Aalto University
School of Arts, Design
and Architecture

Creative Coding with Processing
Department of Art and Media
2025/2026

Programming for Visual Artists

Welcome

Functions

– modularity

– parameters

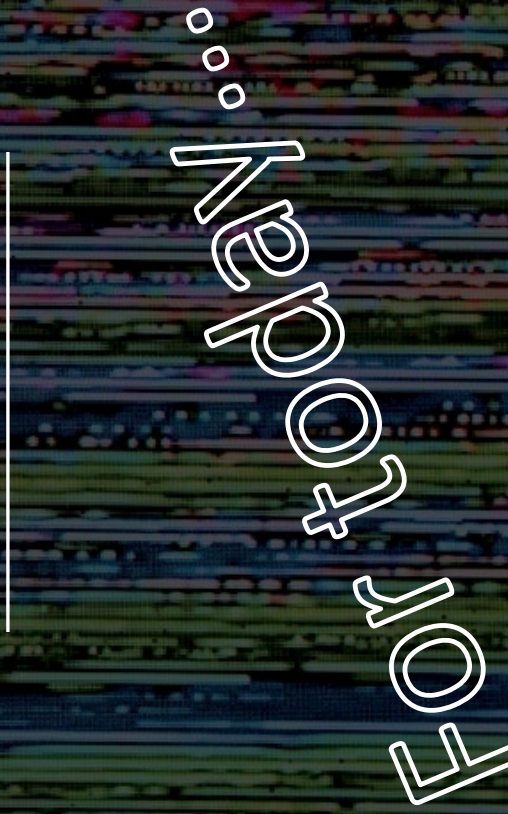
– return values

– recursion

BREAK

Coding tasks

Q&A



Useful Resource

Processing Reference
https://processing.org/reference

Use it to:

- check how a function works
- see examples
- find parameters and syntax

Programmers don’t memorize everything.
We constantly check documentation.

Recap

So far, we have learned:

- Drawing shapes with code
- Coordinates and the screen as a canvas
- Interaction with mouseX and mouseY
- Animation using variables
- Conditions (if / else)
- Loops and generative patterns

Functions

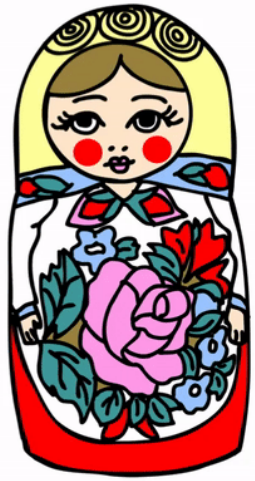
A **function** is a reusable block of code that performs a specific task.

Instead of writing the same code many times, we can **put it inside a function and call it when needed**.

Example idea:

drawCircle()

Whenever we call this function, It draws a circle.



Why use functions?

- Avoid repeating code
- Make programs easier to read
- Organize complex sketches
- Reuse code in different places

Functions - example

- drawCircle() is our **custom function**
- it receives **parameters** (x, y, r)
- it **draws a circle**

What if we call the function multiple times?

```
void setup() {
  size(400, 400);
  background(255);

  drawCircle(width/2, height/2, 50);
}

void drawCircle(float x, float y, float r) {
  ellipse(x, y, r*2, r*2);
}
```

```
void setup() {
  size(400,400);
  background(255);

  drawCircle(100,100,40);
  drawCircle(300,100,40);
  drawCircle(200,300,80);
}

void drawCircle(float x, float y, float r){
  ellipse(x,y,r*2,r*2);
}
```

Functions - modularity

- Modularity means **breaking a program into smaller pieces**.
- Each function performs **one specific task**.
- This makes programs:
  - easier to read
  - easier to modify
  - easier to reuse

```
void setup() {
  size(400, 400);
  background(255);

  drawCircle(100, 100, 50);
  drawSquare(200, 200, 80);
}

void drawCircle(float x, float y, float radius) {
  ellipse(x, y, radius * 2, radius * 2);
}

void drawSquare(float x, float y, float sideLength) {
  rect(x, y, sideLength, sideLength);
}
```

Functions – reusability of code

```
void setup() {
  size(400, 400);
  background(255);

  // Reusing the drawShape function to draw different shapes
  drawShape("circle", 100, 100, 50);
  drawShape("square", 250, 100, 50);
}

void draw() {
  // draw() remains empty since everything is drawn once in setup()
}

// A reusable function that draws either a circle or a square
void drawShape(String shape, float x, float y, float size) {

  if (shape.equals("circle")) {
    ellipse(x, y, size * 2, size * 2);

  } else if (shape.equals("square")) {
    rect(x - size, y - size, size * 2, size * 2);
  }
```

Functions allow us to **reuse code many times**.

Instead of rewriting the same instructions, we call the function again.

What if we change the function?

Now **every circle changes automatically**.

Functions – how to define?

```
void functionName(parameters) {
  // instructions
}
```

In Processing, a function is defined by specifying:

- a **return type** (for example void if the function does not return a value)
- a **function name**
- optional **parameters** inside parentheses
- a **code block** inside curly braces { }

```
void setup() {
  size(400,400);
  background(255);

  ellipse(200,200, randomSize(), randomSize());
}

float randomSize(){
  return random(50,150);
}
```

Example:

- displayMessage() is a **void function** that performs an action but does not return a value.
- add(int a, int b) takes two numbers as parameters and **returns their sum**.

The function setup() is called to run its instructions.
The function randomsize() returns a value that is stored in both size parameters of the ellipse.

Void or not to void

This function **draws a circle**, but it **does not return a value**.

```
void drawCircle(float x, float y, float r) {
  ellipse(x, y, r*2, r*2);
}
```

In Processing, a function can either:

- **perform an action**
- **calculate and return a value**

If a function **does not return anything**, its return type is: Void

Some functions calculate something and return a result.

setup() and draw() are defined as void because they **perform actions** but **do not return a value**.

This function **returns a number**.

```
float randomSize() {
  return random(20, 80);
}
```

Functions - parameters

```
void setup() {
  size(400, 400);
  background(255);

  // Calling the function with different parameters
  drawColoredCircle(100, 200, 40, color(255, 0, 0));
  drawColoredCircle(200, 200, 60, color(0, 255, 0));
  drawColoredCircle(300, 200, 30, color(0, 0, 255));
}

void draw() {
  // No animation needed
}

// Function with parameters: position, size, and color
void drawColoredCircle(float x, float y, float radius, color c) {
  fill(c);
  noStroke();
  ellipse(x, y, radius * 2, radius * 2);
}
```

Parameters allow a function to **receive values** when it is called.

They act as **inputs** that control how the function behaves.

Example:
drawColoredCircle(200, 200, 50, color(255,0,0));

The values passed to the function determine:
position (x, y)

size (radius)

colour (c)

parameters vs **arguments**

- Parameters → variables in the function definition
- Arguments → actual values when calling the function

parameters are the **inputs of the function**.

functions = reusable drawing tools

parameters = controls for the tool

Functions – return values

A function can **send a value back** to the part of the program that called it.

This returned value can then be **used elsewhere in the code**.

Example:
float r = randomRadius();

The function returns a number that becomes the **radius of the circle**.

float → the type of value returned
return → sends the value back

A function can either **do something** (void) or **give something back** (int, float, etc.).

```
void setup() {
  size(400, 400);
  background(255);

  float r = randomRadius();

  fill(100, 150, 255);
  ellipse(200, 200, r * 2, r * 2);
}

// Function that returns a value
float randomRadius() {
  return random(30, 100);
}
```

Functions - recursion

Recursion happens when a **function calls itself**.

Instead of repeating code with a loop, the function **solves the problem step by step**, each time working with a **smaller version of the same problem**.

Think of it like **nested Russian dolls**:
each doll contains a smaller one until you reach the smallest doll.

Base Case - when to stop

The function needs a condition that **stops it from calling itself forever**.

Recursive Call – calling the function again

If the base case is not reached, the function **calls itself with a smaller value**.

This gradually reduces the number until the **base case is reached**.

```
void setup() {
  printCountdown(5);
}

void printCountdown(int n) {

  if (n <= 0) {
    println("Done!");
  } else {
    println(n);
    printCountdown(n - 1);
  }

}
```

Loops vs Recursion

Both loops and recursion can solve **repetitive problems**.

Using a loop

- Loops repeat instructions **until a condition changes**.
- Example: counting down from 5.

Using recursion

- A function **calls itself** with a smaller problem until it reaches a **base case**.

Loops repeat instructions.
Recursion repeats function calls.

Both approaches can sometimes solve **the same problem**.

```
void setup() {
  int n = 5;
  while (n > 0) {
    println(n);
    n--;  // Decrement n
  }
  println("Done!");
}
```

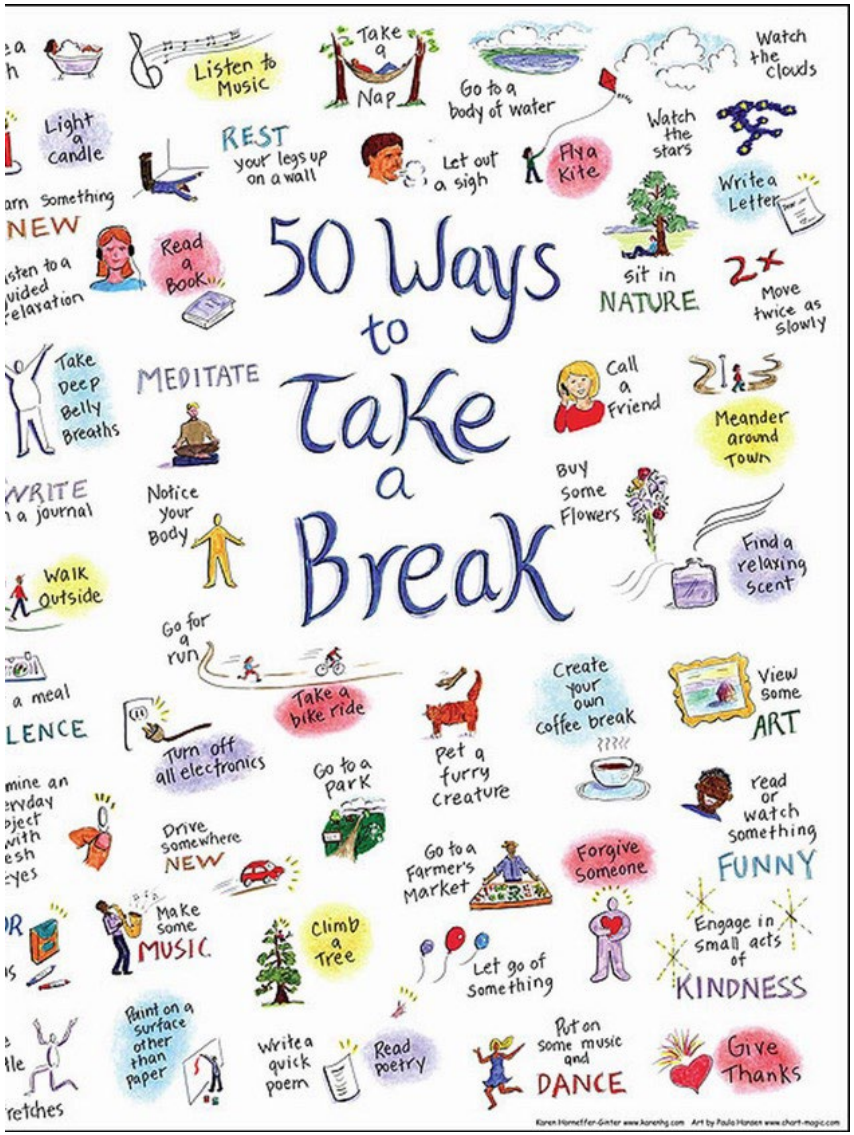
```
void setup() {
  printCountdown(5);
}

void printCountdown(int n) {
  if (n <= 0) {
    println("Done!");
  } else {
    println(n);
    printCountdown(n - 1);
  }
}
```

Break

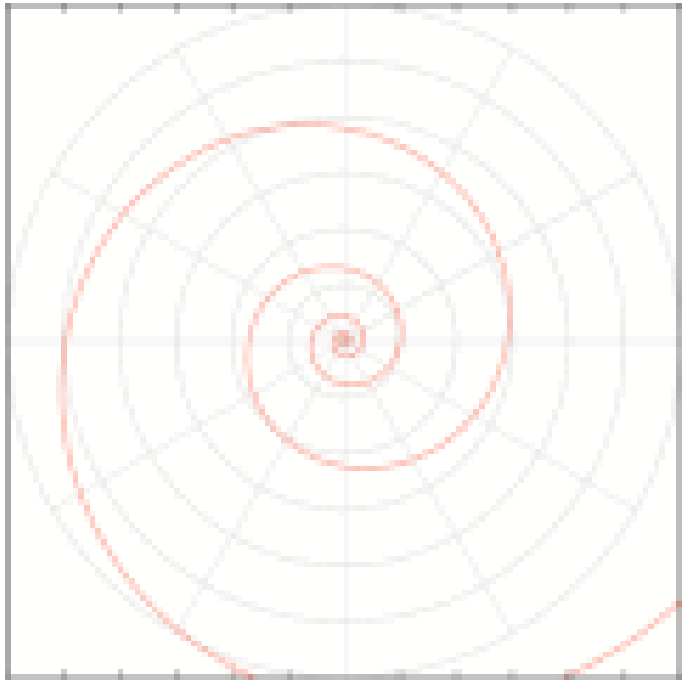
15 minutes

Stretch. Coffee. Reset.

Back at 10:45



Hands-On Exercise!



Your task
Create a **function-based visual system**.
Use **functions** to build reusable drawing components.
Use **parameters** to control variation.
Call your functions many times (loop or recursion).
Optional: animate your system over time.
Your goal is not to write more code.
Your goal is to design reusable tools.

Think about:
What does my function *represent*? (shape, element, rule)
What parameters control its behavior? (x, y, size, color, angle…)
What happens if I change **one** number?
How does repetition become a composition?

Break things. Change numbers. Explore.

Template:
Download from:
https://codeberg.org/ptiagomp/aalto-programming-visual-artists-25-26/src/branch/main/Session-05_09032026

Optional Challenge
If you finish early:
Make a **shape generator** (1 function, 3+ shape types)
Build a **spiral** using a function + loop
Create a **recursive** pattern (nested circles / branching tree)
Add interaction (mouse controls size / speed / color)
Add transparency + layering
Create a limited palette (3–5 colors)
Animate one parameter over time (frameCount / sin)

Discussion and Questions

What changed when we turned code into **functions**?
When did copying code become **reusing code**?
When did a drawing become a **tool**?

Today you didn't just write visuals.
You built **reusable components**.

Tomorrow

One element is interesting. Many elements create systems.
Tomorrow, we move from **single drawings** to **many-element behavior**.
We introduce **arrays** to control multiple elements at once.

Don't forget the assignments!