

# 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:

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

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.

## 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 - 7) 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

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:

$$\begin{array}{l} 735 / 10 = 73 + 5R \\ 73 / 10 = 7 + 3R \\ 7 / 10 = 0 + 7R \end{array}$$

Rest kippen




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.

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:

$$\begin{array}{rcl}
 12 / 2 & = & 6 + 0R \\
 6 / 2 & = & 3 + 0R \\
 3 / 2 & = & 1 + 1R \\
 1 / 2 & = & 0 + 1R
 \end{array}$$

Rest kippen



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):

$  \begin{array}{r}  1100 \\  100 \\  \hline  0  \end{array}  $	$  \begin{array}{r}  1100 \\  100 \\  \hline  00  \end{array}  $	$  \begin{array}{r}  1100 \\  100 \\  1 \\  \hline  000  \end{array}  $	$  \begin{array}{r}  1100 \\  100 \\  11 \\  \hline  0000  \end{array}  $	$  \begin{array}{r}  1100 \\  100 \\  11 \\  \hline  10000  \end{array}  $
---	--	---	---	--

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

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.

## 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

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

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

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  $\frac{1}{3}$ , die als Dezimalzahl 0.3 periodisch ist.

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

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

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.



## 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:

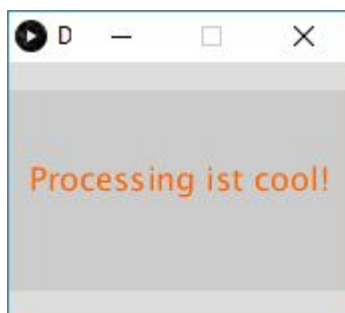
```
char m = 'M';
char questionMark = '?';
char four = '4';
char space = ' ';

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);
}
```

## 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.

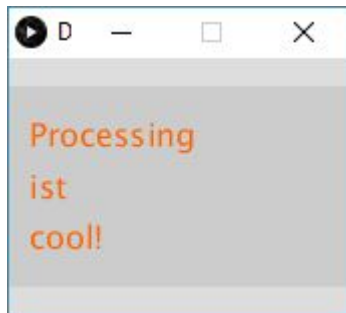
```
String backslash = "\\\";
```

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.

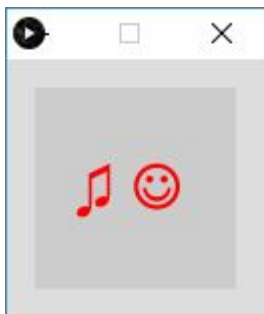
**Steuerzeichen** sind keine am Bildschirm dargestellten Zeichen wie Buchstaben, Zahlen oder Satzzeichen, sondern besitzen bestimmte Bedeutungen. Das bekannteste Beispiel ist der Zeilenumbruch. Zeilenumbrüche wurden bis jetzt durch den Befehl `println()` erreicht. Nach jedem `println()` Befehl wird ein 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](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 Verketteten 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

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));
```

## 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

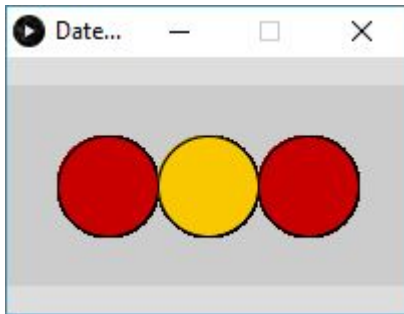
```
void setup() {  
  int alice = 15;  
  int bob = 17;  
  boolean answer = alice > bob;  
  println(answer);  
}
```

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.

## 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);
}
```

## 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:



- 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.

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

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 castet. 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](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.

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.

```
void setup(){
    int a = 20000000000; // 2 Milliarden
    int b = a;
    int c = a + b;

    println(c);
}
```

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:

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.