "Einzelschritte") inhaltlich zusammen. Sie machen also etwas Ähnliches oder erfüllen gemeinsam eine größere Aufgabe (vgl. "Tätigkeit"): Sie zeichnen etwa das Spielfeld mit den Futterpunkten, den Pacman bestehend aus Körper, Mundöffnung, Auge, oder einen Geist, der in sich wieder aus verschiedenen geometrischen Objekten aufgebaut ist (siehe nachfolgende Abbildung links).

Wenn man nun das Pacman-Programm aufgrund dieser inhaltlichen Zusammengehörigkeiten ansieht, dann wird zunächst der setup-Bereich ausgeführt. Im draw-Bereich können wir also inhaltlich drei Abschnitte erkennen: *Futterpunkte/Spielfeld*, *Pacman* und *Ghost* (siehe nachfolgende Abbildung links).
Diese Einteilung des Programms in Abschnitte kann uns dabei helfen, einen besseren Überblick über den Ablauf und den Gesamtaufbau unseres Programms zu erhalten. Wenn wir uns nur die Abschnitte anschauen, sieht der Code auf den ersten Blick gleich übersichtlicher aus. Dies vereinfacht es, die Funktionalität eines Programms zu erfassen, für so einen Überblick braucht man nicht den gesamten Programmcode Zeile für Zeile lesen und nachvollziehen.

Wenn wir also das Pacman-Programm auf dem Level der Code-Abschnitte ansehen (siehe nachfolgende Abbildung rechts), sind wir eine Abstraktionsebene höher gewechselt. Wir können erkennen, dass es Abschnitte gibt, die einen Pacman zeichnen und einen Geist und wie diese Abschnitte im Programm angeordnet sind. Wie sie jedoch im "Inneren" im Detail realisiert sind, wird verschleiert. Wir erhalten somit einen besseren Überblick über das gesamte Programm, aber weniger Detailinformation.

In der Programmierung setzen wir diese Strukturierung mit sogenannten **Unterprogrammen** um. Mit Unterprogrammen fassen wir eben solche Codeabschnitte zusammen, versehen sie mit einem aussagekräftigen Namen und stellen sie zur weiteren Verwendung zur Verfügung. Auf diese Weise strukturieren wir mit Unterprogrammen Programmcode und machen diesen leichter lesbar. Außerdem ersparen wir uns damit Schreibarbeit, denn einmal geschriebene Unterprogramme können beliebig oft an verschiedenen Stellen verwendet werden.

Betrachtet man den gesamten Code dann auf Basis dieser Struktur mit den Unterprogrammen und schaut nicht in die Details, was jedes einzelne macht, dann kann dies als ein Beispiel für Code-Abstraktion in der Programmierung wahrgenommen werden.



Man nennt Unterprogramme auch **Funktionen, Prozeduren** oder auch **Methoden**, je nachdem in welchem Kontext sie vorkommen und was sie genau machen. Umgangssprachlich hat sich in Processing für die Bezeichnung *Unterprogramm* der Begriff **Funktion** eingebürgert, weshalb wir in den Unterlagen daher Unterprogramme synonym als **Funktionen** bezeichnen.

Wenngleich in diesem Handout der Fokus auf dem Programmieren eigener Funktionen liegt, sehen wir uns zunächst an, wie eine Funktion ausgeführt (man sagt auch: aufgerufen) wird:

3 Beispiele für Funktionsaufrufe:

```
fill(255, 0, 0);                Funktion mit Parametern
ellipse(50, 50, 50, 50);        Funktion mit Parametern
noFill();                       Funktion ohne Parameter
```

Zum **Ausführen einer Funktion** ist also deren Funktionsname anzugeben und nachfolgend innerhalb runder Klammern keine, einen, oder mehrere sogenannte Parameter (durch Beistriche getrennt). Abgeschlossen wird der Funktionsaufruf, wie in Processing bei Anweisungen üblich, durch einen Strichpunkt. Das Ausführen einer Funktion wird **Funktionsaufruf bzw. Aufrufen einer Funktion** genannt. Das kennen Sie bereits in Processing, auch wenn Sie vielleicht diese Begriffe nicht gekannt haben.

(cc) BY TU Wien

7.2 Eigene Unterprogramme programmieren

Wie werden nun eigene Funktionen programmiert? So wie Variablen deklariert werden müssen, müssen auch Funktionen dem Computer bekannt gemacht werden. Das bedeutet, auch sie müssen **deklariert** bzw. **definiert** werden, bevor sie in Programmen dann verwendet bzw. aufgerufen werden können.

Im Folgenden sehen Sie zwei Beispiele für Definitionen selbstprogrammierter Funktionen:

```
void drawRedCircle() {
  fill(255, 0, 0);
  ellipse(50, 50, 50, 50);
}
```

Der Aufruf `drawRedCircle();` zeichnet einen roten Kreis mit Durchmesser 50 Pixel an der Position (50, 50).

Die folgende Funktion `maximum(...)` berechnet das Maximum dreier ganzer Zahlen a, b und c:

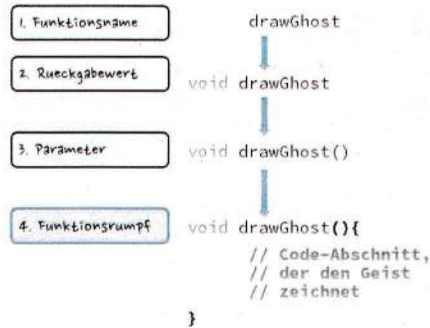
```
int maximum(int a, int b, int c) {
  int max = a;

  if (max < b) {
    max = b;
  }

  if (max < c) {
    max = c;
  }
  return max;
}
```

Die **Definition** einer Funktion in Processing besteht aus vier Teilen, die im Laufe dieses Kapitels im Detail erläutert werden:

	Beispiel
• Datentyp des Rückgabewerts (Rückgabetyp)	`int`
• Funktionsname	`maximum`
• Parameter innerhalb von Klammernpaar ( )	`(int a, int b, int c)`
• Anweisungsblock (Funktionsrumpf)	`{...}`

Wenn wir nun eigene Funktionen programmieren möchten, dann können wir anhand folgender Fragestellungen strukturiert vorgehen:

(cc) BY TU Wien



1. Welche Aufgabe soll die Funktion erfüllen?
   Anhand dieser Fragestellung wählen wir einen aussagekräftigen Namen für die Funktion. Denn der Funktionsname soll das, was die Funktion macht, so gut wie möglich beschreiben. Für erlaubte Funktionsnamen gelten im Übrigen die gleichen Regeln wie für Variablennamen.

2. Soll die Funktion einen Rückgabewert liefern? Wenn ja, von welchem Typ?
   Der Rückgabetyp wird links vom Funktionsnamen geschrieben. Er gibt an, welche Art von Ergebnis eine Funktion liefert. Beispielsweise liefert die Funktion `maximum()` eine ganze Zahl (`int`) als Ergebnis, welches die größte Zahl der übergebenen drei ganzen Zahlen ist. Zeichnen wir mit der Funktion etwas, wie in der Abbildung oben etwa einen Ghost, dann brauchen wir keinen Rückgabewert, den wir an anderer Stelle im Programm weiterverwenden wollen. In diesem Fall ist der Typ des Rückgabewerts `void` (kein Rückgabewert).

3. Welche Informationen/Werte benötigt die Funktion zur Lösung der Aufgabe?
   Mit dieser Frage überlegen wir, ob das Unterprogramm beim Aufruf bestimmte Werte benötigt, um die Aufgabe erfüllen zu können.
   Wollen wir etwa das Maximum von drei Zahlen berechnen, werden diese drei Zahlen zur Ausführung des Unterprogramms benötigt. Soll etwa ein Kreis gezeichnet werden, sind Werte betreffend dessen Position und Größe notwendig.
   Diese Werte, die eine Funktion zur Ausführung der Aufgabe benötigt, werden Parameter genannt. Mit Parametern kann man das Verhalten einer Funktion ein wenig steuern bzw. flexibler gestalten. Sie werden innerhalb runder Klammern unmittelbar nach dem Funktionsnamen geschrieben. Dazu wird in der Definition der Funktion für jeden Parameter vor seinem Namen auch sein Datentyp angegeben. Falls die Funktion keine Parameter benötigt, wird innerhalb der runden Klammern einfach nichts angegeben, so wie bei der Funktion `drawRedCircle()`.

4. Wie wird die Aufgabe erfüllt?
   Diese Fragestellung zielt darauf ab, wie wir zum Ergebnis gelangen. Hier wird - innerhalb geschwungener Klammern - also jener Code programmiert, der die Aufgabe der Funktion löst.

(cc) BY TU Wien

Programmieren mit Processing
Kapitel 7: Unterprogramme

Datentyp des Rückgabewerts (1), Funktionsname (2) und Parameter (3) bilden zusammen den sogenannten **Funktionskopf**:

```
rückgabeTyp funktionsName(parameter1, parameter2, ..., parameterN)
```

Der Grundaufbau einer Funktion ist insgesamt also so gegliedert:

```
rückgabeTyp funktionsName(parameter1, parameter2, ..., parameterN) {
  // Funktionsrumpf, der die auszuführenden Anweisungen enthält
}
```

Achtung: Die Definitionen von Funktionen werden in Processing AUSSERHALB der setup() und draw() Bereiche geschrieben!

(cc) BY TU Wien

7.3 Funktionen ohne Rückgabewert und Parameter

Betrachten wir zunächst eine sehr einfache Variante der Funktionen - nämlich Funktion ohne Rückgabewert und ohne Parameter. Eine sehr einfache selbstgeschriebene Funktion könnte so aussehen:

```
void drawRedCircle() {
  fill(255, 0, 0);
  ellipse(50, 50, 50, 50);
}
```

Diese Funktion zeichnet - wie man am Funktionsnamen schon erkennen kann, einen roten Kreis. Dieser wird fix an der Position (50, 50) gezeichnet. Der Name der Funktion ist `drawRedCircle()`, es gibt keine Parameter, die runden Klammern werden aber jedenfalls geschrieben. Dass die Funktion keinen Rückgabewert besitzt, erkennen Sie am Rückgabetyp `void`. `void` bedeutet so viel wie "leer".

Wenn Sie diesen Code in Processing kopieren und das Programm starten, dann passiert noch nicht viel. Denn genau wie die Processing Funktionen `rect()`, `ellipse()` usw., muss auch `drawRedCircle()` aufgerufen werden, damit der rote Kreis auch tatsächlich gezeichnet wird. Dazu wird der Funktionsname mit nachfolgenden runden Klammern, z.B. in `draw()`, geschrieben.

Hinweis: In diesem Beispiel ist auch ersichtlich, dass die **Definition der Funktion** `drawRedCircle()` **außerhalb der `setup()` und `draw()` Bereiche** programmiert wird.

```
void setup() {
}

void draw() {
  drawRedCircle(); // Funktionsaufruf
}

// Definition der Funktion
void drawRedCircle() {
  fill(255, 0, 0);
  ellipse(50, 50, 50, 50);
}
```

Das "Hinschreiben" des Funktionsnamen, um die entsprechende Funktion auszuführen, nennt man **Funktionsaufruf**. Die Funktion wird also von dieser Stelle im Programm aufgerufen und damit der Code innerhalb des Funktionsrumpfes ausgeführt. Wird jetzt das Programm ausgeführt, dann wird ein roter Kreis gezeichnet.

In diesem kleinen Beispiel scheint eine Funktion noch Mehrarbeit zu sein. Sobald bei komplexeren Programmen jedoch mehr Code im `draw()` Bereich steht, ist es übersichtlicher, zusammenhängenden Code in Funktionen auszulagern, sodass der `draw()` Bereich nur wenige Funktionsaufrufe und die wichtigsten Berechnungen beinhaltet. Damit kann die gesamte Funktionalität des Programms in `draw()` schnell erfasst werden, ohne seitenweise Codezeilen lesen zu müssen.

Funktionen können nicht nur in `setup()` und `draw()` aufgerufen werden. Auch innerhalb von anderen Funktionen sind Funktionsaufrufe möglich, wie das nachfolgende Beispiel illustriert:

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

void draw() {
  drawRobot(); //Funktionsaufruf von drawRobot()
}

//Funktionsdefinition drawRobot()
void drawRobot() {
  drawRobotHead(); //Funktionsaufruf von drawRobotHead()
  drawRobotBody(); //Funktionsaufruf von drawRobotBody()
}

//Funktionsdefinition drawRobotHead()
void drawRobotHead() {
  //head
  fill(200);
  rect(200, 100, 100, 100);

  //eyes
  fill(250, 250, 0);
  ellipse(230, 130, 20, 20);
  ellipse(270, 130, 20, 20);

  //mouth
  fill(0);
  rect(230, 160, 40, 20);
}

//Funktionsdefinition drawRobotBody()
void drawRobotBody() {
  fill(200);
  rect(240, 200, 20, 20); //neck
  rect(180, 220, 140, 200); //body
  rect(160, 220, 20, 150); //left arm
  rect(320, 220, 20, 150); //right arm
}
```

In diesem Beispiel wird die Funktion `drawRobot()` in `draw()` aufgerufen. `drawRobot()` selbst enthält wieder zwei Funktionsaufrufe: `drawRobotHead()` und `drawRobotBody()`. An diesem Beispiel lässt sich auch schön die Abstraktion erkennen. Für den Leser reicht es bereits den Funktionsaufruf im `draw()` zu lesen, um zu verstehen, was hier im Groben passiert - es wird ein Roboter gezeichnet.
Liest man dann die Funktion `drawRobot()` genauer durch, dann erhält man die Information, dass der Roboter aus Kopf und Körper besteht. Geht man dann noch weiter und betrachtet die Funktionen `drawRobotHead()` und `drawRobotBody()`, dann sieht man im Detail, wie Körper und Kopf in sich gezeichnet werden. Je weiter man also in die Funktionen hineinschaut, desto mehr Details erfährt man. Betrachtet man nur die Details, dann weiß man vielleicht nicht, ob beispielsweise der Roboterkopf nur "herumliegt" oder eigentlich ein Teil eines gezeichneten "ganzen Roboters" ist.

Funktionen sind jedoch nicht nur zur besseren Strukturierung da, sondern können auch mehrfach an verschiedenen Stellen des Programms verwendet, d.h. aufgerufen, werden:

Das folgende Beispiel zeichnet mehrere unterschiedlich große und farbige Kreise und Vierecke. Die fürs Zeichnen notwendigen Variablen werden global deklariert und in `setup()` initialisiert. Die Variable `numBodyParts` enthält dabei die Anzahl der noch zu zeichnenden Kreise und Vierecke. Des Weiteren ruft `setup()` die `updateValues()`-Funktion auf, um zufällige Werte für die Variablen `radius` und `diameter` festzulegen.
Das eigentliche Zeichnen passiert in der Funktion `bodyParts()`, die pro Funktionsaufruf einen Kreis und ein Viereck zeichnet und die Anzahl an Figuren, die noch zu zeichnen sind, um 1 verringert. Sie verwendet auch `updateValues()`, um die Koordinaten und Größe der nächsten zwei Figuren zu berechnen. In `draw()` selbst passiert nicht viel: Es wird lediglich die Funktion `bodyParts()` genau `numBodyParts`-mal aufgerufen.

An diesem Beispiel sehen Sie, dass die Funktion `updateValues()` an zwei unterschiedlichen Stellen verwendet wird.

```
float radius;
float diameter;
float x;
float y;
int numBodyParts;

void setup()
{
  size(600, 400);
  x = 0.25f * width;
  y = 0.5f * height;
  updateValues();
  numBodyParts = 6;
}
```

(cc) BY TU Wien

Programmieren mit Processing
Kapitel 7: Unterprogramme

```
void draw()
{
  if(numBodyParts > 0) // wir brauchen keine Schleife,
                       // da draw wiederholt ausgeführt wird.
  {
    bodyParts();
  }
}

void bodyParts()
{
  int r = int(random(255));
  int g = int(random(255));
  int b = int(random(255));

  fill(r, g, b);
  ellipse(x, y, diameter, diameter);

  fill(255 - r, 255 - g, 255 - b);
  rect(x - radius, y, diameter, diameter);

  float oldRadius = radius;
  updateValues();
  x = x + radius + oldRadius;
  numBodyParts--;
}

void updateValues()
{
  radius = random(40) + 10;
  diameter = radius * 2;
}
```

(cc) BY TU Wien

7.4 Funktionen mit Parametern

Funktionen sind aber noch mächtiger. Sie können noch mehr, als nur Programmcode auslagern oder zusammenfassen. Durch den Einsatz von **Parametern** lässt sich die Wirkung einer Funktion variieren und damit individueller gestalten. Parameter sind Variablen innerhalb von Funktionen. Diese besonderen Variablen erhalten ihre Werte nicht durch direkte Zuweisung, sondern die Werte werden beim Funktionsaufruf **übergeben**. Dadurch geben sie der Funktion ein wenig Spielraum, sodass eine Funktion nicht nur auf ein sehr konkretes Problem zugeschnitten ist, sondern ähnliche Probleme durch Aufruf der gleichen Funktion (aber mit jeweils anderen Parametern) lösbar sind.

Beispielsweise hat die Processing-Funktion `ellipse()` vier Parameter. Die ersten zwei geben die Position der Ellipse an und die letzten zwei Parameter die Breite und die Höhe. Der Funktionskopf für diese Funktion sieht wie folgt aus:

```
void ellipse(float a, float b, float c, float d)
```

Der Aufruf der Funktion

```
ellipse(10, 20, 50, 30);
```

bestimmt hier bereits welche Werte die Parameter a (= 10), b (= 20), c (= 50) und d (= 30) erhalten. Dadurch, dass diese Werte beim Aufruf bestimmt werden können, ist es möglich die Ellipse an verschiedene Orte zu platzieren und in verschiedenen Größen darzustellen. Ist die Länge und Breite der Ellipse gleich, erhalten wir die Sonderform der Ellipse, einen Kreis.

Beispiel: Der rote Kreis aus dem vorhergehenden Beispiel soll an verschiedenen Positionen gezeichnet werden können. Um das zu ermöglichen, führen wir zwei Parameter ein, einen für die x-Koordinate und einen für die y-Koordinate der gewünschten Position des Kreises. Wir erweitern dazu den Funktionskopf um zwei Parameter:

```
void drawRedCircle(int x, int y) {
  fill(255, 0, 0);
  ellipse(x, y, 50, 50);
}
```

Die Parameter sind vom Typ `int`, da ganzzahlige Koordinaten ausreichen. Statt den fixen Koordinaten (50, 50) verwenden wir nun die Parameter x und y. Beim Funktionsaufruf ist es jetzt möglich, Koordinaten zu übergeben und rote Kreise an verschiedenen Positionen zu zeichnen:

```
void draw() {
  drawRedCircle(25, 100);
```

(cc) BY


```
  drawRedCircle(75, 40);
}
```

Im `draw()` wurde diesmal zweimal die Funktion `drawRedCircle()` aufgerufen, einmal soll der Kreis an der Stelle (25, 100) gezeichnet werden und einmal an der Stelle (75, 40).
Mit nur kleinen Änderung ist es jetzt möglich diesen Kreis mehrmals an verschiedenen Stellen zu zeichnen. Diese Funktion kann noch mit weiteren Parametern erweitert werden. Z.B. kann auch die Größe des Kreises als Parameter gesetzt werden oder auch der Rotton (oder allgemein die Farbe) des Kreises.

```
void drawRedCircle(int x, int y, int size, int redValue) {
  fill(redValue, 0, 0);
  ellipse(x, y, size, size);
}
```

Es ist auch möglich die Parameter an sich noch komplexer berechnen zu lassen, bevor diese an die Funktion übergeben werden. So lassen sich zum Beispiel einfache Berechnungen innerhalb des Funktionsaufrufs durchführen:

```
void draw() {
  int size = 50;

  drawRedCircle(size - 25, 100);
  drawRedCircle(size + 25, 40);
}
```

Hier wird zuerst die Berechnung innerhalb der Klammern durchgeführt. Anschließend wird das jeweilige Ergebnis als Parameterwert übergeben und die Funktion damit ausgeführt.

Auch Funktionsaufrufe können als Parameter eingesetzt werden. Hier wird der Rückgabewert der Funktion als Parameterwert an die neue Funktion übergeben. Wie genau so eine Funktion mit Rückgabewert aussieht, wird im übernächsten Abschnitt im Detail beschrieben.

(cc) BY

7.5 Funktionen und Rückgabewerte

Der erste Teil des Funktionskopfs ist der Rückgabetyp. Der Rückgabetyp gibt Auskunft, welche Art von Information von dieser Funktion als "Ergebnis" zurückgeliefert wird. Der Rückgabewert steht an erster Stelle im Funktionskopf. Betrachten wir noch einmal die Funktion `drawRedCircle()`.

```
void drawRedCircle(int x, int y, int size, int redValue)
```

Der Funktionskopf für diese Funktion beginnt mit dem Schlüsselwort `void`. `void` bedeutet "leer", bzw. hier so viel wie "kein Rückgabetyp", das heißt, diese Funktion liefert keinen Rückgabewert bzw. keine Information zurück. Funktionen, die keinen Wert zurückliefern, nennt man auch **Prozeduren**. Keinen Rückgabewert braucht man zum Beispiel zum Zeichnen geometrischer Objekte im Sketch-Fenster, wie `line()`, `rect()`, `triangle()`, `text()`, oder auch wenn nur Einstellungen im Speicher verändert werden oder etwas ausgegeben wird, z.B. `fill()`, `noFill()`, `background()` oder `print()`.
Ein Beispiel für eine in Processing verfügbare Funktion *mit* Rückgabewert ist die Funktion `random()`, siehe auch https://processing.org/reference/random_.html

Gemäß der Dokumentation erzeugt die Funktion `random()` eine Zufallszahl. Sie erwarten sich von der Funktion also, wenn sie aufgerufen wird, dass diese eine Zufallszahl zurückliefert, die dann weiter verwendet werden kann. Der vorletzte Absatz in dieser Dokumentation beinhaltet:

Returns float

Returns gibt in der Processing Dokumentation immer den Datentyp des Rückgabewerts der Funktion an. Bei `random()` ist der Rückgabewert also vom Datentyp `float`. Das bedeutet, dass die Zufallszahl, die die Funktion generiert und die Sie dann als Ergebnis zurückgeliefert bekommen, eine Gleitkommazahl ist.

Weitere Beispiele für Processing Funktionen mit Rückgabetyp sind etwa:

`int round(float num)`
: rundet eine Gleitkommazahl *num* auf die nächstgelegene ganze Zahl (kaufmännisches Runden) und liefert diese ganze Zahl zurück

`int abs(int num)`
: berechnet den Absolutbetrag einer ganzen Zahl *num*: $|num|$ und liefert diesen als ganze Zahl zurück

`float sqrt(float num)`
: berechnet die Quadratwurzel einer Gleitkommazahl *num* $\sqrt{num}$ und liefert diese als Gleitkommazahl zurück

`float pow(float n, float e)`
: berechnet n^e und liefert das Ergebnis dieser Berechnung als Gleitkommazahl zurück

Von all diesen Funktionen erwarten Sie ein Ergebnis, das Sie in Form eines Wertes eines bestimmten Datentyps, z.B. als `float` oder `int`, erhalten. Alle oben genannten Funktionen liefern einen Wert. Diesen können Sie zum Beispiel in einer Variable speichern und dann weiter verarbeiten.

Beispiel: Die folgende Funktion summiert in einer Schleife alle Zahlen von 0 bis inklusive der übergebenen Zahl auf und gibt das Ergebnis zurück. Da die Summanden alle vom Typ `int` sind, wird auch das Ergebnis, welches in `sum` gespeichert ist, wieder vom Typ `int` sein.

```
int sumNumbers(int end) {
  int sum = 0;

  for (int i = 1; i <= end; i++) {
    sum += i;
  }

  return sum;
}

void setup() {
  int result = sumNumbers(5);
  println(result);
}
```

Ein wichtiges Schlüsselwort ist in diesem Zusammenhang `return`. Jede Funktion, die einen Rückgabetyp hat, hat im Rumpf mindestens ein `return`, um Processing mitzuteilen, welcher Wert zurückgeliefert werden soll. Im Beispiel bedeutet `return sum;` dass der Wert in der Variable `sum` zurück geliefert werden soll. Den Wert, der zurückgegeben wird, nennt man **Rückgabewert**.

Eine besondere Eigenschaft von `return` ist, dass nicht nur der Rückgabewert übergeben wird, sondern auch der Ablauf der Funktion an dieser Stelle beendet wird und im Programm an die Stelle nach dem Funktionsaufruf zurückgesprungen wird und es von dort aus weiter ausgeführt wird.

Funktionen

setup(){ ~~printInt~~ funktionsname (input 1, input 2)
als Zahl

```
int funktionsname (int input1, int input2){
    return input2;
}
```

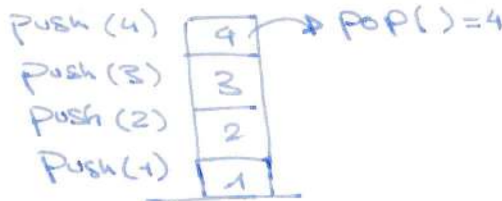
definiert output → ~~return~~ return input2;

Beispiel:
größter gemeinsamer
Teiler

Rekursive Funktionen

Rekursiv: Funktion ruft sich selbst wieder auf: ~~Endlosschleife~~

Die Idee von Stack:
(Quasi wie die Todo Liste vom Computer als ein Stapel an Blöcken)



Beispiel: Code, Ablauf
für n=3

Stack	
rec(0)	0 = 0
rec(1)	0 + rec(1) = 1
rec(2)	1 + rec(2) = 3
rec(3)	3 + rec(3)
~~...~~	print
setup	

danach
rec = 0 = 0
also pop(rec 0)

~~input für~~

Error:
Stackoverflow

7.6 Typische Fehler bei Funktionen

In diesem Abschnitt werden die häufigsten Denkfehler und Fehlermeldungen im Zusammenhang mit Rückgabetypen und Rückgabewerten vorgestellt.

Datentypkonflikt beim Rückgabewert

Der Rückgabewert muss immer vom gleichen Datentyp sein, wie er im Funktionskopf angegeben ist oder implizit gecastet werden können. Verändert man beispielsweise den Datentyp der Variable `sum` im obigen Beispiel auf `float`, dann meldet Processing

```
Type mismatch: cannot convert from float to int
```

weil Processing bei `return sum;` versucht, den Rückgabewert in den Datentyp `int` umzuwandeln (`float` kann nicht implizit auf `int` gecastet werden).

Alle Datentypen, die Sie bisher kennen, können als Rückgabetypen verwendet werden. Wenn die Funktion eine Zahl zurückliefert, dann sollte der Rückgabetyp `int` oder `float` sein. Falls die Funktion eine "Ja"/"Nein" bzw. "Wahr"/"Falsch" Frage beantwortet, dann ist der Typ `boolean` ein geeigneter Rückgabetyp.

Speichern von Rückgabewerten

Wenn Sie eine Funktion mit Rückgabewert aufrufen, denken Sie daran, den Rückgabewert dann auch zu speichern bzw. direkt zu verwenden! Benötigen Sie den Rückgabewert nur an der Stelle des Funktionsaufrufs, können Sie ihn direkt verwenden. Möchten Sie jedoch den Rückgabewert mehrfach weiter verwenden, speichern Sie ihn in einer Variable:

Beispiel: Die Funktion `sumOfEvenNum()` liefert die Summe aller geraden Zahlen zwischen dem Startwert `start` und dem Endwert `end` inklusive.

```
void setup() {
  sumOfEvenNum(1, 5);
}

//sumOfEvenNum Definition
int sumOfEvenNum(int start, int end) {
  int sum = 0;

  for (int i = start; i <= end; i++) {
    if (i % 2 == 0) {    //wenn i eine gerade Zahl ist
      sum += i;          //addiere zur Summe die gerade Zahl i dazu
    }
  }

  return sum;
}
```


Das erwartete Ergebnis nach dem Funktionsaufruf im `setup()` ist 6 (weil 2 + 4 = 6). Beim Ausführen erscheint dieser Wert aber weder im Sketch-Fenster noch in der Konsole.

Sie müssen das Ergebnis, das Sie von der Funktion erhalten, auch explizit speichern (1) oder zumindest verwenden (2):

(1) Ergebnis wird in einer Variable, hier `int result`, gespeichert und ist somit wiederverwendbar:

```
void setup() {
  int result = sumOfEvenNum(1, 5);
  println(result);
}
```

(2) Ergebnis wird direkt verwendet, z.B. innerhalb von println (und somit auf der Konsole ausgegeben):

```
void setup() {
  println(sumOfEvenNum(1, 5));
}
```

In diesem Fall wird der Rückgabewert nicht gespeichert. Daher kann er nur an dieser einen Stelle einmalig verwendet und nicht weiterverwendet werden.

Falls Sie also nach einem Funktionsaufruf ein Ergebnis erwarten, aber keines sehen, könnte das eine mögliche Fehlerquelle sein.

Kein Rückgabewert / Fehlendes return

Eine Fehlermeldung, die beim Erstellen einer Funktion mit Rückgabewert auftauchen könnte, ist:

```
This method must return a result of type int
```

Diese Fehlermeldung weist darauf hin, dass für die Funktion (Processing verwendet hier ausnahmsweise die Bezeichnung method statt wie üblich function) ein Rückgabewert fehlt, in diesem Beispiel etwa vom Typ `int`. Diese Fehlermeldung tritt auf, wenn innerhalb des Funktionsrumpfs überhaupt kein `return` steht. Es ist hier also im Funktionsrumpf die `return` Anweisung hinzuzufügen. Auch unter folgendem Umstand kann dieser Fehler auftreten:

Beispiel: Die Funktion `isEven()` liefert einen Wahrheitswert. Falls der Wert in `n` eine gerade Zahl ist, liefert die Funktion als Antwort `true`.

```
boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
  }
}
```

Auch hier wird dieselbe Fehlermeldung angezeigt, obwohl ein `return` vorhanden ist. Denn was passiert, wenn `n` keine gerade Zahl ist - dann wird die `if`-Anweisung übersprungen und das `return` gar nicht erst ausgeführt. Processing "erreicht" somit das `return` nicht. Damit gäbe es keinen Rückgabewert. Aus diesem Grund wird auch in diesem Fall die Fehlermeldung "This method must return a result of type boolean" ausgegeben. Es ist also sicherzustellen, dass - besonders bei Schleifen, Verzweigungen und verschachtelten Funktionen, in jedem möglichen Programmablauf jedenfalls eine return Anweisung ausgeführt wird.

Um den Fehler zu beheben, wäre im obigen Beispiel folgender Ansatz möglich.

```
boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
  } else {
    return false;
  }
}
```

Aber auch dieser Ansatz ist zulässig:

```
boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
  }
  return false;
}
```

Diese Variante ist möglich, da nach der Ausführung von `return` die Funktion an jener Stelle gleich abbricht und der Rest der Funktion nicht mehr ausgeführt wird. Daher wird nur für ungerade Zahlen der Wert `false` zurückgeliefert. Ist `n` gerade, so wird das `return` innerhalb der `if` Anweisung ausgeführt und die Funktion wird beendet ohne, dass das zweite `return` ausgeführt wird.

Es geht aber auch noch kürzer:

```
boolean isEven(int n) {
  return n % 2 == 0;
}
```

Das Ergebnis des Vergleichs (das ja stets ein boolean Wert ist!) wird ohne Zwischenspeicherung als Ergebnis geliefert.

(cc) BY TU Wien

Allerdings gibt es auch Fälle, wo (rein logisch gesehen) in jedem Fall ein Rückgabewert geliefert wird, aber dennoch Processing jammert, dass ein Rückgabewert fehlt:

```
boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
  } else if (n % 2 != 0) {
    return false;
  }
}
```

Für uns Menschen ist klar, dass n nur entweder gerade oder ungerade sein kann und dass daher unsere Funktion `isEven` in beiden Fällen einen Wahrheitswert zurückliefert. Processing "irrt" in diesem Fall, weil Processing die Bedingungen bei seiner Programmanalyse nicht auswertet, sondern nur nach Programmzweigen ohne `return` Ausschau hält. So könnte zum Beispiel die erste Bedingung auch `if (n % 2 == 0)` und die zweite Bedingung `else if (n % 3 == 0)` lauten. In diesem Fall würden wir einen `else`-Zweig zwingend benötigen, um immer einen Rückgabewert für diese Funktion gewährleisten zu können.

Jedenfalls weist solch eine Fehlermeldung, wenn der Code für uns Menschen fehlerfrei erscheint, Processing ihn jedoch anders interpretiert, meist auf einen schlechten Programmierstil hin. Grundsätzlich sollte man versuchen immer alle möglichen auftretenden Fälle zu berücksichtigen, um sicherzustellen, dass jedenfalls eine `return`-Anweisung erreicht wird.

Unreachable code

Die Fehlermeldung

Unreachable code

wird dann angezeigt, wenn Processing erkennt, dass es Programmcode gibt, der aufgrund des Funktionsabbruchs an einer `return` Anweisung nie ausgeführt werden kann. Das kann beispielsweise ein Codeblock sein, der direkt nach einem `return` folgt.

```
boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
    println("I returned true");
  } else {
    return false;
    println("I returned false");
  }
}
```

(cc) BY TU Wien

Auch im folgenden Fall erkennt Processing, dass Codeabschnitte nie erreicht werden können:

```
boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
  } else {
    return false;
  }

  println("I returned something");
}
```

Sie sollten sich aber auch hier nicht darauf verlassen, dass Processing immer jeden unerreichbaren Code erkennt. Im folgenden Fall wird kein Fehler angezeigt:

```
boolean isEven(int n) {
  if (n % 2 == 0) {
    return true;
  } else if (n % 2 != 0) {
    return false;
  }

  println("I returned something");
  return false;
}
```

Der `if`-Block und der `else if`-Block decken gemeinsam alle Möglichkeiten ab. Es wäre daher nicht möglich, egal welchen Wert `n` hat, dass das letzte `println()` und das `return false;` ausgeführt wird. Dennoch wird dieser Fall nicht von Processing erkannt. In diesem Fall kaschiert der erste Fehler wie bereits im Abschnitt "Fehlendes Return" beschrieben, den zweiten Fehler, dass bestimmte Codeteile nie erreicht werden können.

(cc) BY TU Wien

7.7 Funktionen ⟵⟶ `draw()` und `setup()`

Betrachtet man die beiden Bereiche `draw()` und `setup()` genauer, dann wird man vermutlich eine Ähnlichkeit zu den gerade erlernten Funktionen erkennen. Und tatsächlich sind sowohl `draw()` als auch `setup()` im Grunde auch Funktionen, ganz präzise: Prozeduren. Beide besitzen keinen Rückgabetyp und benötigen keine Parameter. Aber warum werden diese ausgeführt, ohne dass man den Befehl explizit wo hinschreibt, wie es auch bei allen anderen Funktionen der Fall war?

Das liegt daran, dass der Aufruf durch Processing im Hintergrund selbst geschieht. Wie auch in vielen anderen Programmiersprachen, braucht Processing einen Einstiegspunkt, von dem an beginnend das Programm ausgeführt wird. Es ist eine für uns Programmierer nicht sichtbare Funktion, die zunächst einmal die Funktion `setup()` aufruft, ohne dass Sie es sehen. Und anschließend in einer für uns nicht sichtbaren Schleife, die die `draw()`-Funktion wiederholt aufruft. Diese Schleife wird solange ausgeführt, bis der User z.B. das Fenster schließt. Sie sehen also, auch hier findet bereits eine Abstraktion statt.

Der ungefähre Code im Hintergrund könnte vereinfacht beispielsweise so aussehen:

```
setup();
boolean quit = false;

// Falls der User das Programm noch nicht beendet hat
while(!quit) {
  checkUserInput(); // prüft, ob der User das Programm beenden will
  draw(); // Ihr Code wird ausgeführt
  switchCanvas(); // das in draw() Gezeichnete wird sichtbar gemacht
}
```

Zuerst wird die `setup()` Funktion ausgeführt. Danach wird eine `boolean` Variable `quit` deklariert und initialisiert, die speichert, ob das Programm beendet werden soll oder nicht. In einer Schleife werden mit der Funktion `checkUserInput()` bei jedem Durchlauf die Usereingaben überprüft. In dieser Funktion wird beispielsweise auch geprüft, ob der User das Programm durch den Stop-Button beenden möchte. Falls dies der Fall ist, wird die Variable `quit` auf `true` gesetzt und die Schleife wird danach nicht mehr ausgeführt. Nachdem die Interaktionen mit dem User überprüft wurden, wird die `draw()` Funktion ausgeführt und dann alles, was intern gezeichnet wurde mit der Funktion `switchCanvas()` auch am Sketch-Fenster sichtbar gemacht. Vergessen Sie nicht, dass das Ergebnis von `draw()` daher erst nach dessen Ausführung im Sketch-Fenster ausgegeben wird. Es wird also NICHT jeweils unmittelbar nach Ausführung eines einzelnen (Zeichen-)Befehls innerhalb `draw()`, wie z.B. `rect()`, `arc()`, usw. aktualisiert.

Dieser Code ist nur eine Annäherung an das, was in Processing ablaufen könnte. Da wir keinen Einblick in den Code haben, wissen wir nicht, was genau im Hintergrund von Processing ausgeführt wird.

(cc) BY TU Wien

7.8 Sichtbarkeit von Parametern und Funktionsvariablen

Gerade im Bezug auf Funktionen spielt die Sichtbarkeit der Variablen eine wichtige Rolle. Auch hier gilt die Regel: Alles was innerhalb der geschwungenen Klammern deklariert wird, ist nur innerhalb dieser geschwungenen Klammern sichtbar und verwendbar. Auf eine Variable, die innerhalb einer Funktion deklariert wurde, kann nicht in einer anderen Funktion zugegriffen werden. Dies haben Sie bereits in Kapitel 3 anhand von `draw()` und `setup()`, die ja ebenso Funktionen sind, kennen gelernt. Besonders im Zusammenhang mit Parametern und Rückgabewerten ist dies zu berücksichtigen.

Beispiel: Gehen Sie folgendes Beispiel durch und überlegen Sie sich, welche Werte für die Variablen a und b in `println(a, b);` ausgegeben werden.

```
void setup() {
  int a = 5;
  int b = 10;
  addNumbers(a, b);
  println(a, b);
}

void addNumbers(int a, int b) {
  a += b;
}
```

Am Anfang kommt man vielleicht in die Versuchung zu glauben, die Variable a müsse den Wert 15 haben. Schließlich wird der Wert von a an die Funktion `addNumbers()` übergeben und innerhalb der Funktion verändert. Dies hat aber keine Auswirkung auf die Variablen a und b der `setup()`-Funktion. Denn die **übergebenen Werte** werden für die Funktion **kopiert**. Die Veränderung findet daher in der Kopie statt, nicht aber in der ursprünglichen Variable! Daher sind auch Rückgabewerte notwendig, um Veränderungen wieder zurückzuliefern.

Soll das Ergebnis von `addNumbers(a, b);` in die Variable a in `println(a, b);` "aktualisiert" werden, dann wäre eine korrekte Version für dieses Beispiel folgende:

```
void setup() {
  int a = 5;
  int b = 10;
  a = addNumbers(a, b);
  println(a, b);
}

int addNumbers(int a, int b) {
  a += b;
```

(cc) BY TU Wien


```
  return a;
}
```

Hier wird in `addNumbers()` der in der Kopie von a berechnete Wert mit `return` zurückgeliefert. Entsprechend muss der Rückgabetyp angepasst werden. In `setup()` wird der Rückgabewert an die Variable a zugewiesen. a erhält dadurch diesmal tatsächlich den in `addNumbers()` berechneten Wert.

(cc) BY TU Wien

7.9 Dokumentation von Funktionen

Die Dokumentation einer Funktion mag zuweilen als lästige Arbeit empfunden werden, aber sie ist unerlässlich und von großer Bedeutung, denn sie hilft anderen Programmiererinnen und Programmierern dabei, die Funktion besser zu verstehen und korrekt anzuwenden, ohne den Code Zeile für Zeile zu lesen. Es ist daher ratsam, auf ein sorgfältiges Dokumentieren der selbst programmierten Funktionen von Beginn an Wert zu legen. Nicht zuletzt hilft die Dokumentation auch Ihnen selbst als Programmiererin oder Programmierer - etwa, wenn Sie Programme nach längerer Zeit wieder verwenden möchten und sich nicht mehr an jedes Detail erinnern.

Bei einfachen Funktionen sagt ein aussagekräftiger Funktionsname bereits vieles über die Funktion aus. Trotzdem gehört eine Dokumentation zum guten Programmierstil. Spätestens bei komplexeren Funktionen, wo der Funktionsname nicht mehr alles Wichtige ausdrücken kann und mögliche Parameter nicht selbsterklärend sind, sind Funktionen genau zu dokumentieren, um den Anwenderinnen und Anwendern die Verwendung der Funktion zu erleichtern.

Was ist Dokumentation?

Die Dokumentation einer Funktion beinhaltet nicht nur eine Beschreibung, was die Funktion macht, sondern auch, welche Parameter sie verwendet, wofür diese stehen und Informationen zu ihrem Rückgabewert. Außerdem sollten alle **Vorbedingungen und besonderen Eigenheiten**, die alleine vom Funktionsnamen nicht abgeleitet werden können, notiert werden. Einen Anhaltspunkt zur Dokumentation liefert die Processing API. Diese besteht nämlich zu einem guten Teil aus der Dokumentation jener Funktionen, die in Processing verfügbar sind.

Als mögliches Beispiel die Dokumentation zur Processing Funktion `size()`:

https://processing.org/reference/size_.html

Die Beschreibung der Funktion beinhaltet neben ihrer eigentlichen Funktion auch Anwendungsbeispiele sowie Hinweise zu besonderen Eigenheiten: Im Falle von `size()` etwa kann man aus der Dokumentation herauslesen, dass diese Funktion nicht überall und jederzeit und nicht beliebig oft aufgerufen werden kann.
Je besser solche Eigenheiten dokumentiert werden, desto weniger wird es den Anwender/die Anwenderin später überraschen, wenn etwas nicht wie gedacht funktioniert.

(cc) BY TU Wien

Wie dokumentiert man?

Dokumentationen werden in Form von Kommentaren über den zugehörigen Funktionskopf geschrieben. So ist sie auf den ersten Blick verfügbar und ersichtlich.
Wichtig ist, dass sie alle relevanten Informationen enthält, um die Funktion zu verstehen und korrekt einzusetzen. Bei komplexen Funktionen ist außerdem darauf zu achten, keine Eins-zu-Eins-Übersetzung des Codes zu schreiben (z.B. "Es wird eine Variable leftOffset angelegt für den Offset von links. Dann wird Variable topOffset angelegt ... " etc.). Besser ist es die Wirkung der Funktion als Gesamtes in knapper Form so prägnant wie möglich zusammen zu fassen.

Die Dokumentation für die Funktion "maximum" könnte in etwa so aussehen:

```
/*
  Gibt die größte von drei gegebenen ganzen Zahlen zurück
  a: erste Zahl
  b: zweite Zahl
  c: dritte Zahl
*/
int maximum(int a, int b, int c) {
  int max = a;

  if (max < b) {
    max = b;
  }

  if (max < c) {
    max = c;
  }

  return max;
}
```

Dokumentieren mit JavaDoc

Mit speziellen Tools (z.B. Doxygen[1]), können aus Kommentaren von Funktionen auch formatierte und strukturierte Dokumentationen als HTML Seiten oder LaTeX Manuals generiert werden und die Ausgabe als RTF (MS-Word), PostScript oder hyperlinked PDF unterstützen. Dafür sind diese Kommentare in einer bestimmten Form zu beschreiben. Ein weit verbreiteter Stil für solche Generatoren ist der JavaDoc Style[2] von Oracle.
Der allgemeine Aufbau einer Funktionsdokumentation im JavaDoc Style ist im Folgenden anhand des obigen Beispiels dargestellt:

[1] http://www.stack.nl/~dimitri/doxygen/
[2] http://www.oracle.com/technetwork/articles/java/index-137868.html

(cc) BY TU Wien

```
/**
 * Gibt die größte von drei gegebenen Zahlen zurück
 *
 * (funktioniert für positive und negative Werte)
 *
 * @param a erste Zahl
 * @param b zweite Zahl
 * @param c dritte Zahl
 * @return Maximum der drei Zahlen
 */
int maximum(int a, int b, int c) {
  int max = a;

  if (max < b) {
    max = b;
  }

  if (max < c) {
    max = c;
  }

  return max;
}
```

Der Dokumentations-Kommentarblock wird in JavaDoc mit `/**` eingeleitet.
Es werden nach dem Schrägstrich jetzt **zwei Sterne** verwendet anstatt des herkömmlichen einzelnen Sterns für einen herkömmlichen Kommentarblock.
Die erste Zeile beinhaltet eine Kurzbeschreibung der Funktion. Der nächste Absatz enthält detaillierte Informationen. Hier werden z.B. besondere Eigenschaften der Funktion dokumentiert. Danach folgen die Beschreibungen der Parameter. Für jeden Parameter ist ein separater Eintrag anzulegen, welcher mit `@param` eingeleitet wird. Dem folgt der Name und die Verwendung des Parameters. Als letztes wird der Rückgabewert kurz beschrieben. Dieser wird mit `@return` eingeleitet. Gibt es keinen Rückgabewert (steht als Rückgabetyp also `void`) oder werden keine Parameter benötigt, dann werden die entsprechenden Zeilen ausgelassen.

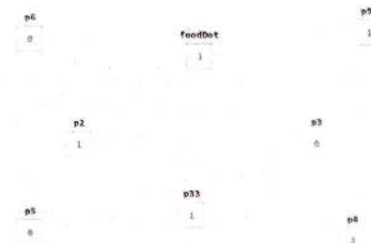
(cc) BY

8. Arrays

8.1 Informationen gruppieren und strukturieren

In unserem Alltag haben wir es häufig eben nicht mit einzelnen Informationen zu tun, sondern mit großen Informationsmengen. Diese werden oft in Listen und Tabellen abgelegt bzw. dargestellt und verarbeitet. So werden bei Sportveranstaltungen wie Marathons oder Schirennen die Teilnehmenden in Listen mit Startnummern geordnet. Kundinnen und Kunden eines Unternehmens wird eine Kundennummer zugeordnet und diese sowie weitere Informationen werden in Tabellen oder Datenbanken gespeichert. Tabellen bzw. Listen finden sich beispielsweise aber auch bei Bestellungen und Rechnungen. Es können auch Einkaufs- oder To-Do-Listen sein oder Tabellen zur Verwaltung von Adressen, Büchern in Bibliotheken oder Zeitaufzeichnungen uvm.
Informationen in Listen oder Tabellen zu strukturieren, erleichtert es uns, den Überblick über größere Informationsmengen zu behalten. Insbesondere dann, wenn Listen bzw. Tabellen nach bestimmten Kriterien oder Werten sortiert oder gefiltert werden. Entsprechend der Reihenfolge der Anordnung werden oft die Einträge der Liste dann abgearbeitet.

In der Programmierung konnten wir mit Variablen bisher jeweils einzelne Werte speichern. Ausschließlich mit Variablen ist es aber nicht praktikabel größere Datenmengen, z.B. Alter oder Namen von 1000 Menschen, effizient zu verarbeiten. Denn dazu müssten wir 1000 Variablen anlegen und wären dadurch auch auf 1000 Einträge beschränkt. Auch das "Durchlaufen" dieser Variablen, um bestimmte Werte aufzufinden, gestaltet sich als sehr aufwändig. Kurzum, wir wären in der Programmierung ohne das Konzept einer Gruppierung wie Listen und Tabellen, wie wir das auch vom Alltag her kennen, ziemlich eingeschränkt. Eine Möglichkeit, wie Informationen in der Programmierung bzw. mit Processing gruppiert werden können, sind sogenannte Arrays (deutsch: Felder bzw. Datenfelder). Wie diese umgesetzt und eingesetzt werden, ist Inhalt dieses Kapitels.

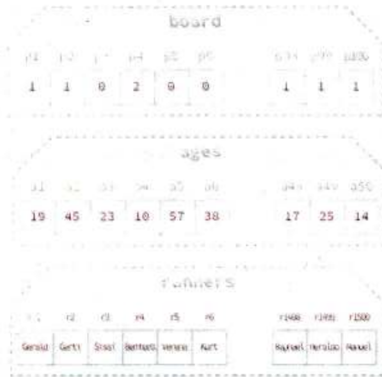


Variablen im Speicher (Schema)

(cc) BY

Beispiele für die Motivation von Arrays wären etwa

- das Speichern, Auslesen und Bearbeiten von 100 Spielfeld-Elementen beim Pacman (vgl. untenstehende Grafik: "board")
- das Speichern des Alters von 50 Personen, um nach bestimmten Werten zu suchen (vgl. untenstehende Grafik: "ages")
- das Speichern, Auslesen und Bearbeiten der Namen von 1500 Marathonläuferinnen und -läufern mit deren Startnummer (vgl. untenstehende Grafik: "runners")



Idee von Listen im Speicher (Schema)

8.2 Was sind Arrays?

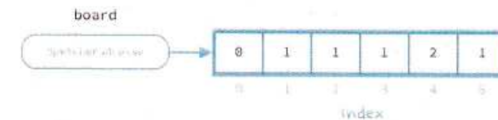
In Kapitel 3 wurden Variablen mit Gefäßen verglichen, welche je nach Typ Inhalte einer bestimmten Art fassen können. Nun könnte man auch Regale bauen, auf welche eine bestimmte Anzahl gleicher solcher Gefäße gestellt werden können. Das Regal wird mit einem Aufkleber mit dessen Namen versehen und das zweite Gefäß auf dem Regal könnte dann mit "im Regal mit dem Namen 'beispielregal', das zweite Gefäß" angesprochen werden. Arrays kann man sich als solche Regale vorstellen. Beim Erstellen wird festgelegt, welche Art von Gefäßen darauf gestellt werden können (der Regaltyp) und wie viele Gefäße darauf Platz haben (die Länge eines solchen Regals).

Ein Array kann man sich zudem vereinfacht auch als nummerierte Liste vorstellen. Bei vielen Sportbewerben, etwa Marathons, Schirennen, Eiskunstlauf oder Radrennen, usw., werden Startnummernlisten der Athletinnen und Athleten erstellt. Für jede Nummer der Startliste wird dann der Name einer Sportlerin bzw. des Sportlers zugelost oder in der Reihenfolge der Anmeldung zugewiesen. Auch das Endergebnis eines Bewerbes wird in einer Liste bzw. Tabelle dargestellt - jede Sportlerin bzw. jeder Sportler erreicht einen bestimmten Platz im Ranking (1. Platz, 2. Platz, etc.) - je nach erreichter Zeit, Punkteanzahl, Distanz bzw. weiterer Kriterien. Weitere bekannte Beispiele sind die Tabellen von Fußball-, Basketball- oder Hockey-Ligen.

Wie bildet man solche Strukturen - Arrays - in der Programmierung ab? Wie können in Processing nun mehrere zusammengehörige Werte geordnet gespeichert und verarbeitet werden?

In der Programmierung ist ein Array ein weiterer Datentyp. Das Besondere an diesem Datentyp ist, dass Array-Variablen nicht nur einen einzelnen Wert eines bestimmten Typs speichern können (wie Variablen das machen), sondern mehrere (wie bei einer Liste) Werte des gleichen Datentyps: zum Beispiel 2.000 Teilnehmerinnen und Teilnehmer eines Marathons, den Inhalt (Futterpunkt, Kraftpille, leeres Feld,...) von 99 Spielfeldelementen im Pacman-Spiel oder die Ergebnisse einer täglichen Blutdruckmessung eines Jahres.

Bei einem einfachen Pacman-Spielfeld mit 6 Elementen könnte das dann folgendermaßen aussehen:



Beispiel für ein Pacman-Spielfeld mit sechs Elementen: Grafische Spielfeldinhalte und deren Codierung (oben) sowie schematische Darstellung als Array mit Namen *board* im Speicher (unten)

Die einzelnen Werte in einem Array sind fortlaufend nummeriert. Dadurch können die

einzelnen Werte über diese fortlaufende Nummer, dem sogenannten **Index**, angesprochen werden. Da die Nummerierung der einzelnen Array-Elemente mit dem Index fortlaufend ist, können Arrays wunderbar mit Schleifen kombiniert werden, um zum Beispiel alle Elemente der Reihe nach abzufragen.

Achtung: Wie das in der Programmierung oft der Fall ist, beginnt der Index bei 0!

8.3 Arrays erstellen und verwenden

Deklaration von Arrays

Das Deklarieren von Arrays ähnelt dem der bereits bekannten Variablen:

```
float[] temperatures;
String[] startingList;
int[] ageList;
```

allgemein:

```
datentyp[] arrayName;
```

An erster Stelle steht auch hier der gewünschte Datentyp. Dieser Datentyp bestimmt bei Arrays, welche Arten von Information in den einzelnen Elementen des Arrays abgelegt werden können. Alle bisher vermittelten Datentypen können als Array-Datentyp eingesetzt werden (`int`, `float`, `boolean`, `String`, etc.).
Um zu kennzeichnen, dass es sich nun eben nicht um eine "herkömmliche" Variable, sondern um ein Array handelt, wird an den Datentyp ein eckiges Klammernpaar angehängt.

Dem folgen, wie gewohnt, der gewünschte Name des Arrays und der Strichpunkt als Abschluss der Anweisung.

Hinweis: Ein Array speichert eine bestimmte Anzahl von Werten **eines einzelnen Datentyps**!

Das Array

- `float[] temperatures;` etwa speichert Temperaturwerte, von denen jeder einzelne eine Gleitkommazahl ist.
- `int[] ageList;` wiederum beinhaltet Altersangaben in ganzen Jahren.

Achtung: Es ist **nicht** möglich, unterschiedliche Arten von Daten in einem einzigen Array zu kombinieren! So können in einem Array etwa nicht ein String als auch ein Integer gespeichert werden.

Bei der Deklaration eines Arrays mit zB. `int[] ageList;` ist nur bekannt, dass das Array `ageList` heißt, aber die Länge des Arrays ist noch nicht bekannt und auch das Array mit den einzelnen Arrayelementen ist im Speicher noch nicht angelegt. Die Speicheradresse des Arrays beinhaltet zu diesem Zeitpunkt den besonderen Wert `null`. Das heißt, dass die Speicheradresse nicht bekannt ist und somit Zugriffe auf die Arrayelemente nicht möglich sind.

```
int[] ageList;
```

ageList

Deklaration eines Arrays (Speicherschema)

Instanziieren und Initialisieren

Aber anders als bei Variablen handelt es sich bei dem Datentyp Array nun um eine Erweiterung der bereits bekannten Datentypen für Variablen - um ein Objekt, das aus mehreren einzelnen Elementen desselben Datentyps, zum Beispiel Integer, Float, String oder Boolean, besteht.

Daher kommt nach der Deklaration noch ein Schritt hinzu - das Array muss (im Speicherbereich) zusätzlich erzeugt, also erstellt, werden. Dieses Erzeugen nennt man beim Programmieren oft auch "Instanziieren". Ist das Array instanziiert, erfolgt auch bei Arrays eine erste Wertzuweisung - die Initialisierung. Für die Instanziierung und Initialisierung von Arrays gibt es verschiedene Möglichkeiten:

Instanziierung mit fixer Länge:
Ein Array kann mit der gewünschten Anzahl von Elementen instanziiert werden. Die einzelnen Elemente des Arrays werden dabei automatisch mit Standardwerten initialisiert. Die Anzahl der Elemente eines Arrays wird die "Länge des Arrays" genannt.

```
int[] ageList;            Deklaration des Arrays ageList
ageList = new int[5];     Instanziierung und implizite Initialisierung mit Nullen
```

Schemadarstellung im Speicher: Deklaration des Arrays ageList (links), Instanziierung und implizite Initialisierung des Arrays ageList mit Nullen (rechts).

Es können Deklaration, Instanziierung und implizite Initialisierung auch zu einer einzigen Anweisung zusammengefasst werden:

```
int[] arrayName = new int[5];
```

Bei dieser Variante erfolgt auf der rechten Seite der Anweisung die Instanziierung durch Angabe, dass ein neues Integer-Array (`new int[5]`) angelegt werden soll. Innerhalb der eckigen Klammern wird die Größe des Arrays in Anzahl von Elementen angegeben. Das hier angelegte Array kann beispielsweise genau 5 Elemente speichern.
Die Größe des Arrays kann auch über eine Variable bestimmt werden:

(cc) BY TU Wien


```
int arraySize = 10;
int[] arrayName = new int[arraySize];
```

Wird für ein Array eine Größe bzw. Länge festgelegt, egal ob direkt mit einem Wert oder über eine Variable, so ist diese Länge nicht mehr veränderbar. Wird das Array über die Größe bzw. Länge instanziiert, dann wird jedem Element im Array automatisch ein Standardwert (engl.: default value) zugewiesen. Das bedeutet, ohne explizite Zuweisungen wird jedes Element des Arrays automatisch mit dem Standardwert initialisiert. In der Tabelle ist zusammengefasst, welche Standardwerte den Array-Elementen je nach Datentyp zugewiesen werden:

Datentyp	Standardwert
int	0
float	0.0
boolean	false
char	'\u0000'
String	null

Den einzelnen Elementen eines Arrays können natürlich nach dieser anfänglichen Initialisierung jederzeit später neue Werte zugewiesen werden.

Initialisierung durch Angabe der konkreten Werte
Eine andere Möglichkeit ein Array zu initialisieren ist es, direkt die gewünschten Werte anzugeben. Hier erfolgt die Instanziierung mittels `new int[]` und unmittelbar nachfolgend die Initialisierung durch Angabe der Werte in geschwungenen Klammern:

```
int[] array2 = new int[]{5, 3, 4, 5, 1};
```

Zwischen den eckigen Klammern bei der Instanziierung wird nun **nicht** die Länge des Arrays angegeben. Stattdessen werden in den nachfolgenden geschwungenen Klammern die gewünschten Werte gelistet. Die Anzahl der Werte in den geschwungenen Klammern bestimmt die Länge des Arrays. Auch hier kann man entweder alles in einer Zeile schreiben, wie oben, oder auf 2 Schritte aufteilen:

```
int[] array2;
array2 = new int[]{5, 3, 4, 5, 1};
```

In diesem Beispiel hat das Array 5 Elemente. Diese Länge ist später nicht mehr veränderbar. Die Werte der einzelnen Elemente selbst können hingegen jederzeit verändert werden. Zu beachten ist, dass die Reihenfolge der angegebenen Werte genau wie

(cc) BY TU Wien

angegeben übernommen werden. Im obigen Beispiel bedeutet dies, dass der Wert 5 dem ersten Array-Element zugewiesen wird, der Wert 3 dem zweiten, usw.

Eine einfachere bzw. noch kürzere Schreibvariante ist:

```
int[] array2 = {5, 3, 4, 5, 1};
```

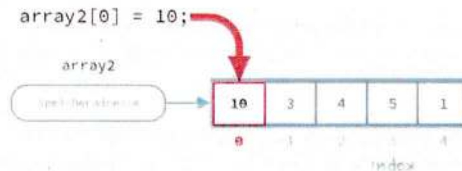
Diese Form der Initialisierung (mit impliziter Instanziierung) ist jedoch nur erlaubt, wenn Deklaration und Initialisierung in einer Zeile geschrieben werden, denn Processing führt im Hintergrund automatisch die Instanziierung durch. Wird die Deklaration getrennt, zeigt Processing eine Fehlermeldung an.

Auf Array Elemente zugreifen

Ist ein Array deklariert, instanziiert und initialisiert, kann der Wert jedes Elements im Array ausgelesen werden und es können natürlich auch jedem einzelnen Element des Arrays neue Werte zugewiesen werden. Um ein einzelnes Element in einem Array anzusprechen, werden der Name des Arrays und der Index des entsprechenden Elements benötigt. Dabei ist zu beachten, dass das erste Element eines Arrays in Processing immer den Index 0 hat.

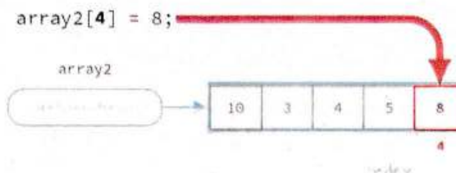
Soll dem ersten Element des Arrays `array2` der Wert 10 zugewiesen werden, schreibt man:

```
array2[0] = 10;
```



Innerhalb der eckigen Klammern nach dem Arraynamen wird nun also der Index jenes Elements geschrieben, das "angesprochen" werden soll (siehe nachfolgende Visualisierung). Die rechte Seite der Zuweisung wird genau wie bei Zuweisungen an Variablen programmiert.

Da der Index bei 0 beginnt, ergibt sich, dass das **letzte Element von insgesamt 5 Elementen den Index 4** hat:



Die gespeicherten Werte können auch für Berechnungen oder einfache Zuweisungen verwendet werden:

```
int[] array3 = {5, 3, 4, 5, 1};
```



```
int min = array3[4];          // weist den 5. Wert von array3 der
                              // Variable min zu.

int sum = array3[0] + array3[1];
                              // addiert den 1. und 2. Wert von array3
                              // und weist die Summe der Variable sum zu.
```

Beispiel: Das folgende Code-Beispiel illustriert die Verwendung von Arrays anhand einer grafischen Anwendung. Der Programmcode zeichnet konzentrische Kreise. Die Größen der Kreise sind in diesem Beispiel im Array `circleSizes` gespeichert.

```
void setup() {
  size(200, 200);

  int[] circleSizes = {220, 130, 80, 60, 10};
  int x = width/2;
  int y = height/2;

  for (int i = 0; i < circleSizes.length; i++) {
    fill(circleSizes[i]);
    ellipse(x, y, circleSizes[i], circleSizes[i]);
  }
}
```

Der Ausdruck `circleSizes.length` liefert in Processing die Länge bzw. Größe des Arrays `circleSizes` - das heißt, die Anzahl der Elemente im Array `circleSizes` - als `int`-Wert. Daher kann mit der Abbruchbedingung `i < circleSizes.length` die Schleife genau so oft ausgeführt werden, wie es Elemente im Array `circleSizes` gibt. In jedem Schleifendurchlauf wird schließlich das Array `circleSizes` über den Index i ausgelesen und zuerst als Grauwert bei der Füllfarbe und danach auch für die Größe des Kreises verwendet.

Um also die Elemente eines Array (hier: `arrayName`) der Reihe nach zu "durchlaufen", kann eine `for`-Schleife mit folgendem Aufbau verwendet werden (der `arrayName` ist dem tatsächlichen Namen des Arrays anzupassen):

```
for (int i = 0; i < arrayName.length; i++){
      print(arrayName[i]);    // gibt den Wert des "aktuellen"
                              // Elements (am Index i) im Array aus
}
```

(cc) BY

Zuweisung eines Arrays an ein anderes Array

Ein Array kann auch einem anderen Array zugewiesen werden:

```
int[] arrayName = new int[]{5, 3, 4, 5, 1};
int[] array2 = arrayName;
```

Allerdings verhalten sich Arrays hier anders als Variablen!

Lesen Sie dazu folgenden Code durch und schreiben Sie sich die von Ihnen erwartete Ausgabe dieses Programmcodes auf:

```
void setup() {
  int[] firstArray = {5, 3};
  int[] secondArray = firstArray;
  firstArray[0] = 10;

  println("Erste Ausgabe");
  println(firstArray[0]);
  println(secondArray[0]);

  println(" ------------------ ");

  firstArray = new int[]{5, 3};
  secondArray = new int[2];
  secondArray[0] = firstArray[0];
  firstArray[0] = 15;

  println("Zweite Ausgabe");
  println(firstArray[0]);
  println(secondArray[0]);

}
```

Möglicherweise haben Sie erwartet, dass im ersten Block der println() - Ausgabe, in der Konsole folgendes ausgegeben wird:

```
Erste Ausgabe
10
5
 ------------------
```

Allerdings ist die tatsächliche Ausgabe:

```
Erste Ausgabe
10
10
 ------------------
```

(cc) BY TU Wien

Was ist hier passiert? Arrays verhalten sich bei der Zuweisung anders als etwa die bisher bekannten einfachen Typen!

Im Gegensatz zu den bisher bekannten Datentypen wird bei einer Zuweisung mit Array nicht der ganze Inhalt kopiert. Vielmehr wird bei dieser Zuweisung nur die Speicheradresse des Arrays kopiert, der dahinter liegende Speicher hat dann zwei unterschiedliche Namen. Also zwei Variablennamen verweisen auf das gleiche Array, auf "denselben Inhalt".

Stellen Sie sich das ganze anhand der Analogie mit Gefäßen für Variablen vor. Ein Array könnte dann abgebildet werden als ein Regal mit Unterteilungen, auf welchem eine bestimmte Anzahl derselben Gefäße Platz hat, so wie in der folgenden Abbildung dargestellt.
Die Anzahl der Gefäße wird durch die Größe des Arrays angegeben, die Art der Gefäße durch den Datentyp der Array-Elemente.

Bei der Zuweisung wird nun nicht ein zweites Regal mit denselben Gefäßen und Inhalten erstellt! Sondern, es wird ein zweites Etikett mit dem "neuen" Array-Namen an das vorhandene Regal dazu geklebt. Wenn das Regal zuerst Toni gehört hat, könnten Sie zum Beispiel noch "Maries Regal" hinzufügen, dann würden das Regal dann beiden Personen, sowohl Toni als auch Marie "gehören" - sie würden es quasi teilen.

Zuweisung: Maries Regal = Tonis Regal (Schema)

Das bedeutet aber auch: Wenn der Inhalt eines Gefäßes (d.h. Array-Elements) für "Tonis Regal" bzw. der gesamte Inhalt des Arrays geändert wird, dann wird auch Maries Regal von der Änderung betroffen sein. Wird zum Beispiel das Gefäß 0 auf "Maries Regal" mit Wasser angefüllt, dann ist auch Tonis Gefäß 0 mit Wasser gefüllt, denn - beide Personen beziehen sich auf ein und dasselbe Regal und damit auch auf das identische Gefäß.

Anders ist es, wenn Sie das Array mit `new int[arraySize]` initialisieren. Hier legen Sie tatsächlich ein neues Regal an, bzw. neuen Speicherplatz.
Bei der Zuweisung `secondArray[0] = firstArray[0];` wird der Inhalt des Elements (hier mit dem Index 0) kopiert. Daher hat `firstArray[0] = 15;` keine Auswirkung am Array `secondArray`. Die Ausgabe vom zweiten `println()`-Block ist daher:

```
Zweite Ausgabe
15
5
```

(cc) BY TU Wien

Wann immer Sie ein Array einem anderen Array zuweisen, müssen Sie im Hinterkopf behalten, dass die Änderung eines Elements in einem Array auch das zweite Array automatisch betrifft, da hier KEINE Kopie des Arrays entstanden ist.
Dies gilt auch, wenn Sie Arrays als Parameter einer Funktion verwenden.

```
void setup() {
  int[] firstArray = {5, 3};
  println("before change: " + firstArray[0]);
  changeArray(firstArray);
  println("after change: " + firstArray[0]);
}

void changeArray(int[] a) {
  a[0] = 15;
}
```

Beim Ausführen dieses Programms wird die folgende Ausgabe erzeugt:

```
before change: 5
after change: 15
```

Beim Funktionsaufruf `changeArray(firstArray);` wird nicht das gesamte Array bei der Parameterübergabe kopiert, sondern nur die "Speicheradresse" übergeben. Die Zuweisung `a[0] = 15;` in der Funktion `changeArray()` findet daher nicht an einer Kopie statt sondern am Original, weswegen der Inhalt von `firstArray[0]` verändert wird.

Arrays kombinieren mit Schleifen

Der große Vorteil eines Arrays ist, dass es mehrere "zusammengehörige" Elemente geordnet speichert. Jedes einzelne Element kann über seinen Index angesprochen werden. Der Index ist eine fortlaufende Nummerierung der Elemente im Array. Das kann dazu genutzt werden, um mit Schleifen elegant und effizient auf alle Werte im Array der Reihe nach zuzugreifen. Im Gegensatz zu den bisher bekannten Datentypen, welche jeweils nur einen einzelnen Wert speichern konnten, ermöglichen Arrays eine Gruppierung von Werten. Dies soll das folgende Beispiel illustrieren:

Beispiel: Fünf Personen wurden nach ihrem Alter gefragt und dieses wurde jeweils notiert. Nun soll das größte Alter gefunden werden. Ohne Arrays, d.h. mit "einfachen" Variablen, würde das Beispiel folgendermaßen aussehen:

```
void setup() {
  int age1 = 18;
  int age2 = 30;
  int age3 = 24;
  int age4 = 19;
  int age5 = 18;
```

(cc) BY


```
  int maxAge = age1;

  if (maxAge < age2) {
    maxAge = age2;
  }

  if (maxAge < age3) {
    maxAge = age3;
  }

  if (maxAge < age4) {
    maxAge = age4;
  }

  if (maxAge < age5) {
    maxAge = age5;
  }

  println(maxAge);
}
```

Zur Ermittlung der ältesten von fünf Personen sind vier `if`-Anweisung notwendig. Sind mehr Altersangaben zu vergleichen, dann sind die Fallunterscheidungen entsprechend zu erweitern. Dabei sind diese Abfragen immer ähnlich: "Falls das momentane maximale Alter kleiner ist als ..., dann setze das maximale Alter auf ...".

Daher könnte diese Abfrage effizienter mit einer Schleife automatisiert wiederholt umgesetzt werden. Mit Variablen ist dies jedoch so nicht umsetzbar, da für den Computer zwischen den angelegten Variablen kein Bezug steht - auch wenn die Variablen für uns Menschen fortlaufend nummeriert sind.

Folgendes Codebeispiel ist von der Grundidee her richtig, würde aufgrund der fehlenden Gruppierung der Werte aber nicht funktionieren:

```
void setup() {
  int age0 = 18;
  int age1 = 30;
  int age2 = 24;
  int age3 = 19;
  int age4 = 18;

  int maxAge = age0;

  for (int i = 1; i < 5; i++) {
    if (maxAge < age[i]) {
      maxAge = age[i];
    }
  }
```

(cc) BY

```
  println(maxAge);
}
```

Um eben diese notwendige Gruppierung zu realisieren, werden Arrays verwendet.

```
void setup() {
  int[] age = {18, 30, 24, 19, 18};

  int maxAge = age[0];

  for (int i = 1; i < 5; i++) {
    if (maxAge < age[i]) {
      maxAge = age[i];
    }
  }

  println(maxAge);
}
```

Dank des Arrays sind nun die einzelnen Elemente intern mit einem Index, also einer Zahl, versehen. Diesen Zusammenhang versteht auch der Computer. Damit lässt sich das Array sehr gut mit Schleifen kombinieren. In diesem Beispiel wird nun tatsächlich jedes einzelne Element des Arrays durchgegangen und mit dem momentan größten Alter verglichen.

Die Länge des Arrays, also die Anzahl seiner Elemente, wird meist direkt oder indirekt als Abbruchbedingung verwendet, um das gesamte Array elementweise zu durchlaufen. Die Länge eines Arrays kann ganz einfach mit folgender `int` Variable abgefragt werden:

```
arrayName.length
```

Das heißt, an den Array-Namen wird die Endung "`.length`" angefügt. In diesem Fall ist `length` ausnahmsweise kein reserviertes Wort, trotz der farblichen Unterlegung.

Mit dieser Variablen lässt sich nun ein Unterprogramm für die Maximumsuche schreiben, das für beliebige int-Arrays mit Mindestlänge 1 funktioniert.

```
void setup() {
  int[] age = {18, 30, 24, 19, 18};
  println(arrMax(age));
}

int arrMax (int[] a) {
  int max = a[0];
  for (int i = 1; i < a.length; i++) {
    if (max < a[i]) {
      max = a[i];
    }
  }
```

(cc) BY

Programmieren mit Processing
Kapitel 8: Arrays

```
  return max;
}
```

Die integrierte Variable `length` wird bei der Instanziierung eines Arrays automatisch angelegt und beinhaltet die Arraygröße.
Im obigen Beispiel wird etwa das Array `age` mit 5 Elementen, d.h. der Länge 5 angelegt. Wird das Programm gestartet, speichert Processing im Hintergrund als Länge den Wert 5 ab und wir können diesen Wert über `age.length` abfragen.
Stoppen wir das Programm, verändern das Array zB auf `int[] age = new int[3]` und starten es erneut, dann speichert Processing wieder im Hintergrund die veränderte Länge in `length`, also den Wert 3 für das erzeugte Array.

(cc) BY

8.4 Typische Fehlermeldungen

Da in Zusammenhang mit der Verwendung von Arrays typische Fehlermeldungen auftreten können, wird im Folgenden als Unterstützung für die systematische Fehlersuche auf zwei solcher "klassischer" Processing Fehlermeldungen eingegangen.

ArrayIndexOutOfBoundsException

Der Fehler "ArrayIndexOutOfBoundsException" wird angezeigt, wenn versucht wird, auf einen Index zuzugreifen, der nicht existiert. Dieser Fehler tritt typischerweise im Zusammenhang mit Schleifen auf, wenn die Abbruchbedingung über den Arraybereich hinausgeht.

Beispiel: Hier sehen Sie noch einmal den Code für die Maximumsuche, leicht verändert.

```
void setup() {
  int[] age = {18, 30, 24, 19, 18};

  int maxAge = age[0];

  for (int i = 1; i <= age.length; i++) {
    if (maxAge < age[i]) {
      maxAge = age[i];
    }
  }

  println(maxAge);
}
```

Statt einem `<` (kleiner) wurde nun ein `<=` (kleiner gleich) verwendet. Die Konsequenz ist, dass der Index um eins zu weit gezählt wird. Der Wert von `i` erhält im letzten Schleifendurchlauf den Wert 5, da das Array `age` fünf Elemente hat. Im Schleifenrumpf wird daher versucht, auf `age[5]` zuzugreifen. Dieser Index existiert jedoch nicht, da das letzte Element des Arrays nur den Index 4 hat. Im unteren Bild wurde der Vorgang auch noch einmal grafisch dargestellt:

Führen Sie den oberen Code aus, wird Processing folgende Fehlermeldung anzeigen:

An diesem Fehler sehen Sie, dass ein ungültiger Indexzugriff stattgefunden hat und welcher Index ungültig war (hier: 5). Dieser Fehler wird allerdings erst zur Laufzeit angezeigt. Das bedeutet, das Programm muss zuerst gestartet und die entsprechende Zeile ausgeführt werden, damit dieser Fehler entdeckt werden kann.

NegativeArraySizeException

Der "NegativeArraySizeException" Fehler tritt dann auf, falls für die Array-Länge eine negative Zahl übergeben wird und versucht wird, auf irgendeine Weise mit dem Array zu arbeiten. Dieser Fehler tritt meist implizit auf, wenn etwa eine Arraygröße mit einer Variable angegeben wird, die durch fehlerhafte Berechnung einen negativen Wert annimmt.

Auch dieser Fehler taucht erst dann auf, wenn das Programm tatsächlich ausgeführt wird.

Beispiel für eine NegativeArraySizeException Ursache:

```
void setup() {
  int a = 5;
  int b = -1;

  int[] array = new int[a*b];

}
```

Beispiel:
Achtung, im folgenden Beispiel wirft Processing trotz des negativen Indexes keine NegativeArraySizeException, denn diese wird nur beim Erstellen eines Arrays mit negativem Index hervorgerufen. Stattdessen wird eine ArrayIndexOutOfBoundsException hervorgerufen. Bei letzterer liegt der Index außerhalb des gültigen Bereichs, was sowohl oberhalb als auch unterhalb sein kann.

```
void setup() {
  int[] ages = {18, 30, 24, 19, 18};
  int a = 3;
  int b = -1;

  int oneAge = ages[a*b];

}
```

8.5 Zweidimensionale Arrays

Bisher haben sie sogenannte 1-dimensionale Arrays kennen gelernt, die Sie sich wie Listen vorstellen können. 1-dimensionale Arrays enthalten nur eine bestimmte Art von Information, d.h. jeder Eintrag bezieht sich beispielsweise auf ein Alter, oder ein Gewicht usw.

Da als Typ der Elemente eines Arrays jeder Datentyp genommen werden kann, ist es auch möglich, als Datentyp der Elemente eines Arrays wieder ein Array zu nehmen. Dadurch kann in einem Element mehr als nur eine Information gespeichert werden. Sie können sich das wie eine Tabelle vorstellen: Beispielsweise haben Sie eine Tabelle von allen Teilnehmern eines Marathons. Jede Teilnehmerin bzw. jeder Teilnehmer hat eine eindeutige Nummer (den Index), das ist die erste Dimension. Von jedem Teilnehmer haben Sie aber noch weitere Informationen wie Alter, Größe, Gewicht, usw., die Sie speichern wollen - das wäre die zweite Dimension.

	0	1	2	3
0 (Alter)	30	53	24	37
1 (Größe in cm)	176	185	169	173
2 (Gewicht in kg)	70	78	58	68

Um das Alter von Teilnehmer/in mit der Nummer 2 herauszufinden, müssen Sie zuerst zur Spalte Nr. 2 gehen und dann in die Zeile 0 schauen. Hier können Sie ablesen, dass das Alter 24 Jahre ist.

Ein anderes Beispiel, das besonders wichtig für graphische Information ist, sind Koordinaten. Im Koordinatensystem haben Sie ebenfalls zwei Schritte auszuführen, um einen gewissen Punkt im Koordinatensystem zu finden. Sie müssen zuerst auf der x-Achse, die x-Koordinate finden und dann senkrecht von der x-Koordinate y-Schritte nach oben oder unten gehen. Erst dann haben Sie einen bestimmten Punkt im Koordinatensystem gefunden.

Deklarieren

Das Deklarieren von 2-dimensionalen Arrays ist den von 1-dimensionalen Arrays sehr ähnlich. Um festzulegen, dass es sich bei einem Array um ein 2-dimensionales Array handelt, verwenden Sie zwei eckige Klammernpaare:

```
int[][] arrayName;
```

(cc) BY TU Wien

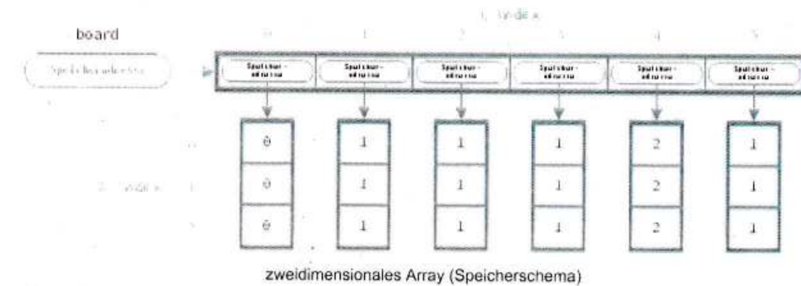
Diese Deklaration sagt aus, dass es sich bei `arrayName` um ein Array handelt, welches Arrays in seinen Elementen speichert. Die einzelnen Arrays wiederum speichern Werte vom Typ `int`.

Initialisierung und Zuweisung

Auch das Instanziieren ist dem vom 1-dimensionalen Array ähnlich. Will man beim Instanziieren vorerst nur die Größe des Arrays definieren, sieht das wie folgt aus:

```
int[][] board = new int[6][3];
```

In diesem Fall hat das Array sechs Arrays und die sechs Arrays können jeweils drei ganze Zahlen speichern.

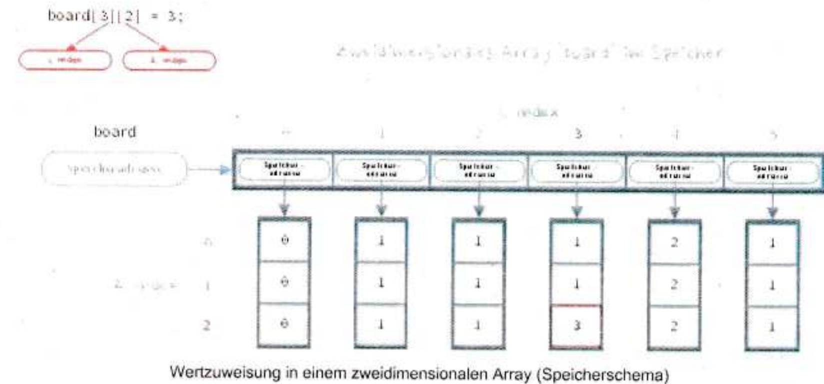


zweidimensionales Array (Speicherschema)

Mit

```
board[3][2] = 3;
```

kann man dem Array mit Index 3 seiner letzten Stelle (Index 2) den Wert 3 zuweisen:



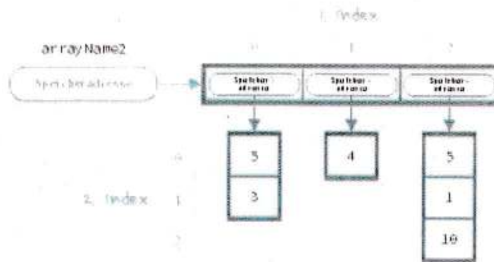
Wertzuweisung in einem zweidimensionalen Array (Speicherschema)

(cc) BY TU Wien

Werden Arrays auf diesem Weg initialisiert, dann haben alle "inneren" Arrays immer gleich viele Elemente, so wie eine einfache Tabelle in der Regel in jeder Zeile gleich viele Spaltenelemente beinhaltet. Das muss aber grundsätzlich nicht sein. Durch die Initialisierung mit direkter Wertübergabe ist es auch möglich, dass in einem Array unterschiedlich lange Arrays gespeichert werden können.

```
int[][] arrayName2 = new int[]{{5, 3}, {4}, {5, 1, 10}};
```

Das Array `arrayName2` speichert drei Arrays: Das erste innere Array besitzt zwei Elemente (`{5, 3}`), das zweite Array hat nur ein Element (`{4}`) und das dritte Array hat drei Elemente (`{5, 1, 10}`). Die geschwungenen Klammernpaare repräsentieren dabei jeweils ein Array.



Die Tabelle von Marathonläufern mit Alter, Größe und Gewicht wäre in einem Array beispielsweise so realisiert:

```
void setup() {
  int[][] marathon = {
    {30, 176, 70},
    {53, 185, 78},
    {24, 169, 69},
    {37, 173, 68}
  };

  println("Alter des dritten Teilnehmers: " + marathon[2][0]);

}
```

Dass für jedes innere Array eine separate Spalte verwendet wird, hat keinerlei Auswirkung auf den Inhalt des Arrays. Es dient lediglich der einfacheren Lesbarkeit und Vermeidung von Fehlern.

Betrachtet man die inneren Arrays als Elemente eines eindimensionalen Arrays, dann ist es verständlich, dass auch die inneren Arrays als Ganzes angesprochen und verändert werden können.


```
int[][] arrayName2 = new int[][]{{5, 3}, {4}, {5, 1, 10}};
arrayName2[0] = new int[]{8, 6, 10, 15};
```

Mit `arrayName2[0]` wird das erste Array angesprochen. Durch die Zuweisung (in der zweiten Zeile des Codes) wird das gesamte erste innere Array überschrieben. Es hat daher nicht mehr die Werte `{5, 3}` sondern die neuen Werte `{8, 6, 10, 15}`. Genauso ist es auch möglich, ein Element des zweidimensionalen Arrays, das ja ein eindimensionales Array ist, einer Array-Variablen zuzuweisen.

```
int[][] arrayName2 = new int[][]{{5, 3}, {4}, {5, 1, 10}};
int[] oneDimArray = arrayName[2];
```

In diesem Codeausschnitt wird der Variable `oneDimArray` das Array mit Index 2 aus `arrayName2` zugewiesen. Ein `println(oneDimArray[2])` würde hier beispielsweise die Zahl `10` auf der Konsole ausgeben.

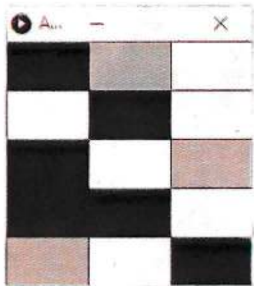
Beachten Sie, dass die Einteilung der Informationen in Zeilen und Spalten eigentlich nur eine Frage des verwendeten "Tabellen-"Denkmodells ist. Das bedeutet: Es kann prinzipiell genauso - etwa im Beispiel der Marathonläufer/innen - für jeden Läufer bzw. jede Läuferin eine Zeile verwendet werden und in Spalten werden dann die Angaben zu Alter, Größe und Gewicht gespeichert. Für das Array selbst ändert sich nichts, nur wie Sie es sich selbst vorstellen.

Zweidimensionale Arrays und Schleifen

Zweidimensionale Arrays sind besonders für die graphische Programmierung in 2D interessant. Denn Sie können sich das Zeichenfenster als ein zweidimensionales Array vorstellen, das x Pixel groß ist für die erste Dimension und y Pixel groß in der zweiten Dimension. Jedes dieser Pixel hat einen Grauwert, den man in einem 2D-Array abspeichern kann.

Um ein zweidimensionales Array mit Schleifen durchzugehen, benötigen Sie verschachtelte Schleifen. Die innere Schleife durchläuft die jeweils inneren Arrays, die äußere Schleife das äußere Array.

Beispiel: In einem zweidimensionalen Array sind Grauwerte gespeichert. Das Sketch-Fenster soll entsprechend der Größe des Arrays in Rechtecke unterteilt werden. Die Länge der ersten Dimension (d.h. die Anzahl der Arrays im "äußeren" Array) gibt an, wie viele Rechtecke es horizontal (x-Richtung) gibt. Die Länge der zweiten Dimension (Anzahl der Elemente in den inneren Arrays) gibt an, wie viele Rechtecke es vertikal (y-Richtung) gibt. Die Rechtecke sollen dann entsprechend der Graustufen im Array eingefärbt werden.



```
void setup() {
  size(180, 180);
  int[][] colors = {
    {0,   255, 0,   0,   150},
    {150, 0,   255, 0,   255},
    {255, 255, 150, 255, 0 }
  };
  int w = width / colors.length;
  int h = height / colors[0].length;

  for (int x = 0; x < colors.length; x++) {
    for (int y = 0; y < colors[x].length; y++) {
      fill(colors[x][y]);
      rect(x * w, y * h, w, h);
    }
  }
}
```

Das Lesen von Programmcode kann herausfordernd sein, wenn verschachtelte Schleifen kombiniert mit zweidimensionalen Arrays vorkommen. Oft hilft es dann - zusätzlich zur guten Dokumentation von Programmen - die Schleifen nicht "von oben nach unten" zu lesen sondern "von innen nach außen":

Betrachten wir dazu zuerst die innere Schleife und versuchen wir sie zu vereinfachen und zu erklären.

```
for (int y = 0; y < colors[x].length; y++) {
  fill(colors[x][y]);
  rect(x * w, y * h, w, h);
}
```

Diese Schleife verwendet die Variable `y` als Zählvariable und die Variable `x`, die sich in der Schleife nicht verändert. Sie können daher für die Variable `x` einen konstanten Wert einsetzen, z.B. 0.

```
for (int y = 0; y < colors[0].length; y++) {
  fill(colors[0][y]);
  rect(0 * w, y * h, w, h);
}
```

(cc) BY TU Wien

Diese Schleife ist jetzt leichter zu lesen. Sie wird so oft wiederholt, bis man am Ende des ersten inneren Arrays (`colors[0]`) angekommen ist. In diesem konkreten Beispiel ist die Länge vom ersten inneren Array gleich fünf. Danach wird die Farbe gesetzt. Diese wird aus dem ersten inneren Array an der Stelle `y` gelesen. Die Rechtecke werden anschließend von oben nach unten gezeichnet, da sich `y` erhöht. Zusammengefasst zeichnet also die innere Schleife eine Spalte an Rechtecken.

Das, was die innere Schleife macht, lässt sich auch in einer Funktion schreiben.

```
/**
 * Zeichnet von oben nach unten eine Spalte von Rechtecken
 *
 * @param colors  Array das Grauwerte der Rechtecke beinhaltet
 * @param xOffset Abstand vom linken Rand des Sketch-Fensters
 * @param w       Breite des Rechtecks
 * @param h       Höhe des Rechtecks
 */
void drawColumn(int[] colors, int xOffset, int w, int h){
    for (int y = 0; y < colors.length; y++) {
      fill(colors[y]);
      rect(xOffset, y * h, w, h);
    }
}
```

Die Funktion `drawColumn` nimmt als Parameter ein eindimensionales Array, eine x-Koordinate, und die Breite und Höhe des Kachels und zeichnet damit eine Spalte aus Rechtecken. Damit lässt sich die verschachtelte Schleife vereinfachen.

```
void setup() {
  size(180, 180);
  int[][] colors = {
    {0,   255, 0,   0,   150},
    {150, 0,   255, 0,   255},
    {255, 255, 150, 255, 0 }
  };
  int w = width / colors.length;
  int h = height / colors[0].length;

  for (int x = 0; x < colors.length; x++) {
    drawColumn(colors[x], x * w, w, h);
  }
}
```

Für jedes Array in `colors` wird eine Spalte gezeichnet, das um eine Rechtecksbreite verschoben ist.

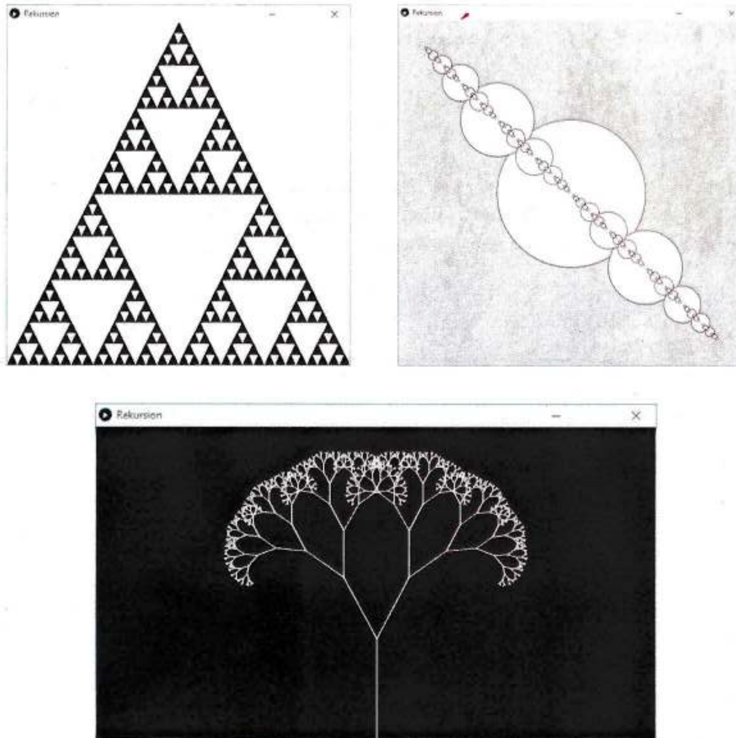
(cc) BY TU Wien

9. Rekursion

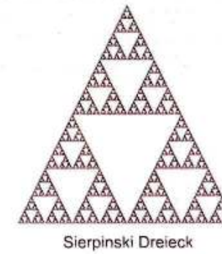
9.1 Motivation

Die bisherigen Kapitel decken ein solides Basiswissen und die Grundbausteine des Programmierens ab. Mit diesem Wissen können Sie bereits vielfältige Aufgaben und Problemstellungen mit dem Werkzeug der Programmierung meistern. Allerdings sind manche Probleme auf diese Weise noch umständlich zu lösen. Mit weiteren Konzepten, unterschiedlichen Herangehensweisen bzw. Denkweisen, können einige Aufgabenstellungen vereinfacht oder effizienter programmiert werden.

Ein solches, ganz bedeutendes Konzept ist die **Rekursion**. Sie kennen das Konzept vielleicht aus dem Grafischen bzw. Visuellen, wenn Sie die folgenden Abbildungen betrachten.



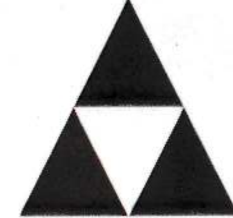
Ihnen allen ist gemeinsam, dass sie aus in sich selbst wiederholten Strukturen, Formen oder Mustern bestehen.
So besteht das Sierpinski Dreieck (links) etwa aus diesem einfachen Grundmuster (rechts):



Sierpinski Dreieck

Grundmuster

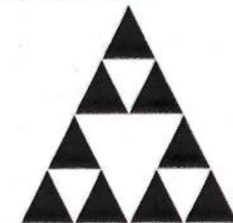
Visualisierung Grundmuster (halbtransparent) auf Sierpinski Dreieck



Resultat (Tiefe 1)

Das Grundmuster wird in jedes der schwarzen Teildreiecke gezeichnet:

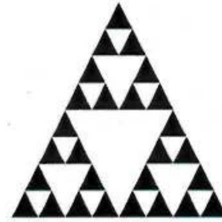
Visualisierung Grundmuster (halbtransparent) auf Teildreieck oben



Resultat (Tiefe 2)

Dies wird dann für jedes der neu entstehenden kleinen schwarzen Teildreiecke wiederholt.

Visualisierung Grundmuster (halbtransparent)
auf neu entstandenem Teildreieck oben



Resultat
(Tiefe 3)

Dies wird immer weiter, bis zu einem gewünschten Level bzw. theoretisch auch unendlich lange, wiederholt. Im Ausgangsbild des Beispiels wurde dies bis Level 7 durchgeführt.

Mittels Rekursion umgesetzt, können solche Grafiken in wenigen Zeilen programmiert werden.

(cc) BY TU Wien

9.2 Aufbau einer Rekursion

Dieser Aufbau aus "**in sich wiederholenden Mustern**" wird in der Programmierung umgesetzt durch eine Funktion, die sich selbst aufruft. Diese Funktion wird dementsprechend eine **rekursive Funktion** genannt. Grundidee der Rekursion ist es, ein Problem so in Teilprobleme zu teilen, dass die Teilprobleme wieder das gleiche Problem beschreiben. Die Teilprobleme sind dabei normalerweise etwas weniger komplex. Die Vorgehensweise der Rekursion könnte man dann auch wie folgt ausdrücken: "Löse die Teilprobleme mit derselben Methode wie das Gesamtproblem". Diese Denkweise ist anfangs gewöhnungsbedürftig und braucht Übung, kann aber in manchen Situationen sehr intuitiv sein.

Das folgende Beispiel soll illustrieren, dass Rekursionen natürlich nicht nur für grafische Visualisierungen eingesetzt werden, sondern auch in anderen Bereichen, wie etwa der Mathematik, Anwendung finden.

Beispiel: Es sollen die natürlichen Zahlen von 1 bis 5 aufaddiert werden. Für den Mathematiker ist die beste Lösung dazu die gaußsche Summenformel. Diese Formel führt am schnellsten zur Lösung. Wer diese aber nicht kennt, wird intuitiv das Problem vielleicht folgendermaßen lösen:

```
sum(5) = 5 + 4 + 3 + 2 + 1
```

`sum(5)` steht dabei für die Summe der natürlichen Zahlen von 1 bis 5. Allgemein würde dann mit `sum(n)` die Summe aller natürlichen Zahlen von 1 bis zur Zahl n gebildet werden.

Allerdings lässt sich die Lösung noch anders schreiben:

```
sum(5) = 5 + sum(4)
```

Das bedeutet, die Summe der natürlichen Zahlen von 1 bis 5 ist gleich 5 plus die Summe der natürlichen Zahlen von 1 bis 4. `sum(4)` steht dabei für die Summe der natürlichen Zahlen von 1 bis 4. Die Summe von 1 bis 4 kann erneut definiert werden als

```
sum(4) = 4 + sum(3)
```

Das kann wiederholt werden bis `sum(1)`, wo die Summe als "1" definiert ist.

```
sum(3) = 3 + sum(2)
sum(2) = 2 + sum(1)
sum(1) = 1
```

Diese Vorgehensweise ist rekursiv definiert. Denn das Problem wird so lange in immer kleinere gleiche Teilprobleme (hier, Summen von natürlichen Zahlen von 1 bis zu einer bestimmten Zahl) geteilt, bis es irgendwann (bei sum(1) = 1) lösbar ist.

(cc) BY TU Wien

Allgemein könnte man dies beschreiben mit:

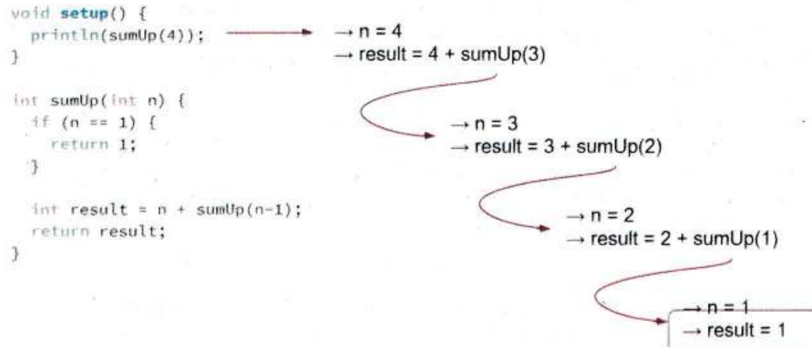
```
sum(n) = n + sum(n-1)        und        sum(1) = 1
```

Die Funktion `sumUp()` im folgenden Code-Beispiel ist eine rekursive Funktion, welche genau dies in Processing umsetzt und die Zahlen von 1 bis n aufsummiert.
Achtung: Für Zahlen kleiner als 1 wird diese Funktion einen Fehler hervorrufen.

```
int sumUp(int n) {
  if (n == 1) {
    return 1;
  }

  int result = n + sumUp(n-1); //Rekursiver Funktionsaufruf sumUp(n-1)
  return result;
}

void setup() {
  println(sumUp(5));
}
```

Man beachte dabei, dass in der Funktion `sumUp()` die Funktion `sumUp()` wieder aufgerufen wird. An dieser Stelle passiert der sogenannte "rekursive Funktionsaufruf". Die aufgerufene Funktion liefert entweder 1 zurück, wenn n bereits gleich 1 war, oder ruft sich selbst nochmals mit einem um 1 kleineren n auf. Die Anzahl dieser rekursiven Funktionsaufrufe, also wie oft die Funktion sich selbst aufruft bis endlich ein Wert (hier der Wert 1) zurückgegeben wird, nennt man auch **Rekursionstiefe**.

Die folgende Grafik visualisiert den Ablauf der Rekursion für `sumUp(4)` Schritt für Schritt im Detail. Analog funktioniert dies für alle anderen positiven ganzen Zahlen n in `sumUp(n)`.



Alternativ zu einer programmiertechnischen Erklärung, stellen Sie sich vor, es gibt vier Freunde, Alice, Bob, Carl und Dave.

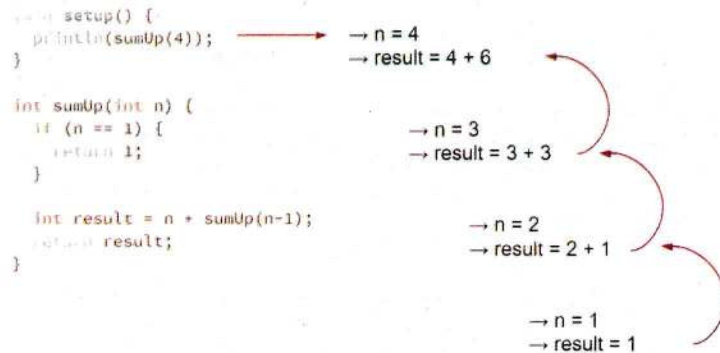
- Alice will die Summe von 1 bis 4 wissen. Sie weiß, dass die Summe `4 + sumUp(3)` ist, also 4 plus der Summe der natürlichen Zahlen von 1 bis 3. Allerdings weiß sie das Ergebnis von `sumUp(3)` nicht. Daher fragt sie Bob, ob er das für sie ausrechnen kann.
- Bob weiß, dass `sumUp(3)` gleich `3 + sumUp(2)` ist. Er weiß aber den Wert von `sumUp(2)` nicht. Daher fragt er Carl um Rat.
- Carl weiß, dass `sumUp(2)` gleich `2 + sumUp(1)` ist, aber was ist `sumUp(1)`? Carl fragt seinen Freund Dave.
- Dieser weiß, dass `sumUp(1)` gleich `1` ist.

Dies war der erste Teil des Ablaufs der Rekursion, die Zerlegung in Teilprobleme. Man kann sich das auch so vorstellen, dass man eben "in die Tiefe" arbeitet bis zum Erreichen der festgelegten Rekursionstiefe. Beim kleinsten Teilproblem (hier `sumUp(1) = 1`) angelangt, wird die Rekursion "zurückgerollt" bzw. muss man sich "zurück nach oben arbeiten", sodass schlussendlich das Ergebnis der Summe ausgegeben werden kann:

- Dave kann Carl sofort antworten und gibt ihm als Antwort `"1"`.
- Damit kann Carl nun das Ergebnis von `sumUp(2) = 2 + sumup(1)` berechnen, nämlich mit `sumUp(2) = 2 + 1`, also `3`. Dieses Ergebnis gibt er weiter an Bob.
- Jetzt weiß Bob, dass `sum(3)` gleich `3 + 3` ist, also `6`. Das erzählt er nun Alice.
- Alice kann sich endlich die Summe von 1 bis 4 ausrechnen, nämlich `4 + 6`. Damit erhält sie den Wert `10`.

An diesem Beispiel ist gut zu erkennen, dass das Problem nicht nur von einer Person gelöst wurde. Vielmehr konnten alle 4 Freunde eine Teillösung zum Problem beitragen und diese an jeweils einen weiteren Freund bzw. Freundin weiterreichen. Durch diese Vorgangsweise, das Problem immer weiter zu zerteilen und das verbleibende Teilproblem an den nächsten Freund bzw. an die nächste Freundin weiter zu reichen, haben sie gemeinsam schließlich das Problem als Ganzes lösen können.

Was passiert nun genau beim Programmablauf?



```
void setup() {
  println(sumUp(4));
}

int sumUp(int n) {
  if (n == 1) {
    return 1;
  }

  int result = n + sumUp(n-1);
  return result;
}
```


Wird die Funktion `sumUp()` für den Wert `4` aufgerufen, dann passiert in der Funktion Folgendes:

- `n` hat für den ersten Aufruf den Wert des übergebenen Parameters, nämlich `4`.
- Die if-Anweisung liefert `false`, da `4` nicht gleich `1` ist. Daher wird der `if`-Block übersprungen. Die Variable `result` erhält den Wert `4 + ` **`sumUp(3)`**.
- `sumUp(3)` ist jedoch ein Funktionsaufruf, das bedeutet, die Funktion `sumUp()` wird für den Wert `3` noch einmal ausgeführt. Währenddessen pausiert die Funktion für den Wert `4`. Sie "wartet" auf den Ergebniswert von `sumUp(3)` (um das Ergebnis berechnen und zurückgeben zu können).
- Im Aufruf von `sumUp(3)` hat `n` nun den Wert `3`. Da `3` nicht `1` ist, erhält die Variable `result` diesmal den Wert "`3 + sumUp(2)`".
- In `sumUp(2)` wird dann schließlich noch die Funktion `sumUp()` für den Parameterwert `1` aufgerufen.
- Für `sumUp(1)` gilt allerdings, dass `n == 1` ist. Daher wird in der if-Anweisung `1` zurückgegeben und dieser Funktionsaufruf beim ersten `return` beendet.
- Das wird nun mit `2` zusammen addiert, das ergibt `3`. Das Resultat wird wiederum an den Aufrufer zurückgegeben.
- Damit erhält auch der Aufruf zu `sumUp(3)` sein Ergebnis für `sumUp(2)` und kann sich für `result` den Wert `6` ausrechnen.
- Dieser wird weiter an den Aufrufer in `sumUp(4)` zurückgegeben. Als `result` wird `4 + 6`, also `10`, ausgerechnet und dieses Ergebnis wird dann zurückgeliefert an den Funktionsaufruf in setup.
- Damit erhält man für den Funktionsaufruf `sumUp(4)` den Wert `10`, welcher schließlich ausgegeben wird.

Auch der Computer arbeitet in dieser ähnlichen Form. Anstatt vieler Freunde hat er jedoch einen Speicher, den sogenannten **Stack**, wo er sich die Zwischenwerte abspeichert. Während die vier Freunde jeweils für sich ein paar Werte merken mussten, wie z.B. welche Summe sie ausrechnen müssen und wen sie um Hilfe gebeten haben, so speichert sich auch der Computer bei jedem Funktionsaufruf separat alle benötigten Variablen, sowie welche Funktionen von wo aus aufgerufen wurden, um später an dieser Stelle fortfahren zu können. Im Computer bedeutet das, dass beispielsweise die Variable `result` mehrmals existiert, aber ganz unabhängig voneinander, da sie bei jedem Funktionsaufruf neu angelegt wird (siehe auch Sichtbarkeit von Variablen bei Funktionen).

9.3 Abbruchbedingung - Endlose Rekursion

Wie mit Schleifen, können auch mit Rekursionen Wiederholungen ausgedrückt werden. Daher ist es wichtig, dass auch Rekursionen eine Abbruchbedingung haben, da sie sonst zu **Endlosrekursionen** ausarten können. Die Funktion `sumUp()` hat als Abbruchbedingung die Abfrage `n == 1` und stellt bei jedem rekursiven Funktionsaufruf sicher, dass `n` immer kleiner wird.

Die Abbruchbedingung einer Rekursion ist meistens jene, die das kleinste Teilproblem löst. Das Problem wird also so oft verkleinert, bis die Lösung einfach genug ist. In `sumUp(1)` wissen wir, dass für die Summe von 1 bis 1 nur 1 rauskommen kann. Hier muss keine weitere Rekursion mehr stattfinden. Für 0 oder negative Zahlen wird hier kein Ergebnis definiert. Der Parameter n wird also bei jedem rekursiven Aufruf um 1 kleiner, bis endlich n == 1 ist und das Ergebnis der Summe bis 1 gleich 1 ist.

Falls eine Abbruchbedingung falsch gewählt ist, dann äußert sie sich auf zwei mögliche Arten. Zum einen kann ein falsches Ergebnis berechnet werden oder die Rekursion endlos laufen, es tut sich also für eine sehr lange Zeit einfach nichts. Bei einer solchen Endlosrekursion kann es zu einem Programmabbruch mit einer sogenannten **StackOverflowException** kommen. Das bedeutet, dass der Stack-Speicher überfüllt ist und die Rekursion daher nicht weiter stattfinden kann. Hätte jedoch der Computer unendlich viel Speicher, dann würde die Rekursion unendlich lange ohne Fehlermeldung laufen.

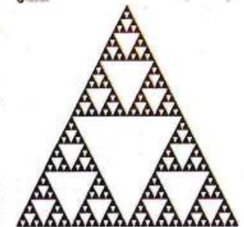
(cc) BY TU Wien

9.4 Unterschied Schleifen und Rekursion

Da sowohl Rekursionen als auch Schleifen Wiederholungen formulieren, stellt sich die Frage, was genau der Unterschied zwischen diesen zwei Konzepten ist bzw. welches Konzept wofür eingesetzt wird.
Grundsätzlich sind beide Varianten austauschbar. Das heißt: Für jede Art von Wiederholung lässt sich das Problem sowohl als Rekursion als auch mittels Schleifen ausdrücken. Auf die Frage, welches der beiden Konzepte für eine bestimmte Problemstellung zu bevorzugen ist, weil es sie etwa eleganter, einfacher oder effizienter löst, lautet die einfachste Antwort: Wann immer Sie merken, dass sich etwas (z.B. ein grafisches oder mathematisches Muster o.ä.) in sich selbst wiederholt, ist dies ein guter Indikator für eine Rekursion.

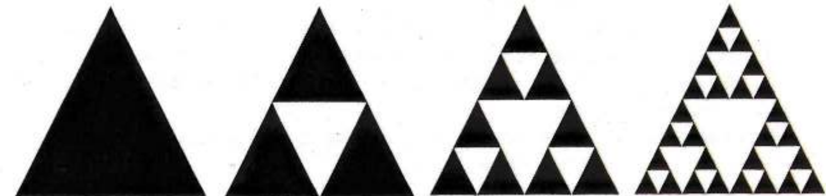
Die Herangehensweise von Schleifen und Rekursion unterscheidet sich grundlegend. Während beim rekursiven Ansatz die Problembeschreibung im Vordergrund steht, wird bei der iterativen (d.h. mit Schleifen) Variante versucht, das Problem Schritt für Schritt zu lösen.

Beispiel: Ein vereinfachter rekursiver Ansatz für das Sierpinski Dreieck, einem typischen grafischen Beispiel für Rekursionen, kann folgendermaßen zusammengefasst werden:



- Ein großes schwarzes Dreieck wird mit 3 Strecken in vier kleinere Dreiecke zerteilt.
- Das mittlere Dreieck wird "herausgeschnitten" (weiß).
- Für jedes der restlichen drei (schwarzen) Dreiecke wird dieser Vorgang wiederholt, bis der gewünschte Detailgrad erreicht ist.

Hier wird also dreimal der gleiche Vorgang wiederholt.



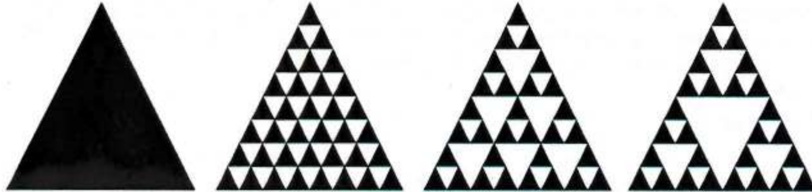
Ein iterativer Denkansatz mit Schleifen für das Sierpinski Dreieck könnte möglicherweise (vereinfacht) so aussehen:

- Beginnend bei der Spitze werden kleine weiße (gleichseitige, nach unten zeigende) Dreiecke derselben Größe in das schwarze Dreiecke gezeichnet. Bei jeder folgenden

(cc) BY TU Wien

Zeile wird ein weißes Dreieck mehr gezeichnet und die Dreiecke sind versetzt angeordnet.

- Dann wird das nächstgrößere Dreieck in gleicher Weise wiederholt über das vorhandene Bild gezeichnet.
- Das wird für alle Dreiecksgrößen wiederholt.



Vergleichen Sie die beiden Lösungsansätze und erklären Sie die Unterschiede noch einmal in eigenen Worten. Überlegen Sie sich auch für beide Varianten, welche Auswirkungen es auf den Code haben könnte, wenn Sie das Sierpinski Dreieck in weitere kleinere Dreiecke aufteilen wollen.

(cc) BY TU Wien

10. Ausblick

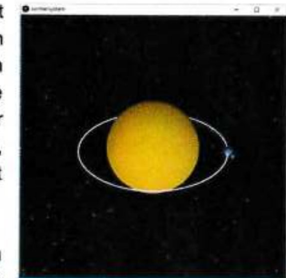
Wenn Sie den Kurs bisher absolviert haben und drangeblieben sind, haben Sie sich schon eine fundierte Basis im Bereich der Programmierung erarbeitet. Damit sind Sie sehr gut ausgerüstet, um verschiedenste Aufgabenstellungen mit dem Werkzeug der Programmierung eigenständig und kreativ zu lösen.

In den vorhergehenden Kapiteln haben Sie etwa dazu gelernt, wie Sie Variablen und verschiedene Datentypen sowie Arrays wirkungsvoll einsetzen. Sie haben mit den gängigsten Kontrollstrukturen, den Verzweigungen und Schleifen bereits Entscheidungen und Wiederholungen im Programmverhalten realisiert. Sie wissen, wie Programmcode leserlich gestaltet, mit Funktionen strukturiert, und dokumentiert werden kann. Außerdem haben Sie gelernt, dass sich manchmal ein Perspektivenwechsel lohnt, um bestimmte Problemstellungen effizienter lösen zu können.

Auch wenn Sie damit schon eine sehr gute Basis erlernt haben, ist das erst der Einstieg in die Welt der Programmierung. In diesem abschließenden Kapitel unseres Kurses wollen wir Ihnen daher Denkanstöße mit auf den Weg geben, wohin Ihre ganz persönliche Entdeckungsreise in der Welt der Programmierung als Nächstes gehen könnte.

10.1 Processing

Wer Interesse im Bereich Medienkunst, grafischer Programmierung, Design und Animation hat, kann in Processing selbst noch viel entdecken. Neben 2D Grafiken und 2D Animationen, wie wir sie im Kurs bereits programmiert haben, können mit Processing auch 3D Szenen, Animationen und Spiele programmiert werden.



Mit Processing können auch Bilder geladen und bearbeitet werden. Denken Sie zum Beispiel an die Bildfilter in Photoshop oder auf Instagram oder ähnlichen Apps. Auch wenn es viele Bildbearbeitungstools gibt, kann man mit Hilfe der Programmierung seine eigenen individuellen Bildfilter erstellen oder bekannte Anwendungen nachprogrammieren, um ihre Funktionsweise besser zu verstehen und effizient auf Bilder anzuwenden.

Eine Kollektion verschiedenster Processing-Projekte zu vielfältigen Themen findet sich auf der Processing Seite unter https://processing.org/exhibition/. Sie wird monatlich aktualisiert und erweitert.

Tutorials

- 3D mit Processing: https://processing.org/tutorials/p3d/
- Bilder laden: https://processing.org/tutorials/pixels/
- alle Processing-Tutorials: https://processing.org/tutorials/

(cc) BY TU Wien
Illustrationen (S. 3): Vaidotas Kinčius

10.2 Java

Wie Sie im ersten Kapitel gehört haben, gibt es neben Processing viele weitere Programmiersprachen. Deren Vielfalt erklärt sich darin, dass jede Programmiersprache andere Ziele verfolgt und daraus resultierend unterschiedliche Stärken und Schwächen besitzen. Deshalb ist für eine bestimmte Aufgabenstellung nicht jede Sprache gleich gut geeignet. Eine Stärke von Processing liegt im schnellen Programmieren graphischer Aufgabenstellungen. Kleine künstlerische Projekte können damit in kurzer Zeit programmiert werden, so wie das Kunstwerke-Beispiel, unser Pacman-Spiel oder andere Animationen. Daher eignet sich die Sprache auch besonders als Einstieg in die Programmierung. Aufgrund der leicht erzeugbaren Grafiken, Animationen und Benutzerinteraktionen macht es Spaß, dran zu bleiben und noch mehr zu entdecken. Allerdings ist Processing nicht so mächtig in punkto Ausführungsgeschwindigkeit oder Rechengenauigkeit. Besonders bei komplexen oder zeitkritischen Aufgaben, großen Programmen oder Applikationen, stößt man mit Processing an seine Grenzen.

Ähnlich wie man in der realen Welt mit mehr Menschen kommunizieren kann, je mehr Sprachen man spricht, ist es auch mit den Programmiersprachen: Wenn Sie mehr als nur eine Programmiersprache beherrschen, können Sie noch vielfältigere und größere Aufgabenstellungen flexibel und effizient lösen.

Ein Blick über den Horizont von Processing hinaus lohnt daher und kann etwa zu Java, einer sehr mächtigen und populären Programmiersprache, führen. Und der Vorteil ist - wer mit Processing in die Welt der Programmierung eingestiegen ist, für den ist es nur ein Katzensprung zur Welt der Java Programmierung. Denn Processing baut auf der Programmiersprache Java auf. Daher ist die Syntax der beiden Programmiersprachen sehr ähnlich, was das Erlernen von Java im Vergleich zu anderen Programmiersprachen wesentlich vereinfacht. Da Java auf einer tieferen Abstraktionsebene als Processing arbeitet, sind manche Aufgaben zwar detaillierter und daher aufwendiger zu lösen, aber dies bietet uns Programmiererinnen und Programmierern mehr Kontrolle und Flexibilität bei der Lösungsfindung. Das Endprodukt kann spezifischer und individueller angepasst werden.

In Java gibt es im Gegensatz zu Processing keine `setup()` Funktion und auch keine sich ständig wiederholende `draw()`Funktion. Dafür besteht ein Java Programm aus zumindest einer sogenannten Klasse (Schlüsselwort `class`). Jede Klasse besteht wiederum aus Variablen und Methoden (in Java werden die Funktionen generell Methoden genannt). Der Programmablauf beginnt dabei immer mit der `main` Methode. Hier ein Beispiel eines Java Programms:

```
public class MyFirstJavaProgram {
  public static void main(String[] args) {
    int sum = 0;
    for(int i = 1; i <= 10; i++)
      sum = sum + i;
    System.out.println("Summe der ersten 10 natürlichen Zahlen ist " + sum);
  }
}
```

(cc) BY TU Wien

Illustrationen (S. 3): Vaidotas Kinčius

Außerdem ist Java eine der am weitesten verbreiteten und am häufigsten eingesetzten und gesuchten Programmiersprachen (https://www.tiobe.com/tiobe-index/). Besonders häufig wird Java zur Entwicklung von Android Apps verwendet. Aber auch im Internet, auf Webseiten, die Berechnungen, Anmeldungen oder Spiele und dergleichen durchführen müssen, ist sie aufzufinden. Aufgrund ihrer Plattformunabhängigkeit ist der Einsatz von Java auch bei Desktop-Anwendungen keine Seltenheit. Beispielsweise wurde das berühmte Computerspiel Minecraft anfangs komplett in Java programmiert. Und nicht zuletzt setzen viele Universitäten und Fachhochschulen in MINT-Studiengängen, vor allem Informatik, Java als Programmiersprache ein - allen voran die TU Wien. ;-)

Java Tutorial: https://docs.oracle.com/javase/tutorial/

10.3 Objektorientierte Programmierung

Ein Perspektivenwechsel lohnt sich oft, effiziente und elegante Problemlösungen für spezifische Probleme zu entwickeln. Dies haben Sie bereits in Kapitel 9 mit der Rekursion (vs. Schleifen) kennengelernt.

Ein weiteres Konzept - eines der am weitesten verbreiteten und bedeutendsten Programmierkonzepte unserer Zeit - ist die **Objektorientierte Programmierung (OOP)**.

Die Perspektive ist eigentlich ganz einfach und aus dem Alltag bekannt: Dinge werden als Objekte aufgefasst, welche Eigenschaften und mögliche Verhaltensweisen haben.

Objekte können zum Beispiel ein Auto, ein Haus, ein Restaurant oder ein Buch sein, aber auch Personen oder Tiere, ein Pacman und ein Ghost können in der Programmierung als Objekte aufgefasst werden. Jedes solche Objekt hat bestimmte **Eigenschaften**, zum Beispiel im Fall von Pacman eine Größe, eine Bewegungsgeschwindigkeit oder eine Farbe. Bei einem Hund könnten es Name, Rasse, Geschlecht oder Fellfarbe sein. Außerdem hat ein Objekt ein **Verhalten**. Der Pacman kann sich z.B. in vier Richtungen bewegen und seinen Mund öffnen und schließen. Der Ghost hingegen hat das Verhalten, dass er dem Pacman nachjagt. Ein Hund kann wiederum unter anderem bellen, ein anderes Objekt fressen oder mit dem Schwanz wedeln.

Objekte können schließlich auch aus anderen Objekten zusammengesetzt werden.

Ein Auto besteht beispielsweise aus

- einer Karosserie (die wiederum bestimmte Eigenschaften, z.B. Farbe, Form, Material, etc. hat)
- ...weiteren Komponenten... und
- vier Rädern (auch diese haben wieder eigene Eigenschaften: Durchmesser, Aufhängung, etc.). Jedes der vier Räder wiederum besteht aus einer

(cc) BY TU Wien

Illustrationen (S. 3): Vaidotas Kinčius

- Felge (Eigenschaften wären hier z.B. das Design oder das Material, genauso wie Dimensionen)
- und dem Reifen (auch dieser hat wieder Eigenschaften, z.B. Material, Profil, Eignung, etc.).

In der objektorientierten Programmierung besteht ein Programm aus Objekten, die gewisse Eigenschaften und bestimmtes Verhalten haben und dabei mit anderen Objekten interagieren. Durch dieses Konzept der Objektorientierung ist es ein Leichtes, die Funktionalität von Programmen zu erweitern und sie für neue Anforderungen oder Aufgabenstellungen anzupassen.
Processing ist eine von vielen Programmiersprachen, die dieses mächtige Konzept der objektorientierten Programmierung unterstützt und Java ist schließlich eine objektorientierte Sprache, in der es nicht einmal möglich ist, ohne Objekte zu arbeiten, auch wenn es Programmieranfängern am Anfang nicht bewusst ist.

Die Objektorientierung können Sie sowohl in Processing als auch in Java und anderen Programmiersprachen verwenden, in Java sind jedoch die Möglichkeiten noch viel breiter gefächert.

Ein Tutorial zur Objektorientierte Programmierung mit Processing findet sich etwa auf der Processing-Seite unter: https://processing.org/tutorials/objects/

10.4 Abschluss

Auf welchen dieser interessanten Bereiche Sie sich auch als Nächstes konzentrieren -
Mit diesem Kurs haben Sie den ersten und damit den wichtigsten Schritt bereits gemacht: Sie sind in die Welt der Programmierung eingetaucht und haben sich eine solide Basis an Wissen, Kompetenzen und Erfahrung rund um die grundlegenden Programmierkonzepte erarbeitet.

Ganz egal für welches Themenfeld Sie sich entscheiden und wohin Ihr Weg Sie in der Welt der Programmierung führt - bleiben Sie dran, bleiben Sie stets neugierig und haben Sie stets Freude am Programmieren!

Das Programmieren mit Processing Team

(cc) BY TU Wien
Illustrationen (S. 3): Vaidotas Kinčius

Zucchini - Schnitte:
Zucchini/

1) Schnee schlagen:

3 Eier, schlagen
35 dag Zucker } dazu
1 Pkg Vanillezucker }
am Schluss 1/8 Öl langsam dazu mixen

2) In einer Schüssel
40 dag Zucchini klein schneiden, oder raspeln,

30 dag Mehl
10 dag Nüsse (Vollkorn / Hasel (Ferner sin cos))
1 TL Zimt
1 Pkg Backpulver

In Schnee ein mixen (Saft aus Zucchini gesaftet)

Rohr 180° = ~40'

Schokoglasur:
200 - 400 g Milch- Vollmilch
Ceres (Kokosfett [nicht viel])
in Wasserbad!

Michael Hahn
011810359