# P.0.0
# Processing—an overview

This book uses the programming language Processing as a way of introducing the concept of generative design; its basic features and functions will be presented on the following pages. This chapter lays no claim to completeness, which would be far beyond the realm of the book. A very good and detailed introduction to programming with Processing can be found in the book *Processing: A Programming Handbook for Visual Designers and Artists* by Casey Reas and Ben Fry.

→ W.201
Processing book

The Processing project was initiated in the spring of 2001 by Ben Fry and Casey Reas and has continually grown and been developed since then with a small group of collaborators. The main goal of Processing is to provide visually oriented people with simple access to programming.

Processing programs are called "sketches." Hence, Processing can be understood as an environment for the quick creation of digital sketches. The main folder where user-created programs are stored is called the "sketchbook" folder.

Processing is an open-source project; users can download it for free and use it for their own projects. Since Processing uses textual notation that consists of up to ninety-five percent Java syntax (with Java considered one of the main standards of the software industry), it is easy to transfer the code and examples to the majority of other textual development environments. Processing is cross-platform, which means the same source code can be used on all operating systems for which a Java platform exists (e.g., Mac OS, Windows, and Linux) and can also be integrated into websites.

→ W.202
Programming language Java

An ever-growing, vital, and supportive online Processing community exists that actively exchanges ideas on the Processing website. The website also contains online references of all language elements and an index of supplementary libraries.

→ W.203
Processing Website

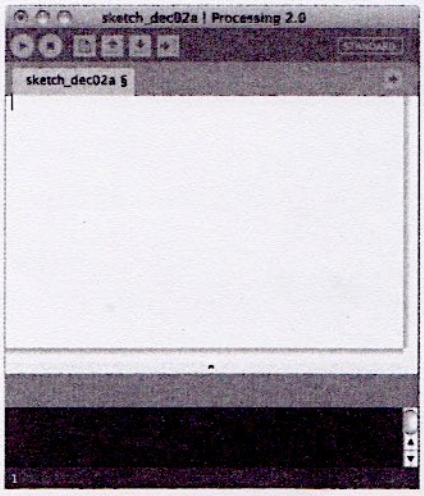
Several of these external libraries are used for the programs in this book. We also developed our own library for this book. The range of functions of the generative design library also includes the support of graphic tablets, the ability to export to Adobe's ASE color palette format, and the transformation of Cartesian coordinates into polar coordinates. The complementary libraries found in the "libraries" folder must be installed for all the programs in this book to work.

→ www.generative-gestaltung.de
→ Generative Design library

Palette exporting
→ Ch.P.1.2
Color palettes

All files required by a program, such as images, texts, or SVG (Scalable Vector Graphics) files, should be saved in a folder called "data" within the sketch folder. This ensures that Processing can find all necessary files, even if the sketch is executed in another operating system or in a web browser.

THE PROCESSING EDITOR As previously mentioned, Processing is based on the programming language Java. Processing is easier to learn than Java because the most important commands are simple, particularly those used for graphic output, and the beginner is not burdened with the confusing constructions Java requires to get a program running.



The Processing development environment.

Processing also provides a development environment that is reduced to the essentials. When Processing is opened, a window like the one shown here appears. The program code is entered in the middle area, the editor. The toolbar above this contains buttons used to start and stop the program or to load, save, or export programs. Information and error messages appear in the console, located at the bottom. Once the program code has been entered, the program starts; if there are no error messages, a display window appears in which the program runs.

The most important language elements and programming concepts are introduced on the following pages. You can immediately enter the presented commands in the editor, press start, and watch what the commands produce. Note, however, that not all printed code examples are functioning stand-alone programs.

# P.0.1
# Language elements

**HELLO, ELLIPSE** A first program. Simply type in the following line into the Processing editor and click start.

```
ellipse(50, 50, 60, 60);
```

A circle appears on the display.

Of course, it is possible to have more than one line of code. In this case, each line will be processed sequentially from top to bottom. Processing reads the following commands in this manner:

```
ellipse(50, 50, 60, 60);
strokeWeight(4);
fill(128);
rect(50, 50, 40, 30);
```

Draw an ellipse at the coordinates (50, 50) with a width and height of 60 pixels. Set the line weight to 4 pixels. Set the fill color to a medium gray. Draw a rectangle at the coordinates (50, 50) with a width of 40 and a height of 30.

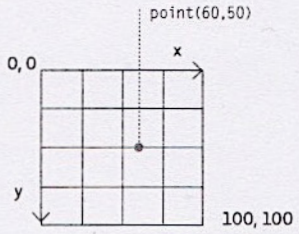
Processing commands are case sensitive. The command `strokeWeight()` would not be recognized when written as `strokeweight()` or `StrokeWeight()`.

There are commands that draw, such as `ellipse`, `rect`, `line`, etc., and commands that specify how the graphic that follows should be drawn, such as `stroke`, `strokeWeight`, `noStroke`, `fill`, `noFill`, etc. Once a drawing mode has been configured, it will apply for all additional drawing commands until the drawing mode is reconfigured.

Most drawing commands require one or more parameters that indicate where something should be drawn and at what size. The unit of measure for this is the pixel. The origin of the coordinate system is located in the upper left corner of the display window.

```
point(60, 50);
```

The pixel generated by the command `point(60,50)` is thus drawn sixty pixels from the left edge and fifty pixels from the upper edge.




Display window with a width and height of one hundred pixels.

**SETUP AND DRAW** The short programs just presented stop when the last line has been processed. However, a program usually continues to run until the user stops it. Only in this mode do animation and interaction become possible.

To achieve this, the `draw()` function has to be implemented. It is called in each drawing step, and all of its commands are processed each time.

```
void draw() { }
```

Although this `draw()` function contains no commands, it keeps the program running.

```
void draw() {
  println(frameCount);
}
```

A `draw()` function with one command. The command `println()` displays text in the console area of the Processing development environment. In this case, it displays the continually increasing number of the actual frame.

The `draw()` function is displayed with a preset frequency that specifies how many images are shown per second. The standard number is set at sixty images per second, but this can be reset using the command `frameRate()`.

```
frameRate(30);
```

This drawing speed is set at 30 images per second.

However, if the computational effort per frame becomes so high that Processing is unable to execute it within the preset time, the set frame rate decreases automatically.

Some actions should only be performed once when the program is started and not repeatedly in each frame. The `setup()` function is used for this.

```
void setup() {
  frameRate(30);
}
```

These commands, which should be executed just once when the program is started, appear in the `setup()` function.

**DISPLAY AND RENDERER** The display window is the stage for the program's visual output and can be set to any size.

```
size(640, 480);
```

The display is now 640 pixels wide and 480 pixels high.

In addition to the parameters' width and height, using the command `size()` makes it possible to enter which renderer should be used for displaying the image. The renderer is responsible for how the different graphic commands are actually translated into pixels. The following rendering options are available:

```
size(640, 480, JAVA2D);
```

The standard renderer: this is used if nothing else has been specified.

```
size(640, 480, P2D);
```

Processing 2D renderer: this is quicker but less accurate.

```
size(640, 480, P3D);
```

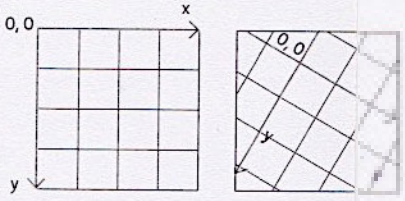
Processing 3D renderer: this is quick and reliable on the Web.

```
size(640, 480, OPENGL);
```

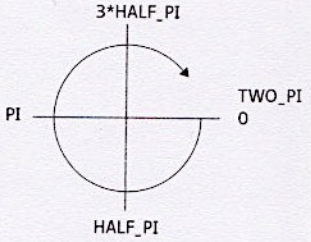
OpenGL renderer: this uses (and requires) OpenGL-compatible graphic hardware. Here the graphic card and CPU share the computational efforts, making it the fastest renderer.

**TRANSFORMATIONS** One of Processing's strengths is that it can move, rotate, and scale the coordinate system. All graphic commands thereafter refer to this altered coordinate system.

```
translate(40, 20);
rotate(0.5);
scale(1.5);
```




In this example, the coordinate system is moved 40 pixels to the right and 20 pixels downwards, then rotated 0.5 radians (about 30°) and finally scaled by a factor of 1.5.




In Processing, angles are generally shown in radians, in which 180° equals the number pi (≈ 3.14) and the rotation is in a clockwise direction.

### VARIABLES AND DATA TYPES

In a program, information is stored in variables so that it is available to other parts of the program. These variables can have any name other than the keywords for Processing functions. Keywords can be recognized by their special color in the editor.

```
int myVariable;
myVariable = 5;
```

The variable `myVariable` is created. The value `5` is then saved there.

When defining a variable, the user must specify its type, meaning what kind of information it will save. The keyword `int` in front of the variable's name informs Processing that this variable should hold integers. Processing provides several data types for various kinds of information. The most important are:

```
boolean myBoolean = true;
```

Logic values (Boolean values): `true` or `false`

```
int myInteger = 7;
```

Integers: e.g., 50, -532

```
float myFloat = -3.219;
```

Floating point value: e.g., 0.02, -73.1, 80.0

```
char myChar = 'A';
```

A single character: e.g., 'a', 'A', '9', '&'

```
String myString = "This is text.";
```

Character string/text: e.g., "Hello, world"

### ARRAYS

When working with a large number of values, it is inconvenient to have to create a variable for each value. An array allows a list of values to be managed.

```
String[] planets = {"Mercury", "Venus", "Earth", "Mars",
                    "Jupiter", "Saturn", "Uranus", "Neptune"};

println(planets[0]);
```

The square brackets after `String` indicate that the variable `planets` should be created or initialized as an array of strings. This array will immediately be filled with eight values (the planet names).

Values can be accessed by entering an index number in the square brackets following the variable name. The index 0 points to the first entry in the array.

If values are not immediately assigned to an array, the array can be created first and filled later:

```
int[] planetDiameter = new int[8];
planetDiameter[0] = 4879;
planetDiameter[1] = 12104;
planetDiameter[2] = 12756;
planetDiameter[3] = 6794;
...
```

The value 8 in the square brackets initialize an array for eight values.

Values are then assigned to the array. This could also happen later while the program running.

OPERATORS AND MATHEMATICS Naturally, calculations can also be performed in Processing. This is possible with simple numbers:

```
float a = (4 + 2.3) / 7;
```

After execution the value 0.9 is saved in a.

. . . with strings:

```
String s = "circumference of Jupiter: " + (142984*PI) + " km";
```

The variable s then contains the string "circumference of Jupiter: 449197.5 km".

. . . and with variables:

```
int i = myVariable * 50;
```

The value in myVariable is multiplied by 50 and the result saved in i.

The following computational operators are available: +, -, *, /, % and !.

A whole series of mathematical functions are also available. Here are a few:

```
float convertedValue = map(aValue, 10,20, 0,1);
```

The value in aValue is converted from a number between 10 and 20 to a number between 0 and 1.

```
int roundedValue = round(2.67);
```

Round a number. The roundedValue is then 3.

```
float randomValue = random(-5, 5);
```

Produce a random number between -5 and

```
float cosineValue = cos(angle);
```

Calculate the cosine of the specified angle.

**MOUSE AND KEYBOARD** There are several ways to access information using the mouse and keyboard as input devices. One option is to query the variables available in Processing.

```
void draw() {
  println("Mouse position: " + mouseX + ", " + mouseY);
  println("Is one of the mouse buttons pressed: " + mousePressed);
  println("Is a key pressed: " + keyPressed);
  println("Last key pressed: " + key);
}
```

The Processing variables `mouseX` and `mouseY` always contain the actual position of the mouse; `mousePressed` is true if one of the mouse buttons is pressed in that moment. For the keyboard, `keyPressed` indicates if a key is pressed; the key last pressed appears in `key`.

Another possibility is to implement one of the event handlers. An event handler is called when the corresponding event occurs—i.e., when a mouse button or key on the keyboard is pressed.

```
void mouseReleased() {
  println("The mouse button has been released.");
}
```

The function `mouseReleased()` is called when the left mouse button is released.

Additional event handlers include `mousePressed()`, `mouseMoved()`, `keyPressed()`, and `keyReleased()`.

**CONDITIONS** It is often necessary to execute lines of code. The `if` conditions are used to do this.

```
if (aNumber == 3) {
  fill(255, 0, 0);
  ellipse(50, 50, 80, 80);
}
```

The two lines of code inside the curly brackets are executed only when the condition is met—i.e., when the value of the variable `aNumber` is 3.

```
if (aNumber == 3) {
  fill(255, 0, 0);
}
else {
  fill(0, 255, 0);
}
```

With `else`, the `if` condition is expanded by one code snippet that is executed when the condition is not met.

```
if (aNumber == 3) fill(255, 0, 0);
else fill(0, 255, 0);
```

If a code snippet consists of only one line, as in the previous example, the curly brackets can be eliminated.

Differentiating between multiple values of a variable can usually be achieved by the `switch` command.

```
switch (aNumber) {
  case 1:
    rect(20, 20, 80, 80);
    break;
  case 2:
    ellipse(50, 50, 80, 80);
    break;
  default:
    line(20, 20, 80, 80);
}
```

The `switch` command tests if the value of the variable `aNumber` corresponds to one of the values listed in the `case` lines, jumps there if it does, and continues code execution until the following `break` statement.

If no corresponding value exists, program execution continues at `default`.

**LOOPS** Loops are used to execute a particular command several times within a program. Here are two versions:

The `for` loop is used to loop a code snippet for a specific number of repetitions.

```
for (int i = 0; i <= 5; i++) {
  line(0, 0, i*20, 100);
  line(100, 0, i*20, 100);
}
```

The two lines of code inside the curly brackets are executed exactly six times. First the variable `i` is set to the value 0, then increased by 1 (`i++`) after each cycle, as long as the value is 5 or less.

A `while` loop will continue as long as a certain condition is fulfilled.

```
float myValue = 0;
while (myValue < 100) {
  myValue = myValue + random(5);
  println("The value of the variable myValue is " + myValue);
}
```

This `while` loop will continue as long as the value of the variable `myValue` is less than 100. A random value between 0 and 5 is added in each new loop cycle.

**FUNCTIONS** There are often parts of a program that appear in a similar way in different places. In many such cases it is wise to encapsulate these parts in a function. For example:

```
void setup() {
  translate(40, 15);
  line(0, -10, 0, 10);
  line(-8, -5, 8, 5);
  line(-8, 5, 8, -5);
  translate(20, 50);
  line(0, -10, 0, 10);
  line(-8, -5, 8, 5);
  line(-8, 5, 8, -5);
}
```

The coordinate system is moved to another location, where a star is made of three lines. The coordinate system is then moved again, and a star is drawn using the same commands at the new location.

To change the star's appearance in this program, the drawing commands have to be changed in two places. It is therefore better to store the corresponding lines in a function.

```
void setup() {
  translate(40, 15);
  drawStar();
  translate(20, 50);
  drawStar();
}

void drawStar() {
  line(0, -10, 0, 10);
  line(-8, -5, 8, 5);
  line(-8, 5, 8, -5);
}
```

Here a function was defined that contains the drawing commands. This function can be called from anywhere in the program in order to execute the code it contains.

Values can be passed in functions, which can also return a value as a result.

```
void setup() {
  println("The factorial of 5 is 1*2*3*4*5 = " + factorial(5));
}

int factorial(int theValue) {
  int result = 1;
  for (int i = 1; i <= theValue; i++) {
    result = result * i;
  }
  return result;
}
```

A function `factorial()` is defined here, in which a value is passed, `theValue` of type `int`. When the function is called the parameter, 5 in this case, is passed in the variable `theValue`.

In the function, various calculations are executed with this value, and the result is returned using the command `return`. The type classifier in front of the function's name (here `int`) indicates the type of the return value. Here `void` indicates when no value will be returned, as in the previous example with the `drawStar()` function.

P.0.2

# Programming beautifully

In programming there are usually many ways to attain a desired result. The fact that a program works, however, is just one part of the solution, albeit a very important one; there are other aspects you should consider.

COMMENTS The more complex and tricky a program is, the harder it is for others—and for you, too, after a short time—to understand it. A program that is not accessible cannot be modified or updated. Comments in the program help to clarify the code.

```
// calculate distance between actual and previous mouse position
// which represents the speed of the mouse
float mouseSpeed = dist(mouseX, mouseY, pmouseX, pmouseY);
```

Text that follows two forward slashes is ignored by Processing and can be used for comments.

The Processing community is spread across the entire world. It is therefore advisable to write comments and variable names in English. This makes it easier for you and others to look for and find suggestions and answers to problems in Internet forums.

USEFUL NAMES AND CLEAR STRUCTURES In addition to the use of comments, sensibly selected variable and function names help users to maintain an overview and to understand what a specific program does.

```
float mixer(float apples, float oranges) {
  float juice = (apples + oranges) / 2;
  return juice;
}
```

In this example it is difficult to recognize just what the function calculates. The fact that the average of two numbers is calculated can only be guessed from the formula.

To this end, it is necessary to structure the program into smaller, logical sections with the help of functions and classes. This encapsulation of functionality also means that the program can be quickly modified and extended.

**PERFORMANCE** Even though computers are becoming faster and faster, it still takes time to execute a command. Even short periods of time add up noticeably when a command is executed often. Try not to burden the program with unnecessary work.

```
for (int i = 0; i < 10000000; i++) {
  float mouseSpeed = dist(mouseX, mouseY, pmouseX, pmouseY);
  doSomethingWithMouseSpeed(mouseSpeed);
}
```

Here `mouseSpeed` is calculated in all ten million loops even though the mouse position cannot be changed during this process.

```
float mouseSpeed = dist(mouseX, mouseY, pmouseX, pmouseY);
for (int i = 0; i < 10000000; i++) {
  doSomethingWithMouseSpeed(mouseSpeed);
}
```

Performance is therefore improved enormously when the mouse speed is calculated outside the loop.

With the programs in this book, we have tried to uphold the aforementioned principles. Sometimes, however, clarity and performance contradict one another, meaning that program code optimized for performance can be more difficult to understand. In such cases we have opted for clear structures.

**USING OUR PROGRAMS** In this chapter we have provided a short overview of the most essential building blocks of the Processing programming language. However, the most important part of programming is to connect these building blocks in such a way that the program does exactly what you want it to do. You will find it helpful to practice using the examples provided.

Learn from the programs in this book—create your own images with them, change the programs, and use them to create something new. We have released the programs under the Apache license. Therefore, you may use, modify, and distribute the programs freely in any context. Naturally, we would be very pleased if you mentioned us in your work.

→ W.204
Apache license