

Creative Coding

Learning to program by making things you can see

Rainer Stropek

2026-08-02

Table of contents

Preface	4
1 Introduction	5
1.1 Welcome	6
1.2 Your first program	9
1.3 From copying to creating	13
1.4 Type less, draw more	16
1.5 From cat to elephant: coding with AI	19
1.6 When the picture is given: Olympic rings	21
1.7 Arcs: drawing parts of a circle	24
1.8 Fixing code you didn't write	27
1.9 Practice projects: flowers and a soccer field	30
1.10 Reading the documentation: a command of your own	33
2 Variables	36
2.1 Variables: values with a name	37
2.2 Your first formula: snapping to a grid	41
2.3 The responsive robot	46
2.4 Mixing colors with numbers	51
2.5 The color worm: a variable of your own	55
2.6 Build your own color picker	60
2.7 Strings that grow: the click protocol	64
2.8 Practice projects: circle tool and rect tool	67
2.9 Back to the reference: a keyboard program of your own	71
3 Conditions	76
3.1 The if statement: programs that decide	77
3.2 Color quadrants: a decision inside a decision	82
3.3 Rolling dice: random numbers and exact comparisons	85
3.4 The bouncing ball: your first animation	90
3.5 Binary and hexadecimal conversions	95
3.6 Number systems in code	104
3.7 Rock, paper, scissors: your first game	111
3.8 The switch statement: many cases, one question	118
4 Loops	122
4.1 The while loop: teaching the computer to repeat	123
4.2 Rays: loops meet the mouse	129
4.3 Pixel X and zebra stripes: the loop variable works harder	133
4.4 Mesh: a loop inside a loop	138
4.5 Chessboard: rows, columns, and one clever loop	142
4.6 The for loop: four steps on one line	146
4.7 Moving the origin: translate, push, and pop	150
4.8 Patterns: repeat, move, repeat	154
4.9 Curves from straight lines	158
5 Arrays	166
5.1 Strings have parts: Word Swirrel	167
5.2 Letters criss-cross: rotate	172
5.3 Your first array	175
5.4 Scale: bigger and smaller	178
5.5 Parsing: numbers hidden in a string	183

5.6	Parallel arrays: the starting eleven	188
5.7	Arrays that grow and shrink	192
5.8	Silos: the array runs the show	197
5.9	Value or reference: what an assignment copies	202
5.10	Reading MDN: a string method of your own	206
6	Functions	209
6.1	Your own commands: functions	210
6.2	Refactoring: from dice to domino	216
6.3	Switch again: a domino of symbols	221
6.4	Decomposition: four functions, one chart	226
6.5	Functions that answer: return values	230
6.6	Callbacks: a function you hand over	234
6.7	A game from functions: Bubble Buster	238
6.8	Many files, one program	243
6.9	Finish the game: logic functions	248
6.10	Images and orthographic worlds	252
6.11	Trains from strings: parsing functions	257
6.12	The earned shortcut: split	262
6.13	Design your own functions: Crossword	267
6.14	Drag and drop: Target Game	272
6.15	Compound types: values that belong together	277
6.16	Practice: Ratio Visualizer	282
6.17	Training: functions and arrays	285
6.18	Test your knowledge: Poker Cards	288
7	Visual Studio Code	291
7.1	A new workshop: Visual Studio Code, Node.js, and npm	292
7.2	Getting the starter code	295
7.3	Your first program, again	299
7.4	Pride in your craft	303
7.5	One file, many pictures: the slot machine	306
7.6	Where you are now	313

Preface

Welcome to **Creative Coding**, a course that teaches you to program a computer by making things you can *see*. Instead of only printing text on a black screen, you'll draw shapes, build small pictures, and grow them into interactive artworks. You don't need any programming experience to start: the course begins at zero and builds up step by step.

Part 1

Introduction

1.1 Welcome

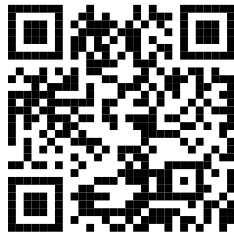
1.1.1 AI tutor

You're not alone with this book. Every chapter of this part links to the same AI tutor, a chat that knows what the course has taught you so far. Ask it whenever an explanation doesn't click, and it helps you with the commands you already have instead of jumping ahead.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/9fxc2eu84z>

1.1.2 Welcome to Creative Coding

Welcome to this course! Over the next months you'll learn how to program a computer, and you'll do it in a creative way. Instead of only printing text on a black screen, you'll draw shapes, build animations, and create small interactive artworks and games. That's why the course is called *Creative Coding*: you learn programming by making things you can see and change, then show to your friends.

You don't need any programming experience to start. The course begins at zero and builds up step by step. If you can use a computer and you're curious, you have everything you need.

1.1.3 The tools you'll use

In this course you'll work with two tools: a programming language called **TypeScript** and a graphics library called **p5.js**. Together they let you write real, professional-style code and see the results on screen right away.

1.1.3.1 TypeScript, the language

TypeScript is one of the most popular programming languages in the world. Many well-known websites and apps are built with it, and professional developers use it every day. So you're not learning a toy language made only for classrooms: the skills you pick up here are the same skills real software teams use.

At the same time, TypeScript is friendly to beginners. It watches your code while you type and warns you about many mistakes before you even run the program. Think of it as a language with a built-in spell checker for code.

1.1.3.2 p5.js, the graphics library

p5.js is a **library** for creating interactive graphics and animations. A library is a collection of ready-made code that you can use in your own programs. Instead of writing hundreds of lines yourself just to draw a circle, you call one short command from the library and the circle appears. Libraries let you build on the work of other programmers, which is how almost all software is made.

p5.js was created for students and artists. It's designed so that your very first program can be just a few lines that draw something on screen. From there you can build up: shapes become patterns, patterns become animations, and animations become games and interactive art.

💡 Simple start, no limits

With p5.js you can draw your first circle in minutes, but people also use it for serious generative art and data visualizations. You won't outgrow it during this course.

1.1.4 Why learn to code when AI writes code?

Maybe you're thinking: "AI tools can write code now. Why should I learn it myself?" That's a fair question, and it deserves an honest answer. Here are four reasons.

1.1.4.1 Programming is more than typing code

When you program, the typing is the smallest part. The real work is thinking: you take a big, fuzzy problem, break it into small clear steps, and put those steps in a logical order. You also make creative decisions, because there is almost never just one way to solve a problem. Breaking problems down, thinking logically, and finding creative solutions help you far beyond programming: in school, in any job, and whenever you face something complicated.

1.1.4.2 AI is like a genie in a lamp

You may know the old story of the genie in the lamp: a magical being appears and grants your wishes. But there's a catch. The genie gives you exactly what you *say*, not what you *mean*. Wish carelessly and you get something you didn't want at all.

AI works the same way. It will happily produce code for you, but the result is only as good as your wish. To get useful code, you need to know what is possible, what to ask for, and how to judge what you get back. Learning to program by hand teaches you exactly that. You learn what a computer can do, how software is built, and what a good solution looks like. In short: you learn what to wish for.

1.1.4.3 Someone has to fix the last few percent

AI makes mistakes. It writes code that looks correct but sometimes isn't, and it can get stuck on the tricky final details of a program. Professionals are the people who spot those mistakes and fix the last few percent that the AI can't get right. You can only do that if you understand the code yourself. The better you understand programming, the more powerful AI becomes as your assistant instead of your replacement.

1.1.4.4 Programming is fun

The last reason is the simplest: programming is fun. There's a special feeling when you imagine something, write the code for it, and watch the computer bring it to life exactly the way you pictured it (or in a surprising way that's even better). In this course that feeling comes quickly, because your programs draw, move, and react from the very first lessons.

1.1.5 Your coding assistant comes later

Two AIs belong to this course, and you meet them at different times. The first one is the **AI tutor** at the top of every chapter. It explains things and helps you read an error message, but it never writes your program for you. That's the AI you use from today on.

The second one is a **coding assistant**. It writes real TypeScript with p5.js together with you, and it waits until the last part of the book, where you move from the web playground to Visual Studio Code (Section 7.3.8). It waits because a coding assistant only helps someone who can read what it writes and spot the line that's wrong. By the last part you can, because every line you typed yourself on the way there is a line you know how to check.

1.1.5.1 Why not just any AI from the internet

Between now and then you'll be tempted to ask a general AI on the internet for code, so you should know what you'd get. That AI knows nothing about your course. It writes code the way it would for a professional, full of things nobody has shown you yet. Code like that might run, but you can't explain it, and you can't repair it when it breaks. You're the one who hands the program in, so you're the one who has to understand every line of it.

The coding assistant of this course is different, because the coding rules of this book are built into it. It writes the kind of code you can read and explain in class, and it stays with the commands you've already learned instead of jumping ahead.

1.1.6 Check your understanding

Before you move on to the next chapter, take the short quiz below. You answer five questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Welcome

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/1xsv90lvoo>

1.2 Your first program

By the end of this chapter you'll have written your very first real program and watched a small arrangement of shapes appear on the canvas. You'll get there by typing the code exactly as it's given, one character at a time, and seeing for yourself how a program comes to life. It's a small result, but it's entirely yours. This is the moment programming stops being an idea and starts being something you do.

1.2.1 AI tutor

Your first program has to be typed exactly right, so small mistakes are part of the deal. When the playground shows an error message you can't make sense of, the AI tutor of this part helps you read it.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/9fxc2eu84z>

1.2.2 Where you'll write code

Professional developers write code in powerful programs like Visual Studio Code. Tools like that are great, but before you can even start, you have to install them, add extensions, download dependencies, and configure many settings. That's a lot of setup before the first line of code.

In this course you'll skip all of that. You'll write your code in a simple **web playground** that runs in your browser. There's nothing to install: you open a link, and you're ready to program. The playground guides you through structured exercises. For each exercise you see a description of your task, an editor where you type your code, and a preview area where your program runs and shows its result.

1.2.3 Every keystroke matters

Here's the most important lesson of your first exercise, and maybe of the whole course: **computers take you literally**. A computer doesn't guess what you meant. It does exactly what your code says, character by character.

For a computer, these two lines are completely different:

```
circle(200, 260, 360);  
Circle(200, 260, 360);
```

The first one draws a circle. The second one is an error, because the command is called `circle` with a lowercase `c`, and `Circle` with an uppercase `C` is a name the computer doesn't know. One single letter decides whether your program works.

The same is true for every parenthesis `(`, every comma `,`, every semicolon `;`, and especially every curly brace `{` and `}`. Curly braces always come in pairs: every opening `{` needs a closing `}`. Forget one, and the computer gets confused about where your code begins and ends.

This exactness can feel strict at first. But it's also good news: there is no hidden magic. If your code is exact, it works, every time, for everyone.

❗ Semicolons are mandatory

TypeScript itself doesn't force you to end every statement with a semicolon: code without them still runs. In this course, semicolons are mandatory anyway: **every statement ends with a `;`**.

There's a good reason for this rule. In later courses you'll meet programming languages like C, Java, and C#, and they all require the semicolon after every statement. If your fingers learn the habit now, those languages will feel familiar from day one.

1.2.4 A program is a sequence of statements

A program is built from **statements**. A statement is one instruction to the computer, like “create a drawing canvas” or “draw a line”. In your code, you write one statement per line, and each statement ends with a semicolon ;.

The computer runs your statements **in order, from top to bottom**, like steps in a recipe. The order matters. For example, a statement like `background("pink")` paints the whole canvas pink. If you draw a circle first and paint the background afterward, the pink paint covers your circle, and you'll see an empty pink canvas.

! The canvas: your sheet of paper

In p5.js, you draw on a **canvas**. Think of it as a sheet of paper lying on your desk: the statement `createCanvas(400, 500)` puts a sheet on the screen that is 400 pixels wide and 500 pixels tall, and every drawing command after it draws on that sheet.

To tell a command *where* to draw, the canvas uses a coordinate system. You know coordinate systems from math class, but the canvas has one important difference: the origin (0, 0) is in the **top left corner** of the canvas.

- The **x axis** works as you expect: higher x values go to the **right**.
- The **y axis** is flipped: higher y values go **down**, not up.

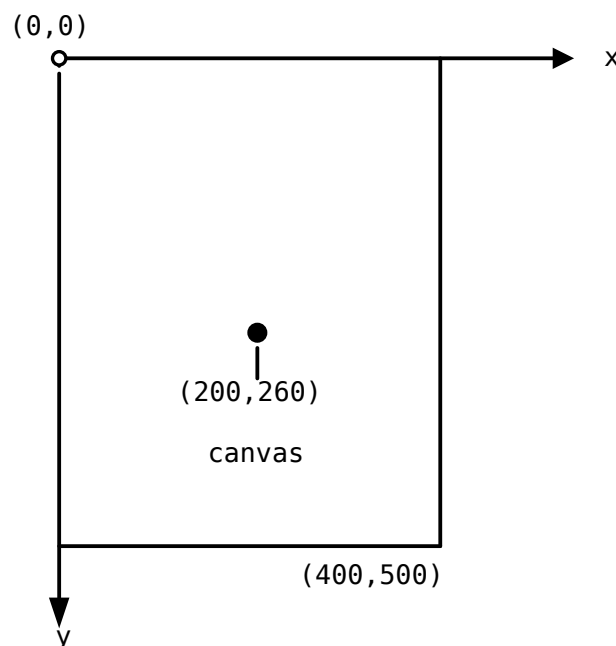


Figure 1.2.1: The canvas coordinate system: the origin sits in the top left corner, x grows to the right, and y grows down.

So the point (200, 260) lies 200 pixels to the right of the left edge and 260 pixels down from the top edge. The flipped y axis surprises almost every beginner, so keep it in mind when a shape doesn't appear where you expected.

In your first exercise, the statements are **drawing commands** from the p5.js library. Each command has a name and, in parentheses, a list of **parameters**: values that tell the command what exactly to do. For example:

```
circle(200, 260, 360);
```

This statement draws a circle. The three parameters are the position of the circle's center (200 across, 260 down) and its diameter (360). Change the numbers, and the circle moves or grows. You'll explore what each parameter means while you play with the code.

1.2.5 Your statements live in the setup function

Your statements don't stand alone in the file. They sit inside a **function**, a named block of statements that runs as a unit. This one is named **setup**, and the computer runs it once when your program starts. Your first exercise has this shape, with a few more statements in the middle:

```
function setup() {  
  createCanvas(400, 500);  
  background("pink");  
  circle(200, 260, 360);  
}
```

The line `function setup() {` opens the block, the closing `}` at the very bottom ends it, and everything between them is your program. The statements inside are shifted to the right so you can see at a glance what belongs to **setup**.

For now, one rule is enough. Every statement you write in this part goes inside **setup**, between the curly braces. Later in the course you'll write functions with other names, and functions that run at other moments.

1.2.6 Red squiggly lines show you your mistakes

While you type, the editor checks your code. When something is wrong, a mistyped name, a missing parenthesis, a lost curly brace, it underlines the problem with a **red squiggly line**, just like a spell checker underlines a typo in a text document.

When you see a red squiggle:

1. Don't panic. Squiggles are completely normal. Every programmer, including professionals, sees them all day long.
2. Move your mouse over the underlined code. A small message appears that describes the problem.
3. Compare your code with the example, character by character. Fix the difference, and the squiggle disappears.

Making mistakes is not failing, it's how programming works. You type, the editor points at a problem, you fix it, and you move on. You'll get faster at this with every exercise.

1.2.7 Format your code properly

Code isn't only for computers, it's also for people: for you next week, for your classmates, and for your teacher. That's why programmers format their code carefully:

- **Indentation:** statements inside curly braces are shifted to the right by a fixed number of spaces. That makes it easy to see what belongs together.
- **One statement per line:** don't squeeze several statements into one line.
- **Consistent spacing:** for example, one space after each comma.

In the exercise, copy the formatting exactly as you see it in the example, including the spaces at the beginning of the lines. Well-formatted code is much easier to read, and mistakes like a missing curly brace are easier to find.

1.2.8 Your first exercise: Shapes

Now you're ready. Open the **Shapes** exercise in the web playground. Your task there has two parts:

1. **Type the code** from the example, exactly as shown. Every character counts: uppercase and lowercase letters, parentheses, commas, semicolons, curly braces, and spaces. Take your time. Typing code is a skill of its own, and speed comes later. Lines that start with `//` are comments, notes for human readers that the computer ignores. Type them in too, as typing practice. A typo in a comment is harmless.
2. **Play with the code.** When your program runs, change the parameters of the drawing commands and watch what happens. Move the circle, stretch the rectangle, try other colors. If you hover your mouse over a command name, a small tooltip shows you its documentation.

If your canvas stays empty or a red squiggle won't go away, compare your code with the example line by line. The difference is always there, even when it's just one character. Finding it is part of the exercise.

💡 Exercise: Shapes

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

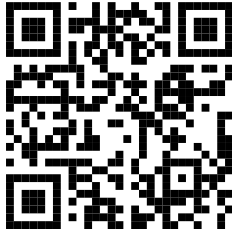
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0010-Basics/shapes.yaml>

1.2.9 Check your understanding

When you've finished the exercise, take the short quiz below. You answer eight questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Your first program

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/emu8e2ik6w>

1.3 From copying to creating

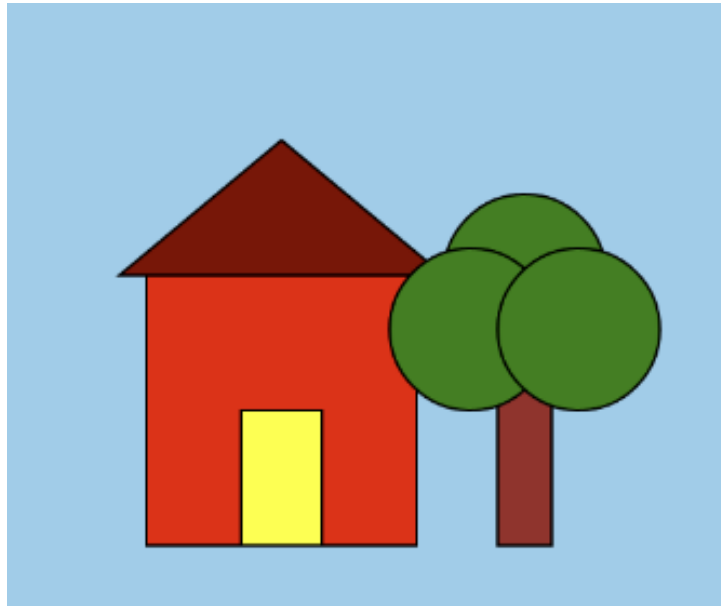


Figure 1.3.1: A house with a tree beside it, drawn entirely from simple shapes.

This is the first time you invent the picture instead of copying it. You'll design your own house on paper and then bring it to life on the canvas, with a tree standing beside it, built entirely from the shapes you already know. It'll be your house and nobody else's.

In the Shapes exercise you typed code that was given to you. Now comes the step that makes programming exciting: you write your own program. Your task will be to draw a house with a tree beside it, using the drawing commands you already know.

There is no single correct solution anymore. Your house doesn't have to look like anyone else's, and that's the point. The rules of exactness still apply, every letter, parenthesis, and curly brace still matters, but *what* you draw is now your decision.

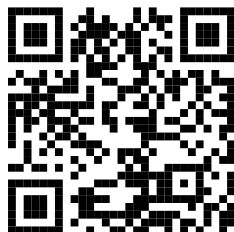
1.3.1 AI tutor

Designing a picture of your own means many small decisions, and there is no sample solution to compare yours with. That makes the AI tutor of this part a good partner here: describe where you're stuck, and it asks the questions that get you moving again.

Your AI tutor

Hints and questions instead of finished programs, in English or German.

► [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/9fxc2eu84z>

1.3.2 Big pictures are made of simple shapes

A house looks complicated until you look closely. Then you see it's just simple shapes stacked together:

- The walls are a rectangle.
- The roof is a triangle sitting on top of the walls.

- The door is a smaller rectangle.
- The windows are small rectangles or circles.
- The tree is a rectangle (the trunk) with a circle on top (the crown).

This is one of the most important ideas in programming: you take something big and break it into small pieces you already know how to build. You'll use this trick again and again, not just for drawings.

1.3.3 Plan before you type

Don't start by typing. Start with a pencil.

! Always have pen and paper at hand

For every exercise in this Creative Coding class, have pen and paper lying on your desk, right next to your keyboard. This is not optional, it's part of your toolkit, just like the editor.

Experienced programmers can sometimes skip the sketch because they can paint the picture in their head. That's a skill they built through years of practice, and you'll get there too. You build it by writing things down: sketches, coordinates, notes. So for now, the rule for you is simple. First the paper, then the keyboard.

Sketch your house on paper and treat the sheet like your canvas: the origin (0, 0) is in the top left corner, x grows to the right, and y grows down. Then write the coordinates next to the important points of your sketch. For example, if your canvas is 400 pixels wide and 500 pixels tall, your walls might be a rectangle starting at (100, 250), 200 wide and 150 tall. Where are the corners of the roof triangle? Where does the door start?

Your first guesses won't be perfect, and they don't have to be. A plan on paper just saves you from guessing blindly at the keyboard.

💡 Sketch on graph paper

The best paper for your sketch is graph paper, the paper from your math notebook with small squares of 5 by 5 millimeters. Let one square stand for 10 pixels. A canvas of 400 by 500 pixels then becomes 40 by 50 squares, which fits comfortably on an A4 page. Draw your house at that size, then use a ruler to find the coordinates: count the squares and multiply by 10, or measure in millimeters and multiply by two, and you have the pixel value for your code.

Working this way trains two skills at once. You learn to handle coordinates with your own hands, not just in code. And you learn to design a picture before you build it, which is how designers and artists work too: first the sketch, then the implementation. In a course called Creative Coding, that design step is just as important as the code.

1.3.4 Draw in the right order

The computer runs your statements from top to bottom, and every new shape is painted **on top of** what's already there, like layers of paint. So the order of your statements decides what covers what:

1. First the background: the sky.
2. Then the large shapes: the walls, the roof, the tree.
3. Then the details that lie on top of other shapes: the door and the windows on the walls, maybe apples on the tree.

If your door disappears, you probably drew it *before* the walls, and the wall rectangle painted over it. Move the door statement further down, and it comes back.

1.3.5 Give your shapes colors

You already know the color commands from the Shapes exercise: `fill` sets the color inside shapes, `stroke` sets the line color, and `strokeWeight` sets the line thickness. These commands are **settings, not actions**. They don't draw anything themselves. They change how every shape *after* them is drawn, until you change the setting again:

```
fill("red");
rect(100, 250, 200, 150); // red walls
rect(180, 330, 40, 70); // still red! fill is still "red"

fill("saddlebrown");
rect(180, 330, 40, 70); // now the door is brown
```

The page [HTML color names](#) lists all the color names you can use, from "skyblue" to "saddlebrown".

1.3.6 Your exercise: House

Now it's your turn. Open the **House** exercise in the web playground and write a program that draws a house with a tree beside it, built from rectangles, circles, triangles, and lines.

A good way to work:

1. Sketch your house on graph paper and note the coordinates.
2. Create the canvas and paint the background.
3. Add one shape at a time. Run the program after every shape, look at the result, and adjust the numbers until it fits. Changing coordinates again and again is completely normal. Professional programmers work exactly like this: try, look, adjust.
4. Keep your code well formatted while you go: indent the statements inside the curly braces and write one statement per line. Tidy code makes it much easier to find the shape you want to change.

The example picture in the exercise is only an idea, not a template. Add whatever you like: a sun, a fence, clouds, a second tree, a cat on the roof. When your program runs and the picture on the screen matches the picture in your head, you've built exactly what you planned.

💡 Exercise: House

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

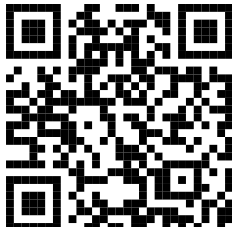
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0010-Basics/house.yaml>

1.3.7 Check your understanding

When you've finished the exercise, take the short quiz below. You answer seven questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: From copying to creating

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/prj4fef0rh>

1.4 Type less, draw more

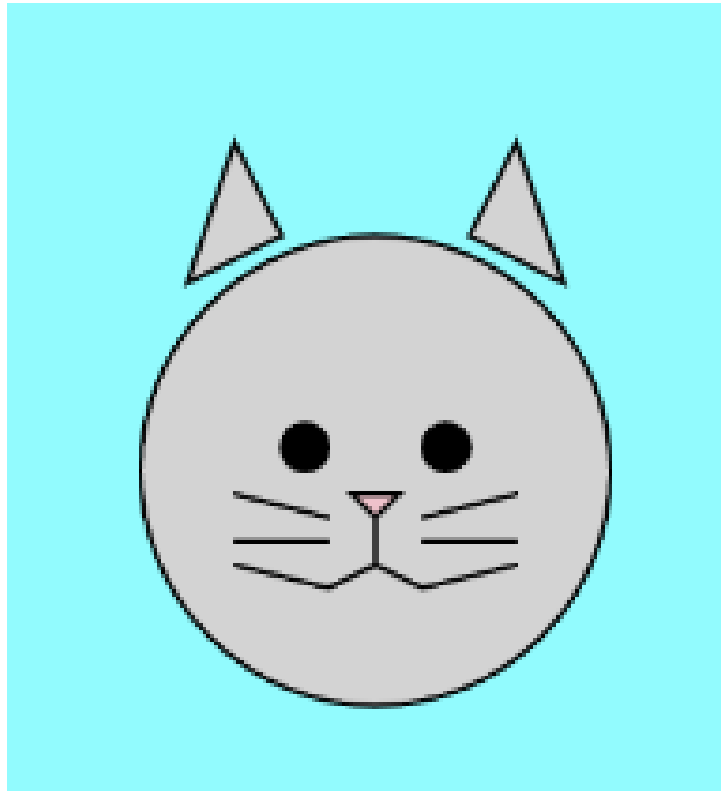


Figure 1.4.1: A cat built from a handful of simple shapes, the kind you'll draw in this chapter.

The cat is your practice piece. You'll put it together from the same rectangles, circles, triangles, and lines you already know, working faster than you did with the house, spending less effort on the typing and more on the picture. The example above is only one idea, your cat can look however you like.

You've typed a program character by character, and you've designed a house of your own. The next exercise is practice: you'll draw a cat, again from rectangles, circles, triangles, and lines. Repetition is how programming skills become natural, the same way regular training works in sports or music. The second picture is already easier than the first.

But practice doesn't mean doing everything the slow way forever. In this chapter you'll learn a few editor tricks that professionals use all day: copying and pasting code, moving lines around, and letting the editor format your code for you.

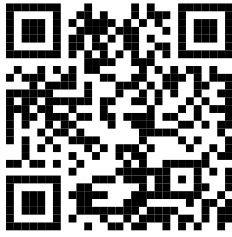
1.4.1 AI tutor

This chapter is about practice and editor tricks. When a keyboard shortcut does nothing on your machine, or a shape refuses to sit where you want it, ask the AI tutor of this part.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/9fxc2eu84z>

1.4.2 Let the clipboard work for you

Pictures are full of repetition. A cat has two ears, which are two very similar triangles. It has two eyes, which are two very similar circles. Its whiskers are several similar lines. Typing each of those statements from scratch is slow and invites typos.

The **clipboard** solves this. Copy a statement with **Ctrl + C** (on German keyboards the key is labeled **Strg**), paste it with **Ctrl + V**, and then change only the numbers:

```
triangle(120, 100, 140, 60, 160, 100); // left ear
triangle(180, 100, 200, 60, 220, 100); // right ear: pasted, then x values changed
```

Two tricks make this even faster:

- You don't have to select a whole line to copy it. Place the cursor anywhere in the line and press **Ctrl + C**, and the editor copies the entire line.
- **Ctrl + V** pastes as often as you like. Paste the same line three times for three whiskers, then adjust the numbers.

Two more shortcuts are worth learning:

- **Alt + Arrow Up** and **Alt + Arrow Down** move the current line up or down. That's perfect for fixing the drawing order: if a shape is hidden behind another one, move its statement further down without any retyping.
- **Ctrl + Z** undoes your last change. Deleted the wrong line? Pasted into the wrong place? **Ctrl + Z** brings it back. You can press it several times to undo several steps, so experimenting is never dangerous.

1.4.3 Let the editor format your code

In the last exercises you formatted your code by hand: indentation inside the curly braces, one statement per line, a space after each comma. That practice was on purpose, because now you know what well-formatted code looks like, and you can judge the result of what comes next.

The editor can format for you. Right-click anywhere in your code and choose **Format Document**. The editor straightens the indentation and the spacing of the whole file in one step.

Two things **Format Document** does *not* do:

- It doesn't fix errors. A missing curly brace or a mistyped command name stays broken, and the red squiggly line stays until you fix it yourself. In fact, when the formatting result looks strange, that's often a hint that a brace is missing somewhere.
- It doesn't think for you. It applies the formatting rules, but you still decide what the code says. Use it to keep your code tidy while you focus on the picture.

! Format your code before you hand it in

In this course, formatting your code before you hand it in is mandatory. That applies to everything you submit: homework, exams, and projects. Whether you format by hand or use **Format Document** is up to you. Code that is handed in unformatted will cost you points.

Why so strict? Because code is read far more often than it is written. Your teacher reads it to grade it, your classmates read it when you work together, and you yourself read it again next week when you want to

change something. Messy code hides mistakes: a missing curly brace is almost invisible in badly indented code and obvious in tidy code. And in professional teams, clean formatting is simply expected, nobody wants to work with code that others can't read. Handing in formatted code is a sign of care, like handing in an essay without crossed-out words and coffee stains.

1.4.4 Your exercise: Cat

Open the **Cat** exercise in the web playground and write a program that draws a simple cat from basic shapes. The example picture is only an idea; your cat can look completely different.

Work the way you practiced with the house, and add the new tricks:

1. Pen and paper first: sketch your cat on graph paper, one square for 10 pixels, and use a ruler to read off the coordinates.
2. Draw one shape at a time and run the program after every step.
3. For repeated parts (ears, eyes, whiskers, paws), write the statement once, copy and paste it, and change the numbers.
4. Use **Alt + Arrow Up** and **Alt + Arrow Down** to put your statements in the right drawing order, and **Format Document** to keep everything tidy.

💡 Exercise: Cat

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0010-Basics/cat.yaml>

1.4.5 Check your understanding

When you've finished the exercise, take the short quiz below. You answer five questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Type less, draw more

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/871pehodxo>

1.5 From cat to elephant: coding with AI

You've now written several programs completely by hand, and that was on purpose: you know what a drawing statement is, how coordinates work, and what tidy code looks like. That makes you ready for the next step. In this exercise you'll work with an AI for the first time, and you'll use it the way professionals do: as a tool for generating code, not just as a toy.

The task: take your finished cat program from the last exercise and get an AI to turn it into a different animal. An elephant, a dog, a penguin, whatever you like. You won't type the new code yourself. Your job is to write the instructions, judge the result, and make it work.

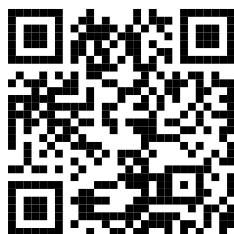
1.5.1 AI tutor

Two different AIs appear in this chapter, and it's worth keeping them apart. The tutor of this part helps you understand the chapter, with hints and questions. The exercise further down has an AI of its own that writes complete code from your prompt, because writing that prompt *is* the exercise.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/9fxc2eu84z>

1.5.2 Writing a good prompt

The message you send to an AI is called a **prompt**, and writing good prompts is a real skill. Remember the genie in the lamp from the start of this course: it grants exactly what you say, not what you mean. An AI works the same way. A vague prompt like “make an elephant” gives the AI room to guess, and it will guess something, just often not what you had in mind.

A good prompt is precise. It says what you want, what the result has to look like, and what the AI must not change. Compare:

Make an elephant.

Here is a TypeScript program that draws a cat using the p5.js library. I am a programming beginner, and this code shows everything I have learned so far. Change the program so that it draws an elephant instead. Keep the same structure: everything stays inside the `setup` function, and the canvas keeps its size of 400 by 500 pixels. Only use the kinds of commands that already appear in my code.

[your cat code]

The second prompt tells the AI the language, the library, the structure to keep, the goal, and, just as important, **your level of knowledge**. The more clearly you describe your wish, the better the result. This skill, describing exactly what you want, is the same skill you practice when you sketch a picture before coding it. Precision is what makes both the computer and the AI do what you actually want.

Telling the AI that you're a beginner is not embarrassing, it's smart. Without that information, the AI writes code the way it would for a professional, full of features you haven't learned yet. Code like that might even run, but it's useless to you: you can't explain it when you hand it in, and, much worse, you can't fix it when something is wrong. Code you don't understand is code you can't control. With the beginner sentence in your prompt, the AI stays inside what you know, and the result remains *your* program.

1.5.3 Show the AI your working code

The most important part of your prompt is your own cat code. Always paste it in. Here's why.

The AI cannot see your screen. It doesn't know that you work in a web playground, that your language is TypeScript, that you draw with the p5.js library, or that your whole program lives inside one `setup` function. If you just ask for "code that draws an elephant", the AI will happily produce something, in some programming language, for some graphics library, in some structure, and there's a good chance it won't run in your environment at all.

Your working cat code solves that problem. Code that runs is the most precise description of your environment that exists: it shows the AI the exact language, the exact library, and the exact structure that your playground expects. When the AI sees it, it will answer with code of the same kind. That's why the rule is: **first make it work, then show it to the AI.**

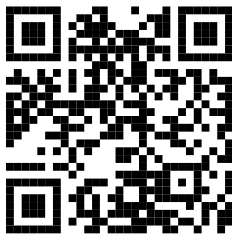
1.5.4 Your exercise: a new animal

This exercise has its own AI, and you'll find it in the box below. Unlike the tutor at the start of this chapter, it doesn't give hints: it writes code from your prompt, exactly the kind of AI this exercise is about. If your teacher picks a different AI tool for the class, use that one instead; the steps stay the same.

💡 Your AI for this exercise

This AI writes code from your prompt. Like the genie in the lamp, it grants what you say, not what you mean.

▶ [Open the exercise AI on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/8uzkn8yyjd>

Work like this:

1. Open your finished cat program in the web playground and check that it still runs. Copy the whole code with `Ctrl + C`.
2. Write your prompt: say which animal you want, what the AI should keep (the `setup` function, the canvas size, the p5.js drawing commands), and paste your cat code below it.
3. Copy the AI's answer, paste it into the playground in place of your old code, and run it.
4. Judge the result. Does it run? Are there red squiggly lines? Does it actually look like the animal you asked for? If something is wrong, you have two options, and both are part of the job: fix small things yourself by adjusting coordinates and colors, or tell the AI in a follow-up prompt what it should change.
5. Keep your code formatted before you show or hand in your result. That rule doesn't change just because an AI wrote the first draft.

Save your cat code somewhere before you replace it, for example in a text file, so you can always go back.

One more thing: read the code the AI gives you. You know every command in it, `rect`, `circle`, `triangle`, and `fill`, so you can follow what it does. You are the programmer here. The AI types faster than you, but you decide what is good enough, and you are responsible for the result.

1.6 When the picture is given: Olympic rings

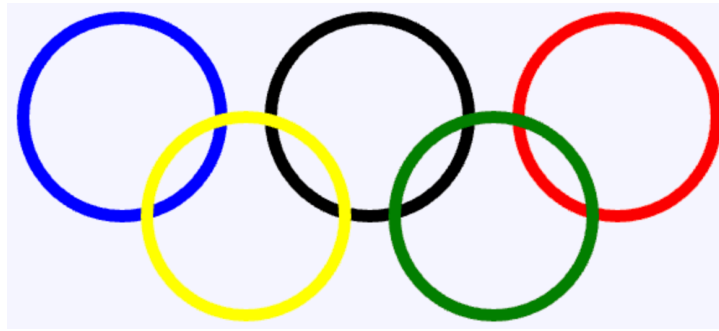


Figure 1.6.1: The five Olympic rings you'll recreate: the right colors, each in its proper place.

This time the picture is given, not invented. You'll recreate the Olympic symbol, the five interlocking rings that everyone recognizes: three rings in a top row, two more below, each in its proper color and its proper place. When you're done, your version should look just like the original.

With the house and the cat, you invented the picture yourself. If a window ended up a little further left than planned, nobody minded. This exercise is different. You'll draw the Olympic symbol, the five interlocking rings. Now the picture is *given*, and your job is to reproduce it: the right colors, the right sizes, the right positions.

This is a very common situation in real projects. A customer brings a logo, a designer delivers a layout, a game needs an existing character, and the programmer's task is to match the original exactly. Working precisely from a given design is a skill of its own, and this exercise trains it.

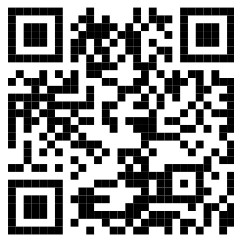
1.6.1 AI tutor

Reproducing a given picture exactly means measuring, computing, and checking. If your rings won't line up, describe the problem to the AI tutor of this part instead of moving numbers around at random.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/9fxc2eu84z>

1.6.2 A ring is a circle without a fill

You already know everything you need to draw a ring. A ring is simply a circle where only the outline is visible:

- `noFill()` switches the filling off, so the inside stays transparent.
- `stroke` sets the color of the outline.
- `strokeWeight` sets the thickness of the outline. Give it a generous value, and the thin circle line becomes a bold ring.

Remember that these three commands are settings: they stay active for every shape that follows until you change them. For the Olympic rings, you set `noFill` and `strokeWeight` once at the top, and then only change the `stroke` color from ring to ring.

1.6.3 Find the pattern before you measure

You could put your sketch on graph paper and measure the position of every single ring with a ruler. That works, but take a closer look at the symbol first, because it's full of regularity:

- All five rings have the same diameter.
- The three upper rings sit on one line, at equal distances from each other.
- The two lower rings sit on a second, deeper line, and each one hangs exactly halfway between two upper rings.

Once you see that, you barely need the ruler. Choose a diameter and a position for the first ring, and the coordinates of the other four follow almost automatically: same y for the upper row, a fixed step in x from ring to ring, and the lower row shifted right by half a step and down by a fixed amount.

Looking for structure before reaching for the ruler is a designer's habit and a programmer's habit at the same time. Regular patterns mean fewer numbers to invent, and fewer numbers mean fewer mistakes.

1.6.4 The right colors

Part of matching a given design is matching its colors. The five Olympic rings are blue, black, red, yellow, and green, and p5.js understands all five as color names: `stroke("blue")`, `stroke("black")`, and so on. Check your result against the original image: if your rings are there but the colors are in the wrong order, the symbol is still wrong.

1.6.5 Your exercise: Olympic rings

Open the **Olympic Rings** exercise in the web playground. Sketch the five rings on graph paper, use the regularities you found to work out the coordinates, and then draw ring by ring, running your program after each one.

One thing you may notice: in the real symbol, the rings don't just lie on top of each other, they are *woven* together, each ring passing over and under its neighbors. Ignore that for now and simply draw five overlapping rings; the exercise doesn't count the overlapping order. Weaving the rings properly is a much trickier puzzle, and it gets its own exercise a little later in the course.

💡 Exercise: Olympic Rings

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0010-Basics/olympic_rings.yaml

1.6.6 Check your understanding

When you've finished the exercise, take the short quiz below. You answer four questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Olympic rings

► Take the quiz on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/b9audajb2e>

1.7 Arcs: drawing parts of a circle



Figure 1.7.1: The friendly smiley face you'll draw in this chapter.

This chapter adds a new shape to your kit: a piece of a circle. You'll use it to draw a smiley face in the middle of the canvas, with two eyes made from shapes you already know and a mouth that is a real curved smile, a single slice of a circle rather than a full one. That curve is the piece you couldn't draw before.

Rectangles, circles, triangles, and lines got you a long way: a house, a cat, the Olympic rings. But some pictures need a shape you can't build from those, a *part* of a circle. Think of a smiling mouth. It's not a full circle, it's just the lower piece of one. In p5.js, that shape is called an **arc**, and it comes with a new command: **arc**.

In this exercise you'll type in a given program again, exactly as it's shown, like you did in the very first exercise. It draws a smiley face, and its mouth is an arc. Typing first and experimenting afterward is on purpose: **arc** is the trickiest command you've met so far, and the best way to understand it is to see it work and then start turning its knobs.

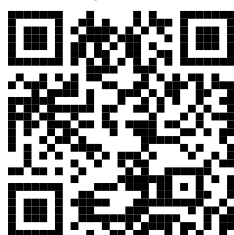
1.7.1 AI tutor

arc is the trickiest command in this part, and its start and end angles confuse almost everybody at first. The AI tutor of this part is a good place to ask why a certain pair of angles draws the piece it draws.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/9fxc2eu84z>

1.7.2 The arc command

`arc` takes six parameters. The first four are like an ellipse: the `x` and `y` of the center, then the width and the height. The last two are new: the **start angle** and the **end angle**. They say where on the outline the visible piece begins and where it ends.

```
// +----- x coordinate of the center
// | +----- y coordinate of the center
// | | +----- width
// | | | +----- height
// | | | | +----- start angle
// | | | | | +----- end angle
// v v v v v v
arc(200, 250, 200, 150, 25, 180);
```

Because width and height are separate, an arc doesn't have to be part of a perfect circle. Make the height smaller than the width, and you get a piece of a flattened circle, an ellipse. That's exactly what makes the smiley's mouth look like a relaxed smile instead of a half circle.

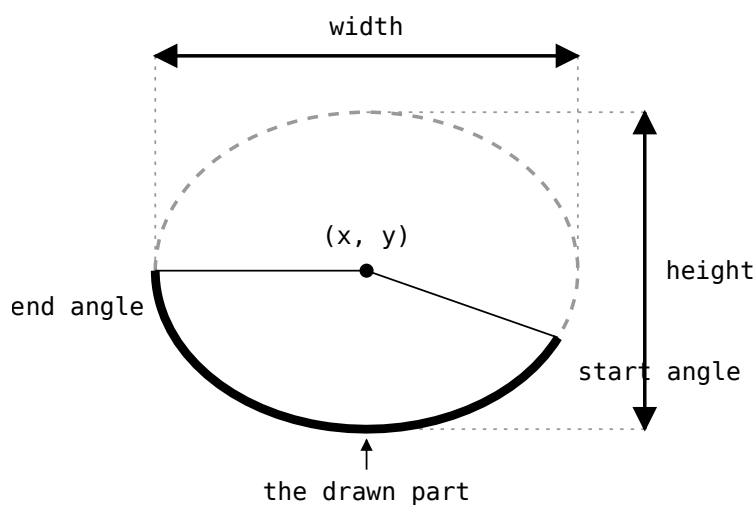


Figure 1.7.2: The six parameters of `arc`, shown for the smiley's mouth. The dashed ellipse is invisible; only the thick piece between the start angle and the end angle is drawn.

1.7.3 Angles on the canvas

Out of the box, p5.js measures angles in **radians**, a unit you'll meet in math class in a few years. The statement `angleMode(DEGREES)` at the top of the program switches p5.js to degrees, the unit you already know: a full circle is 360° .

Where do the angles point? On the canvas, 0° points to the right, and from there the angle grows **clockwise**:

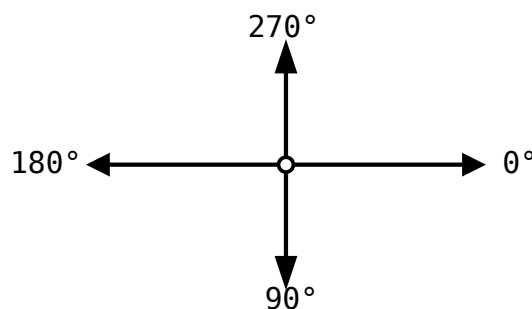


Figure 1.7.3: Angle directions on the canvas: 0° points right, and angles grow clockwise, so 90° points down.

In math class, angles grow counterclockwise, so 90° would point up. On the canvas it's the other way around, and the reason is an old friend: the `y` axis points down. The smiley's mouth shows what that means in practice: `arc(200, 250, 200, 150, 25, 180)` starts at 25° (a little below the 3 o'clock position) and runs clockwise

through 90° (the bottom) to 180° (the 9 o'clock position). The visible piece is the lower part of the outline: a smile.

1.7.4 Learn commands by experimenting

Understanding the parameters of `arc` from a description alone is hard. Nobody does it that way, professionals included. The way to understand a command like this is to experiment with it:

1. Type in the program and run it, so you have a working starting point.
2. Change **one parameter at a time**, run again, and watch what happens. What does a start angle of 0 do? An end angle of 270? A height of 300?
3. If the result gets confusing, **Ctrl + Z** takes you back to the last working state.

Changing one thing at a time is the important part. If you change three numbers at once and the picture jumps, you can't tell which change did what. One change, one run, one observation: that's how programmers explore every unfamiliar command.

And remember the tooltips: hover your mouse over `arc` in the editor, and the playground shows you the documentation of the command, including the list of its parameters.

1.7.5 Your exercise: Smiley face

Open the **Smilie Face (Arcs)** exercise in the web playground and type in the program, exactly as shown, every character. Run it and enjoy your smiley. Then take the time to play with the parameters of `arc` until the command feels familiar: make the smile wider, flatter, upside down, or turn the mouth into a surprised little "o".

The arc knowledge you build here is not just for smileys. In the next exercise, arcs are the key to a puzzle that full circles can't solve: weaving the Olympic rings together properly.

Exercise: Arcs

[▶ Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0010-Basics/arcs.yaml>

1.7.6 Check your understanding

When you've finished the exercise, take the short quiz below. You answer five questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

Quiz: Arcs

[▶ Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/b55a66uhpf>

1.8 Fixing code you didn't write

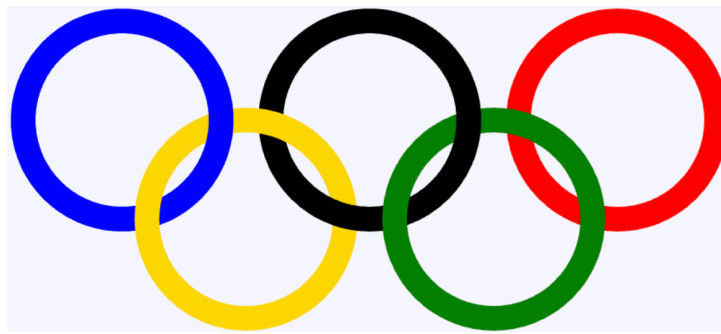


Figure 1.8.1: What the program you're handed already draws: close to the real symbol, but the interlocking isn't quite right yet.

This time you won't write the program from scratch. You get one that someone else already wrote, where the five Olympic rings don't quite link up, and your job is to spot why and repair it, until each ring weaves over one neighbor and under the other the way it does on the real symbol.

So far, every program in this course started with you (or your AI) writing the code. This exercise turns the situation around: you get a program that someone else wrote. It almost works, but not quite, and your job is to find the problem and fix it.

Don't underestimate this exercise. Professional programmers spend most of their time exactly like this: reading code they didn't write, understanding it, and carefully changing it. Whoever wrote the original, a colleague, an AI, or you-three-weeks-ago, the skill is the same.

1.8.1 AI tutor

Reading a program somebody else wrote is a skill of its own, and asking questions about it is part of that skill. The AI tutor of this part explains what a piece of the given code does, without handing you the repaired program.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/9fxc2eu84z>

1.8.2 The puzzle: weaving the rings

In the Olympic Rings exercise you drew five overlapping rings and ignored *how* they overlap. Now look closely at the real symbol: the rings are woven together. Follow the blue ring with your eyes: where it crosses the yellow ring, it passes **over** it in one crossing and **under** it in the other.

Here's the catch: you can't draw that with five full circles. The computer paints shapes in order, so a later circle always lies completely on top of an earlier one. A full circle can be over or under a neighbor, but never both at once.

The trick is to stop thinking of a ring as one shape. Split it into **arcs**: draw most of the ring early, and paint the small missing piece *after* the neighbor ring, right where it has to pass over. The eye sees a woven ring; the code sees a few arcs in a well-chosen order. The layering rule you know, later statements paint over earlier ones, is exactly what makes this work.

1.8.3 Read before you change

The starter code of the exercise already solves most of this puzzle. Before you change anything, spend time understanding it:

1. **Run the program** and put its output next to the target image. Compare crossing by crossing: where do the rings weave correctly, and where not? Finding the difference between “what it should do” and “what it does” is always the first step of fixing a program.
2. **Read the code from top to bottom.** You know every command in it, including `arc` and `angleMode(DEGREES)` from the smiley exercise. The comments tell you which statement belongs to which ring.
3. **Study how the finished rings do it.** The blue and black rings are already woven correctly, and they show the pattern: a ring appears several times in the code, as arcs, with other rings drawn in between. Work out which piece of the ring each arc paints. The angle diagram from the arcs chapter helps: 0° points right, and angles grow clockwise.

Reading code this way takes patience, and that’s normal. Understanding a program you didn’t write is slower than writing your own, even for professionals.

1.8.4 Fix the red ring

The broken part is the red ring: it’s still drawn as one full `circle`, so it can’t weave with its neighbor. That’s your repair job:

1. Look at the target image and decide where the red ring must pass over and where it must pass under the ring it crosses.
2. Replace the red `circle` with arcs, following the same pattern that the blue and black rings use. Think about the angles each arc needs, and about *where in the code* each arc has to stand: before or after the neighbor it crosses.
3. Change a little, run, compare with the image, and repeat. If a change makes things worse, `Ctrl + Z` takes you back.

Be careful to change only what’s broken. The other four rings are correct, and a fix that breaks something else is not a fix. Comparing the picture with the target after every run is your safety net.

When the five rings finally weave like the original, take a moment: you’ve just read a program you didn’t write, understood a genuinely tricky idea in it, and repaired it without breaking the rest.

💡 Exercise: Olympic Rings Advanced

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0010-Basics/olympic_rings_advanced.yaml

1.8.5 Check your understanding

When you’ve finished the exercise, take the short quiz below. You answer five questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Fixing code you didn't write

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/44cpdor3sk>

1.9 Practice projects: flowers and a soccer field

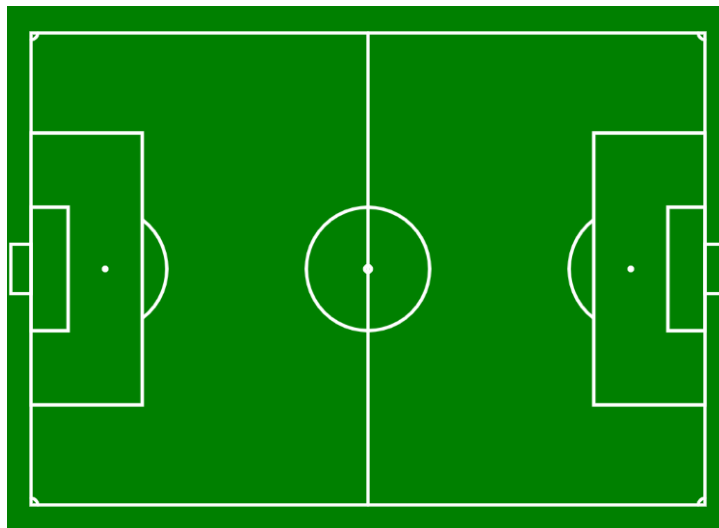


Figure 1.9.1: A soccer field, drawn true to the measurements of a real one

Two projects close out this part, both written like real job briefings. First you'll draw two flowers side by side, each petal placed exactly where a designer's figure says it belongs. Then you'll draw the soccer field above, with its penalty areas, center circle, and corner arcs, sized to match a real field down to the meter.

You now know everything you need for bigger tasks: shapes, arcs, colors, coordinates, planning on paper, and a careful way of working. Time to put it all together. This chapter contains two larger practice projects. Both are written like real job briefings. In one you're part of a game development team, in the other you work for a TV station.

Take these projects seriously. They are practice for exactly the kind of task you'll get in an assessment, and the flower project even states the areas you'd be graded on. Plan on paper, work in small steps, and hand in formatted code, just as you've learned.

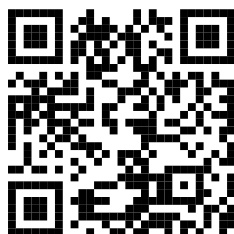
1.9.1 AI tutor

Both projects in this chapter are practice for the kind of task you get in an assessment, so get as far as you can on your own first. When you're truly stuck, the AI tutor of this part gives you the next hint, never the finished project.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/9fxc2eu84z>

1.9.2 Coordinates come in many forms

With the house and the cat, you invented the coordinates yourself. With the Olympic rings, you read them from a symbol. These two projects add the two forms that professionals meet most often:

- **A dimensioned drawing.** The flower project gives you a designer's figure with guide lines, diameters, and distances, the way technical drawings do. Nobody hands you ready-made x and y values; you have

to *compute* them. For example, if a petal with a diameter of 40 sits directly on top of a flower center with a diameter of 60, then the petal's center lies 50 pixels above the flower's center: half of 60 plus half of 40. Working out coordinates like this, from sizes and distances, is a core skill this project trains.

- **Real-world measurements.** The soccer field project gives you the dimensions of a real field, in meters. Your job is to bring them onto a canvas that is measured in pixels, without distorting the proportions. You'll get two new tools for that in a moment.

In both cases, your graph paper sketch is the bridge: draw the figure, write the given measurements next to it, and turn them into canvas coordinates before you type a single statement.

1.9.3 Project 1: Flowers

Your team's designers have specified two flowers. The right flower is the easier one: it's symmetrical both horizontally and vertically, so once you know one petal's position, the mirrored petals follow by symmetry. The left flower is only vertically symmetrical, which makes its petal positions trickier to work out. The stem of each flower is an arc, drawn by eye; not every part of a design is specified to the pixel, and deciding the unspecified parts tastefully is part of the job too.

Use the symmetry when you compute your coordinates: mirrored parts lie at the same distance from the flower's center line, once to the left and once to the right, or once above and once below. If you compute the top petal's position, you get the bottom petal's position almost for free.

💡 Exercise: Flower Basics

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

https://ts-web-playground-server-bfd6bjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0010-Basics/flower_basics.yaml

1.9.4 Project 2: A soccer field

The TV station wants a background graphic of a soccer field, true to the real thing: penalty areas, center circle, penalty points, corner arcs. The task links to a diagram with the real measurements in meters. A real field is about 100 meters wide and 70 meters deep, and your drawing must keep those proportions.

Converting every measurement from meters to pixels by hand would be painful. The starter code of this exercise shows you a much better way, and it introduces three new ideas.

1.9.4.1 Constants give numbers a name

At the top of the starter code, outside the `setup` function, you'll find lines like these:

```
const SCALE = 5;
const MARGIN = 4;
```

The keyword `const` creates a **constant**: a value with a name. From then on you write `SCALE` in your code instead of the number 5. That has two advantages. First, `MARGIN` tells a reader what the number *means*, where a bare 4 explains nothing. Second, the value lives in exactly one place: if you change the 5 to a 10, every part of the program that uses `SCALE` changes with it. By convention, constant names are written in capital letters.

1.9.4.2 `scale` multiplies everything

The statement `scale(SCALE)` stretches everything that is drawn after it by that factor. This is the trick that makes the meter measurements usable: you write all coordinates and sizes *in meters, as if a meter were a pixel*, for example a line of length 70 for the halfway line. With `scale(5)`, that line is drawn 350 pixels long. Want a bigger graphic? Change the constant `SCALE` from 5 to 10, and the whole field doubles, proportions intact, without touching a single coordinate.

1.9.4.3 translate moves the origin

Normally the origin (0, 0) sits in the top left corner of the canvas. The statement `translate(MARGIN, MARGIN)` moves it to the right and down by the margin. After that, the point (0, 0) is the top left corner *of the field*, not of the canvas, and the grass margin around the field appears by itself. You can then use the field's own measurements directly as coordinates, which keeps all your numbers simple.

One more thing you'll see in the starter code: a statement called `push()`. It saves the current drawing state, and it belongs to the `scale` and `translate` machinery. Leave it where it is; you'll learn the details behind it later in the course.

💡 Exercise: Soccer Field Basics

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

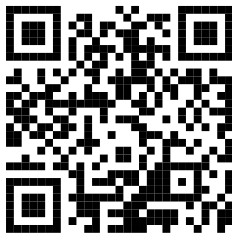
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0010-Basics/soccer-field-basics.yaml>

1.9.5 Check your understanding

When you've finished the exercise, take the short quiz below. You answer five questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Practice projects

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/gxu32sj78u>

1.10 Reading the documentation: a command of your own

Every command you know so far, `rect`, `circle`, `arc`, and the others, was handed to you. The book explained the parameters, and an exercise showed a working example. That's how learning starts, but it's not how it continues. p5.js has hundreds of commands, and no book and no teacher will ever walk you through all of them. Professionals learn a new command by reading its **documentation**: the official description that the makers of a library publish for every command they offer.

Reading documentation is a skill of its own, and this chapter trains it. You'll pick one p5.js command that nobody taught you, learn it from the official documentation, build an example with it, and describe it in your own words. In short: this time, *you* write a little piece of this book.

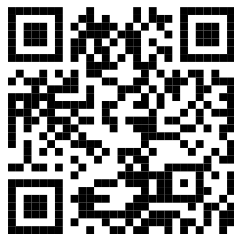
1.10.1 AI tutor

Two AIs appear in this chapter, with different jobs. The AI tutor of this part is here for your questions about the course material, exactly as in every other chapter. The exercise itself sends you somewhere else on purpose, to a general AI on the internet, because meeting material nobody prepared for you is what this chapter trains.

Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/9fxc2eu84z>

1.10.2 The p5.js reference

The official documentation of p5.js is called the [p5.js reference](#). It's a large catalog with one page per command, grouped by topics like shapes, colors, and text. Every page follows the same pattern, and once you can read one, you can read them all:

- **The description** says in a few sentences what the command does.
- **Example programs** show the command in action. You can run them right on the page, and you can even edit them there and run them again. That turns the reference page into a small playground of its own.
- **The syntax and the parameter list** show what goes between the parentheses: the name of each parameter, its order, and what it means.

One detail in the syntax matters: parameters in square brackets are **optional**. You can leave them out, and the command picks a sensible value on its own. For example, the syntax of `rect` is written as `rect(x, y, w, [h])`. Call it with all four values and you set the width and the height separately. Leave the `[h]` out, and p5.js simply uses the width as the height too, so you get a square.

By the way, you've been reading tiny pieces of this documentation for weeks: the tooltip that appears when you hover over a command in the playground editor is a short version of exactly these reference pages.

1.10.3 Reference examples are not playground programs

The examples in the p5.js reference are great for understanding a command, but you can't paste them into the web playground unchanged. Three things to watch out for:

- **They're written in JavaScript, not TypeScript.** The two languages are close relatives, and the drawing commands look exactly the same. Still, the programs around them are structured differently than what your playground expects. So don't copy whole example programs. Read them, understand the idea, and then write your own program in the structure you know: `createCanvas` first, everything inside the `setup` function.
- **The example canvas is tiny.** Most reference examples run on a canvas of only 100 by 100 pixels, so all the coordinates in them look strangely small. On your 400 by 500 canvas, you'll use bigger numbers.
- **Angles are measured in radians.** Reference pages write angles in radians, the unit p5.js uses out of the box, so you'll see values like `PI` or `HALF_PI`. In your programs, `angleMode(DEGREES)` switches to degrees, where a full circle is 360. When a page talks about angles, translate in your head: `PI` is 180 degrees, `HALF_PI` is 90 degrees.

1.10.4 Choose your command

For this exercise, you'll pick **one command** from one of three sections of the p5.js reference. All three belong to the *Shape* group:

- **2D Primitives** is the section where almost all your drawing commands come from. You already know `arc`, `circle`, `line`, `rect`, and `triangle`, so those are off the table, and so is `ellipse`, because a later chapter of this course teaches it. But the section holds more: a command for a single dot, one for squares, one for four-cornered shapes that don't have to be rectangles. A solid choice.
- **Curves** is more advanced. These commands draw lines that bend, and they're controlled by several points at once: some points the curve passes through, others only pull it into shape. Expect to experiment quite a bit before the parameters make sense.
- **Custom Shapes** is also advanced. Here you don't get one command but a small team of commands that work together to build any shape you like, corner by corner. If you choose this section, you learn the whole team.

Pick whatever makes you curious. The advanced sections take more effort, and they're also more impressive to present. The only firm rule: choose a command you haven't used in this course yet.

1.10.5 Your exercise: learn it, build it, describe it

Your task has three parts, and together they work like a small chapter of this book, written by you:

1. **Learn the command from its reference page.** Read the description and the parameter list, then run the examples on the page and experiment with them the way you learned with `arc`: change one value at a time, run, and watch what happens. Keep going until you can predict what a change will do before you run it.
2. **Build your own example.** Sketch a small picture on graph paper that shows your command well, work out the coordinates, and write the program in the web playground: TypeScript, `createCanvas` first, everything inside `setup`, formatted before you hand it in. Don't reuse a picture from the reference page; the point is a picture of your own.
3. **Describe the command in writing.** Write a short text, about half a page, that explains your command to a classmate who has never seen it: what it draws, what each parameter means, and one thing that surprised you while experimenting. Use your own example as the illustration, the same way this book used the smiley's mouth to explain `arc`.

Your teacher will tell you how to hand in your program and your text.

1.10.6 AI is allowed, understanding is required

You may use an AI for any part of this exercise: to explain the reference page, to help with your example, or to improve your text. This time, don't use the tutor from this book. Pick a general AI on the internet instead, for example ChatGPT or Google Gemini. An AI like that doesn't know what you've learned, and its answers will sometimes contain things you've never seen. That's on purpose. You're doing research now, and meeting

the unknown is part of research. When something new shows up, ask follow-up questions until you understand it, or leave it out of your program.

Because of that, the two rules you already know matter even more here. Paste your own working code into the prompt, so the AI answers in the language, library, and structure your playground expects. And say that you're a beginner and that your code shows everything you've learned, so the answer stays inside what you can explain.

There's a second reason to stay inside what you understand: **you'll present your work in class.** You'll show your picture, explain what your command does and what each parameter means, and answer questions. Expect live requests too, like "make it bigger" or "let it bend the other way", with the class watching. An AI can write text for you, but it can't stand in front of the class for you. If there's a single line in your program or a single sentence in your text that you can't explain, replace it or ask until you can. Code you don't understand is code you can't control, and a description you can't defend isn't yours yet.

There's a nice reward at the end. Everyone presents a different command, so after the presentations the class knows many more commands than any one of you learned alone, each one explained by someone who really understands it. That's your class's own little reference, and you wrote one page of it.

Part 2

Variables

2.1 Variables: values with a name

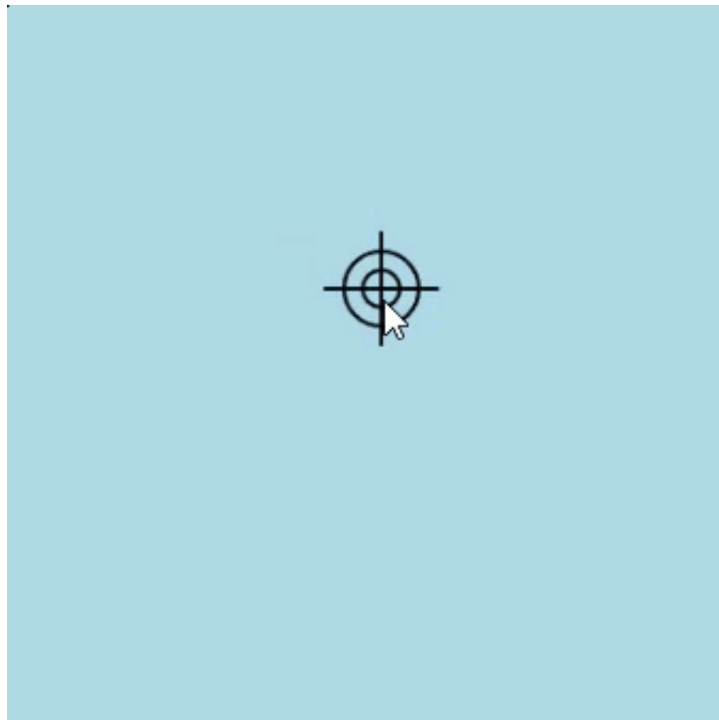


Figure 2.1.1: The finished crosshairs, resting in the middle of the canvas and ready to take aim.

This is your first interactive program. You'll build a set of crosshairs, two thin lines (one horizontal, one vertical) crossing at a single point, wrapped by two circles like an aiming reticle in a game. And they won't sit still: as you slide the pointer around the canvas, the whole crosshairs glides along with it, always centered right where you're pointing.

Every program you've written so far draws one picture, and that picture never changes. The house stands still. The rings don't move, and the smiley keeps smiling. In this part of the course your programs come alive. They react to the mouse, and when a value changes while the program runs, the picture changes with it. The idea that makes all of this possible is the **variable**.

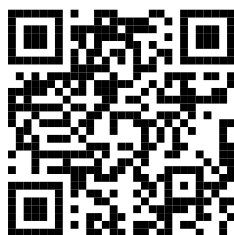
2.1.1 AI tutor

This part of the book has its own AI tutor, and every chapter of the part links to it. It knows the material of the Introduction part plus everything this part teaches, so from now on you can ask it about variables too.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/pl0qyaxsw4>

2.1.2 A variable is a named place for a value

A variable is a place in the computer's memory where a value is stored, with a name attached to it. Think of a small box with a label: the label is the name, and inside the box sits the current value. Whenever your code

uses the name, the computer opens the box and takes out whatever is inside *right now*. The word “variable” comes from *vary*, because the content of the box can change while the program runs.

You’ve already met a close relative. In the soccer field project you used **SCALE** and **MARGIN**, named values that are set once and never change. Those are **constants**. A variable works the same way, with one difference: its value may be replaced while the program is running. That difference sounds small, but it’s exactly what a still picture needs to become an interactive program.

2.1.3 p5.js keeps some variables up to date for you

The easiest way to see a variable in action is to use one that p5.js manages for you. These built-in variables are called **system variables**. You never create them and you never change them yourself: p5.js does that, and your code just reads them by name.

The two system variables you’ll use today are `mouseX` and `mouseY`. They always contain the current position of the mouse on the canvas. `mouseX` is the distance from the left edge, `mouseY` the distance from the top edge. Move the mouse, and p5.js immediately puts the new position into the two boxes. So this statement draws a circle wherever the mouse is at the moment the statement runs:

```
circle(mouseX, mouseY, 20);
```

The value inside `mouseX` is always a *number*. Every value in a program has a **data type**, the kind of value it is, and for `mouseX` and `mouseY` that type is **number**. You’ll learn more about data types in the next chapter.

2.1.4 Code that runs when something happens

Reading `mouseX` is only half the story. All your code so far lives in the `setup` function, and `setup` runs **exactly once**, when the program starts. If you draw a circle at `mouseX` inside `setup`, you get one circle at the mouse’s starting position, and nothing ever moves again.

To follow the mouse, you need code that runs again and again. p5.js gives you a way to do that. Write a function called `mouseMoved`, and p5.js calls it automatically **every time the mouse moves across the canvas**. You write that function exactly like `setup`, with the word **function**, a name, and the statements between curly braces. Only the name is different, and the name is what tells p5.js when to run it:

```
function setup() {  
  createCanvas(200, 200);  
}  
  
function mouseMoved() {  
  background("lightblue");  
  circle(mouseX, mouseY, 20);  
}
```

A function like `mouseMoved` is called an **event function**, because the movement of the mouse is an *event*, and p5.js reacts to the event by running your function. The program now works like this: `setup` runs once and creates the canvas. Then the program waits. Every time you move the mouse even one pixel, p5.js runs `mouseMoved` once, and your drawing code runs with the fresh values in `mouseX` and `mouseY`.

Exactness matters here in a sneaky new way. The name `mouseMoved` is an agreement between you and p5.js. If you type `mousemoved` or `MouseMoved`, you don’t get an error message: you simply get a function that p5.js never calls, and nothing happens when you move the mouse. If your program stays silent, check the spelling of the function name first.

Look at the `background("lightblue")` at the top of `mouseMoved`. It repaints the whole canvas on every mouse movement, and that erases the circle from the previous position. Without that line, every position leaves its circle behind and the canvas fills up with smears. Try removing the line later in the exercise, watch what happens, then put it back.

💡 Only the background at the start? That’s correct

When the Crosshairs program starts, you see nothing but the light blue background. Nothing is broken: all the drawing code sits in `mouseMoved`, so the first crosshairs appears the moment the mouse first moves across the canvas.

2.1.5 Calculating inside a parameter

Crosshairs are more than a dot at the mouse position. You draw a horizontal line, a vertical line, and two circles, all centered on the mouse. The horizontal line should start 15 pixels left of the mouse and end 15 pixels right of it. The command `line` takes four parameters, the x and y of its starting point followed by the x and y of its end point. You don't need to know the mouse position in advance for either of them, because a parameter can contain a small **calculation**:

```
line(mouseX - 15, mouseY, mouseX + 15, mouseY);
```

When the statement runs, the computer takes the current value of `mouseX`, subtracts 15 for the starting point, adds 15 for the end point, and draws the line. Say the mouse is at (120, 80): the line runs from (105, 80) to (135, 80), exactly 30 pixels wide and centered on the mouse. The vertical line works the same way, with the calculation on `mouseY`.

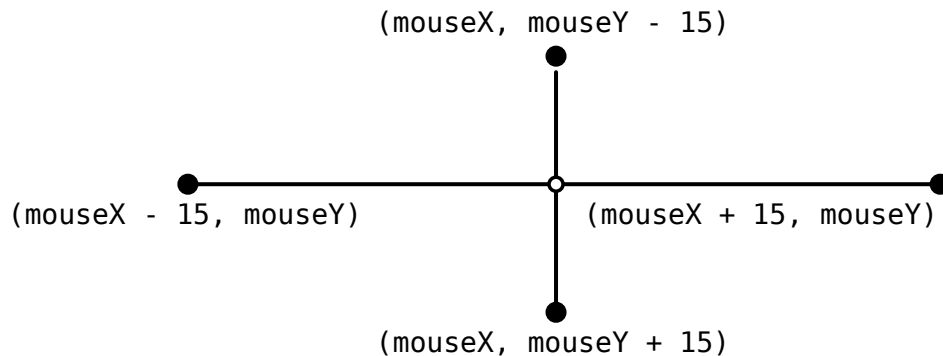


Figure 2.1.2: The two lines of the crosshairs. Every end point is a small calculation with `mouseX` and `mouseY`; the center is the mouse position itself.

The two circles are easier: their center is simply $(\text{mouseX}, \text{mouseY})$, no calculation needed. Use `noFill()` before drawing them, like you did for the Olympic rings, so only their outlines appear and they don't cover the lines underneath.

2.1.6 Your exercise: Crosshairs

Imagine you're building a balloon shooting game. The player needs crosshairs that follow the mouse to aim. That's your task: crosshairs made of two lines and two circles, always centered on the mouse.

Pen and paper first, because this is your first sketch where the coordinates are **formulas instead of numbers**. Draw the crosshairs large on graph paper, mark the center point, and label it $(\text{mouseX}, \text{mouseY})$. Then copy the table below onto your paper and complete it. The first row is done for you as a model:

Point	Coordinates as formulas	Mouse at (120, 80)	Mouse at (40, 150)
Horizontal line, left end	$(\text{mouseX} - 15, \text{mouseY})$	(105, 80)	(25, 150)
Horizontal line, right end			
Vertical line, upper end			
Vertical line, lower end			
Center of both circles			

The second column is the important one: you can't write a plain number there, because the center moves. The formula *is* the coordinate. The last two columns are practice at reading a formula. You play the part of the computer, put a value into `mouseX` and `mouseY`, and work out what the parameter becomes.

Then head to the playground:

1. Put your drawing code into the `mouseMoved` function, below the `background` statement that's already there.
2. Start with the two lines, run the program, and move the mouse across the canvas to test.

3. Add the two circles with `noFill()`. In the sample, the line arms are 15 pixels long and the circles have diameters of 10 and 20, but you can design your own crosshairs, as long as everything stays centered on the mouse.
4. Test what you can actually see. Nobody can place a mouse on the pixel (120, 80) by hand, so test the *behavior* instead. Move the pointer slowly all over the canvas and check that the crossing point stays right under the tip of the pointer, that the arms keep the same length in all four directions, and that the circles stay centered on the crossing point. If the crosshairs drifts away from the pointer, a plus and a minus have swapped places in your formulas.

Exercise: Crosshairs

► [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0020-Variables/crosshairs.yaml>

2.1.7 Check your understanding

When you've finished the exercise, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

Quiz: Variables

► [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/z4doqn6dvt>

2.2 Your first formula: snapping to a grid

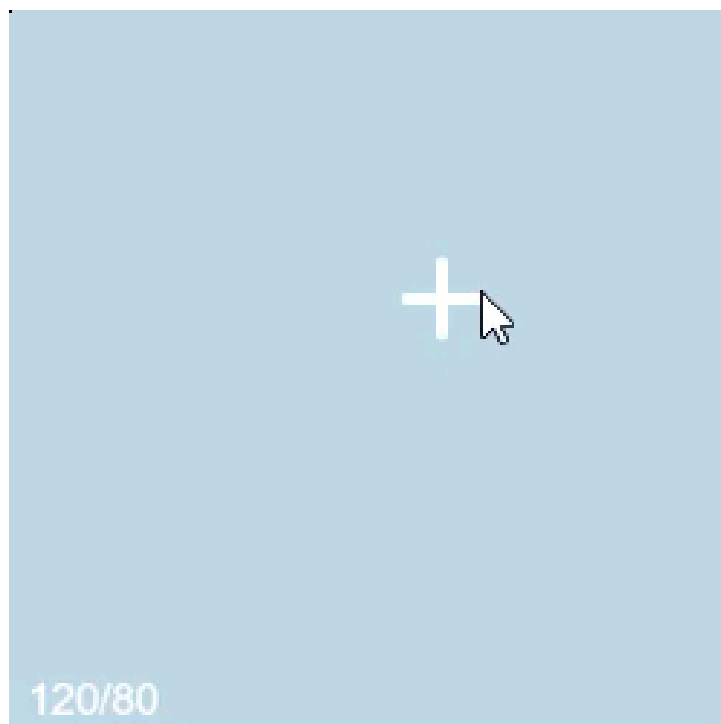


Figure 2.2.1: A white cross-shaped marker that has clicked into place on the grid, its tidy position shown as 120/80.

You'll build a little marker that chases your mouse across the canvas. But it won't glide there smoothly: instead it jumps in fixed steps, clicking into place at the points of an invisible grid. As it moves, a small label in the corner shows its position, always landing on neat, round numbers.

Open a presentation program and drag a shape around. Instead of moving smoothly, the shape jumps in small steps, so that shapes line up neatly without fiddling. This behavior is called **snapping**: the shape snaps to an invisible grid. In this chapter you build exactly that. A marker follows the mouse, but instead of following it pixel by pixel, it jumps from grid corner to grid corner, every 40 pixels. To get there, you'll declare your own constants and meet data types properly. Then you'll write your first real formula.

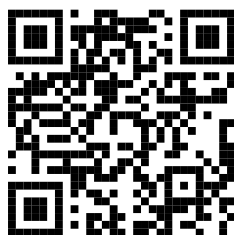
2.2.1 AI tutor

A formula that produces the wrong number is hard to spot, because it looks perfectly fine on the screen. The AI tutor of this part goes through a formula with you step by step, with your own values.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/pl0qyaxsw4>

2.2.2 Constants you declare yourself

The starter code of the Snapping exercise begins with two lines you haven't written yourself before:

```
const snap: number = 40;           // marker must snap every 40 pixels
const markerSize: number = 10;    // size of marker
```

Read the first line from left to right, piece by piece:

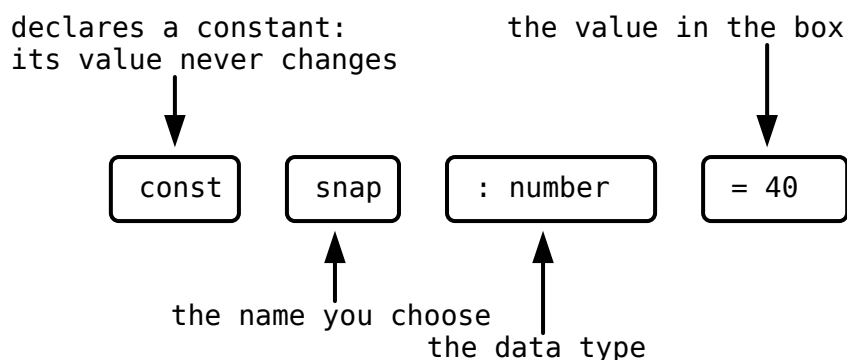


Figure 2.2.2: A `const` declaration piece by piece.

The **data type** names the kind of value the constant holds; it gets a section of its own in a moment. From now on, you write `snap` in your code instead of the number 40. And the semicolon at the end closes the statement, as always.

You saw the advantages of constants in the soccer field project: a name like `snap` tells the reader what the number *means*, and the value lives in one place. Change the 40 to 50 and the whole program uses the new grid size. TypeScript also protects a constant. If a later line tries to give `snap` a new value, you get a red squiggle immediately.

2.2.3 Every value has a data type

Every value in a program has a data type, the kind of value it is. Three types matter most in this course:

- **number** is any numeric value: 40, 3.25, -10, 0. Every coordinate, size, and angle you've used so far is a number.
- **string** is text. A string is written in quotes, and you've been using strings since your very first program: every color name like `"lightblue"` is a string.
- **boolean** is the smallest type of all: its only values are `true` and `false`. Booleans answer yes/no questions, and you'll meet them in a few chapters.

The data type in a declaration is a promise: `const snap: number = 40;` promises that `snap` holds a number. TypeScript holds you to it. Try `const snap: number = "hello";` and the spell checker for code marks the line immediately, because text is not a number.

! Course rule: every declaration gets a data type

In this course, **every constant and every variable you declare must have an explicit data type**, like `: number` or `: string`. Honestly, this is not technically necessary, because TypeScript can usually figure out the type on its own (it sees the 40 and knows it's a number). But you are here to learn, and writing the type out forces you to think, every single time, about what kind of value you're working with. That habit will pay off for years. Like showing your work in math class: the shortcut exists, but not while you're learning.

2.2.4 Two kinds of constants, two kinds of names

Look at the names `SCALE` and `snap`. Both are constants, but one is written in capital letters and the other one is not. Programmers use constants for two different jobs, and the name shows the job:

- **Global configuration constants** like `SCALE` and `MARGIN` in the soccer field project sit at the top of the program, outside every function. They are settings for the whole program, meant to be found and tweaked easily. Their names are written in ALL_CAPS, with underscores between words, for example `MAX_SPEED`.
- **Ordinary constants** like `snap`, `markerSize`, or the `x` and `y` you'll compute in a moment live inside a function. They simply hold a value that doesn't change after it's set. Their names are written in camelCase: first word lowercase, every further word capitalized.

The computer doesn't care about these conventions, but people reading your code do. An ALL_CAPS name signals "program setting, tweak me here". A camelCase `const` signals "this value won't change". Make `const` your default choice. For the rare value that must change while the program runs, TypeScript has the keyword `let`, and you'll meet it soon.

2.2.5 The snapping formula

Now for the heart of the exercise. The marker may only sit on grid corners: 0, 40, 80, 120, 160, or 200. The mouse, however, is somewhere in between, say at `mouseX = 130`. Which corner is the nearest one?

You can find it in three steps, and each step is simple:

1. **Divide** the position by the grid size: $130 / 40 = 3.25$. Read that as "the mouse is 3.25 grid squares from the left edge".
2. **Round** to the nearest whole number: 3.25 becomes 3. Now you know the nearest corner is the third one.
3. **Multiply** back by the grid size: $3 \times 40 = 120$. That's the corner's position in pixels.

The p5.js function `round` does the rounding for you. `round(3.25)` returns 3, `round(3.6)` returns 4, and `round(3.5)` rounds up to 4, just like in math class. Put the three steps together and you get your first formula, written in one line and stored in a constant:

```
const x: number = round(mouseX / snap) * snap;  
const y: number = round(mouseY / snap) * snap;
```

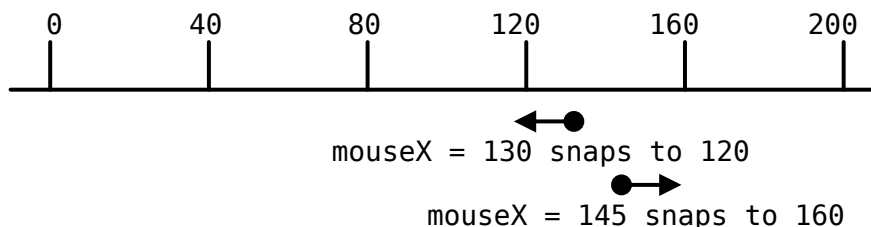


Figure 2.2.3: Two examples of the snapping formula on the x axis: 130 is 3.25 grid squares from the edge and snaps back to 120; 145 is 3.625 squares and snaps forward to 160.

Storing the result in a constant beats writing the formula into every parameter. You need the value several times, for both marker lines and for the text output, and the computer then calculates it only once. A name like `x` with the formula next to it also reads better than the same formula repeated four times. Note the names: `x` and `y` are camelCase constants inside a function, computed fresh on every mouse movement, and never changed after that.

2.2.6 Writing text on the canvas

The exercise has a second task: show the marker's current position, like 120/160, in the lower left corner of the canvas. Two new tools make that possible.

The first is the command `text`. It draws characters on the canvas, at a position you choose:

```
text("hello", 5, 195);
```

The first parameter is a string, and the two numbers are the position of the text on the canvas. Text is drawn in the current `fill` color, so set `fill` before drawing it, and use `noStroke()`, which switches the outline off, so the letters don't get one.

The second tool solves this puzzle. `x` and `y` are numbers, but `text` wants a string. A **template string** builds one from the other:

```
text(`${x}/${y}`, 5, height - 5);
```

A template string is written between backticks ```, not normal quotes. Inside it, `${...}` is a window: the computer inserts the current value of whatever is between the braces. If `x` is 120 and `y` is 160, the result is the string 120/160.

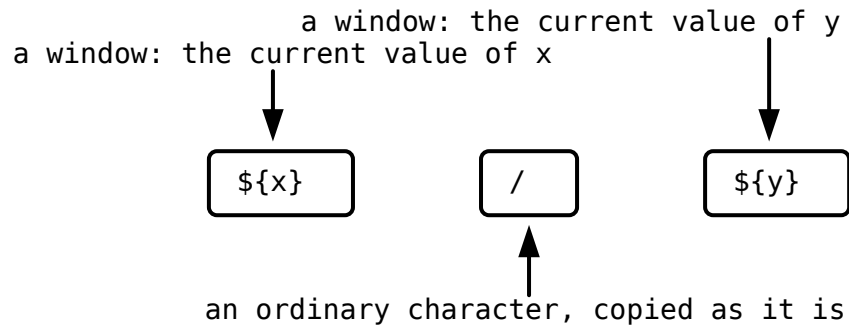


Figure 2.2.4: The template string piece by piece. Each window is replaced by a current value, and the character between them is copied as it is.

💡 Where is the backtick on your keyboard?

On an Austrian or German keyboard, the backtick ``` hides on the key right of the `ß`: press Shift together with that key, then press the space bar once. The extra space is needed because it's an accent key (*Akzenttaste*). The keyboard waits to see if you want to put the accent on a letter, and the space bar says “no letter, just the character itself”.

One detail in that statement deserves a close look: `height - 5`. Where `mouseX` and `mouseY` were your first system variables, `width` and `height` are your next two. They always hold the size of the canvas, as set by `createCanvas`. So `height - 5` means “5 pixels above the bottom edge”, no matter how big the canvas is. Remember these two variables, because they play the main role in the next chapter.

2.2.7 Seeing the exact mouse position

A practical problem shows up the moment you test a program with your paper notes in hand. You cannot put the mouse on the pixel (130, 45) by hand. The canvas has no ruler, one pixel is far smaller than the tip of the pointer, and nothing on the screen tells you where you actually are.

The fix is **debug output**: one temporary line that writes the numbers your program is working with onto the canvas, purely so that you can read them.

```
text(`${mouseX}/${mouseY}`, 5, 15);
```

With that line in `mouseMoved`, the live mouse position appears in the upper left corner and updates as you move. Hitting an exact position is easy now: move roughly to where you want to be, watch the numbers, and nudge the mouse until they read what your paper says. Then check the rest of the picture.

Watch out for one thing. Text is paint like everything else, so debug output that is never erased turns into a smeared blob after a few movements. In a function that repaints the canvas anyway, like `mouseMoved` in this program, the old numbers disappear with everything else. In a function that doesn't repaint, clear just the spot you write in: draw a small filled rectangle first and put the text on top of it.

```
fill("black");
rect(0, 0, 60, 20);
fill("white");
text(`${mouseX}/${mouseY}`, 5, 15);
```

Debug output costs a few lines and answers the question every programmer asks when a program misbehaves: which numbers does it actually have? Add it whenever you want to know, and delete it when you're done. The term is worth remembering, because you'll hear it for the rest of your programming life.

2.2.8 Your exercise: Snapping

Time to build the snapping marker. Work in this order:

- **Calculate by hand.** Five mouse positions on the 200 by 200 canvas are waiting in the table below. Run the formula on paper for each one: divide, round, multiply. The first row is filled in as a model. Watch the last row closely, because 2.5 rounds up, just like in math class.

Mouse position	Snap	x / Snap	y / Snap	Snapped position
(130, 45)	40	$(130 / 40) = 3.25$	$(45 / 40) = 1.125$	$(3 * 40, 1 * 40) \rightarrow (120, 40)$
(199, 60)	40			
(20, 180)	40			
(5, 199)	40			
(100, 100)	40			

- **Type the formulas** for `x` and `y` into `mouseMoved`, then draw the marker: a horizontal and a vertical line through `(x, y)`, each using `markerSize`, just like the crosshairs in the previous exercise.
- **Add the text output** in the lower left corner with a template string.
- **Test against your paper notes.** Add debug output for the mouse position (1), so the raw position sits in the upper left corner while your snapped position sits in the lower left. Steer to each of the five positions from your table and compare the snapped position with your prediction. Delete the debug line when everything matches.

The look of the marker is yours to choose; the sample uses white lines on the light blue background. The snapping behavior, however, must be exact: corners only, every 40 pixels.

💡 Exercise: Snap

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

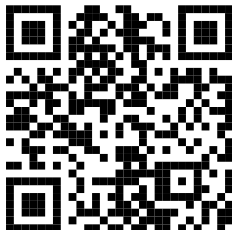
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0020-Variables/snap.yaml>

2.2.9 Check your understanding

When you've finished the exercise, take the short quiz below. You answer seven questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Snapping to a grid

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/vn1ouxszd8>

2.3 The responsive robot

This chapter has two stages. First you draw a robot the way you drew the house and the cat, with coordinates you choose yourself. Then you rebuild the same robot so that it adapts to *any* canvas size, and on the way you learn one of the big ideas of user interface programming: replace fixed numbers with formulas.

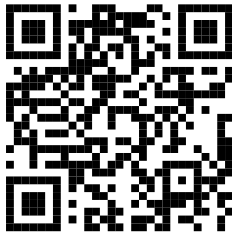
2.3.1 AI tutor

Replacing fixed numbers with formulas is the big idea of this chapter, and it takes a while to click. Describe your robot and the one coordinate you're unsure about to the AI tutor of this part.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/pl0qyaxsw4>

2.3.2 Stage 1: a robot with fixed numbers

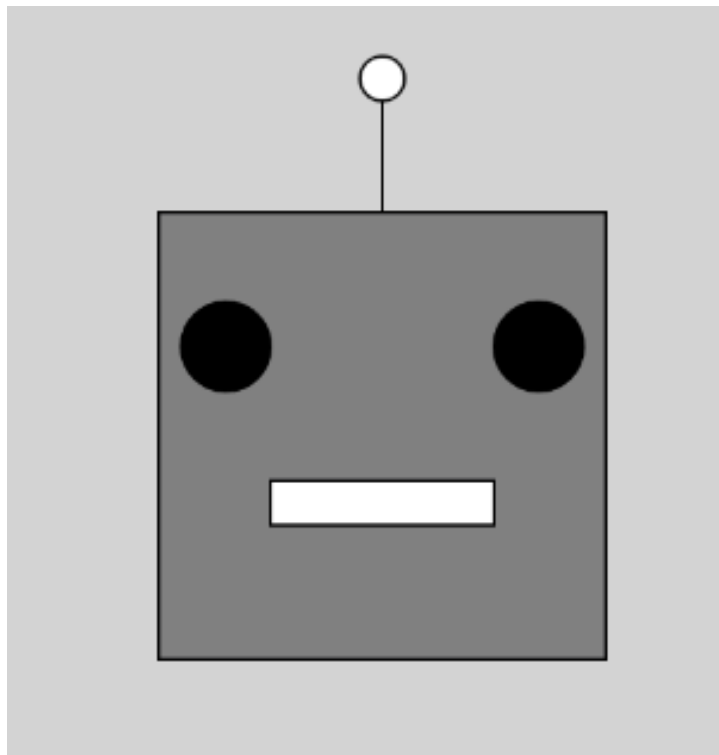


Figure 2.3.1: Stage 1: the robot you draw with coordinates you choose yourself, for one fixed canvas.

You build the robot from shapes you know well: a rectangle for the head, two filled circles for the eyes, a white rectangle for the mouth, and an antenna made of a line with a small circle on top. Your task is to match the shapes of the example picture; the colors are yours to choose.

Graph paper first for this one: sketch the robot, treat the sheet as a 400 by 400 canvas, and write the coordinates next to every shape. A drawing with this many parts is much faster to plan on paper than to guess at the keyboard. Then type your program in the playground, one shape at a time.

Keep your sketch and your finished code close by. The robot comes back in a few minutes, and your fixed coordinates become the “before” picture of this chapter.

💡 Exercise: Robot

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0010-Basics/robot.yaml>

2.3.3 What responsive design is

Programs run on very different screens: a phone held upright, a tablet held sideways, a wide monitor, a projector. A user interface that adapts itself to the available space is called **responsive**. You’ve seen responsive design many times without naming it. Open a news website and slowly drag the browser window narrower. The columns rearrange, pictures shrink, and at some point the menu folds itself into a small button. The page *responds* to the space it gets.

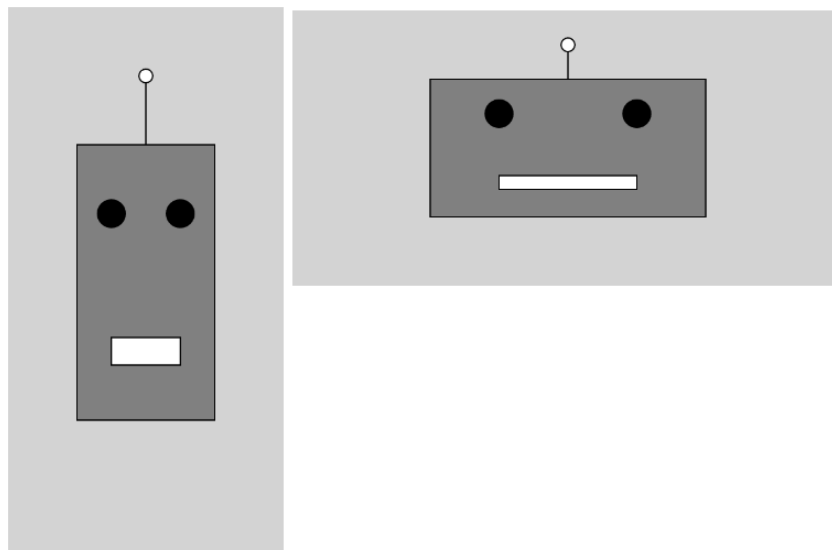


Figure 2.3.2: Stage 2: the same robot on a tall canvas and a wide one, fitting each neatly without a single number changed by hand.

Now look at your robot from stage 1 with that idea in mind. Change `createCanvas(400, 400)` to `createCanvas(200, 400)` and run the program. The canvas shrinks, but every shape stays exactly where its numbers put it: the robot sticks out or sits lost in a corner. Fixed numbers are only correct for the one canvas they were written for. A responsive robot needs something better.

2.3.4 From fixed numbers to formulas

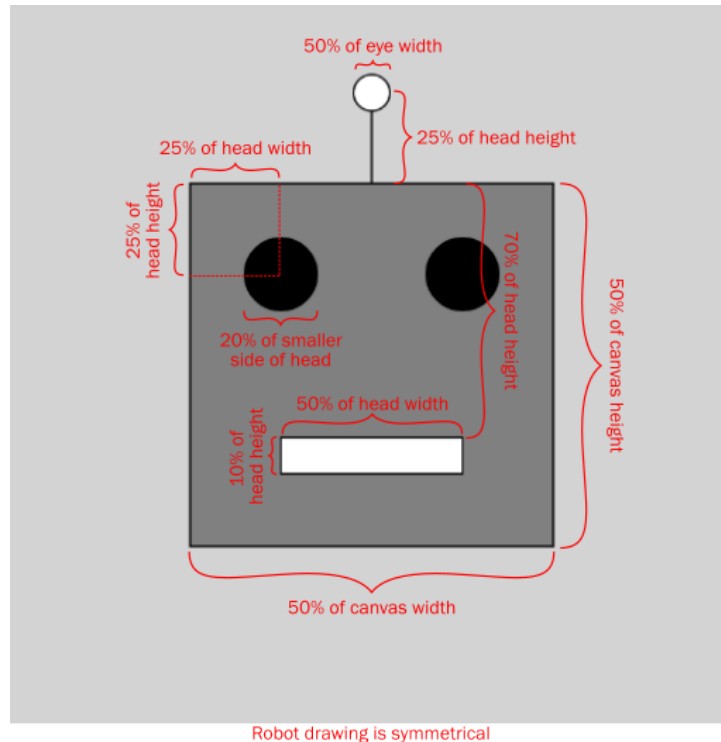


Figure 2.3.3: Every measurement given as a percentage of the canvas: the specification you build the robot from.

The trick is to stop describing the robot in pixels and start describing it in **relationships**: the head is half as wide as the canvas, whatever the canvas is. Percentages become multiplications, exactly as in math class: 50% of the width is `width * 0.5`, 20% is `width * 0.2`. And the values of `width` and `height` come from the two system variables you met in the snapping exercise. They always hold the current canvas size.

```
const headWidth: number = width * 0.5;    // 50% of the canvas width
const headHeight: number = height * 0.5;  // 50% of the canvas height
```

The head should also sit in the middle of the canvas. Centering has a little formula of its own, and you'll reuse it often. The middle of the canvas is `width / 2`, and the head must start half its own width to the left of that point.

```
const headX: number = width / 2 - headWidth / 2;
```

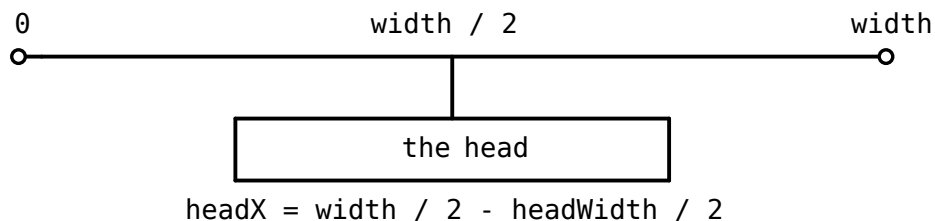


Figure 2.3.4: Centering the head: its left edge sits half a head width to the left of the canvas middle.

Check the formula with real numbers, on paper. On a 400 pixel wide canvas, `headWidth` is 200, and `headX` is $400 / 2 - 200 / 2 = 200 - 100 = 100$. Now make the canvas 200 pixels wide. `headWidth` becomes 100, and `headX` becomes $100 - 50 = 50$. Same formula, new numbers, head centered both times. That is the whole idea of the responsive robot: every coordinate and every size is a formula built from `width` and `height`, so every canvas computes its own robot.

One naming note: `headWidth`, `headHeight`, and `headX` are camelCase constants inside `setup`. They're computed once when the program starts and never change, and they aren't global settings, so `ALL_CAPS` would be the wrong signal.

2.3.5 The eyes and min

The specification says the eye diameter is 20% of the **smaller side** of the head. For that you need a way to pick the smaller of two values, and p5.js has one ready: `min(100, 200)` returns 100, and in general `min(a, b)` returns the smaller of the two. It's a helper function just like `round` from the snapping exercise. Instead of drawing anything, it calculates a value for you.

```
const eyeSize: number = min(headWidth, headHeight) * 0.2;
```

The smaller side is the safe choice. Picture a wide, flat head on a landscape canvas: 20% of its long side could be taller than the head itself, and the eyes would burst out of the face. Tying the eye size to the smaller side keeps the eyes inside the face, whichever way the head is stretched.

2.3.6 Didn't scale already solve this?

Fair question. In the soccer field project, one statement, `scale(SCALE)`, resized the whole drawing, and p5.js did all the multiplying. Now you're doing the multiplications by hand instead.

In practice, you *would* use `scale` whenever it fits the job. When a library offers a ready-made tool, professionals use it instead of rebuilding it; that's the whole point of libraries. The soccer field is a good example: the field must keep its exact proportions, so one scaling factor for everything is the right tool.

In this exercise, however, pretend for a moment that `scale` doesn't exist, for one simple reason: writing size formulas yourself is the skill you're here to practice. Responsive layouts are full of little rules like "center this", "20% of the smaller side", "half of the canvas", and those rules are formulas, whether a `scale` command exists or not. Look closely at the two example robots in the exercise, and you'll spot rules at work that a single scaling factor couldn't express. The tall robot and the wide robot have different proportions, yet the eyes stay round in both. A decision like that comes from the programmer who wrote the formula, not from a magic command.

2.3.7 Your exercise: Responsive robot

The exercise description contains a specification picture with every measurement of the robot as a percentage: head, eyes, mouth, and antenna. The robot is symmetrical, which saves you work.

1. **Read the starter code.** The head is already there, written with formulas. Walk through it line by line and make sure you can explain every formula; the head is the model for everything you'll add.
2. **Paper before keyboard.** Copy the table below and complete it. Take every formula from the percentages in the specification picture, then work out the numbers for both canvas sizes by hand. The four head rows are filled in for you, because the starter code already contains them. This table is your test plan.

Measurement	Formula	400 by 400	200 by 400
Head width	<code>width * 0.5</code>	200	100
Head height	<code>height * 0.5</code>	200	200
Head x	<code>width / 2 - headWidth / 2</code>	100	50
Head y	<code>height / 2 - headHeight / 2</code>	100	100
Eye diameter	<code>min(headWidth, headHeight) * 0.2</code>	40	
Left eye center x			
Eye center y			
Mouth width			
Mouth height			
Mouth y			
Antenna top y			

3. **Type the formulas.** One shape at a time, run after every step. Every constant gets an explicit data type, as always in this course.
4. **The final test.** Change `createCanvas(400, 400)` to `createCanvas(200, 400)`, then `createCanvas(400, 200)`, and run again. The robot must adapt every time, without you touching anything else. If a shape

sticks out or refuses to move, a fixed number is hiding in your code somewhere: find it and turn it into a formula.

Compare what the program draws with the numbers in your table. When they match for both canvas sizes, your robot is responsive.

Exercise: Responsive Robot

► [Open in the playground](#)



Scan the code, or type this address into your browser:

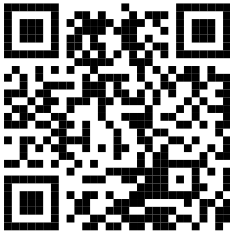
https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0020-Variables/responsive_robot.yaml

2.3.8 Check your understanding

When you've finished the exercise, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

Quiz: The responsive robot

► [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/i57c2wuo1w>

2.4 Mixing colors with numbers

Pick any color you can imagine: the deep orange of a sunset, the pale green of mint ice cream, the exact blue of your favorite hoodie. By the end of this chapter you can put every one of them on the canvas. There's no big program to build this time. This chapter is a series of short color experiments, and what you take away from it is a skill you'll use in every project from now on.

Colors deserve this attention. This is a *creative* coding course, and the colors you choose decide whether a drawing feels calm or loud, warm or cold, cheap or polished. So far you've picked colors from a small list of names. After this chapter you can mix any of the roughly 16.7 million colors your screen can show.

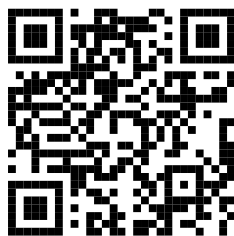
2.4.1 AI tutor

Colors come with a vocabulary of their own: channels, hue, saturation, brightness. Whenever one of those words stays fuzzy, ask the AI tutor of this part for another explanation and another example.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/pl0qyaxsw4>

2.4.2 A color name is a string

You've been using colors since your very first program. Every `background("lightblue")` and every `fill("red")` hands a **string** to p5.js, and the browser looks up the color behind the name. The browser knows about 150 such names, and some of them are wonderfully specific, like "papayawhip", "lemonchiffon", or "mediumspringgreen". You can browse the [complete list of color names](#) and try them right away:

```
function setup() {  
  createCanvas(200, 200);  
  background("lightblue");  
}
```

A color name works everywhere a color is expected: in **background**, in **fill**, and in **stroke**.

Names are comfortable, but they run out quickly. About 150 names stand against 16.7 million colors, and there's no name for "like "lightblue", just a little darker". To reach all the other colors, you describe a color with numbers instead of a name.

2.4.3 Mixing light: RGB

If you look at your screen through a strong magnifying glass, you'll see that every pixel is made of three tiny lights: one red, one green, one blue. Every color the screen shows is a mix of these three lights, each shining at its own strength. That's the **RGB color model**: you describe a color with three numbers, one for red, one for green, and one for blue.

Each of the three values runs from 0 (light off) to 255 (light at full power). The odd-looking maximum has a simple reason: the computer stores each value in one **byte** of memory, and a byte can hold exactly 256 different values, from 0 to 255. Three values with 256 steps each give $256 \times 256 \times 256$ combinations, the 16.7 million colors from the beginning of the chapter.

```
function setup() {  
  createCanvas(200, 200);
```

```
// Switch the color mode to RGB:
colorMode(RGB);

//      +----- red (0..255)
//      | +----- green (0..255)
//      | | +---- blue (0..255)
//      v v v
background(255, 0, 0);
}
```

The statement `colorMode(RGB)` tells p5.js how to interpret the three numbers that follow. RGB is the standard mode, so this line isn't strictly required, but writing it out makes your program easier to read: anyone looking at `background(255, 0, 0)` knows immediately which model the numbers belong to.

💡 Mixing light is not mixing paint

In art class, mixing all colors gives a muddy brown, and yellow comes straight from the tube. Light behaves differently: red and green light together make yellow, all three lights at full power make white, and all three switched off make black. Try `background(255, 255, 0)` and `background(255, 255, 255)` to see both effects. Your screen mixes light and your paint box mixes pigment, so keep the two rule sets apart.

One RGB pattern is worth knowing by heart. When all three values are equal, the result is a shade of gray, from (0, 0, 0) (black) over (128, 128, 128) (medium gray) to (255, 255, 255) (white). Gray is so common that p5.js offers a shortcut: a single number is read as a gray value. `background(0)` paints the canvas black, `background(255)` paints it white.

To find the three values for a color you have in mind, you use a **color picker**, a tool where you point at a color and read off its numbers. Try the [online color picker at colorpicker.me](https://colorpicker.me): it shows the RGB values of every color you click.

2.4.4 Hue, saturation, brightness: HSB

RGB is how the machine thinks. People describe colors differently: “a light, washed-out blue” names the kind of color, how strong it is, and how bright it is. Exactly these three questions form the **HSB color model**: Hue (German *Farbton*), Saturation (German *Sättigung*), and Brightness (German *Helligkeit*).

The hue is the kind of color, given as a position between 0 and 360 on a scale that runs through the whole rainbow:

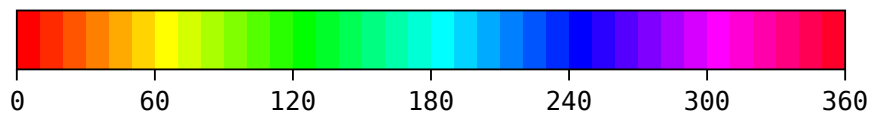


Figure 2.4.1: The hue scale from 0 to 360: every kind of color has its position, and red sits at both ends.

Notice that the scale starts and ends with red. It's really a circle that has been bent open: hue 0 and hue 360 are the same color, and after 360 the colors would simply come around again.

The other two values both run from 0 to 100. Saturation is the intensity of the color: 100 is the full color, and the closer you get to 0, the more washed out and gray it becomes. Brightness is how much light is in the color, so 100 is the brightest version, and 0 is always black, whatever the other two values say.

```
function setup() {
  createCanvas(200, 200);

  // Switch the color mode to HSB:
  colorMode(HSB);

  //      +----- hue (0..360)
  //      | +----- saturation (0..100)
  //      | | +---- brightness (0..100)
  //      v v v
  background(0, 100, 100);
}
```

Why learn a second model when RGB can already mix everything? Because HSB matches the way you *adjust* colors. Want the same color, just darker? Lower the brightness, one number. Want a set of colors that belong together, like a warm autumn palette? Keep saturation and brightness fixed and pick a few nearby hues. In RGB, each of these changes means juggling all three numbers at once. And HSB can do one more trick: a program can walk along the hue scale step by step and travel through the whole rainbow. In the next chapter, your program will do exactly that.

2.4.5 Your exercise: Color basics

This exercise is a laboratory session: you'll run small experiments and train your eye to predict what numbers turn into.

1. **Predict first.** Copy the table below on paper and fill in the middle column *before* you touch the keyboard. Name the color you expect in plain words, for example “bright green” or “dark gray”. The first row shows how much detail is enough.

RGB values	My prediction	What the screen shows
(0, 255, 0)	bright green	
(255, 255, 0)		
(0, 0, 0)		
(255, 255, 255)		
(128, 128, 128)		
(255, 0, 255)		

2. **Start with names.** Use `background` with color names from the [list of color names](#). Find the name you'd use for a sky, for grass, and for a sunset.
3. **Check your predictions.** Switch to RGB and run the six value triples from your table, one by one, filling in the last column as you go. The rows where the screen surprised you are the ones worth a second look.
4. **Mix a favorite.** Find a color you love with the [online color picker](#), read off its RGB values, and reproduce it on your canvas.
5. **Explore HSB.** Switch the mode with `colorMode(HSB)` and start at `background(0, 100, 100)`. First walk the hue in steps of 30 and watch the rainbow go by. Then pick your favorite hue, keep it fixed, and play with saturation and brightness until you understand what each of the two does.

💡 Exercise: Colors Basics

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

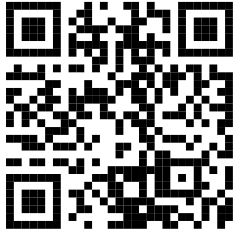
https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0020-Variables/colors_basics.yaml

2.4.6 Check your understanding

When you've finished the exercise, take the short quiz below. You answer seven questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Mixing colors

► Take the quiz on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/35v34cohhg>

2.5 The color worm: a variable of your own



Figure 2.5.1: The color worm: a trail of overlapping circles that follows every move of the mouse and slides through the whole rainbow; here it has written the word “fun”.

Drag your mouse across a black canvas and a glowing worm grows behind it: circle after circle, and every circle has a slightly different color than the one before. The colors wander through the entire rainbow and then start over, as long as you keep moving. To build the worm, you’ll create your first variable of your own, one whose value changes while the program runs.

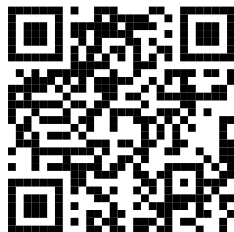
2.5.1 AI tutor

Your first variable of your own carries a big idea: a value that survives from one call of a function to the next. If that idea stays slippery, the AI tutor of this part is the place to ask about it.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/pl0qyaxsw4>

2.5.2 A value that must survive

Think back to the crosshairs: `p5.js` runs `mouseMoved` again and again, once for every movement of the mouse. Any constant you declare inside `mouseMoved` is created fresh in every run and forgotten the moment the run ends. For the crosshairs that was fine, because every run computed everything it needed from `mouseX` and `mouseY`.

The color worm needs something more. Its hue should move along the hue scale step by step: 0 in the first run of `mouseMoved`, then 5, then 10, and so on. Each run must know where the previous run stopped, and a `const`

can't provide that, because its value is set once and never changes, and inside a function it doesn't even survive to the next call. You need a box whose content can be replaced, and one that stays alive between calls.

2.5.3 `let` declares a variable

In the snapping chapter you read that TypeScript has a keyword for values that must change. Here `let` is in action:

```
let colorHue: number = 0;
```

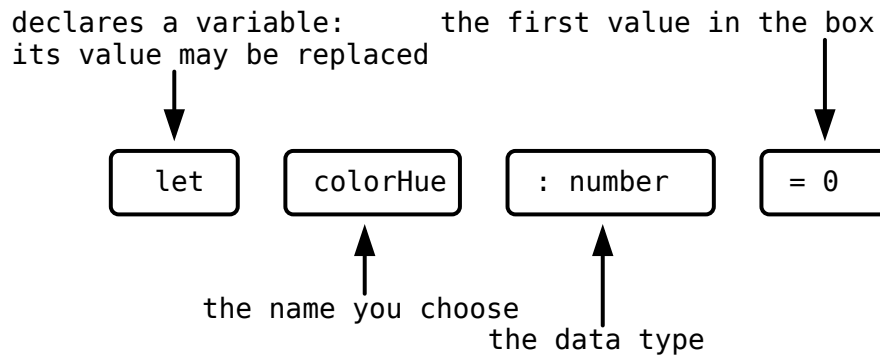


Figure 2.5.2: A `let` declaration piece by piece. As always in this course, the declaration names its data type.

The pattern is exactly the pattern of a `const` declaration; only the keyword changed, and with it the promise. `const` says “this value never changes”, and `let` says “this value may be replaced later”. A named place whose value can change while the program runs is a **variable**, and unlike `mouseX` and `mouseY`, which p5.js manages for you, this one is entirely yours.

i You may also meet `var`

Old JavaScript code declares variables with a third keyword, `var`. It comes from the language's early days, and this course doesn't use it: declare every value with `const` or `let`, never with `var`.

Replacing the value is called an **assignment**, and it's just the name, an equals sign, and the new value:

```
colorHue = 90;
```

No `let` here. The variable is declared exactly once, and every later assignment only uses its name. The assignment the worm needs looks stranger:

```
colorHue = colorHue + 5;
```

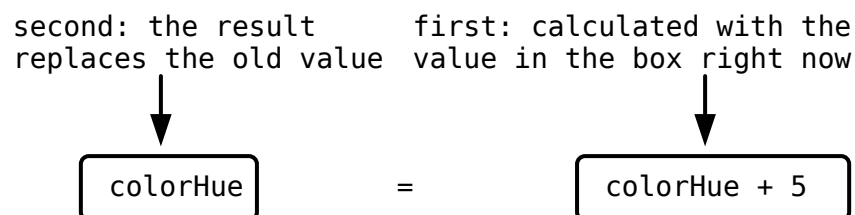


Figure 2.5.3: An assignment works in two steps: the right side is calculated first, then the result replaces the value in the box.

In math class, `colorHue = colorHue + 5` would be nonsense: no number equals itself plus five. But an assignment is not an equation; it's an order, carried out in the two steps of the picture. If `colorHue` holds 40, the right side gives 45, and 45 replaces the 40. Run the line again and 50 replaces the 45. That's the stepping-forward you were missing.

2.5.4 Global variables

The declaration must not go inside `mouseMoved`. There the variable would be newly created, set to 0, in every single run, and the hue would be stuck at the start of the scale forever. The declaration has to sit **outside of every function**, at the top of your program:


```

let colorHue: number = 0;

function setup() {
  // ...
}

function mouseMoved() {
  // ...
}

```

A variable declared outside of every function is called a **global variable**. It's created once, when the program starts. Every function in your program can read and change it, and, most importantly here, it keeps its value between function calls. When one run of `mouseMoved` sets `colorHue` to 45, the next run finds the 45 still in the box.

That power comes with a warning. In a large program, a global variable can be changed from anywhere, and finding out who changed it when becomes detective work. So professionals follow a clear rule, and you should too: prefer `const`, prefer local declarations inside functions, and reach for a global variable only when a value must survive between function calls. The color worm is exactly such a case.

2.5.5 Counting in circles: the % operator

One problem is left. `colorHue + 5` grows without end: 350, 355, 360, 365, and up and up, right past the end of the hue scale. What the worm wants instead is a counter that runs in circles, like the hue scale itself. After 355 it should return to 0 and start the rainbow again.

TypeScript has an operator for exactly that. The **% operator** (called **modulo**) gives the remainder of a division: $7 \% 3$ is 1, because 7 divided by 3 is 2 with a remainder of 1. Two properties make it right for the worm. As long as a value is smaller than 360, $\% 360$ changes nothing, so $355 \% 360$ is simply 355. But the moment the value reaches 360, the remainder wraps around to the start, and $360 \% 360$ is 0.

```
colorHue = (colorHue + 5) \% 360;
```

The parentheses matter, because they make sure the addition happens before the $\%$ is applied. Follow the line through the interesting spot: `colorHue` holds 355, the right side computes $355 + 5 = 360$, then $360 \% 360 = 0$, and 0 lands in the box. The next circle is red again, and the rainbow starts over.

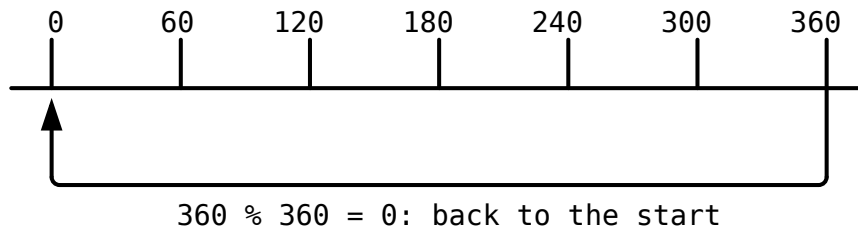


Figure 2.5.4: The hue counter runs in circles: it climbs the scale in steps of 5, and the $\%$ operator sends it from 360 back to 0.

2.5.6 Smears, this time on purpose

One last design decision, and it's an old friend. In the crosshairs program, `mouseMoved` started with a `background` statement to erase the previous drawing, and when you removed it, the canvas filled up with smears. The color worm is that experiment turned into art: there is no `background` call inside `mouseMoved` at all, so every circle stays where it was drawn, and the smears *are* the worm. The background is painted exactly once, in `setup`.

2.5.7 Your exercise: Color worm

💡 Playing computer

So far, the main job of pen and paper in this course was sketching your picture on graph paper before you code. The color worm gives them a second job, and it's just as important: **playing computer**. You step through the code by hand, write down the value of a variable after every step, and predict what the program will do before you run it. If the screen later matches your notes, you know you understood the code. If it doesn't, you've found a bug or a misunderstanding, and both are worth finding.

The hue counter is a good first case. On screen, the wrap from 355 back to 0 flashes by in a single mouse movement, far too fast to see. On paper, you can slow it down: put 355 into the formula and watch the modulo do its work.

Time to grow the worm:

1. **Play computer first.** Copy the table below on paper and complete it. The first four rows are the first four runs of `mouseMoved`, the last two rows jump ahead to the interesting spot, where the counter reaches the end of the hue scale. The first row is filled in as a model.

Run of <code>mouseMoved</code>	<code>colorHue</code> at the start	<code>colorHue + 5</code>	$(\text{colorHue} + 5) \% 360$
1	0	5	5
2			
3			
4			
71	350		
72	355		

The last row is the one you can't see on screen, because the wrap flashes by in a single mouse movement. Your table also answers a question the program never tells you: how many mouse movements does one full rainbow take when the hue grows in steps of 5?

2. **Set the stage in setup.** Create a 500 by 500 canvas, switch to HSB with `colorMode(HSB)`, and paint the background black with `background(0)` (a single number is a gray value, and 0 is black). Add `noStroke()` so the circles have no outlines and melt into one worm.
3. **Declare the global variable `colorHue`** outside of every function, starting at 0, with an explicit data type.
4. **Fill `mouseMoved`.** Set the fill color to `fill(colorHue, 100, 100)` (the current hue at full saturation and full brightness), draw a circle at the mouse position with a diameter of 40, and finish with the assignment that moves the hue forward and wraps it at 360.
5. **Test and play.** Move the mouse slowly, then quickly: fast movements leave gaps between the circles, because `mouseMoved` runs once per movement event, not once per pixel. Then check your paper prediction. Does the rainbow really repeat? To finish, change the step of 5 to 1 and then to 20, and watch what happens to the worm's stripes.

💡 Exercise: Color Worm

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

https://ts-web-playground-server-bfd6b3c7hubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0020-Variables/color_worm.yaml

2.5.8 Check your understanding

When you've finished the exercise, take the short quiz below. You answer seven questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: The color worm

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/2bwce6zilh>

2.6 Build your own color picker



Figure 2.6.1: The finished color picker: the top half previews colors while the mouse moves, the bottom half keeps the color chosen with the last click, and the corner lists its HSB values.

In the chapter about mixing colors you used an online color picker, where you point at a color and read off its numbers. By now you know enough to build one yourself. The project brings together almost everything from this part: mouse events, the HSB color model, the system variables `width` and `height`, constants with explicit types, and text output with a template string. And it hides a small surprise, because the program remembers your chosen color without a single variable of your own.

2.6.1 AI tutor

The color picker brings together almost everything from this part, which also means many places where it can go quiet or show the wrong color. The AI tutor of this part helps you narrow down which half of the program is at fault.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/pl0qyaxsw4>

2.6.2 Two halves, two jobs

The picker splits its canvas into two halves with different jobs. The **top half** is the preview: it changes its color on every mouse movement. Moving the mouse to the right walks the hue along the rainbow, moving it down increases the saturation. The **bottom half** is the result: when the user clicks, it takes on the current color and keeps it until the next click. Next to it, in the lower left corner, the program writes the HSB values of the chosen color, so the user can copy them into another program.

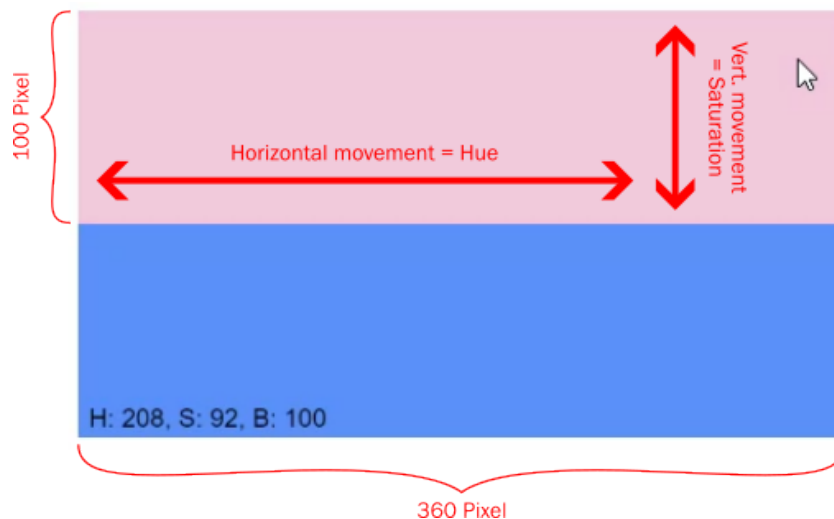


Figure 2.6.2: The layout of the color picker: horizontal movement controls the hue, vertical movement the saturation, and the canvas measures 360 by 200 pixels.

2.6.3 A canvas size with a purpose

Look at the measurements in the layout picture. The canvas is 360 pixels wide, and the hue scale runs from 0 to 360. The top half is 100 pixels tall, and saturation runs from 0 to 100. That is no coincidence. It's the whole trick of this program. The mouse position doesn't have to be converted at all, because it *is* the color. `mouseX` is the hue, `mouseY` is the saturation, and the brightness simply stays at 100:

```
function mouseMoved() {
  fill(mouseX, mouseY, 100);
  rect(0, 0, width, height / 2);
}
```

The `rect` statement covers exactly the top half: its corner is (0, 0), its width is the canvas width, and its height is half the canvas height. Notice what is *not* here. There is no `background` call. The rectangle repaints the whole preview by itself, and the bottom half must not be erased anyway, because it holds the chosen color.

2.6.4 A second mouse event: `mouseClicked`

The mouse can do more than move. When you press and release a mouse button, that's a **click**, and p5.js treats it as an event of its own. Write a function called `mouseClicked`, and p5.js calls it automatically once per click:

```
function mouseClicked() {
  // runs once for every click
}
```

Everything you know about `mouseMoved` applies here too: the name is an agreement between you and p5.js, so a typo like `mouseclicked` gives you a function that is never called, without any error message. Inside `mouseClicked`, the system variables `mouseX` and `mouseY` hold the position where the click happened.

2.6.5 Freezing the chosen color

When the click arrives, the program reads the color from the mouse position and stores the two interesting values in constants:

```
const colorHue: number = mouseX;
const colorSaturation: number = min(100, mouseY);
```

The second line needs a moment of thought. The user may click anywhere on the canvas, including the bottom half, and there `mouseY` runs up to 200. A saturation of 200 doesn't exist; the scale ends at 100. The function `min` from the responsive robot chapter solves that: `min(a, b)` returns the smaller of the two values, so `min(100, mouseY)` caps the saturation at 100. In the top half, where `mouseY` is at most 100, `min` changes nothing. Used like this, `min` is a guard that keeps a value inside its allowed range.

With the two constants in hand, the rest of `mouseClicked` is drawing you already know. Set `fill(colorHue,`

`colorSaturation, 100)`, cover the bottom half with a `rect`, then switch the fill to black and write the values with a template string:

```
text(`H: ${colorHue}, S: ${colorSaturation}, B: 100`, 5, height - 5);
```

Mind the order: the text must be drawn *after* the rectangle, or the freshly painted bottom half covers it up.

2.6.6 A program that remembers without a variable

One thing about the color picker is worth noticing before you build it. The program remembers the chosen color between events, across hundreds of `mouseMoved` calls, and yet it declares no global variable. The memory is the paint itself: nothing ever repaints the bottom half except the next click, so the color simply stays on the canvas. That works here because the program only has to remember one thing, and that thing is literally a color. In the next chapter, a program has to remember a growing list of values, and there paint alone won't be enough.

2.6.7 Your exercise: Color picker

Build the picker in this order:

1. **Set the stage in `setup`.** Create the 360 by 200 canvas, switch to HSB with `colorMode(HSB)`, and add `noStroke()` so the rectangles have no outlines. Paint the whole canvas with a starting color; the sample uses `background(0, 100, 100)`, a fully saturated red.
2. **Fill `mouseMoved`.** Set the fill color from the mouse position and cover the top half with a rectangle.
3. **Fill `mouseClicked`.** Declare the two constants with explicit data types, including the `min` guard, paint the bottom half in the chosen color, and write the HSB values in black in the lower left corner. The text is a format promise: `H: 208, S: 92, B: 100`, with exactly these letters, colons, and commas.
4. **Test the tool.** Move the mouse along the top edge and watch the preview walk through the rainbow, then move it downward and watch the colors fade toward white. Click in the top half and check the bottom half and the text. Finally, click deep in the bottom half: the saturation in the text must never climb above 100, because the `min` guard caps it.

💡 Exercise: Color Picker

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

https://ts-web-playground-server-bfd6bjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0020-Variables/color_picker.yaml

2.6.8 Check your understanding

When you've finished the exercise, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Color picker

► Take the quiz on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/ivqin5iav0>

2.7 Strings that grow: the click protocol

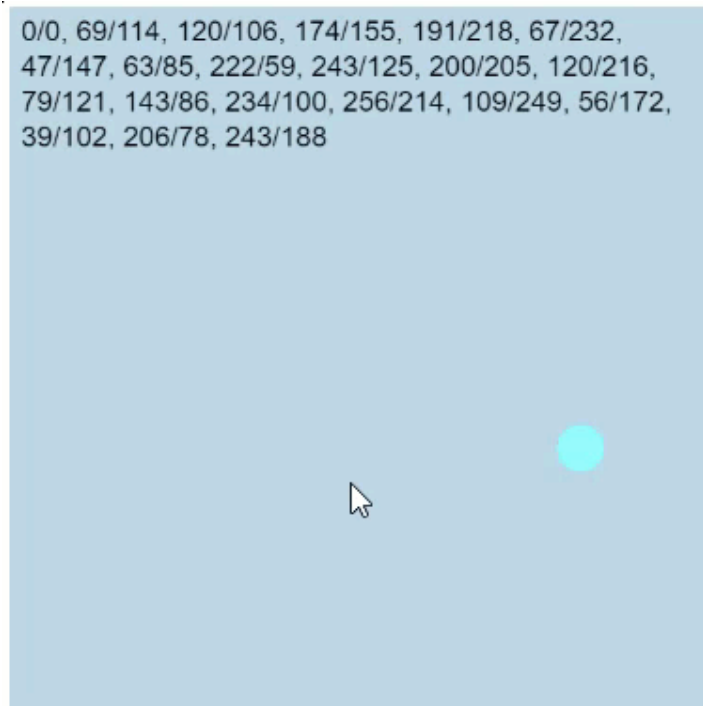


Figure 2.7.1: The click protocol after many clicks: every position the user ever clicked is listed at the top, wrapped over several lines, and a circle marks the newest click.

The color worm counted with numbers; now you'll make text grow. You'll build a program that keeps a written record of everything the user does: each click draws a circle at the click position and adds that position to a protocol at the top of the canvas. Click by click, the protocol gets longer, until it fills line after line, and the program never forgets a single entry.

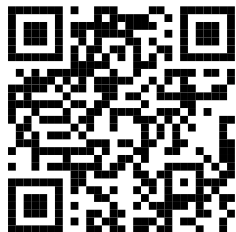
2.7.1 AI tutor

A growing string goes wrong in small ways: a missing space, a protocol that resets, a line that never gets longer. The AI tutor of this part helps you work out which of them you're looking at.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/pl0qyaxsw4>

2.7.2 An event for clicks

The click protocol reacts to clicks with the event function `mouseClicked`, the same one that froze the color in the color picker: `p5.js` calls it automatically once per click, and inside it, `mouseX` and `mouseY` hold the position where the click happened.

```
function mouseClicked() {  
  circle(mouseX, mouseY, 20);  
}
```


As with every event function, the name is an agreement between you and p5.js: a typo like `mouseclicked` gives you a function that is never called, without any error message.

2.7.3 Chaining strings together

Sticking two strings together into one longer string is called **concatenation**, from the Latin word *catena*, a chain. You've already built strings from pieces: in the snapping exercise, the template string ``${x}/${y}`` chained two numbers and a slash into a string like 120/80.

The new idea of the click protocol is that the *old value of a variable* can be one of the pieces. The protocol lives in a global variable, and every click builds a longer string out of it:

```
let clickPositions: string = "0/0";

function mouseClicked() {
  clickPositions = `${clickPositions}, ${mouseX}/${mouseY}`;
  // ...
}
```

Look closely at the assignment. It's the exact same pattern as `colorHue = (colorHue + 5) % 360` in the color worm, just with text instead of a number. The computer first builds the string on the right side, starting with whatever `clickPositions` holds right now, and then stores the result back into the variable. Say the protocol is 0/0 and the user clicks at (120, 80): the right side becomes 0/0, 120/80, and that's the new content of the box. The next click at (30, 190) turns it into 0/0, 120/80, 30/190. The string grows with every click.

And like the worm's hue, the protocol must survive between event calls, so `clickPositions` has to be a global variable, declared with `let` outside of every function. Its data type is `string`, because a variable isn't limited to numbers. It can hold a value of any type.

2.7.4 A text box that wraps

A growing protocol soon becomes wider than the canvas, and `text` with three parameters would happily keep writing past the right edge. For long text, `text` accepts five parameters:

```
text(clickPositions, 5, 5, width - 10, height - 10);
```

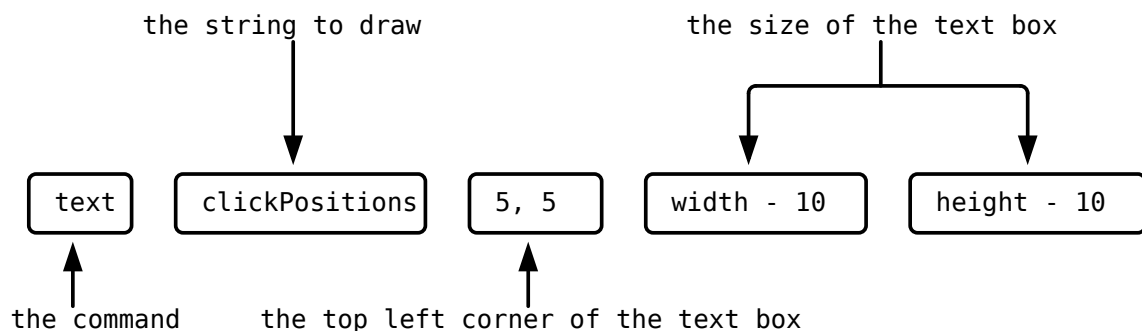


Figure 2.7.2: The five parameters of `text`: the string to draw, the corner where the text box starts, and the size of the box.

The two extra numbers turn the position into an invisible **text box**, and inside the box, p5.js wraps the text for you: when a line reaches the box's right edge, the text continues in the next line, just like in a word processor. In the call above, the box starts at (5, 5) and is 10 pixels smaller than the canvas in each direction, which leaves a 5 pixel margin on every side. Text is drawn in the current `fill` color, so pick one that reads well on your background.

2.7.5 The variable remembers, the canvas forgets

There's one decision in the click protocol program worth a moment of thought. Each run of `mouseClicked` starts by repainting the background, exactly like the crosshairs did, so the circle of the previous click is erased and only the newest circle is visible. And yet the protocol at the top still lists *every* click since the program started.

The two behaviors don't contradict each other. Whatever you draw on the canvas is just paint: repaint the background and it's gone. The protocol survives because it isn't stored on the canvas at all. It's stored in the variable `clickPositions`, and it's drawn onto the canvas again, freshly and completely, after every click. Drawing something is not the same as remembering it. Variables remember, and the canvas only shows.

2.7.6 Your exercise: String concat

Build the click protocol:

1. **Declare the global variable `clickPositions`** with the explicit type `string` and the starting value `"0/0"`.
2. **Simulate the first click in setup.** The starter code already creates the canvas and paints the background. Add what a click at (0, 0) would leave behind: a circle with diameter 20 at (0, 0), of which only a quarter is visible in the corner, and the protocol text. The circle and text colors are yours to choose; in the sample the circle is "aqua" and the text is "black".
3. **Fill `mouseClicked`.** Repaint the background, draw the circle at the click position, extend `clickPositions` with a template string, and draw the protocol into a wrapping text box. Mind the order: extend the protocol *before* drawing it, or the newest click will always be missing from the display.
4. **Test the protocol.** Click a few positions and read the text on screen character by character: every position separated by a comma and a space, x and y separated by a slash, and the newest click at the end. The protocol is a format promise, and exactness decides whether your output matches it. Then keep clicking until the protocol wraps into a second and third line.

💡 Exercise: String Concat

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

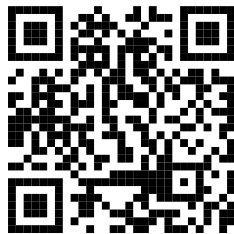
https://ts-web-playground-server-bfd6bjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0020-Variables/string_concat.yaml

2.7.7 Check your understanding

When you've finished the exercise, take the short quiz below. You answer five questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: The click protocol

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/iog30ofmq5>

2.8 Practice projects: circle tool and rect tool

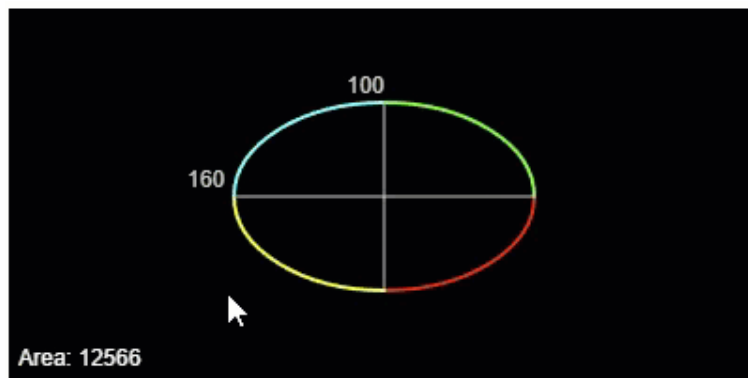


Figure 2.8.1: The finished circle tool: colored quadrants, helper lines, both diameters as labels, and the area in the corner.

This part closes the way the last one did, with practice projects written as job briefings. Imagine you are part of a team building a graphical design application, and your job is to implement two drawing tools, one for ellipses and one for rectangles. Between them, the two tools use almost everything from this part: mouse events, the system variables `width` and `height`, constants with explicit types, the snapping formula, template strings, and text output.

Take these projects seriously. They are practice for exactly the kind of task you'll get in an assessment. Plan on paper, work in small steps, and hand in formatted code, just as you've learned.

2.8.1 AI tutor

Both drawing tools are practice for an assessment, so work on them on your own first and turn to the AI tutor of this part when you're really stuck. It gives you the next hint, never the finished tool.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:
<https://app.novedu.at/pl0qyaxsw4>

2.8.2 The briefing: a circle tool

The circle tool lets the user shape an ellipse with the mouse. The basic version has three requirements:

1. **A centered ellipse.** While the user moves the mouse, an ellipse appears, centered on the canvas. The centering must be responsive: if someone changes the parameters of `createCanvas`, the ellipse stays centered without any other change to the code.
2. **The mouse controls the size.** The width of the ellipse is the distance between `mouseX` and the canvas center, multiplied by two. The height works the same way with `mouseY`. So the further the mouse moves away from the center, the larger the ellipse grows.
3. **The area in the corner.** The lower left corner shows the area of the ellipse in the format **Area: 123**, as a whole number. The area of an ellipse is the horizontal radius times the vertical radius times π .

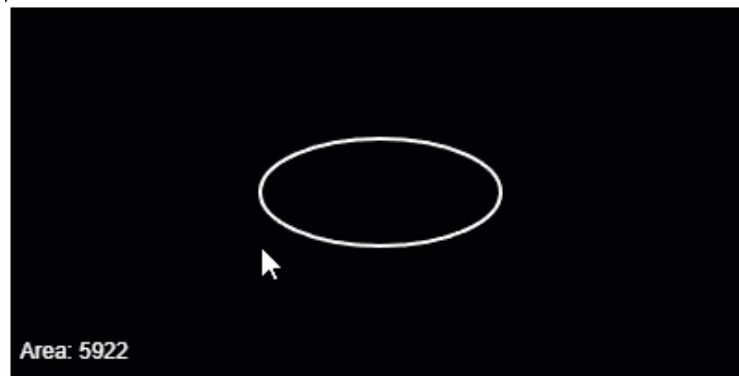


Figure 2.8.2: The basic circle tool: a white ellipse follows the mouse, and the corner shows its area.

2.8.3 Three helpers you haven't met

Three small pieces are new; everything else in the briefing is knowledge you already have.

ellipse draws stretched circles. The command `circle(x, y, d)` can only draw perfect circles, because it has a single diameter. Its bigger sibling `ellipse(x, y, w, h)` takes the center position plus a width and a height of their own, and that is exactly what the tool needs.

abs removes the minus sign. Requirement 2 talks about the *distance* between `mouseX` and the center. The obvious formula, `mouseX - width / 2`, gives a negative number whenever the mouse is left of the center, and a width of -140 makes no sense. The helper function `abs` returns the absolute value of a number: `abs(-70)` is 70, and `abs(70)` is also 70. It belongs to the same family of calculating functions as `round` and `min`. It draws nothing and only computes a value for you.

PI is π , ready to use. For the area you need π , and p5.js provides it as the constant `PI`, holding 3.14159 and so on. You write it without parentheses, because it's a value, not a function. Since π has endless decimal places, the computed area has them too; wrap the calculation in `round` so the corner shows **Area: 12566** and not a number with a long tail.

2.8.4 The polished circle tool

When the basic tool works, the briefing has four upgrades. Each one is a separate step; finish and test one before starting the next.

1. **Helper lines.** Draw a horizontal and a vertical line through the center of the ellipse, dividing it into four equal quadrants.
2. **Snap to grid.** The two diameters must snap to a grid of 20 pixels, so the width and height are always multiples of 20. That is the snapping formula from the snapping chapter, applied to a size instead of a position: divide by 20, round, multiply by 20.
3. **Diameter labels.** Show the horizontal diameter as a number directly left of the ellipse's center line, and the vertical diameter above the ellipse's top center. One tip for the horizontal label: normally a text *starts* at the position you give; after `textAlign(RIGHT)`, it *ends* there, which is exactly right for a label that should sit left of a point.
4. **Colored quadrants.** Draw the outline of each quadrant in its own color. A quadrant's outline is an arc of 90 degrees, and `arc` together with `angleMode(DEGREES)` is smiley knowledge; the starter code already switches the angle mode for you. Remember `noFill` and `strokeWeight` from the Olympic rings for outline-only drawing.

2.8.5 How to work, and when to peek

Work the way you've practiced all through this part:

1. **Paper first.** The canvas is 400 by 200, so its center is (200, 100). Complete the table below by hand before you write a single line of code. Each column header carries the formula, so you only fill in numbers. The last column belongs to the polished tool; the others to the basic one.

Mouse (x, y)	Width: 2 * <code>abs(x - 200)</code>	Height: 2 * <code>abs(y - 100)</code>	Area, rounded	Both diameters snapped to 20
(130, 45)	140	110	12095	140 and 120
(330, 180)				
(200, 100)				
(400, 0)				

The third row is a deliberate trap: the mouse sits exactly on the center. Work out what the tool draws there, and check later that your program does the same.

2. **Build in requirement order.** Centered ellipse first, then the size formulas, then the area text. Run after every step. Remember the crosshairs lesson: `mouseMoved` has to start by repainting the black background, or the old ellipses smear across the canvas.
3. **Test against your table.** Add debug output for the mouse position (1), steer to each position from your table, and compare the area in the corner with your prediction.
4. **Peek with a rule.** The playground shows a sample solution for this exercise. Use it when you've been stuck for more than a few minutes, but use it like this: read only until you understand the idea you were missing, close it, and write the code from memory. Copying with the solution open next to you feels faster and teaches nothing.

💡 Exercise: Circle Tool

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

https://ts-web-playground-server-bfd6bjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0020-Variables/circle_tool.yaml

2.8.6 The transfer: a rect tool

The second briefing comes from the same team. The users also want a rectangle tool. The rules match the circle tool almost line for line. The rectangle stays centered on the canvas, responsive as always. The mouse controls its width and height exactly as before, distance to the center times two. The lower left corner shows **Area: 123**, and a rectangle makes that easier: width times height, no π needed.

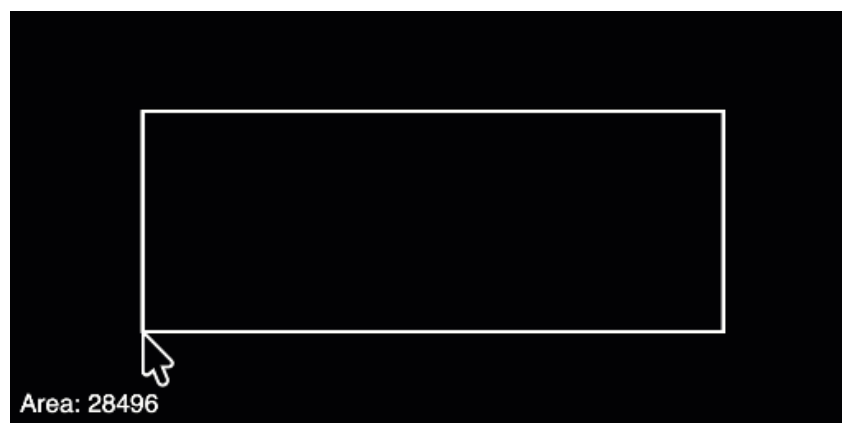


Figure 2.8.3: The basic rect tool: a centered rectangle sized by the mouse, with its area in the corner.

One difference deserves attention before you start. `ellipse` takes the position of its *center*, but `rect` takes the position of its *top left corner*. To center the rectangle you need the centering formula from the responsive robot: the corner sits half the rectangle's width left of the canvas middle, and half its height above it.

The polished rect tool mirrors the circle tool's upgrades: the two diagonals of the rectangle instead of the helper

lines, the same snapping to a 20 pixel grid, the same two dimension labels, and, instead of colored quadrants, each side of the rectangle drawn as a line in its own color.

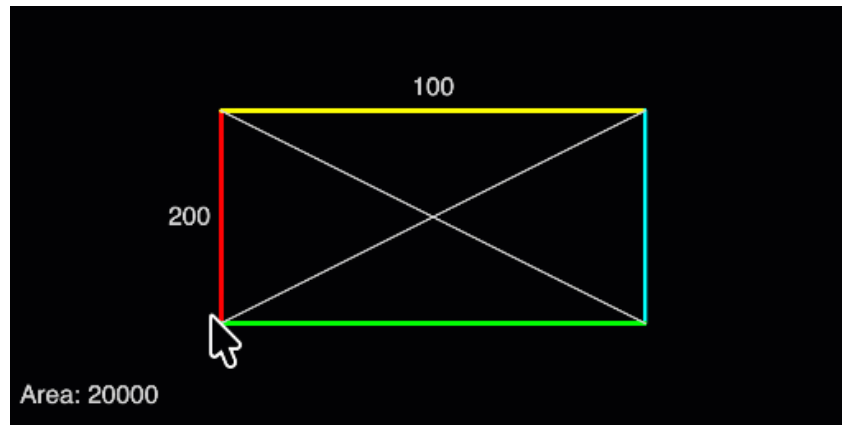


Figure 2.8.4: The finished rect tool: colored sides, diagonals, dimension labels, and the area.

The real challenge of this second project is that you have already solved every sub-problem once. So try to build the rect tool with as few looks at the sample solution as possible, ideally none at all. When you get stuck, don't open the solution first; open your own circle tool code, find the matching piece, and ask yourself what has to change for a rectangle.

💡 Exercise: Rect Tool

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

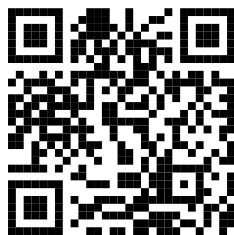
https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0020-Variables/rect_tool.yaml

2.8.7 Check your understanding

When you've finished both projects, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Circle tool and rect tool

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/ru7s99pf3u>

2.9 Back to the reference: a keyboard program of your own

Every program you have written in this course waited for the mouse. The keyboard was sitting there the whole time, and p5.js can hear it just as well. Nobody is going to teach you how, though. At the end of part 1 you learned a p5.js command straight from the official reference, without the book explaining it first, and you can do that again. This time it is not one command but a whole family of them.

This chapter gives you two tasks. First you research the keyboard commands of p5.js, invent a small program of your own that combines at least two of them, and present it in class. Then your own program raises a second question. It gives you a character and a number for the very same key press, and you will want to know which number belongs to which character, and who decided that. The quiz at the end of the chapter checks what you found out.

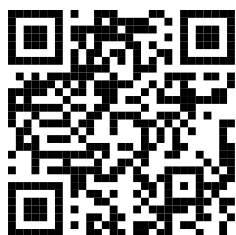
2.9.1 AI tutor

Two AIs appear in this chapter, with different jobs. The AI tutor of this part answers your questions about the course material, exactly as in every other chapter of the Variables part. Both tasks in this chapter send you to a general AI on the internet instead, because working through material that nobody prepared for you is what this chapter trains.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/pl0qyaxsw4>

2.9.2 The keyboard section of the p5.js reference

The [p5.js reference](#) sorts its commands into groups, and the keyboard commands live in the group *Events*, in the section *Keyboard*. That section is small enough to read completely, and it holds three kinds of thing:

- **Three event functions**, `keyPressed`, `keyReleased`, and `keyTyped`. They work like `mouseClicked`, which you already know: you write the function, and p5.js calls it when the matching thing happens.
- **Four system variables**, `key`, `keyCode`, `code`, and `keyIsPressed`. They work like `mouseX` and `mouseY`. You read them; you never set them.
- **One helper function**, `keyIsDown`.

That list is all you get from this book. What each one does, when p5.js calls it, and what exactly it hands you is your research.

Two warnings before you start. The examples on the reference pages are written in JavaScript on a canvas of 100 by 100 pixels, so read them and rebuild the idea in your own structure instead of copying them, exactly as the documentation chapter at the end of part 1 explained. And some sentences in the p5.js reference are left over from older versions of p5.js. If a page says one thing and your running program shows another, your program is right.

2.9.3 Two limits and one practical tip

Two things you have not learned yet shape what you can build, and knowing them before you start saves you an hour of frustration.

You have no repeating drawing loop yet. Your programs run `setup` once and then wait for events, exactly like the click protocol does. That fits `keyPressed`, `keyReleased`, and `keyTyped` perfectly, because p5.js calls them by itself when something happens. Two items in the keyboard section want a loop that runs again and again instead, so read what they do and then leave them for a later part of the course.

You have no conditions yet. A condition is a statement that lets a program react *differently* depending on what happened, and the next part of this course is about nothing else. So the simplest plan is a program that reacts the *same way* to every key and lets the values do the varying. The character, the number, and the code of a key press are different for every key, and you can draw them or calculate with them without deciding anything.

And one practical tip. Click on the canvas once before you type. The drawing area only receives key presses when it has the keyboard focus, and a keyboard program that looks completely dead is usually just not listening yet.

2.9.4 Your exercise: learn it, build it, present it

Your task has three parts:

1. **Learn the keyboard section from the reference.** Read the pages, run the examples that sit on them, and experiment the way you learned with `arc`: change one thing at a time, run, and watch what happens. Keep going until you can predict the result before you press run.
2. **Invent your own program and build it.** The rules are short. At least two of the keyboard commands have to work together, something clearly visible has to happen on the canvas, and the program is yours to think up. Sketch it on paper first, as always, then write it in the playground: TypeScript, `createCanvas` first, variables with explicit types, formatted before you hand it in.
3. **Present it in class.** About five minutes. Show the program running, say which keyboard commands you combined and what each one does, and tell the class one thing that surprised you while you were experimenting. Expect live requests with everybody watching, like “make it react to the space bar too” or “show the number instead of the letter”.

Everyone prepares a presentation, and your teacher decides who gives it. Your teacher will also tell you how to hand in your program.

2.9.5 Ideas, if you need one

Inventing the idea is part of the exercise, and a small program that nobody else thought of is worth more than a perfect copy of somebody else’s. If you are staring at an empty canvas, here are some starting points.

Press and release. Hold down any key and the canvas turns yellow. Let go and it is white again. Two keyboard commands, four lines, and it works:

```
function keyPressed() {  
  background("yellow");  
}  
  
function keyReleased() {  
  background("white");  
}
```

That is an honest, complete answer to this exercise, and it shows how small the program is allowed to be. Start there and then make it yours. Draw a shape instead of a plain color, or pick colors that go well together, or let a third keyboard command add something on top.

A key inspector. There is a website called keycode.info that does exactly one thing. You press a key, and it shows you every piece of information the browser has about that key press. Build your own version. p5.js hands you a character and two different codes for every key, so draw them on the canvas in big letters and watch what changes when you press the A key, the space bar, an arrow key, or shift.

A typewriter. `keyTyped` gives you the character that was typed. Add it to a growing string, exactly the way the click protocol grew with every click, and draw the whole text into a wrapping text box. A second keyboard command can show something extra, for example how many keys have been pressed so far.

A color for every key. The number of a key is just a number, so you can calculate with it. The color worm turned a growing number into a hue with %, and the same trick turns a key number into a color. Every key on your keyboard gets one of its own, and no condition is needed anywhere.

Whatever you pick, at least two keyboard commands have to work together. That is where the interesting behavior comes from.

2.9.6 AI for your program: allowed, understanding required

You may use an AI for every part of this program, and the rules are the ones from the research exercise at the end of part 1. Use a general AI on the internet, for example ChatGPT, Google Gemini, or Claude, and not the tutor of this part, because meeting answers that nobody tailored to this course is part of the research. Paste your own working code into the prompt, so the AI answers in the language, the library, and the structure your playground expects. And say that you are a beginner whose code shows everything they have learned.

One addition for this chapter. An AI will hand you conditions and a **draw** loop without being asked, because that is how it would write a keyboard program for somebody who knows them. Write in your prompt that you have not learned conditions or the **draw** loop yet. And the old test still decides what stays: a line you cannot explain in front of the class gets replaced, or you ask follow-up questions until you can explain it.

2.9.7 The character and the number

Once your keyboard program runs, you will notice something odd. For one single key press, p5.js gives you two different answers. One of them is a character, a string like "a". The other one is a number.

Press the A key, then B, then C, and watch that number. It goes up by one each time, in the order of the alphabet. That ordering was not invented for p5.js, and not for the web either. It is much older, and it comes from an agreement about which number stands for which character that people made in the 1960s. The agreement is called ASCII, and researching it is the second task of this chapter.

Before you start, keep one thing apart in your head. The number that p5.js gives you for a key press identifies a **key on your keyboard**, not a character. That is why the arrow keys have numbers too, although they write nothing at all. For the letters and the digits the two happen to agree, and that is exactly the puzzle worth looking into.

2.9.8 Bits, bytes, and the notation you will meet

Your research will use three pieces of vocabulary from the first minute, so here they are before you start.

A **bit** is the smallest thing a computer can store, a single switch that is either 0 or 1. Eight bits together are one **byte**, and one byte can hold 256 different values, from 0 to 255. You have already used that fact. In the color chapter, each of the red, green, and blue values sits in one byte, which is why they run from 0 to 255 and stop there.

You will also meet a notation that looks like U+00E4 or 0xC3. Those are numbers written in a number system called hexadecimal, and part 3 of this course teaches you to read and calculate with it. For this chapter you do not need any of that. Treat U+00E4 as the name of a character and move on. If your AI starts converting numbers into binary and asks you to follow along bit by bit, stop it and say that you only want the idea, not the arithmetic.

The third piece of vocabulary is a distinction that almost everybody gets wrong at first. The **number** of a character and the **picture** of a character are two different things. The number is what gets stored and sent. The picture comes from a font, and a font can be missing a character even when the number arrived perfectly. This book is a good example. The printed PDF you may be holding cannot show a single emoji, because the font used for printing has no picture for them, and nothing at all is wrong with the encoding.

2.9.9 Research: ASCII, Unicode, and UTF-8

Research these three questions with a general AI on the internet, in this order, because each one builds on the one before it:

1. **What is ASCII?** What is it, why was it necessary, and where does it stop? You should be able to name characters from everyday German writing that ASCII simply does not contain.
2. **What is Unicode?** What is it, and why is it needed when ASCII already exists? Pay attention to what exactly Unicode assigns to a character, and to what Unicode deliberately says nothing about.
3. **What is UTF-8?** What is it, and how does it relate to ASCII and to Unicode? Collect four examples: a character that UTF-8 stores in one byte, one that takes two bytes, one that takes three bytes, and one that takes four bytes.

ASCII, Unicode, and UTF-8 are three different kinds of thing. Understanding what kind of thing each one is matters more than any single number you can memorize.

2.9.10 Measure the bytes yourself

An AI will give you byte counts for your examples, and it will sound confident. Check them. The playground can count the bytes of any character for you:

```
function setup() {
  createCanvas(400, 500);
  background("white");
  fill("black");

  const character: string = "ä";
  const byteCount: number = new TextEncoder().encode(character).length;
  text(`${character} needs ${byteCount} bytes`, 20, 40);
}
```

Two things in that program go beyond what this course has taught you, and that is fine here. `new TextEncoder()` builds a small helper that turns a string into its UTF-8 bytes, and `.length` asks how many bytes came out. Take the two lines as a measuring instrument. You put a character in, you get a byte count out.

Now measure every character from your research. Put it between the quotation marks, run the program, and read the number. Try a plain A, try ä and ß, try the euro sign, try a letter from a language you speak or are learning, and type an emoji straight from your phone or with the emoji keyboard of your computer. When the AI said one number and your program shows another, your program is right.

One warning, so that a surprise does not confuse you. Some emoji are not single characters at all but several characters glued together, for example flags or the ones with a skin tone. Those measure much more than four bytes. If that happens to you, either pick a simpler emoji or research what is going on.

2.9.11 AI for your research: how to ask

A few habits make the difference between an hour of scrolling and an hour of learning:

- **Say who you are and what you need it for.** Something like “I am 16, this is my first year of programming, and I have to answer this in a quiz in my own words” changes the whole answer. Without that sentence, an AI explains encodings the way it would explain them to a software developer.
- **Ask one question at a time,** and finish one of the three research questions before you open the next.
- **Keep asking until you can say it without the screen.** “Can you explain that again with a simpler example” is a completely normal follow-up, and you can send it as often as you need to. Then close the laptop and say the answer out loud to yourself. If you cannot, you are not done.
- **Let the AI test you.** Ask it to give you five questions about UTF-8 and to tell you what you got wrong. That costs two minutes and it is the closest thing you have to the quiz.
- **Stop the AI when it goes too far.** UTF-16, surrogate pairs, byte order marks, and normalization are real topics, and none of them are yours today. Write “that is too advanced for me, please keep it simple” and go back to your question.

2.9.12 Check your understanding

This quiz is different from the others in the book. Its six questions are not about anything this chapter taught you. They are about what you researched yourself, so do the research first and then answer in your own words.

An AI reads your answers and tells you what you already understand and what you should look up again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Characters, numbers, and bytes

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/pjctpd8dr2>

Part 3

Conditions

3.1 The if statement: programs that decide



Figure 3.1.1: The color areas program: three colored stripes, and below them a black bar where the program announces which color was clicked.

You'll build a canvas with three bold stripes of color and a black message bar at the bottom. The program watches where you click: hit the left stripe and the bar shows that color's name, hit the middle or right stripe and the name changes accordingly. It looks simple, but behind it sits the most important idea of this whole part: a program that reacts *differently* depending on what happened.

Look back at everything you've written so far. Every program ran its statements from top to bottom, one after the other, every single time. The crosshairs always drew the same two lines, the color worm always moved its hue forward by the same step. Your programs could already react *when* something happened, thanks to event functions like `mouseMoved`. But they could not yet react differently to *what* happened, and that changes now. In this part of the course, your programs learn to make decisions.

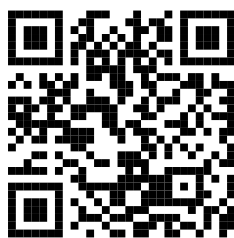
3.1.1 AI tutor

This part of the book has its own AI tutor, and every chapter of the part links to it. It knows the whole course up to here, so you can ask it about conditions as well as about anything from the earlier parts.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:
<https://app.novedu.at/aei6o7ko3h>

3.1.2 Questions with a yes/no answer

In the snapping chapter you met the three data types of this course, and one of them was still waiting for its moment: `boolean`, the type with only two values, `true` and `false`. That moment is now.

A decision always starts with a question, and in a program a question is written as a **comparison**. A comparison takes two values and compares them. The result is a boolean, `true` if the answer is yes and `false` if the answer is no. You write comparisons with **comparison operators**:

Table 3.1.1: The six comparison operators.

Operator	Question it asks
<	is the left value smaller?
>	is the left value greater?

Operator	Question it asks
<code><=</code>	is the left value smaller or equal?
<code>>=</code>	is the left value greater or equal?
<code>===</code>	are the two values exactly equal?
<code>!==</code>	are the two values different?

i Three equals signs are no mistake

Equality really is written with three equals signs, `===`, and “not equal” with an exclamation mark and two of them, `!==`. That’s how TypeScript spells these two questions. A single `=` wouldn’t do, because that character is already taken: it performs an assignment, as in `colorHue = 90`. The two equality operators get their first real job in the dice chapter (Section 3.3.4).

A comparison is a calculation like any other. It just produces a boolean instead of a number. Take `mouseX < width / 3`. The computer first works out `width / 3`, then compares. If the canvas is 400 pixels wide and the mouse sits at 95, the comparison asks “is 95 smaller than 133.33?”, and the result is `true`. At `mouseX = 250`, the same comparison produces `false`. Which brings up the obvious question: what can a program *do* with a `true` or a `false`?

3.1.3 The if statement

The **if statement** is the answer. It wraps a comparison around a block of code, and the block runs only when the comparison is `true`:

```
if (mouseX < width / 3) {
  message = "Orange";
}
```

Read it piece by piece:

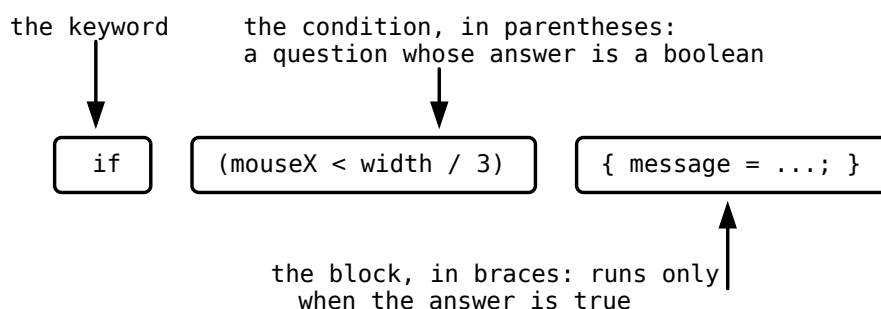


Figure 3.1.2: The if statement piece by piece: the keyword, the condition in parentheses, and the block in braces.

The question inside the parentheses is called the **condition**. When the program reaches the if statement, it evaluates the condition once. If the result is `true`, the block between the braces runs. If it’s `false`, the whole block is skipped as if it weren’t there, and the program continues after the closing brace.

Notice how the statement inside the block is indented one step further than the `if` itself. That indentation is not decoration: it shows the reader at a glance which statements belong to the block. From now on, code has *structure*, with blocks inside functions and soon blocks inside blocks. Correct indentation is how you keep that structure visible. Exactness, as always.

! Course rule: every block gets its braces

TypeScript would let you drop the braces when a block contains only a single statement, and you will meet such brace-less code on the web and in code that an AI writes for you. In this course we never do that: **every if, else if, and else gets a block in curly braces**, even for a single statement. The braces keep the structure visible, and they close a classic trap: adding a second statement to a brace-less branch later and not noticing that it doesn’t belong to the condition at all.

3.1.4 else if and else: one question after another

One stripe down, two to go. For the color areas you need three answers, and you get them by chaining questions. The keyword `else` means “otherwise”, and it adds a second block that runs exactly when the condition was `false`. And `else if` attaches the next question to that otherwise:

```
if (mouseX < width / 3) {  
  message = "Orange";  
} else if (mouseX < 2 * width / 3) {  
  message = "Green";  
} else {  
  message = "Red";  
}
```

The computer walks through the chain from top to bottom and takes the **first** branch whose condition is `true`. All later branches are skipped, even if their conditions would also be true. The final `else` has no condition at all: it catches everything that fell through all the questions above it.

That “first true wins” rule is doing quiet work in the second condition. Shouldn’t the green branch check that `mouseX` is between `width / 3` and `2 * width / 3`? It doesn’t have to. The program only *reaches* the `else if` when the first condition was `false`, so at that point `mouseX` is guaranteed to be at least `width / 3`. One comparison per branch is enough:

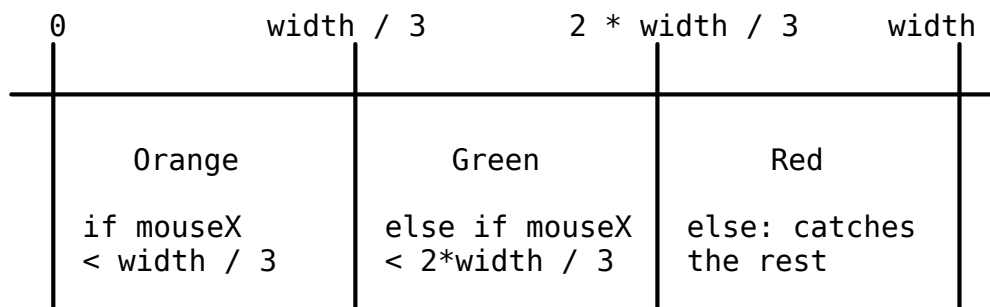


Figure 3.1.3: How the chain sorts every click into one of three answers: each condition only has to draw the boundary to its left neighbor, because earlier branches have already taken everything below it.

Exactly one of the three blocks runs, always. That’s the pattern to remember: a chain of `if`, `else if`, `else` picks one branch out of several, like a switch yard sorting a train onto one of three tracks.

3.1.5 A variable that gets its value in a branch

Look at the chain again. Every branch assigns to `message`, so the variable must be declared *before* the chain. The declaration looks slightly new:

```
let message: string;
```

There is no `=` and no value: the variable is declared empty, and the `if` chain fills it a moment later. That’s allowed, with one strict rule attached. You must assign the variable a value before you *read* it. TypeScript enforces the rule. Delete the `else` branch and the line `text(message, ...)` further down gets a red squiggle, because now there is a path through the program on which `message` never received a value. The spell checker for code is reading your branches, and it notices what you might miss.

Why collect the answer in a variable at all, instead of calling `text` three times, once in each branch? Because the three branches differ in one thing only: the text. Everything else, where the text goes and how it looks, is the same. So the chain decides only the *what*, and a single `text` statement after the chain does the *drawing*. Less repeated code, and if you later move the message bar, you change one line instead of three.

3.1.6 Dressing up text

Speaking of how the text looks, so far the output from `text` was small and plain. The message bar deserves better, and three formatting commands help. Like `fill` or `stroke`, you call them before drawing, and they stay active until changed:

- `textSize(30)` sets the height of the letters in pixels.
- `textStyle(BOLD)` switches to bold letters.
- `textAlign(CENTER, CENTER)` you half know from the circle tool, where `textAlign(RIGHT)` made a text end at its position. With two parameters, it centers the text on the given point, horizontally and vertically. Perfect here: pass the middle of the black bar, and the message sits centered in it, whatever its length.

3.1.7 Your exercise: Color areas

Now build the program. Work in this order:

1. **Paper first.** On the 400 by 200 canvas, `width / 3` is 133.33 and `2 * width / 3` is 266.67. Copy the table below and walk the if chain through five clicks by hand. Four of the five x values sit right next to a boundary, and that's on purpose: boundary clicks are where wrong comparisons hide. The first row is filled in as a model.

Click at x	<code>mouseX < width / 3</code>	<code>mouseX < 2 * width / 3</code>	Message
95	<code>95 < 133.33 -> true</code>	not asked	Orange
133			
134			
266			
267			

The “not asked” in the first row is the point of the whole table: once a branch has been taken, the program never looks at the questions below it.

2. **Pick your three colors.** They should be easy to tell apart. If you'd like a color combination that really works, Adobe's [color wheel](#) suggests matching sets: try the *Split Complementary* option under Color Harmony.
3. **Set the stage in setup.** Draw the three stripes as rectangles using the `width / 3` formulas, then the black message bar along the bottom, 50 pixels high.
4. **Write mouseClicked.** Repaint the black bar first, so the previous message disappears. Then set up the text formatting, declare `let message: string;`, and write the if chain that fills it. Your messages must name the colors *you* chose in step 2. Finish with one `text` statement that draws the message centered in the bar.
5. **Test like a professional.** Click in all three stripes first. Then hunt for the boundaries, and for that you need to know where you clicked: add debug output (1) by writing `mouseX` into the message bar next to the message. Now click your way slowly across the first boundary, a few pixels at a time, and watch where the message flips. The last x with the left color and the first x with the middle color must be the 133 and 134 from your table. Do the same at the second boundary.

💡 Exercise: Color Areas

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/color-areas.yaml>

3.1.8 Check your understanding

When your color areas program works, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: The if statement

► Take the quiz on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/rbobhh0k8x>

3.2 Color quadrants: a decision inside a decision



Figure 3.2.1: The color quadrants program: four colored quarters, and a message box in the center that names the clicked color.

The second decision program looks like a close relative of the first one, and it is: the canvas is now split into four colored quadrants, and a click anywhere makes a black box in the center announce the color's name. But there is a twist hiding in the geometry. Three stripes could be sorted with questions about `mouseX` alone. Four quadrants can't, because no single question about one coordinate can tell all four areas apart.

3.2.1 AI tutor

An if statement inside another one is where many beginners lose track of which branch runs when. The AI tutor of this part walks through your decision with you, condition by condition.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/aei6o7ko3h>

3.2.2 Two questions, four answers

Think about how you would decide as a human. Someone points at a spot on the canvas, and you want to name the quadrant. You'd naturally ask two questions, one after the other: is the spot in the left half or the right half? And is it in the upper half or the lower half? Each of the two questions has two possible answers, so two times two makes four combinations. That's exactly one combination per quadrant.

The first question is a comparison you could write immediately: `mouseX < width / 2`. The second one is its twin on the other axis, `mouseY < height / 2`. What's new is how the two questions work together. The second question is only asked *after* the first one is answered, inside the branch that the first answer picked.

3.2.3 An if statement inside an if statement

The block of an if statement can contain *any* statements, and that includes other if statements. Putting one if statement inside another is called **nesting**. The whole decision looks like this:

```
if (mouseX < width / 2) {  
  // Left half  
  if (mouseY < height / 2) {  
    message = "Orange";  
  }  
}
```

```

    } else {
      message = "Blue";
    }
  } else {
    // Right half
    if (mouseY < height / 2) {
      message = "Green";
    } else {
      message = "Red";
    }
  }
}

```

The outer if statement splits the canvas into left and right. Each of its two branches contains a complete inner if statement that splits its half into top and bottom. Four leaves at the end, one for each quadrant:

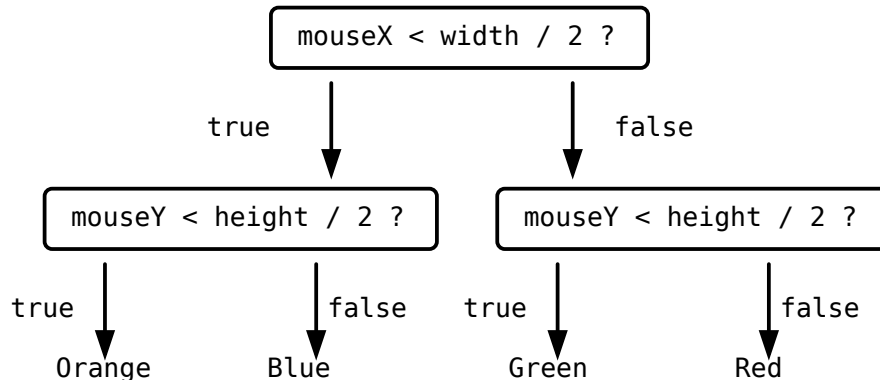


Figure 3.2.2: The nested decision as a tree: the first question picks a half, the second question picks the quadrant within that half.

Play computer with one click before you code. Say the canvas is 400 by 200 and the click lands at (300, 60). The outer condition asks: is 300 smaller than 200? No, so the whole first block is skipped and the program enters the `else` branch. There, the inner condition asks: is 60 smaller than 100? Yes, so `message` becomes "Green". Two questions answered, one quadrant found.

Watch the indentation as the nesting grows: the inner if statement is indented one step, its blocks two steps. Every opening brace pushes the code one step to the right, every closing brace comes back one step. Without that discipline, nested code becomes a riddle; with it, you can see the structure of the decision at a glance.

i Couldn't I ask both questions at once?

You might feel that "left half *and* upper half" should be a single condition, and you're right. TypeScript can combine two comparisons into one question. Those combining operators are worth their own chapters, and they're coming: the first one appears with the bouncing ball, the second one in the binary guesser project.

3.2.4 The message box in the center

One more difference to the color areas is the message box. It now sits in the middle of the canvas, on top of the quadrant boundaries. Its position is a centering job, and you know the formula from the responsive robot: the box's corner sits half the box's width left of the canvas center, and half its height above it. Give the box its size as two constants, for example `textAreaWidth` and `textAreaHeight`, and use them both for drawing the box in `setup` and for clearing it in `mouseClicked`, so the previous message disappears before the new one is written.

3.2.5 Your exercise: Color quadrants

The color areas chapter walked you through the code. This time the logic is yours to build; the previous chapter is your reference if you get stuck.

1. **Pick your four colors** and decide which quadrant gets which. On the 400 by 200 canvas, `width / 2` is 200 and `height / 2` is 100, so the two boundary lines run right through the middle.

2. **Play computer on paper.** Copy the table below and walk both questions through five clicks, one per quadrant plus one that lands exactly on the center. The first row is filled in as a model.

Click at	<code>mouseX < width / 2</code>	<code>mouseY < height / 2</code>	Quadrant	Message
(50, 50)	<code>50 < 200 -> true</code>	<code>50 < 100 -> true</code>	upper left	Orange
(350, 50)				
(350, 150)				
(50, 150)				
(200, 100)				

The last row is the tricky one. A click exactly on the center belongs to one single quadrant, and your two conditions decide which. Work out on paper which one it is, then remember the answer for the test.

3. **Set the stage in setup.** Draw the four quadrants as rectangles and the black message box centered on the canvas.
4. **Write mouseClicked.** Clear the message box, set up the text formatting, then write the nested if statements that fill `message` with the name of the clicked color. One `text` statement at the end draws it.
5. **Test against your notes.** Click once in each quadrant and compare the message with your prediction. The fifth row is the interesting one, and you can't hit the center pixel by feel: add debug output
- (1) by writing `mouseX` and `mouseY` into the message box next to the message. Then click near the center, read the numbers, nudge, and click again until the debug output says exactly 200/100. The message next to it decides whether your two conditions are exact.

💡 Exercise: Color Quadrants

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

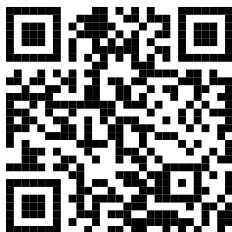
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/color-quadrants.yaml>

3.2.6 Check your understanding

When your quadrants answer every click correctly, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Color quadrants

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/gbzale3qqr>

3.3 Rolling dice: random numbers and exact comparisons

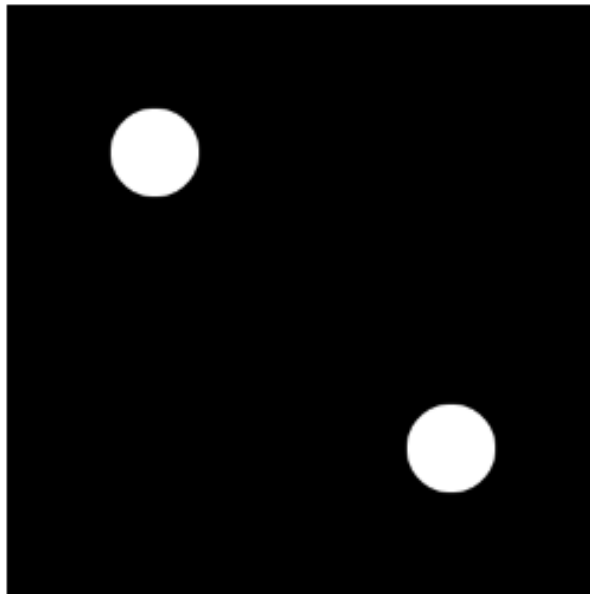


Figure 3.3.1: The dice program after one run: the computer rolled a two, so two white points appear on the black canvas.

This program is a die you can roll as often as you like. Every time it runs, the computer picks a number from 1 to 6 that nobody, not even you as the programmer, knows in advance, and then draws the matching dice face. One centered point for a one, two corner points for a two, and so on up to six. To pull that off, you need two new tools: a way to produce random numbers, and a comparison that asks “is it exactly this value?”.

3.3.1 AI tutor

Two AIs appear in this chapter, with different jobs. The tutor of this part answers your questions about random numbers, rounding, and comparisons, with hints. The advanced section at the end has an AI of its own that writes code from your prompt, so you can have your finished program improved and then take the improvement apart.

Your AI tutor

Hints and questions instead of finished programs, in English or German.

► [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:
<https://app.novedu.at/aei6o7ko3h>

3.3.2 A number nobody knows in advance

Everything your programs computed so far was predictable. The same input always led to the same picture. Games need the opposite: dice rolls, shuffled cards, surprise positions. p5.js provides it with the function `random`. A call like `random(1, 7)` returns a random number that is at least 1 and *less than* 7. The lower bound is included, the upper bound is not. Every call produces a fresh number, so running your program again means rolling again.

There is a catch. `random` returns a *decimal* number, and a call to `random(1, 7)` might give 3.874 or 6.999. There is no die in the world that shows 3.874 points.

3.3.3 Rounding down with floor

To turn 3.874 into a proper dice value, you need to cut off the decimal part, and that’s a kind of rounding you haven’t used yet. You know `round` from the snapping exercise; its sibling `floor` always rounds **down** to the next whole number. `floor(3.874)` is 3, and `floor(6.999)` is 6. Combined with `random`, the roll becomes one line:

```
const dice: number = floor(random(1, 7));
```

Follow a value through. `random(1, 7)` produces something between 1 and 6.999..., and `floor` maps everything from 1 up to 1.999... onto 1, everything from 2 up to 2.999... onto 2, and so on up to 6. Six equally wide ranges, six equally likely results: a fair die.

i Why not `round(random(1, 6))`?

It looks tempting, but play computer with it. `round` sends everything from 1 to 1.499... to 1, a range of width 0.5, while 2 gets everything from 1.5 to 2.499..., a range of width 1. The same squeeze happens at the top for the 6. With `round`, the 1 and the 6 would appear only half as often as the other numbers: a loaded die. The pattern `floor(random(min, max + 1))` is the standard recipe for fair whole-number randomness, worth memorizing.

3.3.4 Asking “is it exactly this value?”

The comparisons you’ve used so far, `<`, `>`, `<=`, and `>=`, ask about order. The dice program needs a different question: did I roll exactly a 1? The comparison table in the color areas chapter has the operator for it. The **equality operator** `===` asks “are the two values exactly equal?”, and its counterpart `!==` asks “are they different?”. Both produce a boolean, like every comparison.

Here is the question that decides whether the die shows a one. All points of a dice face have the same size, so their diameter sits in a constant named `dicePointDiameter`, and a one is a single point in the middle of the canvas:

```
if (dice === 1) {
  circle(width / 2, height / 2, dicePointDiameter);
}
```

The count of equals signs matters. A single `=` is an *assignment*, an order that says “put this value into the box”. The triple `===` is a *comparison*, a question: “are these equal?”. Writing `if (dice = 1)` instead of `if (dice === 1)` is one of the most classic mistakes in programming. TypeScript catches many of these slips with a red squiggle, but not all of them, so train your eyes now. Inside a condition, an `=` alone is almost always wrong. You may also spot a two-character `==` in code on the web; in this course, we always compare with `===` and `!==`.

With `===` in hand, the whole program becomes one chain that picks exactly one face out of six:

```
if (dice === 1) {
  // draw one point
} else if (dice === 2) {
  // draw two points
} else if (dice === 3) {
  // draw three points
} else if (dice === 4) {
  // draw four points
} else if (dice === 5) {
  // draw five points
} else {
  // must be 6: draw six points
}
```

Look at the last branch. It’s a plain `else`, without a condition. That’s safe reasoning, not laziness: `dice` can only hold 1 to 6, and if five questions have been answered with no, only the 6 is left. The final `else` in a chain is the “everything else” branch, and here “everything else” happens to be exactly one value.

3.3.5 The geometry of a dice face

Now for the creative half. Look closely at a real die. All six faces place their points on the same invisible grid of three columns and three rows. On your 200 by 200 canvas, the columns sit at `width / 4`, `width / 2`, and `3 * width / 4`, and the rows at the matching heights:

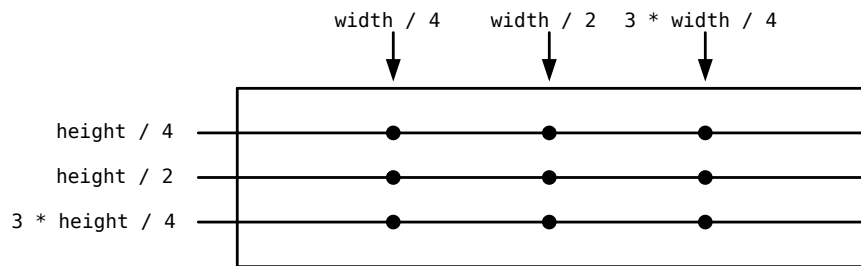


Figure 3.3.2: The nine anchor points every dice face is built from: three columns and three rows, at one quarter, one half, and three quarters of the canvas size.

Each face is just a selection of these nine anchors: the 1 uses only the center, the 2 takes two opposite corners, the 4 takes all four corners. And the faces reuse each other. A 3 is a 2 plus the center, a 5 is a 4 plus the center, and a 6 is a 4 plus the two middle anchors of the left and right columns. Spotting this reuse on paper first will save you a lot of typing and a lot of mistakes.

3.3.6 Your exercise: Dice

The starter code rolls the die and already draws the faces for 1 and 2. Your job is the cases 3 to 6.

1. **Paper first.** Give every anchor point a short name: the column first (L, C, R for left, center, right), then the row (T, M, B for top, middle, bottom). So LT is the upper left anchor at (50, 50), CM is the center at (100, 100), and RB is the lower right anchor at (150, 150). Now sketch the six faces on paper and complete this table. The faces for 1 and 2 are filled in because the starter code already draws them, and the 3 is filled in to show you what the last column is for.

Face	Anchors it uses	Built from an earlier face
1	CM	—
2	LT, RB	—
3	LT, RB, CM	the 2 plus the 1
4		
5		
6		

The last column is where the work gets saved: every face you still have to build is an earlier face plus one or two extra anchors. Find those before you type anything.

2. **Read the starter code.** Find the roll, the chain, and the two finished cases. Check the code for the 2 against your paper sketch: it uses `width / 4` and `3 * width / 4`, exactly your corner anchors. Add the missing data type to the `dice` declaration while you're there; every declaration gets one, as always.
3. **Add the cases 3 to 6** to the chain, one `circle` call per point. Work on one case at a time.
4. **Test each case deliberately.** A random program is awkward to test, because you can't tell it what to roll. So force the roll: replace the right side of the `dice` declaration with a fixed value, for example `const dice: number = 3;` check the face, then test 4, 5, and 6 the same way. When every face is right, put the `random` line back and roll a few times for the joy of it. Professionals test exactly like this. Make the random part predictable for a moment, check every case, then switch the randomness back on.

💡 Exercise: Dice

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/dice.yaml>

3.3.7 Advanced: the same dot, drawn five times

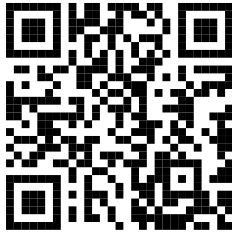
Get your paper sketches of the six faces out again and look at them as a group instead of one by one. The dot in the upper left corner appears on the 2, the 3, the 4, the 5, and the 6. Five faces share it, and your finished chain draws it with five separate `circle` statements. The center dot appears on the 1, the 3, and the 5. Count all the calls in your chain and you land at 21 `circle` statements for only nine distinct anchor points. Repetition like that is a warning sign in any program, and the conditions you already know are enough to remove it: the chain can be rewritten with far fewer `circle` statements.

You won't do the rewrite alone: this is a job for an AI, used the way you learned in the elephant exercise (Section 1.5.4). This exercise has its own AI, and you'll find it in the box below. The two rules from the elephant exercise apply word for word. Paste your complete, working dice program into the prompt, because running code is the most precise description of your environment. And say that you are a beginner and that the code shows everything you have learned so far, so the answer stays inside what you can understand. Then ask the AI to rewrite the program so that it needs as few `circle` statements as possible.

💡 Your AI for this exercise

This AI writes code from your prompt. Like the genie in the lamp, it grants what you say, not what you mean.

▶ [Open the exercise AI on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/pymqyk796z>

The AI will solve this, no surprise there. The real exercise is what happens next: don't just accept the answer. Read the new code line by line, and every time something is unclear, ask a follow-up question. Why is this comparison suddenly `>=` instead of `===`? Which faces does this `circle` call serve? What happened to the final `else`? Keep asking until you could have written the improvement yourself.

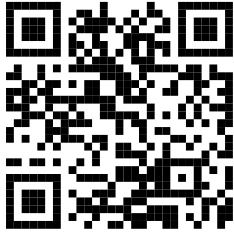
There's a reason to be that strict: **be prepared to describe your optimization in class**, without the AI and without your notes. You should be able to explain which dots the faces share and how the rewritten conditions use that overlap. When you can do that, the code is yours, no matter who typed it first.

3.3.8 Check your understanding

When all six faces roll correctly, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Rolling dice

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/g9ulmi6t1q>

3.4 The bouncing ball: your first animation

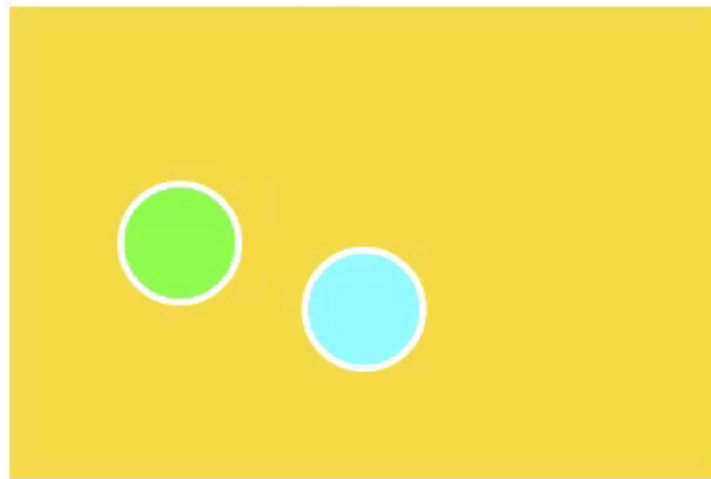


Figure 3.4.1: The finished first stage: two balls on their endless journey, one traveling left and right, the other up and down.

So far, movement in your programs always came from you: no mouse, no motion. This chapter breaks that rule. A ball flies across the canvas all by itself, and when it reaches a wall, a condition notices and sends it back the other way, forever. You'll first study and extend a one-ball starter program, adding a second ball that bounces vertically. Then, in the advanced stage, you'll set a single ball loose diagonally, bouncing off all four walls, endlessly crisscrossing the canvas.

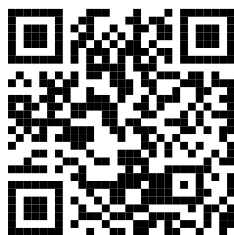
3.4.1 AI tutor

An animation that misbehaves is hard to read, because it moves too fast to see what went wrong. Tell the AI tutor of this part what your ball does and what you expected instead.

Your AI tutor

Hints and questions instead of finished programs, in English or German.

► [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/aei6o7ko3h>

3.4.2 An animation is many pictures

Every movie ever made is a trick: a rapid series of still pictures, each slightly different from the one before. Show about 60 of them per second and your eyes see motion. Computer animation works exactly the same way, and each still picture is called a **frame**.

Frames in your hands: the flip book

You've probably held a frame-by-frame movie before without calling it that: a **flip book**, a small pad of paper where every page shows a drawing slightly different from the page before. Flip through it with your thumb and the drawing moves. A frame is exactly one page of such a flip book, and an animated program is a flip book whose pages the computer draws for you, about 60 pages per second. To see the idea in action, watch the video [How to make a flipbook](#).

p5.js has a function slot ready for this, and it's the last of the big ones: **draw**. You know the pattern from `mouseMoved` and `mouseClicked`. You write a function with the agreed name, and p5.js calls it at the right

moments. For **draw**, the right moment is *every frame*. p5.js calls it around 60 times per second, from program start until you close the page, no events needed. Whatever **draw** paints is one frame of your movie.

Compare your three kinds of functions: **setup** runs exactly once, at the start. Event functions like **mouseClicked** run when the user does something. And **draw** runs constantly, on its own. From one frame to the next, your global variables keep their values, and that's the whole secret of animation. Change a variable a little every frame, and the picture it controls comes alive.

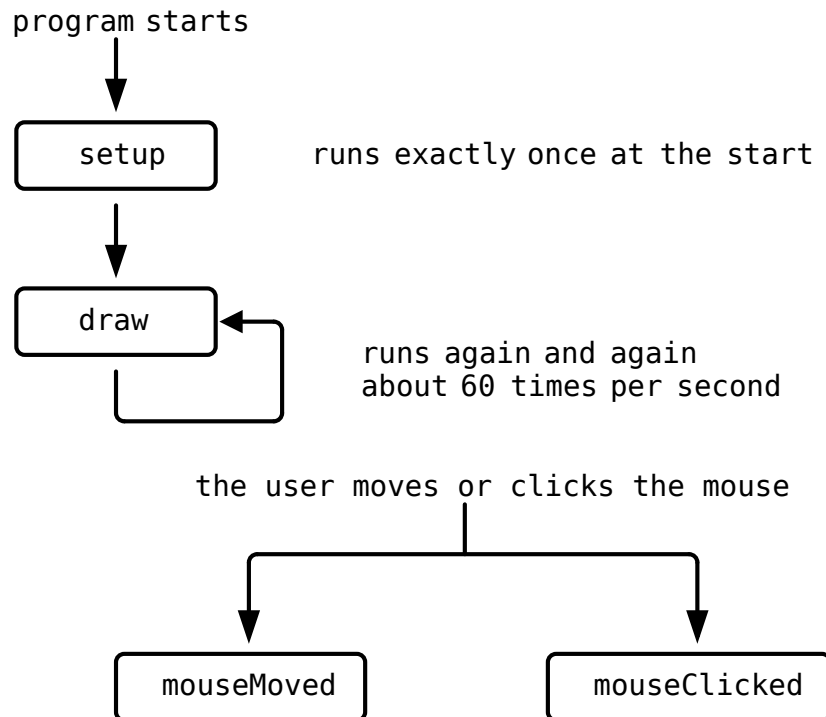


Figure 3.4.2: The life of a p5.js program: **setup** runs exactly once, then **draw** runs in an endless loop, and event functions like **mouseMoved** and **mouseClicked** run in between whenever the user acts.

3.4.3 A ball that moves itself

Here is the recipe in its smallest form. One ball on a 300 by 200 canvas, flying from left to right at half the canvas height. Three global variables carry it. The ball's horizontal position and its current travel direction change from frame to frame, so both are declared with **let**; the ball's size never changes, so it is a **const**:

```
const circleDiameter: number = 50;
let circleCenterX: number = 0;
let direction: number = 2;
```

Each frame, **draw** does three things: repaint, draw, move.

```
function draw() {
  background("gold");

  stroke("white");
  strokeWeight(3);
  fill("lime");
  circle(circleCenterX, height / 2, circleDiameter);

  circleCenterX += direction;
}
```

The **background** call comes first. It repaints the whole canvas and erases the previous frame. You know this decision from the crosshairs and the color worm: without it, every frame leaves its ball behind and the path smears across the screen. Then the ball is drawn at its current position, halfway up the canvas.

The last line moves the ball, using an operator you can almost guess. **circleCenterX += direction** is the short form of **circleCenterX = circleCenterX + direction**: take the value in the box, add **direction**, put

the result back. In the color worm you wrote the long form; the short form is what programmers actually type. Every calculation operator you know has such a short form, and together they are called the **compound assignment operators**:

Table 3.4.1: The five compound assignment operators.

Operator	Short form	Means the same as
<code>+=</code>	<code>score += 10</code>	<code>score = score + 10</code>
<code>-=</code>	<code>lives -= 1</code>	<code>lives = lives - 1</code>
<code>*=</code>	<code>direction *= -1</code>	<code>direction = direction * -1</code>
<code>/=</code>	<code>speed /= 2</code>	<code>speed = speed / 2</code>
<code>%=</code>	<code>colorHue %= 360</code>	<code>colorHue = colorHue % 360</code>

The pattern never changes: take the value the variable currently holds, calculate with it, and store the result in the same variable. Only variables declared with `let` can be used this way, because a `const` never changes its value. The last row uses the `%` operator from the color worm (Section 2.5.5), the one that gives the remainder of a division.

Now the animation adds up. With `direction` holding 2, every frame shifts the ball 2 pixels to the right. At 60 frames per second, that’s about 120 pixels per second of smooth flight. The value of `direction` is the ball’s speed, in pixels per frame.

3.4.4 Turning around at the wall

Left alone, `circleCenterX` grows past the right edge and the ball flies off the canvas, never to return. Time for a condition: when the ball reaches a wall, its direction must turn around.

Turning around is beautifully cheap. A ball moving right has `direction` 2; moving left means `direction` -2. One multiplication flips between the two: `direction *= -1` turns 2 into -2 and, at the other wall, -2 back into 2. The same line bounces off both walls.

And when should it run? When the ball has reached the right wall *or* the left wall. That little word “or” is a new operator, `||`. The **OR operator** combines two boolean questions into one, and the combined result is `true` when at least one of the two parts is true:

```
if (circleCenterX >= width || circleCenterX <= 0) {
  direction *= -1;
}
```

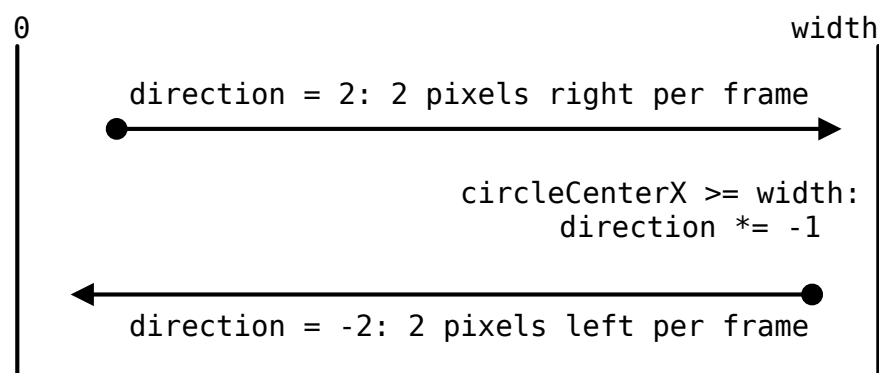


Figure 3.4.3: One condition guards both walls: past the left edge or past the right edge, either answer flips the direction.

One detail deserves a professional’s eye: the comparisons are `>=` and `<=`, not `===`. Play computer to see why. With a speed of 2 starting at 0, the positions are 0, 2, 4, ..., and on the 300 pixel wide canvas the ball lands exactly on 300, so `=== width` would happen to work. Now change the speed to 7. The positions jump 294, 301, and the ball is never *exactly* at 300. An `===` condition would never fire, and the ball would sail off the canvas forever. Conditions that guard a border should ask “at or beyond”, not “exactly at”, so even a fast ball that overshoots gets caught.

💡 The two vertical bars

On an Austrian or German keyboard, the | character is typed with AltGr and the key left of Z (the one with < on it). For ||, type it twice. The character is called a *pipe*, and yes, the OR operator really is two of them; a single | exists too but does something entirely different.

3.4.5 Your exercise, part 1: a second ball

The playground hands you the single-ball program from this chapter as your starting point: the three globals, the `draw` function, and the bounce condition, all of them lines you have just read. Your job is to give that ball a partner.

1. **Read the starter code** until every line has a job you can name. One small chore while you read: the playground's starter declares its globals without data types; add them so the code matches the listings in this chapter (every declaration gets one, as always).
2. **Play computer around the bounce.** Copy the table below and follow the ball through five frames on the 300 pixel wide canvas, starting two frames before the right wall. Keep the order of `draw` in mind: first the ball is drawn, then it moves, then the condition is checked. The first row is filled in as a model.

Frame	circleCenterX at the start	After the move	Wall reached?	direction at the end
1	296	298	298 >= 300 -> no	2
2				
3				
4				
5				

One row in your table is the bounce. Now change the starting value to 297 and redo the five rows. The ball never lands exactly on 300 this time, and that is the reason the condition uses `>=` instead of `===`.

3. **Add a second ball** that travels vertically: it starts at the top, moves down, and bounces between top and bottom walls. It needs its own position variable and its own direction variable, and it should wear a different fill color. Both balls are drawn in the same `draw` function, after the one `background` call. The second ball's condition guards `height` and 0 instead of `width` and 0.
4. **Test both directions.** Watch at least one full bounce at every wall: right, left, bottom, top.

💡 Exercise: Bouncing Ball

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/bouncing-ball.yaml>

💡 A ball with more manners

Look closely at a bounce. The ball's center reaches the wall, so half the ball sinks into it before turning around. If that bothers your designer's eye, refine the conditions so the ball's *edge* touches the wall instead: the center then must stay `circleDiameter / 2` away from each border. Optional, but satisfying.

3.4.6 Your exercise, part 2: the diagonal ball

Now the advanced stage, and this time the playground gives you an almost empty file: one ball, starting at a random position, flying diagonally, bouncing off all four walls.

Diagonal flight sounds new, but it's only the two motions you just built, happening to one ball at the same time: the x position changes by a horizontal direction, the y position by a vertical one, every frame. So the ball

needs four global variables, two for its position and two for the two directions.

Two things to work out on paper before you code:

- **The starting position is random**, so the position variables can't get their values in the declaration. `random(0, width)` needs the canvas, and the canvas exists only after `createCanvas` has run. You know the pattern from the color areas' **message**: declare the globals without a value, `let circleCenterX: number;`, and assign them in `setup`.
- **Keep the two bounce conditions separate**. It's tempting to chain all four wall checks into one big `||` condition, but think about what must happen: hitting the left or right wall flips only the horizontal direction; hitting the top or bottom flips only the vertical one. One if statement per axis, each flipping its own direction variable. If you merged them, a touch of the side wall would also reverse the vertical motion, and the ball would stutter along the walls instead of bouncing cleanly.

Build it in this order: canvas and random start in `setup`; in `draw` repaint, draw the ball, move both coordinates, then the two bounce conditions. Test by watching for a while. The ball should visit all four walls and leave each one at the mirrored angle. If you kept the edge refinement from part 1, carry it over here too.

💡 Exercise: Bouncing Ball Advanced

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

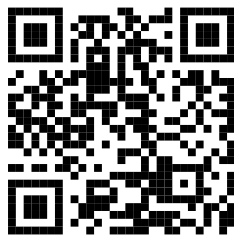
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/bouncing-ball-advanced.yaml>

3.4.7 Check your understanding

When you've finished both stages, take the short quiz below. You answer seven questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: The bouncing ball

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/ievjp8iozf>

3.5 Binary and hexadecimal conversions

Deep inside, a computer stores everything as numbers written with only the digits 0 and 1: your dice rolls, the colors on your canvas, this very text. In this chapter you learn to read and write that notation yourself. You start from the decimal system you use every day, build the binary system from it, and then meet hexadecimal, the compact notation programmers use when binary gets too long to read.

This chapter is pure pen and paper: no new p5.js functions, no TypeScript. Get a pencil, because every conversion here is a small calculation you do by hand. The next chapter turns these calculations into working programs.

Maybe you already know these conversions from math class. Test yourself: jump to the mixed practice at the end of this chapter (Section 3.5.13) and work through the six conversions there. If they all come out right, you can move on to the next chapter with a clear conscience.

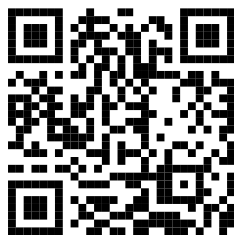
3.5.1 AI tutor

Converting between number systems is a skill you build by doing, and a book chapter can only print a handful of practice numbers. Ask the AI tutor for a fresh number whenever you want one more round. Show your calculation, not only your final answer. The tutor can then help you find the exact step that went wrong.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/o3uxgq8zsv>

3.5.2 Decimal, the system you already know

You read decimal numbers without thinking about the process. Still, that process is the key to every number system in this chapter, so let's slow it down once.

Why do people count with ten digits? We can't know the origin for certain, but a likely explanation is right in front of you: humans have ten fingers. Fingers are always available, so people probably used them as early counting tools, and our base-10 system may have grown from that habit.

Take the decimal number 352. Your brain reads it in one glance, but behind that glance sits a small calculation: each digit is worth its face value times a **place value**, and the place values are the powers of ten.

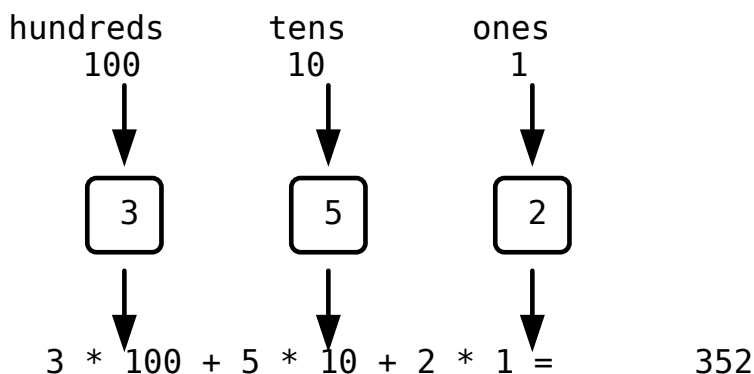


Figure 3.5.1: The number 352 taken apart: each digit is worth its face value times its place value, and the place values are the powers of ten.

Decimal is a **base-10** system. It has ten digits, 0 to 9, and each place is worth ten times as much as the place to its right: 1, then 10, then 100, then 1000, and so on.

! Remember the power of zero

Every nonzero number to the power of 0 is 1. You may have learned this rule before, but it is easy to forget:

- $10^0 = 1$
- $2^0 = 1$
- $16^0 = 1$

That is why the rightmost place has the value 1 in decimal, binary, and hexadecimal.

Binary and hexadecimal follow exactly the same place-value idea. Only the base and the available digits change. Everything you calculate in this chapter is this one rule, applied to different bases.

3.5.3 Binary: counting with two digits

Now imagine an alien with only one finger. The finger has two states: it can be inactive or active. With that finger, the alien can show 0 or 1, but it can't continue to 2, 3, and all the way to 9. After 1, the alien already needs a new place to keep counting.

A computer is a little like this one-finger alien. A tiny electronic switch inside a computer also has exactly two states, off and on. Computers therefore write all their numbers with two digits instead of ten.

Binary is this **base-2** system. It uses only the digits 0 and 1, and one binary digit is called a **bit**. Each binary place is worth twice as much as the place to its right:

Position from the right	Power of 2	Place value
0	2^0	1
1	2^1	2
2	2^2	4
3	2^3	8
4	2^4	16
5	2^5	32
6	2^6	64
7	2^7	128

A 1 means the place value is present, a 0 means it is absent. The binary number 1101 contains one 8, one 4, no 2, and one 1:

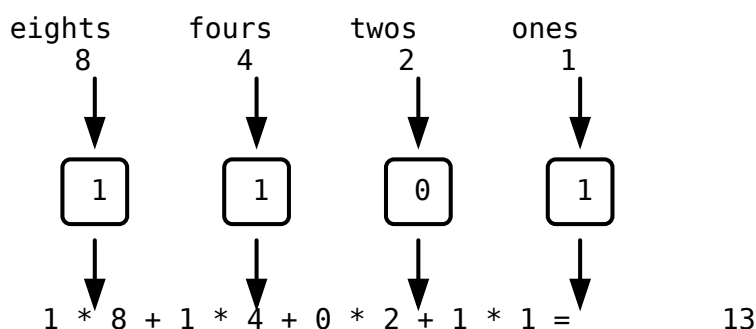


Figure 3.5.2: The binary number 1101 taken apart: the place values are the powers of two, and they add up to 13.

3.5.4 Convert binary to decimal

The picture of 1101 is already the whole method. To convert any binary number to decimal:

1. Write the place values under the bits, from right to left: 1, 2, 4, 8, 16, and so on.
2. Multiply each bit by its place value.

3. Add the contributions.

Convert binary 101010 to decimal. Start with the rightmost bit, whose place value is 1:

Step	Bit	Place value	Contribution	Running total
1	0	1	$0 \times 1 = 0$	0
2	1	2	$1 \times 2 = 2$	$0 + 2 = 2$
3	0	4	$0 \times 4 = 0$	$2 + 0 = 2$
4	1	8	$1 \times 8 = 8$	$2 + 8 = 10$
5	0	16	$0 \times 16 = 0$	$10 + 0 = 10$
6	1	32	$1 \times 32 = 32$	$10 + 32 = 42$

Binary 101010 is decimal 42.

💡 Why developers keep using 42

In Douglas Adams's novel *The Hitchhiker's Guide to the Galaxy*, 42 is the "Answer to the Ultimate Question of Life, the Universe, and Everything." The book is popular among software developers, so 42 appears frequently in coding examples, puzzles, and exercises.

If you enjoy fantasy or science fiction novels, consider reading the book. You'll understand the reference the next time 42 appears in a programming exercise.

3.5.4.1 Practice binary to decimal

Convert these five binary numbers on paper. Draw the same columns as in the worked example: bit, place value, contribution. The first row is filled in as a model.

Binary	Contributions	Decimal
101	$1 \times 4 + 0 \times 2 + 1 \times 1$	5
1110		
10011		
110101		
1111111		

Compare only when your table is complete: the results are 14, 19, 53, and 255.

Look closely at the last one. 255 should feel familiar: it's the largest value of a color channel (Section 2.4.3). That's no coincidence, and a few sections from now you'll see exactly why.

3.5.5 Convert decimal to binary

The opposite direction uses division instead of multiplication. Divide by 2 repeatedly. Each division gives you a whole-number **quotient** and a **remainder**, and because you divide by 2, every remainder is either 0 or 1. Those remainders are the bits.

Convert decimal 45 to binary:

Step	Number to divide	Division by 2	Quotient	Remainder
1	45	$45 = 22 \times 2 + 1$	22	1
2	22	$22 = 11 \times 2 + 0$	11	0
3	11	$11 = 5 \times 2 + 1$	5	1
4	5	$5 = 2 \times 2 + 1$	2	1
5	2	$2 = 1 \times 2 + 0$	1	0
6	1	$1 = 0 \times 2 + 1$	0	1

The divisions produce the bits from right to left, so read the remainder column from bottom to top: 1, 0, 1, 1, 0, 1. Decimal 45 is binary 101101.

Use the same process every time:

1. Divide the decimal number by 2.
2. Write down the whole-number quotient and the remainder.
3. Divide the quotient by 2 again.
4. Stop when the quotient is zero.
5. Read the remainders from bottom to top.

Here is a second, shorter run. Convert decimal 26:

Step	Number to divide	Division by 2	Quotient	Remainder
1	26	$26 = 13 \times 2 + 0$	13	0
2	13	$13 = 6 \times 2 + 1$	6	1
3	6	$6 = 3 \times 2 + 0$	3	0
4	3	$3 = 1 \times 2 + 1$	1	1
5	1	$1 = 0 \times 2 + 1$	0	1

Read the remainders from bottom to top: decimal 26 is binary 11010.

3.5.5.1 No answer key needed

Here's the best thing about conversions. You never have to wonder whether your result is right, because you can always convert it back. Take the 11010 from the last example and send it in the opposite direction, multiplying bits by place values:

Bit	Place value	Contribution
0	1	$0 \times 1 = 0$
1	2	$1 \times 2 = 2$
0	4	$0 \times 4 = 0$
1	8	$1 \times 8 = 8$
1	16	$1 \times 16 = 16$

The contributions add up to $0 + 2 + 0 + 8 + 16 = 26$, the number we started from. The conversion is confirmed.

Make this reverse trip a habit: finish every conversion by traveling back. If you land on your starting number, both directions were right. If you don't, one of the two trips holds a mistake, and finding it is exactly the practice you need.

3.5.5.2 Practice decimal to binary

Convert 21, 37, and 52 to binary with the division table. Then check each result yourself with the reverse trip: multiply the bits by their place values and confirm that you land back on the starting number. Only after that self-check, compare with the results 10101, 100101, and 110100.

3.5.6 Powers of two are everywhere

In decimal, you know the round numbers 10, 100, and 1000 by heart. For computers, the round numbers are the powers of two, and they show up everywhere in computing. You must be able to name the powers of two from 2^0 through 2^{16} without calculating each one. Practice until they feel as familiar as 10, 100, and 1000.

Power	Value	Power	Value
2^0	1	2^9	512
2^1	2	2^{10}	1,024
2^2	4	2^{11}	2,048
2^3	8	2^{12}	4,096

Power	Value	Power	Value
2^4	16	2^{13}	8,192
2^5	32	2^{14}	16,384
2^6	64	2^{15}	32,768
2^7	128	2^{16}	65,536
2^8	256		

Two larger powers are especially important:

- $2^{31} = 2,147,483,648$, which is approximately 2.15 billion.
- $2^{32} = 4,294,967,296$, which is a little less than 4.3 billion.

Why do these numbers matter so much? A group of n bits can form exactly 2^n different patterns. A group of eight bits is called a **byte**, and a byte can form $2^8 = 256$ patterns. Remember the color basics chapter (Section 2.4.3)? It said a color channel is stored in one byte and runs from 0 to 255, and now you know where those numbers come from: 256 patterns, one of them used for zero, so the largest value is 255. The binary number 11111111 from your practice table is one full color channel.

The same rule scales up. An unsigned 16-bit number has 2^{16} patterns and holds the values 0 through 65,535. With 32 bits there are 2^{32} patterns. What a 32-bit number can hold depends on one more idea: whether one of its bits is spent on the sign.

3.5.7 Shorten binary with hexadecimal

Binary is honest but long-winded: even a small number like 214 needs the eight bits 11010110, and long rows of ones and zeros are hard to scan. Hexadecimal, usually shortened to **hex**, is the compact notation programmers use instead.

Hexadecimal is a **base-16** system, so it needs 16 digits. The symbols 0 to 9 cover the first ten values, and the letters A to F cover the values 10 to 15:

Decimal value	Binary	Hex digit	Decimal value	Binary	Hex digit
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

The counting continues past F the same way it does in decimal past 9. The next number after hex F is hex 10, which has the decimal value 16. The place values of hex are the powers of 16: 1, 16, 256, 4096, and so on. The calculation rule itself never changes. Multiply each digit by its place value, then add the contributions. Only the base is different.

The binary column in the digit table shows why programmers love hex. Exactly four bits can represent the values 0 to 15, so one hex digit always matches one group of four bits, and two hex digits match one byte. That makes hex a perfect abbreviation for binary: divide the bit count by four and you have the hex digit count.

3.5.8 Convert binary to hexadecimal

To convert binary to hex, split the bits into groups of four, starting at the right, and translate each group with the digit table (Section 3.5.7). Add zeros on the far left if the first group is too short.

Convert binary 10110110 to hex:

Step	What you do	Result
1	Group four bits from the right	1011 0110
2	Convert 1011 to its value	11
3	Write decimal 11 as a hex digit	B
4	Convert 0110 to its value	6
5	Write both hex digits in the same order	B6

Binary 10110110 is hex B6.

For a shorter example, binary 10110 becomes 0001 0110 after you add leading zeros. The groups are hex 1 and hex 6, so the result is hex 16. The added zeros don't change the value.

3.5.8.1 Practice binary to hexadecimal

Group and translate these binary numbers on paper. The first row is filled in as a model.

Binary	Grouped binary	Hex
1010	1010	A
1111		
101101		
11001010		
11111111		

The results, for after your own attempt: F, 2D, CA, and FF. The last row is an old friend by now. 11111111, 255, and FF are three ways to write one full color channel.

3.5.9 Convert hexadecimal to binary

The reverse direction is even quicker: replace every hex digit with its four-bit group from the digit table (Section 3.5.7).

Convert hex 3D to binary:

Step	Hex digit	Four-bit group	Result so far
1	3	0011	0011
2	D	1101	0011 1101

Hex 3D is binary 00111101. You may remove zeros only from the far left, so binary 111101 has the same value. Inside the number, every group keeps all four bits.

3.5.9.1 Practice hexadecimal to binary

Translate these hex numbers into binary, keeping the four-bit groups visible. The first row is filled in as a model.

Hex	Binary in four-bit groups
7	0111
B	
2F	
A5	
FF	

The results, for after your own attempt: 1011, 0010 1111, 1010 0101, and 1111 1111.

3.5.10 Convert decimal to hexadecimal

You can convert decimal to hex in two ways.

The first way travels through binary:

1. Convert decimal to binary with repeated division by 2.
2. Group the bits in groups of four from the right.
3. Replace each group with one hex digit.

For example, decimal 45 is binary 101101. Group it as 0010 1101, and the groups become hex 2 and hex D: decimal 45 is hex 2D.

For larger numbers, the direct road is shorter: divide by 16 repeatedly, exactly like the repeated division by 2, and translate remainders from 10 to 15 into the letters A to F.

Convert decimal 713 directly to hex:

Step	Number to divide	Division by 16	Quotient	Remainder	Hex digit
1	713	$713 = 44 \times 16 + 9$	44	9	9
2	44	$44 = 2 \times 16 + 12$	2	12	C
3	2	$2 = 0 \times 16 + 2$	0	2	2

Read the hex digits from bottom to top: decimal 713 is hex 2C9.

3.5.11 Convert hexadecimal to decimal

Again there are two roads. Through binary: hex 2D becomes binary 0010 1101, which has the decimal value $32 + 8 + 4 + 1 = 45$.

The direct road uses the hex place values 1, 16, and 256. Multiply every digit value by its place value and add the contributions, exactly like in decimal and binary.

Convert hex 2D7 to decimal:

Step	Hex digit	Digit value	Place value	Contribution	Running total
1	7	7	1	$7 \times 1 = 7$	7
2	D	13	16	$13 \times 16 = 208$	$7 + 208 = 215$
3	2	2	256	$2 \times 256 = 512$	$215 + 512 = 727$

Hex 2D7 is decimal 727.



You can now decode web colors

On the web, colors are usually written as hex codes such as `#FF8800`. That code is three bytes side by side, two hex digits each: `FF` for red, `88` for green, `00` for blue. With this chapter, you can decode it. `FF` is $15 \times 16 + 15 = 255$, red at full power. `88` is $8 \times 16 + 8 = 136$, a medium green. `00` is no blue at all. In `p5.js`, the same color is `fill(255, 136, 0)`, a strong orange. You can now read every color code you meet on the web, two hex digits per channel.

3.5.12 The conversion map

You now have a path for every direction. Binary sits in the middle of the map, because both neighbors connect to it so easily:

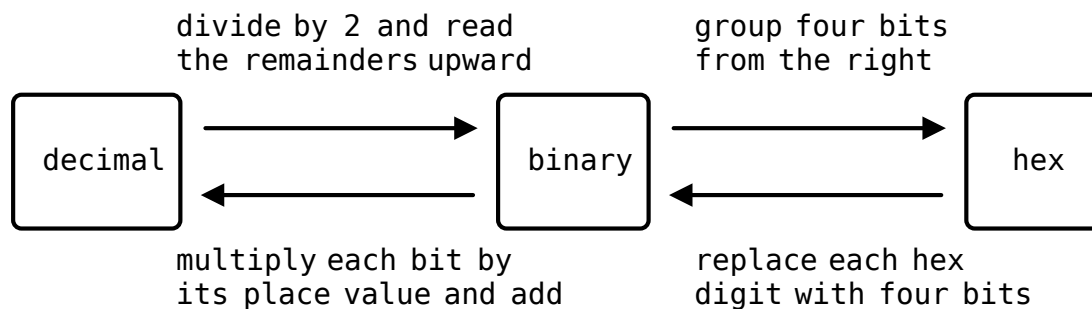


Figure 3.5.3: The conversion map: decimal and hexadecimal both connect to binary. Between decimal and hex you can travel through binary, or take the direct road with divisions by 16 and powers of 16.

Starting system	Target system	Process
Binary	Decimal	Multiply bits by powers of 2, then add
Decimal	Binary	Repeatedly divide by 2, then read remainders upward
Binary	Hex	Group four bits from the right
Hex	Binary	Replace every hex digit with four bits
Decimal	Hex	Convert through binary or repeatedly divide by 16
Hex	Decimal	Convert through binary or use powers of 16

And remember the built-in answer key (Section 3.5.5.1): whatever direction you travel, travel back to check. If decimal 45 becomes binary 101101, convert 101101 back to decimal. If hex A5 becomes binary 10100101, group those bits again and check that you get A5.

3.5.13 Mixed practice

One conversion per direction, in mixed order. For each line, show every step on paper, and then do the reverse trip before you look anything up.

Start	Direction	Your result	Reverse check
decimal 31	decimal to binary		
binary 111100	binary to decimal		
binary 10011110	binary to hex		
hex 6C	hex to binary		
decimal 128	decimal to hex		
hex A5	hex to decimal		

When both trips agree, compare with the results:

Start	Result
decimal 31	binary 11111
binary 111100	decimal 60
binary 10011110	hex 9E

Start	Result
hex 6C	binary 01101100
decimal 128	hex 80
hex A5	decimal 165

To finish, treat yourself to a real-world conversion: the web color **#40E0D0**. Decode its three channels to decimal, then check your result in the playground with a quick **background** call using your three numbers. If a friendly turquoise fills the canvas, your conversion was right. Enjoy the moment. You just read a color the way a computer does.

3.5.14 Check your understanding

When your conversion table is filled in and checked, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should practice again. Show your calculation steps, not only the final answer. The quiz is anonymous, and answering in German is fine too.

i Quiz: Binary and hexadecimal conversions

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:
<https://app.novedu.at/11c3e0hzgz>

3.6 Number systems in code

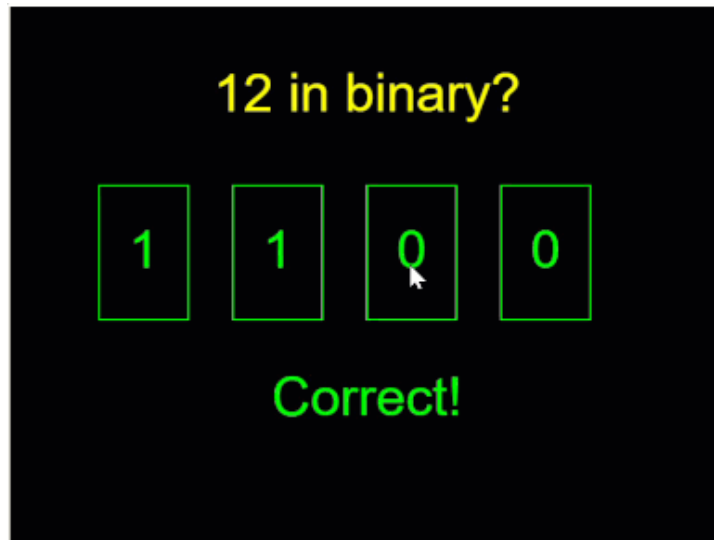


Figure 3.6.1: The binary guesser, solved: the player toggled the four binary digits to 1100, the program checked the guess and everything turns green.

In the previous chapter you converted numbers with pen and paper. Now the conversions become working TypeScript. You'll take decimal, binary, and hexadecimal numbers apart digit by digit, and the chapter ends with a game you can challenge your friends with: the program shows a decimal number, the player clicks together its binary representation, and when the guess is right, the display flashes green. Three smaller exercises prepare you for the game. First you take an ordinary decimal number apart, then a binary number, and finally a hexadecimal number.

These number systems are everyday tools in programming, not just theory. Bits represent switches, flags, and pixel data. Hex codes such as `#FF8800` name colors on the web, and debugging tools show memory addresses in hex. Whenever you meet them, the digit-by-digit thinking from this chapter is what makes them readable.

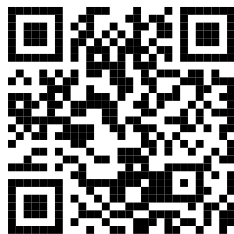
3.6.1 AI tutor

Use the tutor from the conversion chapter (Section 3.5.1) when a conversion step is still unclear. Use the Conditions tutor below when you need help applying the conversion in TypeScript, laying out the digit boxes, or checking the binary guesser.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/aei6o7ko3h>

3.6.2 Taking a number apart

The first challenge sounds almost silly: the program rolls a random number, stores it in a variable named `num`, and must draw each digit in its own box. For 4711 that means a 4, a 7, a 1, and a 1. But `num` holds the single *value* 4711, not four separate digits. How do you get single digits out of a number? With two operations you already know, working as a team.

The `%` operator from the color worm gives the remainder of a division, and that makes it a digit extractor: `num % 10` is the remainder when dividing by 10, which is exactly the last digit. `4711 % 10` is 1.

`floor` from the dice chapter rounds down, and that makes it a digit remover: `floor(num / 10)` chops off the last digit. `4711 / 10` is 471.1, and flooring it leaves 471.

Chain the two and you can reach every digit. Divide first to shift the digit you want to the last place, then `% 10` to grab it:

```
const digit1: number = num % 10;           // ones
const digit2: number = floor(num / 10) % 10; // tens
const digit3: number = floor(num / 100) % 10; // hundreds
```

Why 10, 100, and their bigger siblings? Because that's how our number system is built. The digits of 352 are worth their face value times a **place value**, and the place values are the powers of ten:

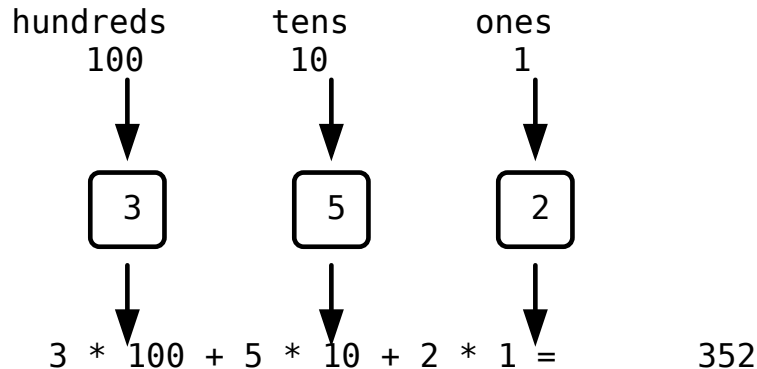


Figure 3.6.2: The number 352 taken apart: each digit is worth its face value times its place value, and the place values are the powers of ten.

Check the recipe against the picture. Dividing 352 by 100 and flooring gives 3, the hundreds digit. Dividing by 10 and flooring gives 35, and `35 % 10` is 5, the tens digit. The recipe works for any place: divide by the place value, floor, take `% 10`.

The number to take apart comes from a roll, using the fair-whole-number recipe from the dice chapter: `floor(random(0, 1_000_000))` gives a whole number from 0 to 999999. Those underscores are not a typo. TypeScript lets you group the digits of a long number the same way you would group them with spaces on paper. They mean nothing to the computer and everything to human eyes.

3.6.3 A row of boxes, placed by formula

All three digit exercises in this chapter show the same picture: a row of boxes on a black canvas 300 pixels high, each box 50 pixels wide and 75 pixels high, the row starting 100 pixels below the top edge, one digit per box. Decimal and binary need six boxes and a canvas 550 pixels wide; hexadecimal needs only three boxes on a 300 pixel wide canvas.

The interesting part is the x coordinate of each box. You could work the six numbers out by hand, 50, 125, 200, and so on, and type them in. Better not. Write each position as a formula with the box number in it:

```
rect(50 + 0 * 75, 100, 50, 75);
rect(50 + 1 * 75, 100, 50, 75);
rect(50 + 2 * 75, 100, 50, 75);
```

The 50 is the left margin, the 75 is the step from one box to the next (50 pixels of box plus a 25 pixel gap), and the 0, 1, 2 count the boxes from the left. TypeScript works out the numbers for you, and the code shows the *rule* behind the layout instead of six finished results. Want wider boxes later? Change one number per line instead of recomputing everything. The binary guesser at the end of this chapter reuses this layout, and there the formula earns its keep a second time: it tells you exactly where each box starts and ends when you have to find out which box the player clicked.

3.6.4 Your exercise: Digits

Now build it: a random number up to six digits, each digit drawn in its own yellow box.

1. **Play computer first.** Complete this table on paper for `num = 4711`. The first two rows are filled in as a model.

Constant	Formula	Result for 4711
<code>digit1</code>	<code>num % 10</code>	1
<code>digit2</code>	<code>floor(num / 10) % 10</code>	1
<code>digit3</code>		
<code>digit4</code>		
<code>digit5</code>		
<code>digit6</code>		

The last two rows are worth predicting carefully: what do `digit5` and `digit6` hold when the number has only four digits?

2. **Read the starter code.** The roll is already there, and so is a small `text` call that prints the whole number at the bottom of the canvas, so you can check your digits against it at a glance.
3. **Extract the six digits** into constants `digit1` to `digit6`, with explicit data types.
4. **Draw the six boxes** with `rect`, using the `50 + i * 75` formula. You'll thank yourself in the next two exercises.
5. **Draw the digits** into the boxes with `text`, largest place value on the left: `digit6` in the leftmost box, `digit1` in the rightmost.
6. **Test with the paper number.** Force `num` to 4711 for a moment, like you forced the dice, and compare every box with your paper table. Numbers with fewer than six digits show leading zeros, 004711, and that's correct: the empty places hold zeros.

💡 Exercise: Digits

► [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfd6fbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/digits.yaml>

3.6.5 Apply binary in code

You already practiced the conversion. Binary uses the digits 0 and 1, and its place values are 1, 2, 4, 8, 16, 32, and so on. A single binary digit is called a **bit**. The binary number 1101 means one 8, one 4, no 2, and one 1:

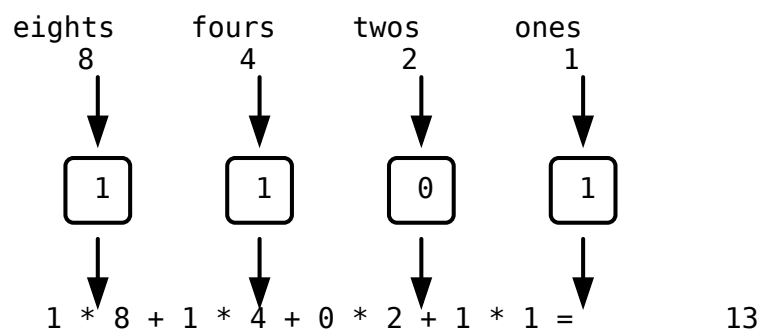


Figure 3.6.3: The binary number 1101 taken apart: the place values are the powers of two, and they add up to 13.

With six binary digits, the largest number you can write is 111111, which is $32 + 16 + 8 + 4 + 2 + 1 = 63$. If that conversion is not clear yet, return to the previous chapter (Section 3.5.4) before you continue. In this exercise, the new question is how your program can extract those six bits.

And the digit-extraction recipe? It carries over beautifully. In decimal you divided by the place values, the powers of ten, and took `% 10`. In binary you divide by the powers of two and take `% 2`:

```
const digit1: number = num % 2;
const digit2: number = floor(num / 2) % 2;
const digit3: number = floor(num / 4) % 2;
```

3.6.6 Your exercise: Binary digits

Same picture, new number system: six boxes, but now they show the random number in binary.

1. **Paper first.** Convert three numbers to binary by hand with the table below. Go from left to right: does a 32 fit into the number? Put a 1 under it and subtract, otherwise put a 0. Then ask the same about the 16, and so on down to the 1. The first row is filled in as a model.

Number	32	16	8	4	2	1	Binary
13	0	0	1	1	0	1	001101
42							
63							

Check each row by adding the place values under your ones back together: $8 + 4 + 1$ is 13, so the first row is right.

2. **Copy your digits solution** into the empty starter and change the number range: the boxes can show at most 63, so the roll becomes `floor(random(0, 64))`.
3. **Change the extraction** from base ten to base two: the divisors 1, 10, 100, ... become 1, 2, 4, 8, 16, 32, and every `% 10` becomes `% 2`.
4. **Test against your paper conversions**, again by forcing `num` to your three practice values. The small decimal number at the bottom is your reference.

💡 Exercise: Binary Digits

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/binary-digits.yaml>

3.6.7 Apply hexadecimal in code

You also practiced the hexadecimal system in the previous chapter. Hex uses 16 digits, with A to F standing for the values 10 to 15. Its place values are 1, 16, 256, and so on. One hex digit represents four bits, so two hex digits represent one byte, from 00 to FF.

Now apply those facts in code. The program must extract three hex digits as numbers, and then translate the values 10 to 15 into the visible letters A to F. This is where number conversion meets the `if` and `else if` chains from this part.

Extracting hex digits is the same recipe a third time, with the divisors 1, 16, and 256, and `% 16` at the end. Typing those place values by hand gets tiring, so here is one more `p5.js` function, `pow`. A call to `pow(base, exponent)` computes a power, so `pow(16, 0)` is 1, `pow(16, 1)` is 16, and `pow(16, 2)` is 256. Written that way, the place number stays visible in the code instead of hiding inside a finished result:

```
const digit1: number = floor(num / pow(16, 0)) % 16;
const digit2: number = floor(num / pow(16, 1)) % 16;
const digit3: number = floor(num / pow(16, 2)) % 16;
```

One thing is genuinely new. Such a digit is a *number* from 0 to 15, but on the screen, 10 to 15 must appear as the letters A to F. A number-to-symbol translation with exact cases is an `if/else if` chain, like the dice faces. The two position arguments of `text` are the center of the box the digit belongs in, from your layout formula:

```
if (digit1 < 10) {
  text(digit1, boxCenterX, boxCenterY);
} else if (digit1 === 10) {
  text("A", boxCenterX, boxCenterY);
} else if (digit1 === 11) {
  text("B", boxCenterX, boxCenterY);
} // ... and so on up to F
```

3.6.8 Your exercise: Hex digits

Three boxes this time, showing the random number in hex.

1. **Paper first.** Convert three numbers to hex by hand with the table below: divide by 256, then by 16, then by 1, each time taking % 16 of the result, and translate 10 to 15 into A to F. The first row is filled in as a model.

Number	256s	16s	1s	Hex
47	0	2	15 -> F	02F
255				
3000				

2. **Adapt your solution:** three digits instead of six on a 300 pixel wide canvas, divisors 1, 16, and 256, extraction with % 16. The three boxes keep their size and their $50 + i * 75$ positions. The biggest three-digit hex number is FFF, which is 4095, so the roll is `floor(random(0, pow(16, 3)))`.
3. **Write the translation chain** for each box: digits below 10 are printed as they are, 10 to 15 as A to F.
4. **Test with forced values**, at least 255 (should read 0FF) and 4095 (FFF).

💡 Exercise: Hex Digits

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdafbcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/hex-digits.yaml>

3.6.9 The project: Binary guesser

Time to put everything from this part into one game. The computer shows a number from 1 to 15 and four boxes holding 0000. The player clicks boxes to flip their bits between 0 and 1 until the four bits show the number in binary. After every click the program checks the guess, and when it's right, the display turns lime green and announces "Correct!". Random numbers, drawing, conditions, binary: everything you've built in this part, working together.

One tool is still missing, and it's the sibling of `||` from the bouncing ball. To decide whether a click landed inside a box, two things must *both* be true: the horizontal position must be in the box's x range *and* the vertical position in its y range. The **AND operator** `&&` combines two boolean questions and is `true` only when both parts are true. Keep the box layout from the digits exercise (Section 3.6.3) and the vertical test writes itself. The row of boxes starts 100 pixels below the top edge and the boxes are 75 pixels high, so their lower edge sits at 175.

```

if (mouseY >= 100 && mouseY <= 175) {
  // the click is at the height of the boxes
}

```

Compare the two siblings. `||` is satisfied by *at least one* true part; `&&` insists on *both*. For the boxes, that means two `&&` checks per click: first whether `mouseY` is in the boxes' row, then, for each box, whether `mouseX` lies between the box's left and right edge. The box-position formula gives you those edges too. Box `i` starts at $50 + i * 75$ and, being 50 pixels wide, ends at $50 + i * 75 + 50$.

i Both operators take a shortcut

`&&` and `||` stop asking as soon as the answer is certain. Take `mouseY >= 100 && mouseY <= 175` and a click at `y = 40`: the first comparison is **false**, and `&&` needs both parts, so the condition can no longer become true and the second comparison is never even looked at. `||` does the mirror image. In the bouncing ball's `circleCenterX >= width || circleCenterX <= 0`, a ball past the right wall makes the first part **true**, one true part is already enough, and the second comparison is skipped. Programmers call this **short-circuit evaluation**. It never changes the result and only saves work, so it costs you nothing to put the question that decides most often on the left.

Two smaller building blocks complete the plan. Flipping a bit is a two-case decision on a global variable:

```

if (guessedDigit1 === 1) {
  guessedDigit1 = 0;
} else {
  guessedDigit1 = 1;
}

```

And checking the guess means converting the four bits back into a number, place value times digit, exactly like the diagrams in this chapter:

```

let guessedNumber: number = guessedDigit4 * 8;
guessedNumber += guessedDigit3 * 4;
guessedNumber += guessedDigit2 * 2;
guessedNumber += guessedDigit1 * 1;

```

Then one `===` comparison against the target number decides between yellow and lime.

The playground gives you an almost empty file for this project, just the roll and the canvas. Build the game in five steps, testing after each:

1. **The start screen.** The starter already rolls the target number into a global variable, because that number has to survive from `setup` into `mouseClicked`. Check its range: a rolled 0 means the four zeros on screen are already the right answer, so make the roll run from 1 to 15. Then draw the question, the four boxes, and the four zeros.

Before you write the click test, work out the four boxes' edges on paper. Each box is 50 pixels wide and starts at $50 + i * 75$:

Box i	Left edge $50 + i * 75$	Right edge, 50 further
0	50	100
1		
2		
3		

These eight numbers are exactly what your `&&` conditions compare `mouseX` against, so a mistake here shows up as a box that ignores every click.

2. **The boxes.** Place the four boxes with the $50 + i * 75$ formula, so their edges match the numbers in your click test.
3. **Click detection.** In `mouseClicked`, write the `&&` conditions, and before any game logic, just print which box was hit. Verify all four boxes and, importantly, that clicks *outside* the boxes do nothing. Only then continue: debugging click ranges is much easier while the program does nothing else.
4. **Flip the bits.** Add the four global bit variables, flip the right one on each click, and redraw the boxes and digits so the player sees the change.

5. **Check the guess.** After each click, convert the bits to a number and compare with the target. On a match, redraw everything in lime and print “Correct!”. Play a few rounds, and then hand the game to someone else: watching another person use your program is the oldest and best test in software.

💡 Exercise: Binary Guesser

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

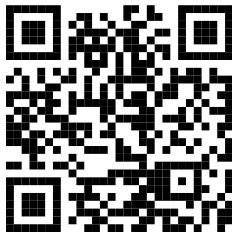
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/binary-guesser.yaml>

3.6.10 Check your understanding

When your binary guesser has survived its first player, take the short quiz below. You answer seven questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Number systems

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/qwawygmofoq>

3.7 Rock, paper, scissors: your first game

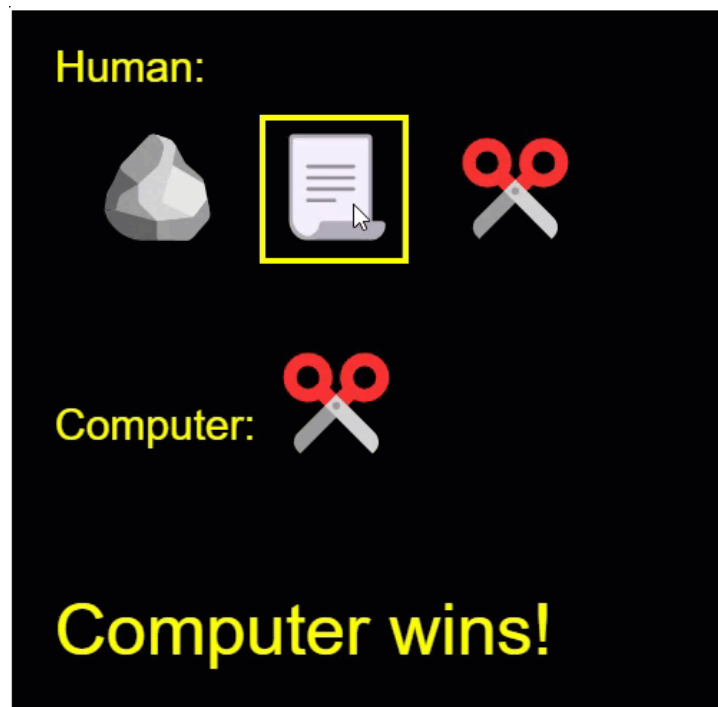


Figure 3.7.1: A finished round: the player picked paper, the computer rolled scissors, and the program announces the winner.

Schere, Stein, Papier. You’ve almost certainly played it in the schoolyard. Two players choose a symbol at the same time, and three simple rules decide who wins. In this chapter the game becomes a program, and your opponent is the computer. You pick rock, paper, or scissors with the mouse, the computer picks by chance, and the program announces the winner. It’s the biggest program of this part, and everything you’ve built so far plays a role in it: event functions, global variables, random numbers, and conditions in every shape you know. You’ll build the game in two stages, and for fast finishers there’s a famous five-symbol variant waiting at the end.

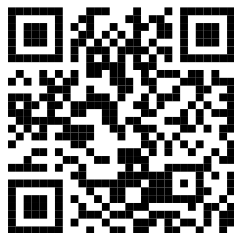
3.7.1 AI tutor

A game is harder to debug than a drawing, because much of what it knows sits invisibly in variables instead of on the canvas. When your game misbehaves, tell the AI tutor what you clicked, what the program showed, and what you expected instead. The tutor knows this chapter and both stages of the exercise.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/aei6o7ko3h>

3.7.2 A plan from the designer

Here is how one round of the finished game plays out:

1. The program shows three symbols next to the label “Human:”.

2. While the mouse rests on a symbol, a yellow frame appears around it. When the mouse leaves, the frame disappears.
3. The player clicks one symbol. That choice is final.
4. The computer immediately shows its own, randomly chosen symbol next to the label “Computer:”.
5. The program prints the result at the bottom: the player won, the computer won, or it’s a tie.

In a software team, a designer would hand you a picture like the one below, and your job as the programmer is to match it as closely as you can. Every red measurement is a number your code will need:

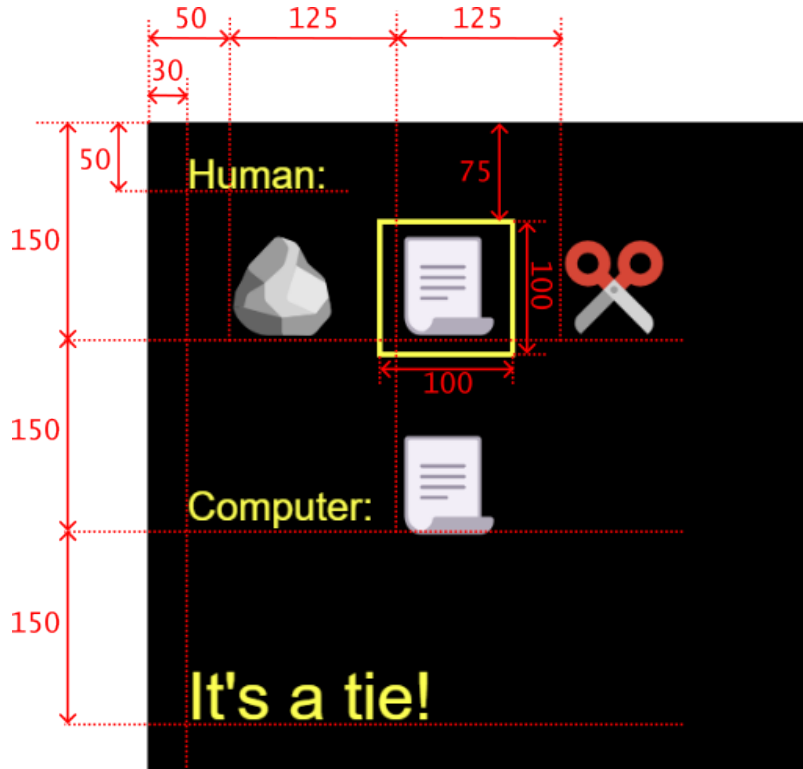


Figure 3.7.2: The UI design of the game. The red lines give every position and size in pixels.

You could type those numbers straight into your drawing calls, but numbers sprinkled through code with no explanation are hard to read. Programmers call them **magic numbers**, and the cure is to give each one a name. The playground starter for this chapter does exactly that:

```
const TEXT_LEFT: number = 30;
const STONE_LEFT: number = 50;
const PAPER_LEFT: number = 175;
const SCISSORS_LEFT: number = 300;
const ICON_WIDTH: number = 100;
const ICON_TOP: number = 75;
const ICON_HEIGHT: number = 100;
```

Constants and their two naming styles are old friends from the snapping chapter (Section 2.2.4). The names are written in ALL_CAPS because these are global configuration constants, settings for the whole program, meant to be found and tweaked easily. Read them against the design picture, where each one is a red measurement with a name. One naming detail so you don’t stumble: the starter says *stone* where this chapter says *rock*; they mean the same symbol.

3.7.3 Emoji live in strings

The three symbols on the canvas are emoji, the same little pictures you know from chats. For your program, an emoji is nothing special. It’s a character, and characters live in strings. A string can hold an emoji alone, and `text` draws it like any other text. The bigger the text size, the bigger the picture:

```
textSize(75);
text("rock", 50, 150);
```


i Emoji on printed paper

The printed book can't show emoji, so the code listings and tables in this chapter print the three symbols as the plain words rock, paper, and scissors. On screen, in the playground and in the web version of this book, you'll see the real symbols instead. The logic of the game is exactly the same either way.

💡 Typing an emoji

On Windows, the key combination Windows + period (.) opens an emoji picker right in the playground editor. On a Mac it's Cmd + Ctrl + Space. And copying an emoji from the exercise description and pasting it into your code works everywhere.

Strings can also do something you've so far only done with numbers: they can be compared. In the dice chapter (Section 3.3.4) you compared numbers with `===` and `!==`. The same two operators work on strings, and they are just as exact. Two strings are equal when they match character for character. So `"rock" === "rock"` is `true`, but `"rock" === "Rock"` is `false`, because `r` and `R` are different characters. Asking “did the player pick the rock?” will be one string comparison.

3.7.4 The hover effect

The yellow frame that follows the mouse is the first thing the game needs, and the right function slot for it is one of your oldest: `p5.js` calls `mouseMoved` on every movement of the mouse across the canvas. Inside it, the program must answer whether the mouse is over a symbol right now.

That's a box test, and you wrote one in the binary guesser (Section 3.6.9). The mouse is inside a box when the horizontal position is in the box's x range *and* the vertical position is in its y range, two questions joined by `&&`. The design gives all three symbols the same vertical band, from `ICON_TOP` down `ICON_HEIGHT` pixels, so the vertical half of the question has the same answer for all three boxes. Answer it once and keep the result:

```
const isInVertical: boolean = mouseY >= ICON_TOP && mouseY < ICON_TOP + ICON_HEIGHT;
```

Look closely at that line, because it shows something new. Until now, a boolean appeared only for a blink: born inside an `if` condition, checked, and gone. But a boolean is a value like any number or string, and a value can be stored. The constant `isInVertical` holds `true` or `false`, its type annotation is `boolean`, and the three box tests can all use it instead of asking the same vertical question three times.

The second new idea is how the frame disappears again. A condition alone can put the yellow frame on, but when the mouse slides off a symbol, something must take the frame off. The trick is to always draw the frame, and let the condition choose only the pen color. Yellow when the mouse is inside; black, the background color, when it isn't. A black frame on a black canvas is invisible, so drawing it erases the highlight:

```
noFill();
strokeWeight(3);
if (isInVertical && mouseX >= STONE_LEFT && mouseX < STONE_LEFT + ICON_WIDTH) {
  stroke("yellow");
} else {
  stroke("black");
}
rect(STONE_LEFT, ICON_TOP, ICON_WIDTH, ICON_HEIGHT);
```

The `rect` call sits *after* the `if/else`, so the frame is drawn on every mouse move, always in the right color. And the `noFill()` at the top matters: without it, the rectangle would be filled and paint over the emoji it's supposed to frame. Paper and scissors each get the same block with their own left edge.

3.7.5 Your exercise, part 1: the three options

Build the first stage: the three symbols and the hover effect. The computer's choice and the winner come in part 2.

1. **Paper first.** Work out the edges of the three boxes from the design picture and the constants. The first row is filled in as a model:

Symbol	Left edge	Right edge	Top	Bottom
rock	50	150	75	175
paper				
scissors				

These twelve numbers are what your `&&` conditions will compare `mouseX` and `mouseY` against. A mistake here shows up later as a box that ignores the mouse, so check each right edge: it's the left edge plus `ICON_WIDTH`.

2. **Read the starter code.** The canvas, the label "Human:", and the constants are already there. One chore while you read: the starter declares its constants without data types, so add the `: number` annotations, like every declaration in this course.
3. **Draw the three symbols** with `text` at text size 75, using the `_LEFT` constants for x and the baseline 150 for y.
4. **Build the hover effect** in `mouseMoved`: the `isInVertical` constant, then one `if/else` plus one `rect` per symbol.
5. **Test with your table.** Glide the mouse slowly across all three symbols and watch the frame follow. Then check the gaps: between the boxes, for example at x values from 150 to 175, no frame should light up.

One warning before you start: the second part of this exercise contains a sample solution for this part. Don't look at part 2 until your part 1 works.

💡 Exercise: Rock, paper, scissors: part 1

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/rps-part-1.yaml>

3.7.6 A game remembers its state

Between the click and the end of a round, the game has to remember things: which symbol the player picked, and which one the computer rolled. Values that must survive from one event to the next belong in global variables, the ones from the color worm (Section 2.5.4). The game needs two, and both hold strings:

```
let selected: string = "";
let computer: string = "";
```

Those quotes with nothing between them are the **empty string**: a string with zero characters. It's a perfectly normal value, and here it plays the role of "nothing yet". As long as `selected` holds the empty string, the player hasn't chosen; the moment a click stores `"rock"` in it, the choice is made. Programmers say these variables hold the program's **state**: everything the program has to remember about where it currently stands.

The state also decides which events still matter, because the rules of the game say a choice is final:

- The hover effect must work only *before* the player has chosen. So the whole body of `mouseMoved` moves inside `if (selected === "") { ... }`.
- A click must count only while the game is still running. So the body of `mouseClicked` sits inside `if (computer === "") { ... }`.

An event function that first asks "am I still allowed to do anything?" is a pattern you'll meet in every game you ever write.

3.7.7 The computer's choice

The computer picks one of three symbols, with equal chances. Rolling a fair whole number is the dice recipe (Section 3.3.2), and `floor(random(0, 3))` gives 0, 1, or 2. But the roll is a *number*, and the game works with symbol *strings*, so the roll must be translated. A number-to-symbol translation with exact cases is an `if/else if` chain, the same move as the hex letters (Section 3.6.7):

```
const computerSymbolId: number = floor(random(0, 3));
if (computerSymbolId === 0) {
  computer = "rock";
} else if (computerSymbolId === 1) {
  computer = "paper";
} else if (computerSymbolId === 2) {
  computer = "scissors";
}
```

After the chain, `computer` holds the computer's symbol, ready to be drawn next to the label "Computer:" and compared against the player's choice.

3.7.8 Who wins?

Three rules decide the game, and every arrow in the picture below means "beats": rock crushes scissors, scissors cut paper, and paper covers rock.

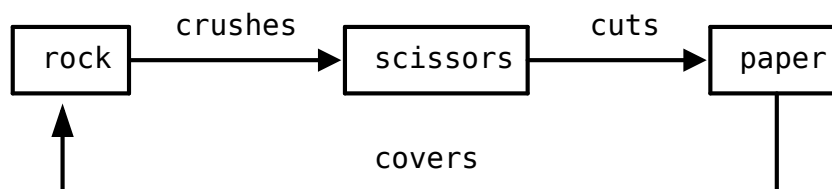


Figure 3.7.3: Who beats whom: each symbol beats exactly one other symbol and loses to exactly one.

Count the possibilities before writing any code. The player has three choices, the computer has three, so a round has 3 times 3, or nine combinations. In three of them both picked the same symbol, a tie. In three others the player's symbol beats the computer's: the player wins. The remaining three go to the computer.

The tie is one comparison, and an elegant one: `selected === computer` compares the two variables *with each other*, and one line covers all three tie combinations at once.

The player's three winning combinations each need two things to be true at the same time, the player's symbol *and* the computer's symbol, so each one is an `&&` pair. And the player wins if the first pair is true *or* the second *or* the third, which is exactly what `||` says (Section 3.4.4). For the first time, both operators meet in one condition:

```
let winner: string = "";
if (selected === computer) {
  winner = "It's a tie!";
} else if (
  (selected === "rock" && computer === "scissors") ||
  (selected === "paper" && computer === "rock") ||
  (selected === "scissors" && computer === "paper")
) {
  winner = "You win!";
} else {
  winner = "Computer wins!";
}
```

Two details in that listing deserve a close look:

- **The parentheses around each `&&` pair are not decoration.** They seal each winning combination into one closed question, so any reader sees immediately which comparisons belong together. Whenever `&&` and `||` meet in one condition, put parentheses around the groups.
- **The condition spans several lines.** TypeScript doesn't mind line breaks inside parentheses, and a long condition becomes readable when every `||` starts a new line.

And the final `else`? That's the "everything else" branch you know from the dice chapter: nine combinations, minus three ties, minus three player wins, leaves exactly the three combinations where the computer wins. No condition needed.

3.7.9 Your exercise, part 2: the whole game

Now finish the game: the click, the computer's roll, and the winner.

1. **Compare first.** The starter code of part 2 is a sample solution for part 1. Before you write anything, compare it with your own part 1: where did you decide differently? Keep whichever version you like better, but carry over your data type annotations.
2. **Play computer on the winner logic.** Copy the table and fill it in using the listing from Section 3.7.8: which branch is the first to be true, and what does the program print? The first row is filled in as a model.

selected	computer	First true branch	Message
rock	scissors	first <code>&&</code> pair of the <code>else if</code>	You win!
paper	paper		
scissors	rock		
paper	rock		
scissors	paper		

3. **Recognize the click.** Add the two global state variables. In `mouseClicked`, reuse the box test from the hover effect to find out which symbol was clicked, and store that symbol in `selected`. A click that hits no symbol must leave the game unchanged, so everything that follows, the roll and the result, runs only when `selected !== ""`. Test it: clicking the empty canvas must do nothing.
4. **Make the choice final.** Guard the two event functions with the state, as described in Section 3.7.6: hover only while `selected === ""`, clicks only while `computer === ""`.
5. **Roll the computer's symbol** with the chain from Section 3.7.7 and draw it as the design shows: the label "Computer:" at text size 30, the symbol at text size 75.
6. **Announce the winner** with the logic from Section 3.7.8, at text size 50 near the bottom edge.
7. **Test all nine combinations.** Chance won't show you all three messages quickly, so force the roll like you forced the dice: replace `floor(random(0, 3))` with 0, 1, or 2 for a moment and play against a computer whose move you know. Check your play-computer table row by row. Then restore the roll and hand the game to a classmate: watching another person use your program is the oldest and best test in software.

💡 Exercise: Rock, paper, scissors: part 2

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdafbcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/rps-part-2.yaml>

3.7.10 For fast finishers: rock, paper, scissors, lizard, Spock

Maybe you know the series *The Big Bang Theory*, where Sheldon Cooper plays a five-symbol variant of the game. You can [watch him explain it](#) ([German version](#)). The rules:

- rock crushes scissors, and rock crushes lizard
- paper covers rock, and paper disproves Spock
- scissors cut paper, and scissors decapitate lizard
- lizard eats paper, and lizard poisons Spock

- Spock smashes scissors, and Spock vaporizes rock

Every symbol beats two others and loses to two others.

If the game left you wanting more, rebuild it with five symbols. There is no walkthrough and no sample solution for this stage, on purpose. This time you are the designer. Plan on paper before you code, the way this course always does, and let these questions guide you: which emoji will you use for lizard and Spock? Where do five boxes sit, when the current canvas comfortably fits three? How many combinations does a round have now, and how many of them make the player win? The starter code is the finished three-symbol game, so everything you change, you understand.

💡 Exercise: Rock, paper, scissors, lizard, Spock

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

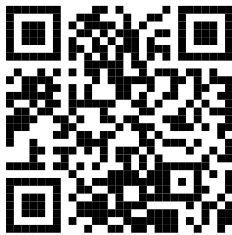
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0030-Conditions/rps-part-3.yaml>

3.7.11 Check your understanding

When your game has won (or graciously lost) its first round against a classmate, take the short quiz below. You answer seven questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Rock, paper, scissors

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/0924i0kd11>

3.8 The switch statement: many cases, one question

```
function setup() {
  createCanvas(200, 200);
  background("black");

  const dice = Math.floor(random(1, 7));

  let message: string;
  if (dice === 1) { message = "Eins"; }
  else if (dice === 2) { message = "Zwei"; }
  else if (dice === 3) { message = "Drei"; }
  else if (dice === 4) { message = "Vier"; }
  else if (dice === 5) { message = "Fünf"; }
  else { message = "Sechs"; }

  textAlign(CENTER, CENTER);
  noStroke();
  fill("yellow");
  textSize(50);
  text(message, width / 2, height / 2);
}

function setup() {
  createCanvas(200, 200);
  background("black");

  const dice = Math.floor(random(1, 7));

  let message: string;
  switch (dice) {
    case 1: // <<< The switc
            // <<< Add one c
            message = "Eins";
            break; // <<< At the en
    case 2:
            message = "Zwei";
            break;
    // ...
    default: // <<< The defau
            message = "Sechs";
            break;
  }

  textAlign(CENTER, CENTER);
  noStroke();
  fill("yellow");
  textSize(50);
  text(message, width / 2, height / 2);
}
```

Figure 3.8.1: The same dice program twice: on the left the if/else-if chain you know, on the right the new switch statement doing exactly the same job.

Look back at this part for a moment. The dice program picked one face out of six with a chain of `===` comparisons. The hex exercise translated the digit values 10 to 15 into the letters A to F with a chain (Section 3.6.7). The rock-paper-scissors game turned the computer's roll into a symbol, with a chain again (Section 3.7.7). Three programs share one shape, a single variable compared against one exact value after another. Programmers meet this shape so often that TypeScript has a statement built just for it: **switch**. This closing chapter of the Conditions part teaches you nothing new to compute, only a tidier way to write decisions you already make. And you'll practice it on an old friend, the dice.

3.8.1 AI tutor

The tutor of this part knows the switch statement too. If a case never seems to run or a message comes out wrong, show the tutor your switch and tell it what you rolled and what appeared.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/aei6o7ko3h>

3.8.2 From chain to switch

To have something concrete to rewrite, here is a small variant of the dice program. It rolls the die exactly like the dice chapter (Section 3.3.3), but instead of drawing a face, it prints the rolled number as a German word:

```
const dice: number = floor(random(1, 7));

let message: string;
if (dice === 1) { message = "Eins"; }
else if (dice === 2) { message = "Zwei"; }
else if (dice === 3) { message = "Drei"; }
else if (dice === 4) { message = "Vier"; }
else if (dice === 5) { message = "Fünf"; }
else { message = "Sechs"; }
```

Nothing in this chain is new: exact comparisons with `===` (Section 3.3.4), a final `else` that is safe because `dice` can only hold 1 to 6, and every branch with its braces, even on one line.

But read the chain out loud and you'll hear the repetition, "dice equals, dice equals, dice equals", five times the same variable, five times the same question. The **switch statement** says the repeated part once:

```
switch (dice) {
  case 1:
    message = "Eins";
    break;
  case 2:
    message = "Zwei";
    break;
  // ... the cases 3, 4, and 5 ...
  default:
    message = "Sechs";
    break;
}
```

Read it top to bottom:

- `switch (dice)` names the value to look at, once. Everything inside the braces asks questions about `dice`.
- Each `case` is one exact comparison: the line `case 3:` asks the same question as `dice === 3`. The computer walks the cases from top to bottom and jumps to the first one that matches.
- After the colon come the statements of that case, and `break` ends the case: leave the switch, continue after its closing brace.
- `default` is the case without a value. More about it below.

3.8.3 break: the way out

A case does not end by itself. The `break` statement is what sends the computer out of the switch, and forgetting it is the classic switch mistake. Without a `break`, the computer finishes the case's statements and simply keeps running, straight into the statements of the *next* case, without asking whether that case matches. Programmers say the execution *falls through*.

Play computer with it: delete the `break` after `message = "Eins";` and roll a 1. The computer jumps to `case 1`, stores "Eins", and then, with no `break` to stop it, runs on into `case 2` and stores "Zwei". Only the `break` there ends the switch. The program shows "Zwei" although it rolled a 1.

A sneaky bug

A missing `break` is not an error for TypeScript: the program runs, there is no red squiggle, and the drawing simply shows the wrong result. When a switch behaves strangely, check the ends of your cases first, one `break` per case.

3.8.4 default: the else of the switch

The `default` case runs when no other case matched. That is exactly the job the plain `else` did at the end of the chain: after five nos, only the 6 is left, so the word `dice` can write `default` instead of `case 6`. Like every case, the `default` ends with a `break`, so all branches of the switch look the same.

The `default` case is optional. A switch without one is fine: when no case matches, the switch simply does nothing, and the program continues after the closing brace. You'll see both designs in this chapter's exercises, one switch with a `default` and one without.

3.8.5 When switch fits, and when it doesn't

A switch asks only one kind of question: "is the value exactly this?", the `===` question. That makes it great for chains of exact cases, and useless for everything else. A range like `mouseX < 100` from the color areas cannot be a case, and neither can the `||` conditions of the bouncing ball or the `&&` box tests of your games. Those decisions stay with `if`.

Here's the rule of thumb. When you catch yourself writing three or more `else if` lines that compare the *same* variable with `===`, the chain is a switch candidate. The compared value doesn't have to be a number, by the way: a switch can also ask about a string, with the same character-for-character exactness you know from comparing strings with `===`.

One more reason to learn the shape by heart: a similar `switch` exists in C, Java, and C#, the languages your semicolon habit is already preparing you for. Some languages grow the idea further. C# and Rust, for example, offer more powerful variants that can match whole patterns, not just single values. The shape you learn here is the foundation for all of them.

3.8.6 Your exercise: the dice in words

The first exercise is the word dice from this chapter. The starter code contains the finished `if/else if` chain; your job is to replace it with a switch.

1. **Read the starter code.** Two small chores while you read: the starter rounds with `Math.floor`, a spelling you haven't met. It does the same as the `floor` you know, but this course always uses the `p5.js` helpers, so change it to `floor`. And the `dice` declaration is missing its data type; add `: number`, like every declaration in this course.
2. **Rewrite the chain as a switch**, one case at a time. Start with `switch (dice) {` and the case for the 1, run the program, and only then add the next case. One change, one run: exactness first.
3. **Turn the final else into the default.** The chain's last branch has no condition, and the switch's `default` is its twin.
4. **Test with forced rolls**, like you tested the dice faces (Section 3.3.6): replace the roll with `const dice: number = 1`; for a moment, check the word, and work through all six values. Then put the `random` line back.

💡 Exercise: Simple dice

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0040-Switch/simple-dice.yaml>

3.8.7 Your exercise: the dice faces with switch

The second exercise is a program you have already built once: the dice with its six faces from the dice exercise (Section 3.3.6). The roll, the anchors, and the `circle` calls all stay as they were; only the decision around them changes from a chain to a switch. Get your paper table of the nine anchor points out again; it is still the plan.

1. **Read the starter code.** The switch is started, and the cases for the 1 and the 2 are done. Compare the case for the 2 with your own dice solution: the `circle` calls are identical, only the frame around them changed. Add the missing `: number` to the `dice` declaration while you're there.

2. **Add the cases 3 to 6**, one at a time, with a **break** at the end of each. Your table tells you which anchors each face uses, and the reuse you found on paper still works: the 3 is the 2 plus the center, the 5 is the 4 plus the center.
3. **Write the 6 as a normal case.** This switch has no **default**, and it doesn't need one: **dice** always holds one of the six values, so one case per value covers every roll.
4. **Test with forced rolls**, face by face, against your paper table. Then restore the roll and enjoy a die that runs on a switch.

💡 Exercise: Dice with switch

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0040-Switch/dice-switch.yaml>

3.8.8 For fast finishers: one more chain

One switch candidate from this part is still out there. The rock-paper-scissors game translates the computer's roll into a symbol with an **if/else if** chain (Section 3.7.7), three exact cases on one variable. Open your finished game and rewrite that chain as a switch. The game must behave exactly as before, and you have a tester who knows it well: you.

3.8.9 Check your understanding

When both dice run on a switch, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: The switch statement

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/s6f6hdt01x>

Part 4

Loops

4.1 The while loop: teaching the computer to repeat

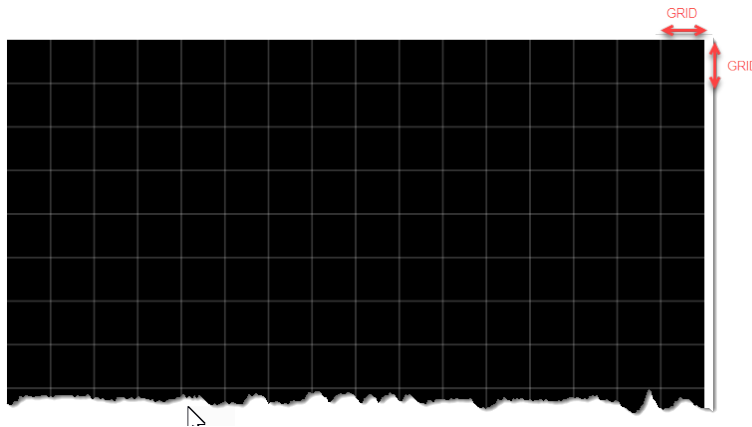


Figure 4.1.1: A gray grid on a black canvas: 15 vertical and 15 horizontal lines. Each set of lines comes out of a loop that is only four lines of code long.

Look at the drawing above and count what it takes. It needs 15 vertical and 15 horizontal lines, 30 `line` statements that differ in exactly one number. You could type them all. You typed six `circle` calls for the dice faces, and it already felt like work. This part of the book gives you the tool that makes such typing unnecessary: the **loop**, a way to tell the computer “do this again, with the next number, until I say stop”. Loops are the biggest step in this course so far. A program with loops can do thousands of things with a few lines of code, and from now on your drawings can be as rich as you like without your fingers paying for it.

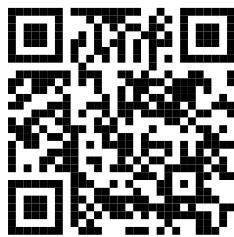
4.1.1 AI tutor

Loops are new territory, and the first loop that runs away or draws nothing can be confusing. The tutor of this part knows everything about the while loop. Show it your loop and tell it what you expected and what you got.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/ctck301mbv>

Watch out for one danger before you write your very first loop. A loop repeats until its condition says stop, and it is surprisingly easy to write a loop whose condition never says stop. Such a loop is called an **endless loop** (programmers also say *infinite loop*), and everyone who learns loops produces one sooner or later. The playground cannot rescue you from it: the browser tab freezes, the buttons stop reacting, and the only way out is to close the tab and open the playground again. Everything you typed since your last save is gone at that moment.

❗ Save before you run a loop

The playground can save your work locally in the browser or in GitHub. Save often, and always save right before you run code with a loop you are not completely sure about. Saving takes a second; typing a lost program in again takes a lot longer.

4.1.2 Thirty statements, one idea

Here is the honest, loop-free way to draw the vertical lines of the grid, with the canvas 400 pixels wide and a line every 25 pixels:

```
line(25, 0, 25, 400);
line(50, 0, 50, 400);
line(75, 0, 75, 400);
// ... and twelve more of these ...
```

Every statement is the same except for one number, and that number grows by 25 each time. Reading it out loud sounds like a robot: “line at 25, line at 50, line at 75”. And the worst part comes later. If you want a finer grid with a line every 20 pixels, you have to rewrite every single statement.

What you’d rather tell the computer is the *idea* behind the column: “start at 25; as long as you haven’t reached the right edge, draw a line and move 25 to the right.” That sentence is a while loop, almost word for word.

4.1.3 The while loop: four steps

Here is that sentence in TypeScript. `SIZE` is the canvas size (400) and `GRID` is the distance between lines (25), both constants:

```
let i: number = GRID;    // STEP 1: Initialize the loop variable
while (i < SIZE) {        // STEP 2: Check the loop condition
    line(i, 0, i, SIZE);  // STEP 3: Do whatever you want to do repeatedly
    i += GRID;            // STEP 4: Update the loop variable
}
```

The variable `i` is called the **loop variable**. It is the loop’s memory: it stores how far the work has come. Here it holds the `x` position of the next line to draw. The four steps around it:

1. **Initialize.** Before the loop starts, the loop variable gets its starting value. The first line belongs at `x = GRID`, so `i` starts at `GRID`.
2. **Check.** The **condition** between the parentheses works exactly like the condition of an `if`: a comparison that is `true` or `false`. As long as it is `true`, the loop keeps going. `i < SIZE` reads as “the next line is still on the canvas”.
3. **Do the work.** The statements between the braces are the **loop body**. They run once per round. Our body draws one vertical line at `x = i`.
4. **Update.** The last statement of the body moves the loop variable forward: `i += GRID` shifts the position one grid step to the right. You know `+=` from the bouncing ball (Section 3.4.3).

After the update, the computer jumps back to step 2 and checks the condition again, with the new value of `i`. Check, work, update, check, work, update, until the check says `false`. Then the computer skips past the closing brace and the program continues.

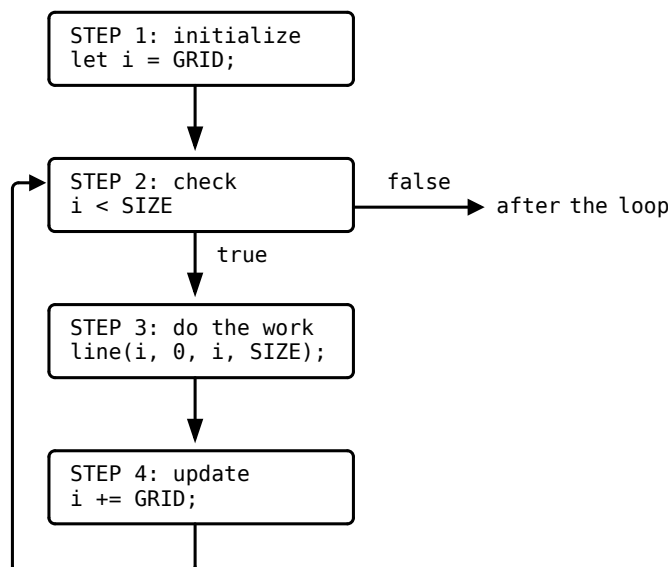


Figure 4.1.2: The while loop as a cycle: initialize once, then check the condition, do the work, and update, around and around until the check says false.

One detail of the picture deserves a second look. The check comes *before* the work, every time. When the condition is **false** at the very first check, the body never runs at all. A while loop can run zero times, and sometimes that is exactly right: on a canvas smaller than one grid step, the correct number of lines is none.

! Course rule: the loop variable gets a data type too

`let i: number = GRID;` declares a variable like every other declaration, so it carries its data type. The rule from the Variables part (Section 2.2.3) has no exception for loops, and that stays true for every loop shape in this book. Code you find on the web often leaves the type out in a loop, because TypeScript can work it out on its own. In this course you write it, every time.

i Why is the variable called *i*?

i is short for *index* and is the traditional name for a loop variable; programmers have used it for generations, and everyone reads *i* as “the loop is counting here”. For a loop variable with a clear meaning, a speaking name like *x* or *diameter* is just as good. Meaningless names like *temp*, *helper*, *thing*, or *stuff* stay forbidden for everything else, because the naming rules from the Variables part (Section 2.2.4) still apply.

4.1.4 Play computer: the loop table

You can’t watch the computer run a loop, but you can *be* the computer. Take paper and pencil and walk through the loop one round at a time, writing down the loop variable, the check, and the work. The result is a **loop table**. To keep it short, shrink the numbers first: here is the vertical-line loop with `SIZE = 100` and `GRID = 25`.

Table 4.1.1: The loop table for the vertical lines with `SIZE = 100` and `GRID = 25`.

i	i < 100?	what happens
25	true	line at x = 25, then i becomes 50
50	true	line at x = 50, then i becomes 75
75	true	line at x = 75, then i becomes 100
100	false	the loop ends

Three lines, at 25, 50, and 75. Notice the two edges of the table. The loop starts at `GRID`, not at 0, because a line at `x = 0` would lie on the canvas border where nobody sees it. And the line at `x = 100` is never drawn, because `100 < 100` is **false**: the condition `<` keeps the last line off the right border. Both edges are design decisions, and the loop table makes them visible before the program ever runs.

! Course rule: play computer with every new loop

Before you run a loop you just wrote, trace it in a loop table with small numbers, like the table above. The table costs two minutes and shows you the first value, the last value, and the number of rounds. Most loop bugs, wrong start, wrong condition, forgotten update, are visible on paper before the browser ever sees the code. The following chapters won't repeat this instruction every time; the technique is yours now, and it helps with every loop you write from now on.

4.1.5 The loop that never stops

What happens when step 4 is missing?

```
let i: number = GRID;
while (i < SIZE) {
  line(i, 0, i, SIZE);
  // the update is missing
}
```

Play computer: *i* is 25, the check says **true**, a line appears, and *i* is... still 25. Check **true**, line, still 25. The condition can never become **false**, so the loop never ends. This is the endless loop from the start of the chapter (Section 4.1.1), and now you can see exactly how one happens. The classic cause is a forgotten update.

! The frozen tab

An endless loop shows no error message and no red squiggle: the code is legal, it just never finishes. The browser tab simply freezes, the drawing stays blank, and clicking does nothing. When that happens, close the tab, open the playground again, and read your loop in four-step order: is the variable initialized? Can the condition ever become **false**? Does the update actually move the variable toward the end? And save before you run the repaired loop, in case the second try freezes too. Try to produce an endless loop on purpose in the playground and see what happens. It will prepare you for a mistake that you will make sooner or later.

4.1.6 Two loops, one variable

The grid needs a second set of lines, and the horizontal loop is the mirror image of the vertical one. Only the `line` arguments change. A horizontal line at height *i* runs from the left edge to the right edge, so start and end swap their roles: `line(0, i, SIZE, i)`.

The second loop can reuse the variable of the first one. But careful: *i* already exists, so declaring it again with `let` gives you a red squiggle. A plain assignment resets it instead:

```
i = GRID; // back to the start: assignment, no let
while (i < SIZE) {
  line(0, i, SIZE, i);
  i += GRID;
}
```

`let i` happens once; after that, the variable belongs to the whole function and every loop in it may use it. Resetting the loop variable before the loop is just step 1 in different clothes.

Reusing *i* works here because the two loops run one after the other. When a program really needs two counter variables at the same time, programmers traditionally call the second one *j*, simply the next letter of the alphabet. Everyone reads *i* and *j* as a pair of counters, just like everyone reads *i* alone as “the loop is counting here”.

4.1.7 Your exercise: Grid

Now build the grid yourself. The starter code contains the finished vertical-line loop, with the four steps marked as comments. Your job is the horizontal lines.

1. **Read the starter code.** Find the four steps of the vertical loop and match them to the comments. Say the loop out loud: “start at 25, while the edge isn’t reached, draw and move on.”
2. **Write the horizontal loop.** Reset *i* with an assignment, keep the condition and the update, and swap the `line` arguments as shown above (Section 4.1.6). Run after every few keystrokes.

3. **Change the constants.** Set `GRID` to 50, then to 20, then put `SIZE` to 300. The grid must adapt by itself every time; that is the point of writing loops with constants instead of 30 statements with numbers in them.

💡 Exercise: Grid

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/grid.yaml>

4.1.8 Your exercise, part 2: Step Lines

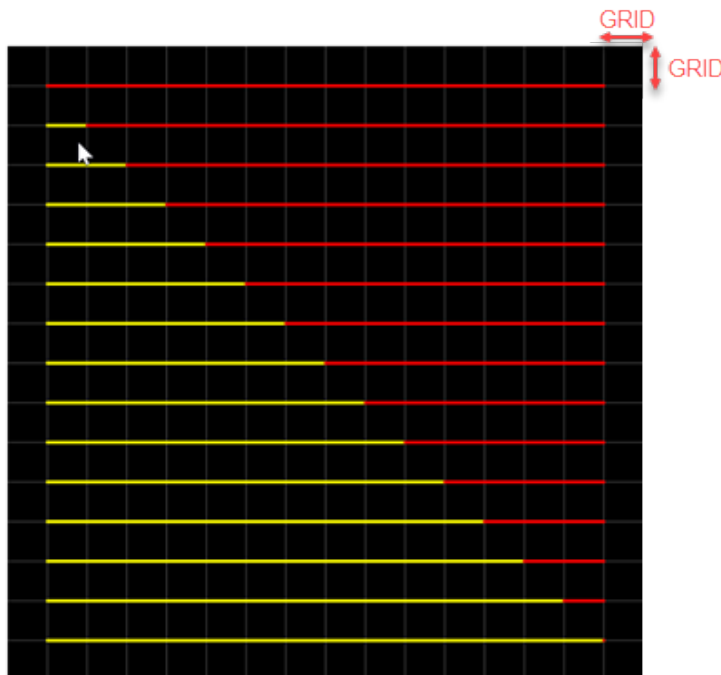


Figure 4.1.3: The grid again, now with a yellow staircase growing from the left and a red staircase shrinking toward the right. The two staircases meet on the diagonal.

The second exercise starts where the grid ends, with both grid loops already in the starter code. On top of the grid you draw colored lines, one per row, and their lengths change from row to row. That is the new idea of this exercise: the loop variable can do more than one job. i is the height of the row, and at the same time it decides how long the yellow part is.

Look at the picture: in every row, yellow and red meet on the diagonal, and the diagonal is exactly where x equals y . So in the row at height i , the yellow line runs from $x = \text{GRID}$ to $x = i$, and the red line continues from $x = i$ to $x = \text{SIZE} - \text{GRID}$.

1. **Read the given code.** The two grid loops are your own solution. Below them, the starter sets `strokeWeight(2)` and `stroke("yellow")` once, before your loop. Settings placed before a loop apply to every round, so all yellow lines get the same look without repeating the command inside the body.
2. **Make a loop table** for the yellow loop with `SIZE = 100` and `GRID = 25`, with one extra column: where does the yellow line start, where does it end? The first row's line is tiny (from 25 to 25), and that is fine.
3. **Yellow first.** Write the yellow loop and run it. Only when the yellow staircase looks right, add the red loop after the `stroke("red")` line.
4. **Change the constants.** `GRID = 50`, `GRID = 20`, `SIZE = 300`: the staircases must follow the grid by themselves.

💡 Exercise: Step Lines

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/step-lines.yaml>

4.1.9 Check your understanding

When your grid and your staircases work, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: The while loop

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/uhm8axeg61>

4.2 Rays: loops meet the mouse

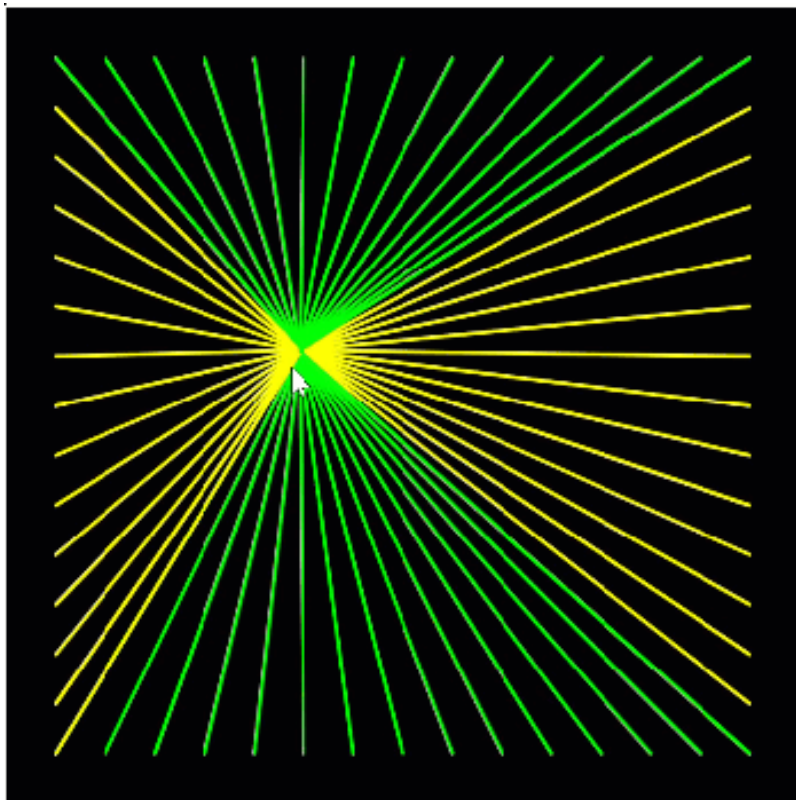


Figure 4.2.1: Sixty rays follow the mouse: lines run from anchor points on all four canvas edges to wherever the mouse is right now. In the playground, the picture moves with the mouse.

The grid from the last chapter is calm. Two loops run once inside `setup`, the picture is done, nothing moves. This chapter wakes the loops up. You put a while loop inside an event function, and suddenly 60 lines redraw themselves every time the mouse moves, like threads pinned to the edges of the canvas. Then you go one step further and give the rays a color that never stands still, cycling through the color wheel like the lighting of a gaming keyboard. The loop itself stays exactly the one you know: four steps, nothing more. What changes is where it lives and what happens around it.

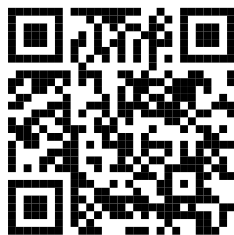
4.2.1 AI tutor

Two ideas meet in this chapter: a loop and an event. If your rays don't show up, smear across the canvas, or wear the wrong colors, tell the tutor which of the chapter's two exercises you are in and show it your loop.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:
<https://app.novedu.at/ctck301mbv>

4.2.2 A loop inside an event

So far, loops ran once, in `setup`. But a loop is a statement like any other: it may live in any function. When a loop lives in `mouseMoved` (Section 2.1.4), it runs again on every mouse move, drawing its whole picture fresh

each time. That is exactly how the rays work. Here is the plan of such a program, with the loop itself still missing:

```
function mouseMoved() {
    background("black");
    strokeWeight(2);

    // the loop draws all rays for the current mouse position
}
```

The first statement repaints the background. You know that move from the bouncing ball (Section 3.4.2): repaint first, then draw. Without it, every mouse position would leave its rays on the canvas, and after a second of movement the picture would be a smeared tangle of old lines.

4.2.3 One variable, four lines

Where do the rays start? Look at the goal picture. The starting points sit evenly spaced along all four edges, one grid step apart, with a margin of 25 pixels. Those are the same positions your grid loop visited. The new idea is that in each round, the loop variable *i* marks four points at once, one per edge:

- (*MARGIN*, *i*) on the left edge, and (*width* - *MARGIN*, *i*) on the right edge: *i* is the *height* of the point.
- (*i*, *MARGIN*) on the top edge, and (*i*, *height* - *MARGIN*) on the bottom edge: *i* is the *horizontal position* of the point.

Every ray ends at the same point: (*mouseX*, *mouseY*). So one round of the loop draws four lines, and the body sets two colors while it works, yellow before the left-and-right pair and lime before the top-and-bottom pair. In the grid, the color was set once before the loop, because every line looked the same. Here two colors take turns inside every round, so the `stroke` calls move into the body.

One more difference from the grid loop is easy to miss: the condition. The grid stopped with *i* < *SIZE*, keeping the last line off the border. The rays want anchor points *on* the last position, *SIZE* - *MARGIN*, so the loop runs while *i* <= *SIZE* - *MARGIN*. Whether < or <= is right is not a grammar question; it is a design decision, and your loop table shows you which one you need. Write down the last value you want, and check which comparison still lets it through.

4.2.4 Your exercise: Rays

1. **Make a loop table** for *SIZE* = 100 and *MARGIN* = 25, with the condition *i* <= *SIZE* - *MARGIN*. Which values does *i* take, and how many rounds is that? Does the last anchor position, *SIZE* - *MARGIN*, get through?
2. **Read the starter code.** It is the plan from the beginning of the chapter (Section 4.2.2), ready to go: the constants *SIZE* and *MARGIN* are declared, and `mouseMoved` already repaints the background and sets the stroke weight. A comment marks the spot where your loop belongs.
3. **Start with one edge.** Set `stroke("yellow")` before the loop for now, and write the loop with a single statement: `line(MARGIN, i, mouseX, mouseY);`. Run it and move the mouse. One fan of yellow rays from the left edge, nothing else.
4. **Add the other three edges**, one `line` statement per edge, and move the colors into the loop body: yellow before the left-and-right pair, lime before the top-and-bottom pair.
5. **Change the constants.** *MARGIN* = 50, then *SIZE* = 300. The fans must follow by themselves.

💡 Exercise: Rays

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/rays.yaml>

4.2.5 A color that never stands still

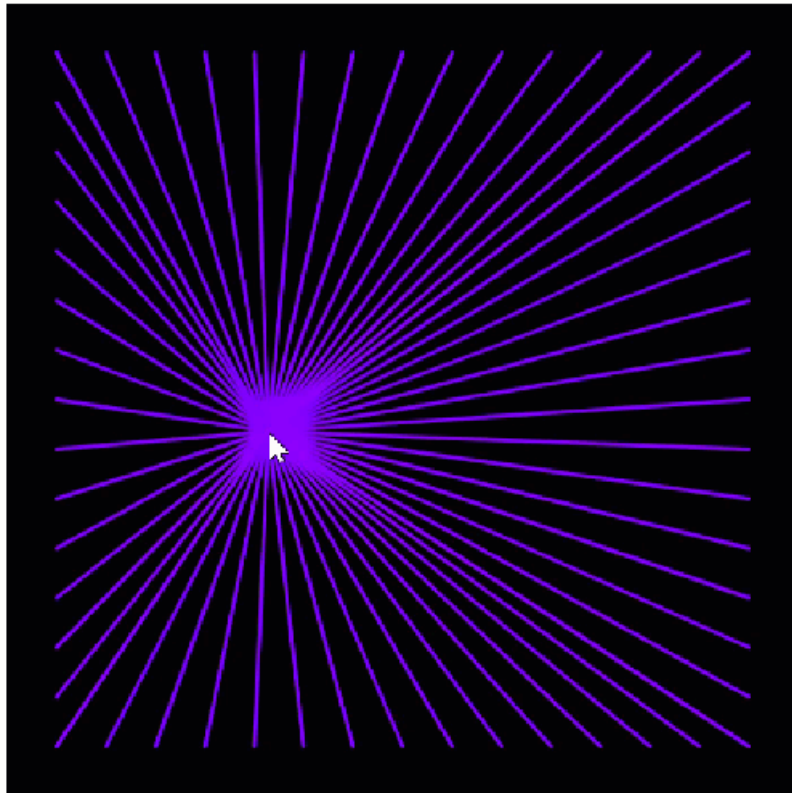


Figure 4.2.2: The same rays, but the color wanders through the color wheel on its own, whether the mouse moves or not. The printed page shows one moment; in the playground the color keeps changing.

For the second exercise, the rays should change color continuously, like RGB lighting. That needs two changes in how the program is built, and both reach back to earlier chapters.

First, the loop moves from `mouseMoved` into `draw` (Section 3.4.2). `mouseMoved` runs only when the mouse moves, so a color animation would freeze the moment you let go of the mouse. `draw` runs about 60 times per second, no matter what, and the starter code of the exercise has this change already made. The rays still follow `mouseX` and `mouseY`; those variables are always up to date, in any function.

Second, the color needs a memory. A color that *changes* is a value that must survive from one frame to the next, and you know the tool for that: a global variable (Section 2.5.4), declared outside every function.

```
// The current hue value of the rays
let rayColor: number = 0;
```

The color itself is a hue on the HSB color wheel (Section 2.4.4), so `setup` gets a `colorMode(HSB)`; and the loop paints with `stroke(rayColor, 100, 100)`; full saturation, full brightness. And after every ray, the hue takes a tiny step:

```
rayColor = (rayColor + 0.25) % 360;
```

Read the right side inside out: take the current hue, add a quarter step, and then `% 360`. The modulo operator is an old friend from digit extraction (Section 2.5.5), and here it does its second job, wrapping. As long as the

sum stays below 360, % 360 changes nothing. The moment it reaches 360, the remainder snaps back to 0, which is exactly right, because 360 and 0 are the same point on the color wheel. A variable that grows step by step like this, keeping its value between rounds and between frames, is called a **running value**, and you will meet it in many programs, counting points, summing prices, or turning a color wheel.

i Why do the rays make a rainbow?

The update line sits *inside* the loop, so the hue moves a quarter step from one ray to the next. Within a single frame, neighboring rays differ slightly in color, which gives the fan its soft gradient. If the update ran *after* the loop instead, every ray in one frame would get the same color, and the fan would change color as a whole, one hue per frame. Both are valid designs. The exercise builds the first one, and trying the second for a moment is a great experiment.

4.2.6 Your exercise, part 2: Colored Rays

The starter code is a finished rays program, already living in **draw**. The exercise text walks you through every snippet; your job is to place them correctly and understand what each one does.

1. **Read the starter code.** Find the one difference to your own rays solution: the function is called **draw** now, not **mouseMoved**.
2. **Set the color mode.** Add `colorMode(HSB);` to **setup**.
3. **Add the global variable.** Declare `rayColor` outside every function, and give it its data type: the exercise shows `let rayColor = 0;`, and in this course every declaration carries its type, so write `let rayColor: number = 0;`.
4. **Recolor the loop.** Remove the two old **stroke** calls and put `stroke(rayColor, 100, 100);` at the top of the loop body.
5. **Add the update** `rayColor = (rayColor + 0.25) % 360;` at the end of the loop body, and run. The fan should slowly turn through the color wheel.
6. **Experiment.** Change 0.25 to 5: faster cycling, stronger gradient. Then move the update after the loop for a moment and watch the gradient disappear. Put it back where the exercise wants it.

i Exercise: Colored Rays

► [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/colored-rays.yaml>

4.2.7 Check your understanding

When your rays follow the mouse in ever-changing colors, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Rays

► [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/c7euteh7kp>

4.3 Pixel X and zebra stripes: the loop variable works harder

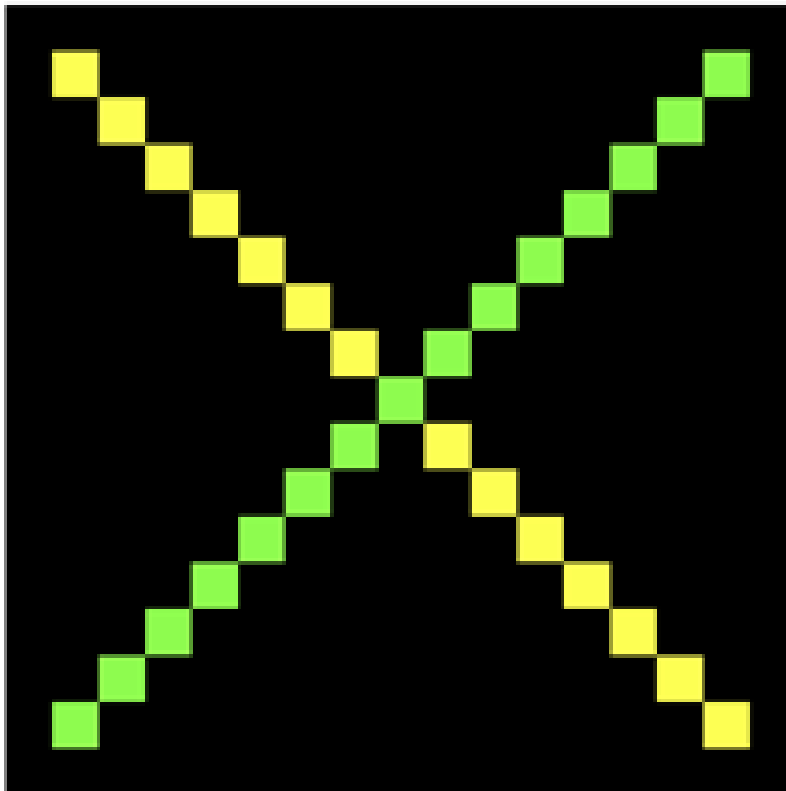


Figure 4.3.1: An X made of yellow and lime squares, like giant pixels: one diagonal runs from the top left down to the bottom right, the other from the bottom left up to the top right.

The grid gave you a finished loop to copy, and the rays came with a half-written program. This chapter takes the training wheels off: two exercises, two empty canvases, and you write every loop yourself. Both drawings look simple, and both hide a new skill. The pixel X teaches you to *compute* with the loop variable, so that one variable places two squares in one round. The zebra stripes teach you a second style of loop, where the variable counts rounds instead of measuring pixels, and where an `if` inside the loop body lets every second round act differently.

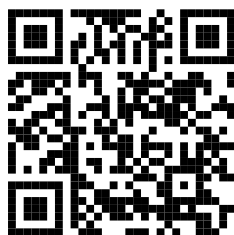
4.3.1 AI tutor

Writing a loop from scratch is different from completing one. If your diagonal bends the wrong way or your stripes are all one color, show the tutor your loop and your loop table; the mistake is usually visible in the table.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/ctck301mbv>

4.3.2 Computing with the loop variable

Start with the first diagonal of the X. The canvas is 425 pixels, each square 25, and the X keeps one empty cell on every side, so the squares sit at 25, 50, and so on up to 375. On the first diagonal a square sits at every position where x and y are equal, so one `rect` with the loop variable in both slots draws the whole diagonal:

```
rect(i, i, GRID, GRID);
```

The second diagonal is the mirror image. When the yellow square is near the top, the lime square is near the bottom. Mirroring a position means subtracting it from the canvas height, roughly `height - i`. But remember how `rect` measures: its first two arguments are the *top left corner* of the rectangle. So `height - i` would put the square's top edge where its bottom edge belongs, one cell too low. The correct top edge sits one grid step higher:

```
rect(i, height - GRID - i, GRID, GRID);
```

One loop, one variable, and each round places a yellow square at `(i, i)` and a lime square at `(i, height - GRID - i)`. The loop variable is no longer just a position you draw at; it is a value you compute with.

💡 Test expressions at the edges

Before you trust an expression like `height - GRID - i`, feed it the first and the last value of the loop. In the first round, with `i = 25` and height 425, the lime square lands at `y = 375`, one cell above the bottom edge. In the last round, with `i = 375`, it lands at `y = 25`, one cell below the top edge. Both ends look right, so the middle will be right too. Two small calculations catch most mirror mistakes.

4.3.3 Your exercise: Pixel X

The canvas is 425 pixels, the squares are 25, and the X keeps a margin of one empty cell on every side. The starter code contains only the canvas setup and `noStroke()`.

1. **Plan the loop.** The canvas is 17 cells of 25 pixels, and the margin eats the first and the last cell: the first square sits at (25, 25), the last at (375, 375). So your loop runs from `GRID` while `i < SIZE - GRID`.
2. **Draw the yellow diagonal alone.** Four steps, one `rect(i, i, GRID, GRID)` in the body, `fill("yellow")` before it. Run.
3. **Derive the lime expression,** as in the section above, then add `fill("lime")` and the mirrored `rect` to the same loop body. Like the rays, the body switches colors every round: fill, yellow square, fill, lime square.
4. **Check the crossing.** In the middle of the X, the two diagonals meet. Which color wins there, and why? Think about the order of the two `rect` calls in your body.

💡 Exercise: Pixel X

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/pixel-x.yaml>

4.3.4 Counting instead of measuring



Figure 4.3.2: Eight horizontal stripes filling the canvas, alternating lime and yellow from top to bottom.

All your loops so far measured pixels; *i* was an x or y coordinate, and the update jumped one grid step. For the zebra stripes, a second style is more natural: let the variable count the stripes, 0, 1, 2, 3, and compute the position from the count when you need it. Stripe number *i* starts at `y = i * STRIPE_THICKNESS`:

```
let i: number = 0;
while (i * STRIPE_THICKNESS < SIZE) {
  // draw stripe number i
  i += 1;
}
```

Both styles are correct, and you will use both. The measuring style is direct: the variable *is* the position. The counting style keeps the round number, and that is what you need here, because the stripe's number decides its color.

Counting up by one is so common that TypeScript has an extra-short form for it: `i++` does exactly what `i += 1` does. Until now the course held the `++` back; from this chapter on you may use it, but only for steps of one. For every other step size, `+=` stays the right tool.

4.3.5 A decision inside the loop

The loop body may contain any statements, including an `if`. Every round can look at the loop variable and act differently:

```
if (i % 2 === 0) {
  fill("lime");
} else {
  fill("yellow");
}
```

The test `i % 2 === 0` is the modulo operator in its parity role. You used `% 2` in the binary chapter to split off the last binary digit (Section 3.6.2); the answer is 0 for even numbers and 1 for odd ones, even, odd, even, odd, in exactly the rhythm a zebra needs. Stripes 0, 2, 4, and 6 come out lime, stripes 1, 3, 5, and 7 yellow.

4.3.6 Your exercise: Zebra Stripes

The canvas is 400 pixels, each stripe 50 thick, and the rule of the exercise is strict: your output must match the goal picture exactly, and you may use only a **single** while loop.

1. **Make a loop table** with four columns: `i`, the stripe's y position `i * STRIPE_THICKNESS`, the value of `i % 2`, and the color. 400 divided by 50 is eight stripes, so `i` counts 0 to 7, and the goal picture tells you which color stripe 0 must get.
2. **Write the loop:** the counting skeleton from above, the `if` deciding the fill, then one `rect` for stripe number `i`. It spans the full canvas width and is `STRIPE_THICKNESS` tall.
3. **Change the constants.** `STRIPE_THICKNESS = 25` must give 16 thinner stripes, `STRIPE_THICKNESS = 100` four fat ones, with the colors still alternating from the same starting color.

💡 Exercise: Zebra Stripes

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfd6b3c2hubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/zebra-stripes.yaml>

4.3.7 Make it smarter: draw only half the stripes

Look at your finished zebra once more. The loop draws eight stripes, and four of them are the same color. If the whole canvas were lime *before* the loop starts, the lime stripes would already be there. One `background("lime")` paints the canvas, and the loop only has to add the yellow stripes on top.

That changes the loop, and for the better. It no longer needs to visit every stripe; it can jump straight from one yellow stripe to the next: start at stripe 1 and count in steps of two, `i += 2`. And the `if` in your loop body loses its job, because a loop that only visits yellow stripes has no decision left to make; the whole `if` and `else` can go. Same picture, half the rounds, and one branch less.

Try it. Rewrite your zebra program into this form: paint the canvas lime, remove the `if`, let the loop count in twos. Check that the picture stays exactly the same, also after you change `STRIPE_THICKNESS`.

i First obvious, then smarter

This rewrite shows how a lot of real programming works. First you write the obvious version, the one that follows the task word for word. Eight stripes, so draw eight stripes. Then you look at the working code and ask whether it could be simpler. The obvious version is not wasted work; it is the step that lets you *see* the shortcut. And simpler code is more than pretty, because fewer lines and fewer decisions mean fewer places for a bug to hide. You met this rhythm once before, when an AI shrank the dice's 21 `circle` statements down to a handful (Section 3.3.7), and you will meet it again and again. First make it work, then make it simpler.

4.3.8 Check your understanding

When your X crosses cleanly and your zebra wears its stripes, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Pixel X and zebra stripes

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/rwywus75mr>

4.4 Mesh: a loop inside a loop

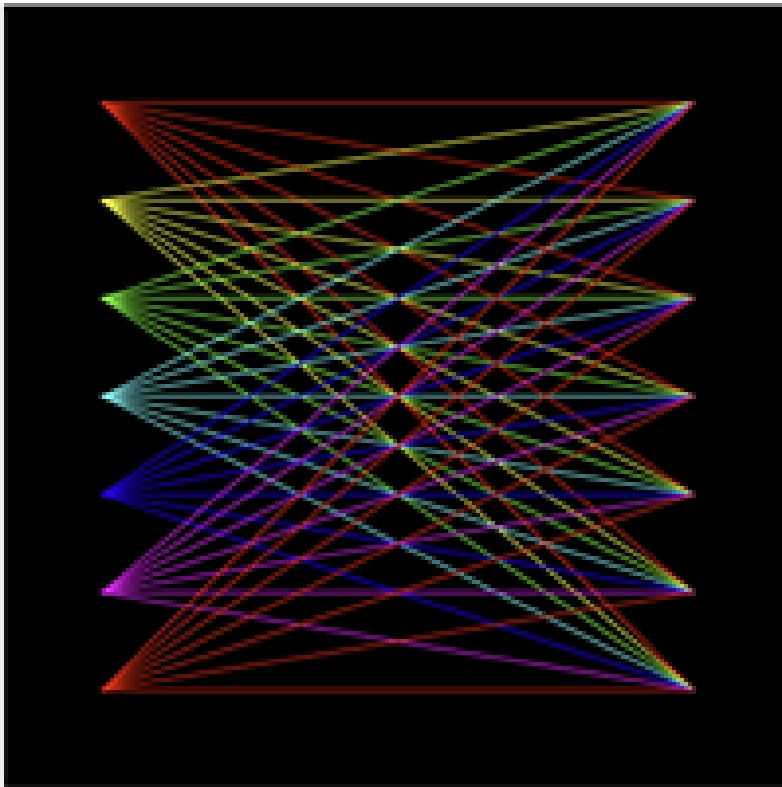


Figure 4.4.1: The finished mesh: every point on the left edge is connected to every point on the right edge, 49 lines in total, with one color per left point.

Look at the picture and try to count the lines. Seven points on the left edge, seven on the right, and a line from *every* left point to *every* right point, 49 lines in total. Typing them would be madness, and a single loop won't do it either, because a single loop can only walk one row of points. This chapter introduces the tool for “every-with-every” drawings: the **nested loop**, a loop inside another loop. Nested loops draw grids, game boards, and patterns, and they come with one famous beginner bug that you'll meet in a controlled experiment before it can meet you.

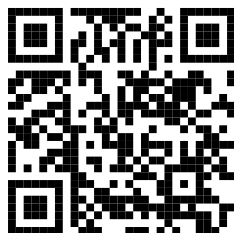
4.4.1 AI tutor

Two loops working inside each other are harder to read than any single loop. If your mesh shows only one bundle of lines instead of the full weave, or the colors don't land where you wanted them, show the tutor your two loops and tell it which picture you expected.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/ctck301mbv>

4.4.2 One fan first

Start small. The first stage of the mesh connects the seven left points to a *single* point on the right edge. The left points sit at (MARGIN, left) with left running from 50 to 350 in steps of 50, and the target sits at (width - MARGIN, right):

```
let left: number = MARGIN;
while (left <= SIZE - MARGIN) {
  let right: number = MARGIN;
  line(MARGIN, left, width - MARGIN, right);

  left += MARGIN;
}
```

One thing looks odd on purpose: the target's height lives in a variable `right`, although it never changes. `right` is 50, every line ends at the same point, and the loop draws one fan. The variable is preparation. In the next stage, `right` starts moving, and the `line` statement is already written for it.

4.4.3 The loop inside the loop

Now the real mesh, where every left point connects to every right point. Think it through for a single left point, say `left = 50`. You need lines to `right = 50, 100, ... , 350`. That is a while loop, and you can already write it. The trick is *where* it goes. The loop body of the outer loop may contain any statements, and a loop is a statement. So the whole inner loop moves into the outer body:

```
let left: number = MARGIN;
while (left <= SIZE - MARGIN) {
  let right: number = MARGIN;
  while (right <= SIZE - MARGIN) {
    line(MARGIN, left, width - MARGIN, right);
    right += MARGIN;
  }

  left += MARGIN;
}
```

Play computer, but zoom out. For the outer loop, the entire inner loop is just one statement of its body. Outer round one sets `left = 50`, and the body runs; `right` starts at 50, the inner loop draws all seven lines of the first fan, and ends. Then `left += 50`, the outer check passes, and the body runs *again from the top*. `let right = MARGIN`; makes the inner loop start fresh, and the second fan appears. Seven fans of seven lines: 49.

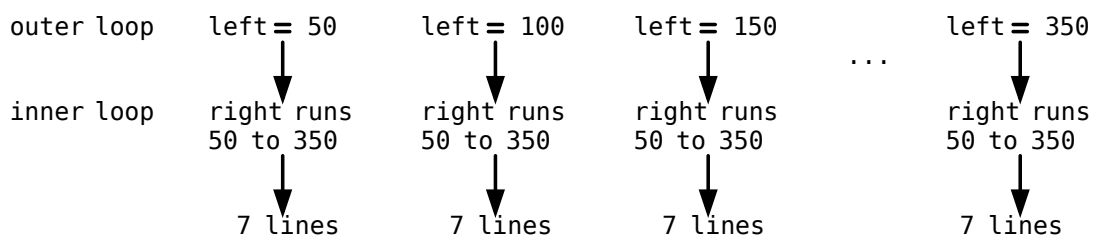


Figure 4.4.2: Every round of the outer loop runs the complete inner loop: left takes one step, and right runs its whole range, again and again.

Watch the two variables move: `right` runs fast, a full lap per fan, while `left` moves slowly, one step per lap. You have seen this rhythm before, in the number systems chapter, where the ones digit turns fast and the tens digit slow (Section 3.5.2). Nested loops count exactly like digits do.

⚠ The classic nested-loop bug

Imagine the line `let right = MARGIN`; moved above the *outer* loop. Only the first fan would appear. Why? The inner loop runs once, leaves `right` at 400, and nothing ever resets it. In every later outer round, the inner check `right <= 350` is false immediately, and the inner loop runs zero times. The rule: initialize the inner loop variable *inside* the outer loop's body, so every fan starts fresh. Once your own mesh runs, produce this bug on purpose; you will recognize the “only the first row shows up” picture forever.

4.4.4 Where a statement sits decides how often it runs

The last stage colors the mesh. All lines of one fan share a color, and each fan's hue is 60 steps further around the HSB color wheel (Section 2.4.4). You know the tool, a running value:

```
let lineColor: number = 0;
```

The interesting question is where the `stroke` call goes. In a nested loop, there are three different places, and they mean three different things:

- **Before both loops:** runs once. One color for the whole mesh.
- **In the outer body, before the inner loop:** runs once per fan. All seven lines of a fan share a color; the next fan gets the next color. This is what the mesh needs.
- **In the inner body:** runs once per line, 49 times. Every line gets its own color, like the gradient of the colored rays.

The same choice you made for the rays' update line (inside the loop or after it) returns here, one level bigger. Placement is not cosmetics; it decides how often a statement runs and what the picture shows.

4.4.5 Your exercise: Mesh

The exercise walks the same three stages as this chapter, and in the playground each stage has its own goal picture. Solve them strictly in order.

1. **Step 1: one fan.** The starter code brings the canvas, the constants, and a fixed `stroke("lime")`. Write a single while loop over `left`, with the target height in a variable `right` that stays at `MARGIN` for now, exactly as in the fan section above (Section 4.4.2). The anchor points sit at 50, 100, and so on up to 350, seven per edge.
2. **Step 2: the mesh.** Wrap the `line` statement in the inner loop over `right`. Keep `let right = MARGIN;` inside the outer body, and give the inner loop its own update. Run it; the full mesh with its 49 lines should appear.
3. **Step 3 (advanced): colors.** Put `colorMode(HSB);` into `setup` and remove the fixed `stroke("lime")`. Declare `let lineColor: number = 0;` before the outer loop, set `stroke(lineColor, 100, 100);` in the outer body before the inner loop, and add `lineColor += 60;` at the end of the outer body. One color per fan, around the color wheel.
4. **Experiment.** Move the `stroke` call and the `+= 60` into the inner body and run again; now every one of the 49 lines gets its own color. Decide which version you find more beautiful, then restore the exercise's version.

💡 Exercise: Mesh

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdafbcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/mesh.yaml>

4.4.6 Check your understanding

When all 49 lines are in place and every fan wears its own color, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Nested loops

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/ud07730c80>

4.5 Chessboard: rows, columns, and one clever loop

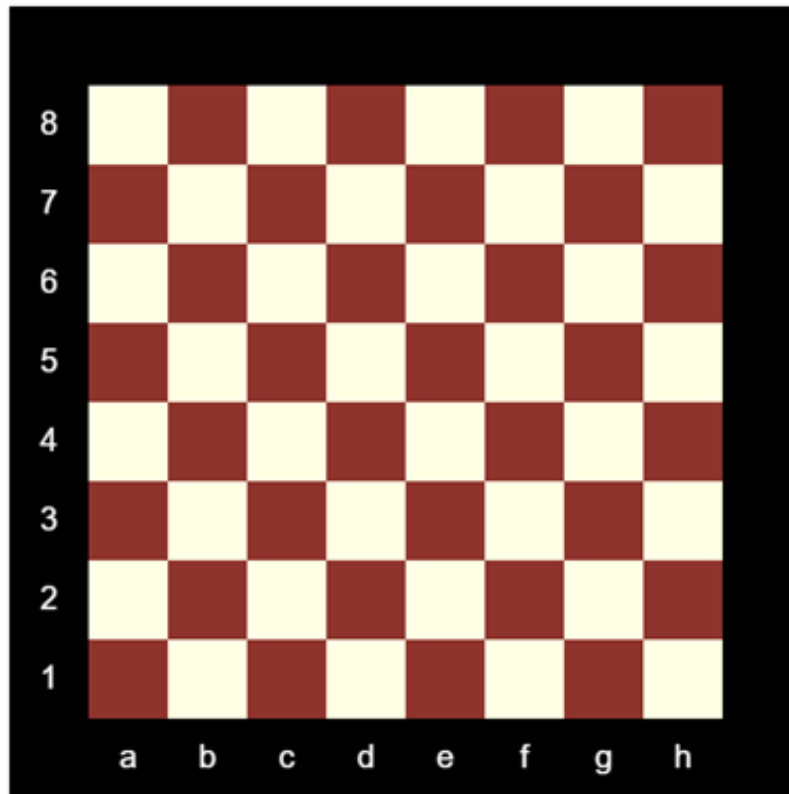


Figure 4.5.1: A chessboard with brown and light yellow squares, the files a to h labeled along the bottom margin and the ranks 1 to 8 up the left margin.

A chessboard is a small exam in disguise. It needs nested loops for the 64 squares, the parity trick for the two colors, `text` for the markings in the margin, and a `switch` to turn the numbers 1 to 8 into the letters a to h. Every one of those tools is already in your toolbox; this chapter is about making them work together. And at the end, for the ambitious, comes a clever connection: you can draw a chessboard with a *single* loop, using the same arithmetic you used to take numbers apart in the number systems chapter.

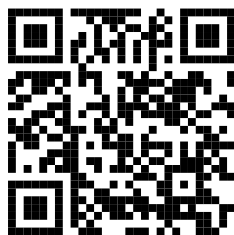
4.5.1 AI tutor

This drawing has many small parts: squares, colors, numbers, letters. Work on one part at a time, and when a part misbehaves, show the tutor only that part. “My colors are striped instead of checkered” is a great opening sentence.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/ctck301mbv>

4.5.2 Rows and columns

The board is a grid of cells, and for grids of cells the counting style of loop (Section 4.3.4) is the natural choice: an outer loop counts the rows 0 to 7, an inner loop counts the columns 0 to 7. The size of one square gets a constant, `SQUARE_SIZE`, worth 50 pixels here. You then compute the positions from the counts, like the zebra stripes, only in two directions:

```
x = SQUARE_SIZE + col * SQUARE_SIZE
y = SQUARE_SIZE + row * SQUARE_SIZE
```

The extra `SQUARE_SIZE` at the front shifts the whole board one square in from the edge. On the 500 pixel canvas the board covers 400, and the remaining border leaves room for the letters and numbers. Row 0, column 0 is the top left square at (50, 50); row 7, column 7 is the bottom right square at (400, 400).

4.5.3 Coloring by parity

Which squares are brown? Walk along a row and the colors alternate with every step, so the column's parity decides. Walk down a column and they alternate again, so the row's parity matters too. The neat trick is to add the two counts. Moving one square in any direction changes `row + col` by exactly one, so the sum flips between even and odd just like the colors flip:

```
if ((row + col) % 2 === 0) {
    fill("lightyellow");
} else {
    fill("brown");
}
```

This is the zebra's % 2 (Section 4.3.5), grown into two dimensions. Check it against the goal picture: the top left square has `row + col` equal to 0, even, and is indeed light yellow.

4.5.4 Letters and numbers in the margin

The markings need one more loop, counting `i` from 1 to 8. Each round places two things: the rank number `i` in the left margin, and a file letter at the bottom.

The numbers are positions to compute. Chess counts its ranks from the bottom, so rank 1 sits next to the *lowest* row. Rank `i` sits `i` squares above the bottom margin, centered in the margin column; work the expression out before you type it, and test it with `i = 1` and `i = 8` like you tested the mirror expression of the pixel X.

The letters are a translation, where 1 becomes "a", 2 becomes "b", and so on, one exact case per value. That is exactly what the `switch` statement is for (Section 3.8.2):

```
let file: string = "";
switch (i) {
    case 1: file = "a"; break;
    case 2: file = "b"; break;
    // ... the cases 3 to 8 ...
}
text(file, SQUARE_SIZE * i + SQUARE_SIZE / 2, height - SQUARE_SIZE / 2);
```

The variable starts as the empty string, the "nothing yet" value you know from rock-paper-scissors, and the `switch` fills it. A `switch` inside a loop body runs once per round, with the current `i`, so each round translates its own number.

💡 text prints numbers too

For the rank numbers, pass `i` to `text` directly: `text(i, x, y)` draws the number without any template string around it. Set `textSize(20)`, `textAlign(CENTER, CENTER)`, and `fill("white")` once, before the loop; they apply to every marking.

4.5.5 Advanced: one loop for the whole board

Now the clever part, for the ambitious. Number the squares 0 to 63, row by row, so squares 0 to 7 form the top row, 8 to 15 the next, and so on. Can one loop over `i` from 0 to 63 recover the row and column from the

number alone? You did exactly this in the number systems chapter, splitting a number into digits with `%` and `floor` (Section 3.6.2). A square number works like a two-digit number in base 8:

```
const row: number = floor(i / 8);
const col: number = i % 8;
```

`floor(i / 8)` counts how many *complete* rows of eight fit below `i`, and `i % 8` is what's left over, the position inside the row. Check it with a small table:

Table 4.5.1: Recovering row and column from the square number.

i	floor(i / 8)	i % 8	square
0	0	0	first row, first column
7	0	7	first row, last column
8	1	0	second row, first column
42	5	2	sixth row, third column
63	7	7	last row, last column

One warning for later, when you compare your work with the playground's sample solution. That solution rounds with `Math.floor`. You know the chore from the switch chapter (Section 3.8.6): this course always uses the `p5.js` helper `floor`, which does the same job.

4.5.6 Your exercise: Chessboard

The starter code contains only the canvas, the two constants, and `noStroke()`. Build the board in the exercise's three steps.

1. **Step 1: the squares.** Nested counting loops over `row` and `col`, the parity `if`, one `rect` per square. Run and compare corners with the goal picture.
2. **Step 2: the markings.** The 1-to-8 loop with the rank number and the switch-translated letter, text settings before the loop. Getting the rank numbers to count from the bottom is the fiddly part; the margin section above (Section 4.5.4) shows how to find the expression.
3. **Step 3 (advanced): one loop.** Replace the nested square loops with a single loop over the square numbers 0 to 63, and compute `row` and `col` with `floor` and `%` as shown above (Section 4.5.5). The picture must not change at all.

Exercise: Chessboard

► [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/chessboard.yaml>

4.5.7 Check your understanding

When your board is checkered and labeled, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Chessboard

► Take the quiz on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/zr9xqctms0>

4.6 The for loop: four steps on one line

```
// Part 1: Draw vertical lines
// Here you see how to use a while loop.
let i = GRID;           // STEP 1: Initialize the loop variable
while (i < SIZE) {       // STEP 2: Check the loop condition
    line(i, 0, i, SIZE); // STEP 3: Do whatever you want to do repeatedly
    i += GRID;           // STEP 4: Update the loop variable
}

// Part 2: Draw horizontal lines
// Note that this time, we are using a for loop.
// +----- STEP 1: Initialize the loop variable
// |           +----- STEP 2: Check the loop condition
// |           |           +--- STEP 4: Update the loop variable
// v           v           v
for (let i = GRID; i < SIZE; i += GRID) {
    line(0, i, SIZE, i); // STEP 3: Do whatever you want to do repeatedly
}
```

Figure 4.6.1: The while loop and the for loop side by side: the initialize, check, and update parts of the while loop move into the parentheses of the for loop; only the work stays in the body.

Count your while loops. The grid, the staircases, the rays, the X, the zebra, the mesh, the chessboard. Every single one had the same skeleton, initialize above the loop, check in the parentheses, update at the bottom of the body, and by now your fingers type it half asleep. Programmers noticed this repetition a long time ago and built a loop for it: the **for loop**. Like the switch chapter, this chapter teaches you nothing new to compute, only a tidier way to write what you already write. And because the pattern is the point, the second half of the exercise sends you back to your own programs to convert them.

4.6.1 AI tutor

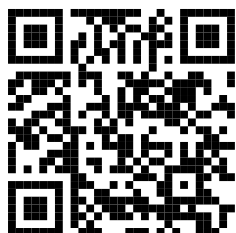
Converting a loop is mechanical once you see the mapping, and confusing until then. If your for loop draws nothing or TypeScript complains about the header, paste the while version and your for version and let the tutor compare them with you.



Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/ctck301mbv>

4.6.2 Packing the four steps

Here is the horizontal-line loop of the grid, first as the while loop you wrote, then as a for loop:

```
let i: number = GRID; // STEP 1: Initialize the loop variable
while (i < SIZE) {     // STEP 2: Check the loop condition
    line(0, i, SIZE, i); // STEP 3: Do whatever you want to do repeatedly
    i += GRID;          // STEP 4: Update the loop variable
}
```

```

}

// +----- STEP 1: Initialize the loop variable
// |               +----- STEP 2: Check the loop condition
// |               |               +--- STEP 4: Update the loop variable
// |               |               v
// v               v               v
for (let i: number = GRID; i < SIZE; i += GRID) {
    line(0, i, SIZE, i); // STEP 3: Do whatever you want to do repeatedly
}

```

Steps 1, 2, and 4 move into the parentheses, the **header** of the for loop, separated by two semicolons. Only step 3, the work, stays in the body. Read the header as a sentence: “starting with `i` at `GRID`, as long as `i < SIZE`, stepping `i` by `GRID`.”

Nothing about the *execution* changes. The computer still initializes once, then checks, works, updates, checks, works, updates, exactly like the while loop. Your loop tables work unchanged, an endless loop is still possible (leave the update slot empty and nothing moves), and the condition is still checked before every round. The for loop is packaging, not new machinery.

i Why two semicolons?

The header always has exactly three slots and two semicolons: `for (initialize; check; update)`. The slots are separated by semicolons, not commas, because initialize and update are ordinary statements. When a for loop won’t compile, count the semicolons in the header first.

4.6.3 The loop variable belongs to the loop

The `let i` inside the header comes with a pleasant surprise. A variable declared in a for header exists *only inside that loop*: it is born when the loop starts and gone after the closing brace.

Remember the grid chapter, where the second while loop was not allowed to say `let i` again and had to reset the existing variable with a plain assignment (Section 4.1.6)? With for loops, that worry disappears. Two loops in a row may each declare their own `let i` in their own header, and TypeScript is happy, because the two variables never exist at the same time:

```

for (let i: number = GRID; i < SIZE; i += GRID) {
    line(i, 0, i, SIZE);
}

for (let i: number = GRID; i < SIZE; i += GRID) {
    line(0, i, SIZE, i);
}

```

Each loop brings its own variable and cleans it up afterward. No leftovers, no accidental reuse, one reason less for a red squiggle.

4.6.4 Nested loops love for

The conversion works for nested loops too, and there it pays off double. Here is the mesh with both loops converted:

```

for (let left: number = MARGIN; left <= SIZE - MARGIN; left += MARGIN) {
    for (let right: number = MARGIN; right <= SIZE - MARGIN; right += MARGIN) {
        line(MARGIN, left, width - MARGIN, right);
    }
}

```

Look at the inner header: `let right = MARGIN` runs every time the inner loop starts, which is once per outer round. The initialization sits *inside* the outer loop automatically, because the header belongs to the loop. The classic nested-loop bug from the mesh chapter, the forgotten reset that leaves only the first fan (Section 4.4.3), cannot even be written this way. The for loop’s shape puts the initialization where it belongs.

4.6.5 while or for?

Both loops can do everything the other can, so this is a question of style, and the course’s rule of thumb is simple. When a loop counts, with a clear start, a clear condition, and a fixed step, write a for loop; from this

chapter on, that is the default for drawing loops. The while loop stays in your toolbox for loops that don't count, where the update isn't a fixed step or you can't say in advance how many rounds it takes. You'll meet loops like that later in the course.

The counting-style loops convert nicely too: the zebra's counter becomes `for (let i: number = 0; i * STRIPE_THICKNESS < SIZE; i++)`, with the `++` shortcut (Section 4.3.4) sitting naturally in the update slot.

4.6.6 Your exercise: While to For

The starter code is the finished grid, with the horizontal lines already converted to a for loop.

1. **Read the given for loop.** It carries the same STEP comments as the picture at the top of this chapter; match each header slot to its comment until the mapping feels obvious.
2. **Convert the vertical-line loop.** Move the three while parts into a for header, delete the old `let i` line and the update line, and run. The grid must look exactly as before.
3. **Say both loops out loud.** "Starting at GRID, while below SIZE, stepping by GRID." If the sentence matches for both, you are done.

💡 Exercise: While to For

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

`https://ts-web-playground-server-bfd6bjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/while-to-for.yaml`

4.6.7 Your exercise, part 2: convert your own programs

Task 2 of the same exercise sends you back to your own work. Pick **at least three** of your earlier loop exercises and convert your while solutions to for loops. Choose so that every kind of loop shows up once:

1. **A measuring loop:** Pixel X or Step Lines, where the variable is a coordinate.
2. **A counting loop:** Zebra Stripes, where the header ends in `i++`.
3. **A nested loop:** Mesh or your chessboard, where the inner header takes over the re-initialization (Section 4.6.4).

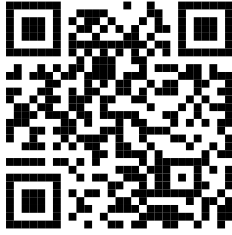
After each conversion, run the program. The picture must not change at all; the loop table of the for loop is the same as the while loop's. Converting your own working code is the fastest way to make the new shape feel like home.

4.6.8 Check your understanding

When your grid runs on for loops and three old friends are converted, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: The for loop

► Take the quiz on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/j1rokfr061>

4.7 Moving the origin: `translate`, `push`, and `pop`

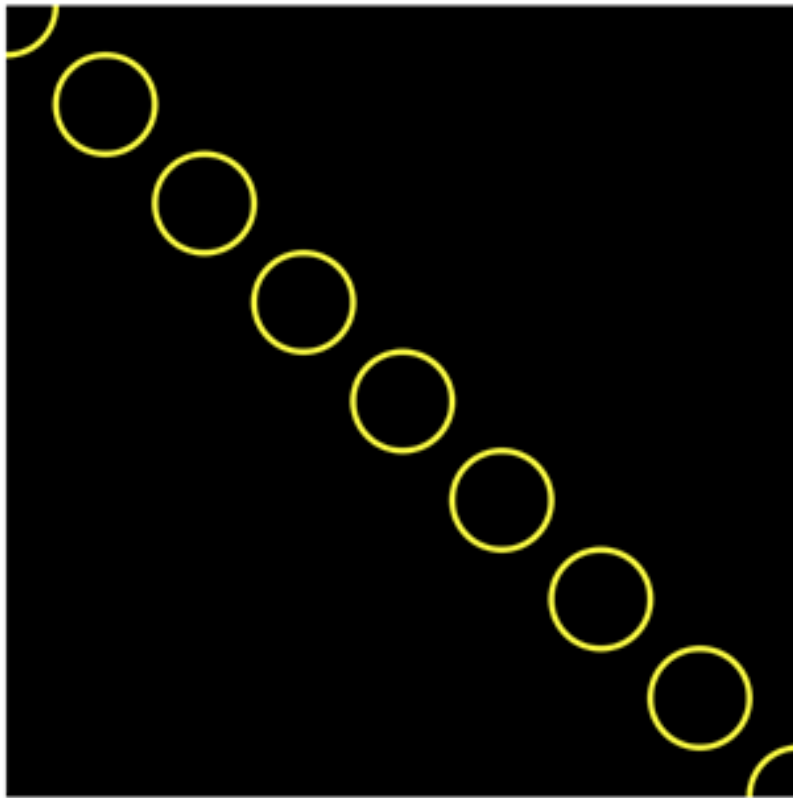


Figure 4.7.1: A diagonal of yellow circles from the top left to the bottom right corner. Every circle is drawn with the same command, `circle(0, 0, 50)`; only the origin moves between them.

Way back in the practice projects of part 1, the soccer field came with a recipe, a few lines with `push`, `translate`, and `scale` that you copied without questions, magic words with a promise that the details would come later. Later is now. In this chapter the magic words become tools you understand. `translate` moves the origin of the coordinate system, and `push` and `pop` save and restore it. Combined with loops, they open a new way of drawing: instead of computing every position, you draw in one place and move the paper under the pen. Both exercises are discovery scripts, in which you uncomment code line by line, predict, and watch.

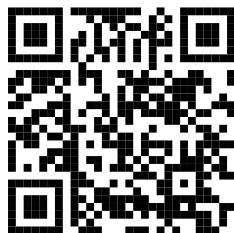
4.7.1 AI tutor

Moving the origin confuses everyone at first, because the code names one position and the shape appears at another. When that happens to you, tell the tutor which line you uncommented and where the circle landed; the tutor knows the way of thinking that untangles it.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/ctck301mbv>

4.7.2 The origin moves

Every coordinate you have ever written is measured from the **origin**, the point (0, 0), and so far the origin has always been the top left corner of the canvas. The `translate` command moves it:

```
translate(100, 100);  
circle(0, 0, 50);
```

After the `translate`, the point (0, 0) sits 100 pixels right and 100 pixels down from the corner, and every drawing command that follows measures from there. The `circle` names the position (0, 0), and appears at the center of what used to be (100, 100). The command did not move the circle; it moved the ruler the circle is measured with.

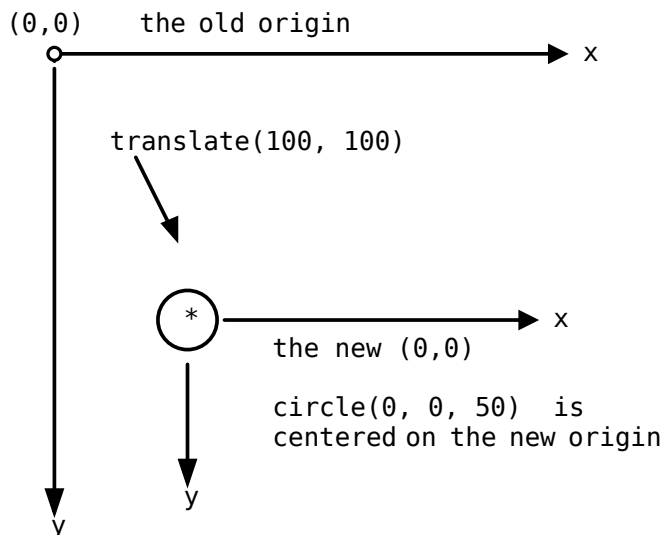


Figure 4.7.2: `translate` moves the origin: after `translate(100, 100)`, the point (0, 0) sits 100 pixels right and down from the corner, and `circle(0, 0, 50)` is centered exactly on that new origin.

Three facts complete the picture, and the first exercise will let you discover each one:

- **translate is relative.** A second `translate(100, 100)` moves the origin another 100 right and down, measured from where it *currently* is. Translations add up.
- **Negative values move back.** `translate(-50, 0)` moves the origin 50 pixels to the left, `translate(0, -50)` moves it up.
- **resetMatrix starts over.** `resetMatrix()` puts the origin back in the top left corner of the canvas, as if no `translate` had ever happened.

4.7.3 Drawing without coordinates

Now combine `translate` with a loop. The goal picture at the top of the chapter shows a diagonal of circles, and the rule is that you draw every circle with the very same command, `circle(0, 0, CIRCLE_DIAMETER)`, no other coordinates allowed. `SIZE` is the canvas size and `CIRCLE_DIAMETER` the diameter of one circle, both constants:

```
circle(0, 0, CIRCLE_DIAMETER);  
for (let i: number = 0; i <= SIZE; i += CIRCLE_DIAMETER) {  
  translate(CIRCLE_DIAMETER, CIRCLE_DIAMETER);  
  circle(0, 0, CIRCLE_DIAMETER);  
}
```

Each round moves the origin one step down the diagonal and draws “here”. The position is nowhere in the code; it lives in the canvas itself, accumulated by all the translates so far, the way the hue accumulated in the colored rays. Notice the loop variable `i`: it appears nowhere in the body. Its only job is counting rounds, so that the loop runs often enough to cross the canvas. A loop that just counts rounds is perfectly normal from now on.

You could draw this diagonal with the pixel X technique instead, computing `circle(i, i, ...)`. Same picture, two ways of thinking: move the pen, or move the paper. For a single diagonal, both are fine. For the patterns of the next chapter, moving the paper wins, and you’ll see why.

4.7.4 A note before you wander: push and pop

Translating is a one-way street. After five translates, how do you get back? `resetMatrix` goes all the way back to the corner, but usually you want something gentler, a “remember this spot, let me wander, then bring me back”. That is exactly what `push` and `pop` do:

- `push()` writes a note: it saves the current origin **and** the current style settings (`fill`, `stroke`, `strokeWeight`, and the other style commands).
- `pop()` takes the most recent note and restores everything on it.

One concrete timeline shows the note-taking at work:

Table 4.7.1: A push and pop timeline, with the origin after every statement.

statement	origin afterward
<code>translate(100, 100);</code>	(100, 100)
<code>push();</code>	(100, 100), and a note saves this spot
<code>translate(100, 0);</code>	(200, 100)
<code>translate(0, 100);</code>	(200, 200)
<code>pop();</code>	(100, 100) again, restored from the note
<code>translate(0, 100);</code>	(100, 200)

The wandering between `push` and `pop` leaves no trace: after the `pop`, the program continues as if the detour never happened. And because `pop` also restores the styles, a color change inside the detour vanishes with it.

Notes stack. If you `push` twice, you hold two notes, and each `pop` takes back the most recent one: the first `pop` returns you to the second `push`, the next `pop` to the first. Like sticky notes piled on a desk, you always take the top one.

! Course rule: every push gets its pop

Treat `push` and `pop` as a pair, like opening and closing braces. A `push` without its `pop` leaks: every drawing command that runs later inherits your moved origin and changed styles, and the bug shows up far away from its cause. Write the `pop` immediately after the `push`, then fill the middle.

4.7.5 Your exercise: Move Origin

1. **Task 1: uncomment and predict.** The starter code is a stack of commented blocks. Before you uncomment each one, play computer: write down where you expect the next circle, then uncomment, run, and compare. Work through all the blocks, including the negative translate and the `resetMatrix` at the end.
2. **Task 2: write the loop.** The goal is a diagonal of circles: diameter 50, the first centered at the corner (0, 0), each next one 50 right and 50 down. One circle at (0, 0) before the loop, then a for loop whose body is exactly two statements, the translate and the circle, as in the section above (Section 4.7.3). No computed coordinates allowed.
3. **Experiment.** Change the translate to `(CIRCLE_DIAMETER, 0)` for a horizontal chain, then try a negative y for an upward staircase from the bottom (you’ll need a translate before the loop to start at the bottom left).

💡 Exercise: Move Origin

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/move-origin.yaml>

4.7.6 Your exercise, part 2: Push and Pop

The second discovery script has no drawing task and no sample solution; its only goal is that you *see* the note-taking happen.

1. **Uncomment and predict**, block by block, like in task 1. The script translates to (100, 100), pushes, wanders right and down, pops, and continues from the saved spot. Before each run, say out loud where the next circle will appear.
2. **Check the styles.** Add `stroke("red")` somewhere between `push` and `pop`, and see that the circles after the `pop` are yellow again: the note restored the stroke color along with the origin.
3. **Stack two notes.** Add a second `push/pop` pair inside the first one, with its own small translate, and confirm: each `pop` returns to the most recent `push`.

💡 Exercise: Push and Pop

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

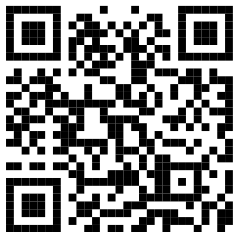
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/push-pop.yaml>

4.7.7 Check your understanding

When your diagonal runs on translate alone and the push-pop script holds no more surprises, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Moving the origin

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/b0f2kwzb7n>

4.8 Patterns: repeat, move, repeat

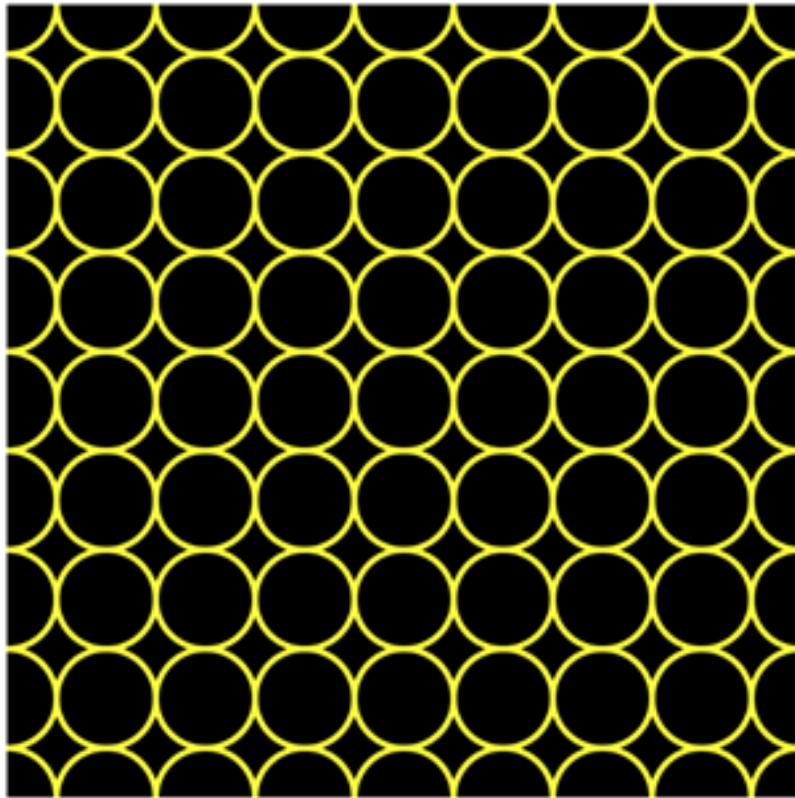


Figure 4.8.1: A grid of overlapping yellow circles covering the whole canvas, drawn by nested loops that never name a coordinate: the origin wanders, row by row.

This chapter brings the loop tools together: nested for loops, `translate`, and the `push/pop` pair combine into one recipe that draws wallpapers, tilings, and textile patterns. Learn the recipe by heart, because you will use it every time a drawing repeats in rows and columns. You meet it twice, first in a finished program that you study and convert, then in a triangle pattern that you build yourself, including the professional touch of letting the pattern run over the edges.

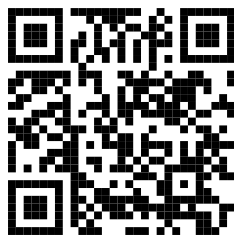
4.8.1 AI tutor

Patterns built from loops and translates have many moving parts, and swapping any two of them changes the picture completely. If your pattern drifts, stacks, or vanishes, tell the tutor what you expected, what you see, and paste your loops.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:
<https://app.novedu.at/ctck301mbv>

4.8.2 The row recipe

Here is a finished pattern program, still with while loops; it is also the starting point of this chapter's first exercise. Read it slowly; every line has a job:

```

let y: number = 0;
while (y <= SIZE) {
  // Save the current origin (start of the current row)
  push();

  let x: number = 0;
  while (x <= SIZE) {
    circle(0, 0, CIRCLE_DIAMETER);
    translate(CIRCLE_DIAMETER, 0); // move one step right
    x += CIRCLE_DIAMETER;
  }

  // Back to the start of the row
  pop();

  // Move one row down
  translate(0, CIRCLE_DIAMETER);
  y += CIRCLE_DIAMETER;
}

```

The rhythm of one outer round: **note, draw across, back, down.**

1. `push()` writes the note: “this is where the row starts.”
2. The inner loop draws the row: circle at the origin, step right, circle, step right, like a typewriter.
3. After the row, the origin sits far off to the right. `pop()` brings it back to the note, the row’s start.
4. `translate(0, CIRCLE_DIAMETER)` moves one row down, and the next outer round begins there.

Could you skip the `push/pop` and translate back to the left by hand? You would have to know exactly how far the inner loop walked and undo it precisely. The note is simpler and always exact; this is the situation `push` and `pop` were made for.

Two details deserve a second look. First, the loop variables `x` and `y` never appear in a drawing command; they only count how far the row and column have come, while the real position lives in the origin, exactly like in the diagonal of circles (Section 4.7.3). Second, the conditions use `<=`, so the last circle of each row sits at the canvas edge and is half cut off. That is on purpose: patterns run to the edge, like wallpaper, not like a framed picture with margins.

4.8.3 Your exercise: Pattern with Push and Pop

The starter code is the finished while version. Your job is to understand it, then convert it.

1. **Read the starter code closely.** Walk through the first outer round in your head: where is the origin after the third circle? After the `pop`? After the row translate?
2. **Convert both loops to for loops.** The headers take over initialize, check, and update (Section 4.6.2); the `push`, `pop`, and both translates keep their places. Run: the picture must not change.
3. **Break it on purpose.** Comment out the `pop()` and predict the picture before you run. The rows start where the previous row ended, and the pattern staircases off the canvas. Put the `pop` back; you have just seen why the note matters.

💡 Exercise: Pattern with Push and Pop

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/push-pop-pattern.yaml>

4.8.4 Triangles in local coordinates

The second exercise tiles the canvas with triangles, and you build the recipe yourself. You need one new skill, drawing a shape *around* the origin. The pattern’s triangle looks like this:

```
triangle(0, 0, SIDE_LENGTH / 2, -HEIGHT, SIDE_LENGTH, 0);
```

You measure all three corners from the current origin: the left corner of the base sits *on* the origin at (0, 0), the right corner one side length further at (SIDE_LENGTH, 0), and the tip above the middle of the base, at (SIDE_LENGTH / 2, -HEIGHT). The y value of the tip is negative, and that is no trick, because negative coordinates lie left of or above the origin. Since the origin moves with every translate, “above the origin” is exactly where a tip should be.

The height of the triangle comes from geometry. The pattern uses equilateral triangles (in German: gleichseitiges Dreieck, all three sides the same length), and an equilateral triangle with side length *s* has the height $s * \sqrt{3} / 2$:

```
const HEIGHT: number = SIDE_LENGTH * sqrt(3) / 2;
```

`sqrt` is the square root, a p5.js helper like `floor` and `round`. The exercise’s starter code spells it `Math.sqrt`; you know the chore (Section 3.8.6), and changing the spelling will be your first small task. The constant then does double duty, because the canvas itself is created with the height `HEIGHT * 9`; the drawing surface is computed from the pattern, not the other way around.

4.8.5 Level 2: over the edges

The triangle exercise has two levels. Level 1 places the first triangle exactly at the corner, and the pattern looks planned. Level 2 shifts the whole grid, half a side length to the left and half a height up. Now the edge rows and columns are cut in half, and the pattern looks like a piece of endless wallpaper.

The shift will need two small changes to your finished level 1 program, and both use ideas you already own. A `translate` *before* the loops moves the whole pattern’s starting point by half a side length and half a height, using negative values where the shift goes left or up (Section 4.7.2). And the loop bounds change a little so the cut shapes at the edges still get drawn, because loop bounds are ordinary numbers, and nothing stops a loop from starting at `-SIDE_LENGTH / 2` or running one extra round. Fiddle with the pre-loop translate and the bounds until the picture matches the level 2 goal; that fiddling *is* the exercise.

4.8.6 Your exercise, part 2: Triangles

1. **The chore first.** Change `Math.sqrt` in the starter code to `sqrt`.
2. **Understand the triangle line.** Read the `triangle` call in the local-coordinates section (Section 4.8.4) until you can say where each of the three corners sits relative to the origin, and why the tip’s y value is negative.
3. **Level 1: the grid.** Build the row recipe around the triangle: inner loop draws and translates right by `SIDE_LENGTH`, outer loop pushes, pops, and translates down by `HEIGHT`. Compare with the level 1 goal picture.
4. **Level 2: the shift.** Add the pre-loop translate and widen the bounds until the edges show clean half-triangles, as in the level 2 picture.

💡 Exercise: Triangles

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0050-Loops/triangles.yaml>

4.8.7 Check your understanding

The row recipe pulls the whole loop toolbox together: with it, you can repeat, decide inside loops, nest loops, and move the paper under the pen. Take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Patterns

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/9zhgxdsem>

4.9 Curves from straight lines

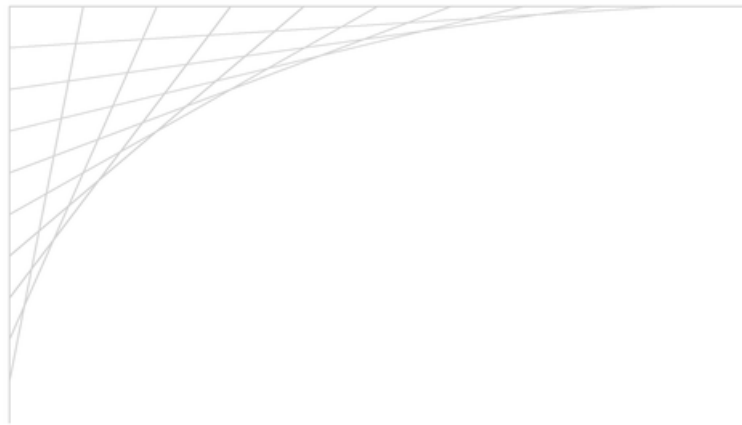


Figure 4.9.1: The first program of this chapter at work: nothing but straight gray lines drawn between a horizontal and a vertical axis, and yet a smooth curve shows up along the open edge of the fan, where the lines crowd together.

This closing chapter of the Loops part starts with a small surprise. A for loop draws straight lines, only straight lines, and a curve appears anyway. From there you give that curve its name, bend it by hand in a professional design program, let p5.js draw it with a single command, and finish with an AI that turns loops and curves into a pattern of your own. Much of this chapter follows the exercise [Spaß mit Bezier-Kurven by CoderDojo Linz](#).

4.9.1 AI tutor

Curves like to go somewhere you did not expect. If your fan comes out as a box of crossing lines, or if your curve runs off the canvas instead of bending nicely, tell the tutor what you expected, what you actually see, and paste your loop or your `bezier` call.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:
<https://app.novedu.at/ctck301mbv>

4.9.2 A fan of straight lines

The first program of this chapter draws a fan, and every command in it is one you already know. Two `line` calls draw an axis to the right and an axis downward, `translate` moves the origin away from the canvas corner so the drawing gets some air around it, and a for loop with a typed loop variable draws the fan lines. The loop computes both endpoints of every fan line from the counter `i`.

This chapter has no prepared exercise, so you type the program into the Empty Playground yourself:

```
function setup() {  
  createCanvas(500, 300);  
  background("white");  
  stroke("lightgray");  
  translate(20, 20);  
}
```

```

const xWidth = 460;
const yHeight = 260;
line(0, 0, xWidth, 0);
line(0, 0, 0, yHeight);

const points = 10;
const deltaX = xWidth / points;
const deltaY = yHeight / points;
for (let i: number = 1; i < points; i++) {
    line(deltaX * i, 0, 0, yHeight - (deltaY * i));
}
}

```

The loop body is the interesting line. Each round connects one point on the horizontal axis with one point on the vertical axis. `deltaX * i` walks to the right along the horizontal axis, and `yHeight - (deltaY * i)` climbs upward along the vertical axis. So the first round joins a point close to the corner with a point far down, the last round joins a point far right with a point close to the corner, and in between the fan opens up.

1. **Type the program in.** Open the Empty Playground below and write the code out. The five `const` declarations above have no type annotations, and that is your first job: add `: number` to each of them while you type, the way this course writes every declaration. Then format your code, check that there are no red squiggles, and run it.
2. **Play with points.** The constant `points` decides how many fan lines the loop draws. Try 5, then 20, then 50, then 100, and run after every change.
3. **Find the curve.** Every line your program draws is straight, and still a curve shows along the open edge of the fan, right where the drawn lines stop and the empty part of the canvas begins. The more lines you draw, the smoother that curve looks. No single line draws it. It appears where the lines crowd together, and each line just touches it at one spot.

💡 Exercise: Empty Playground

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/emptyPlayground.yaml>

4.9.3 The curve has a name

The curve at the edge of the fan is a **Bezier curve**, spoken roughly like “bay-zee-eh”. It is named after Pierre Bézier, a French engineer who worked for the car maker Renault in the 1960s. He needed a way to describe the shape of a car body exactly, precise enough for a machine to cut the metal, and these curves were his answer.

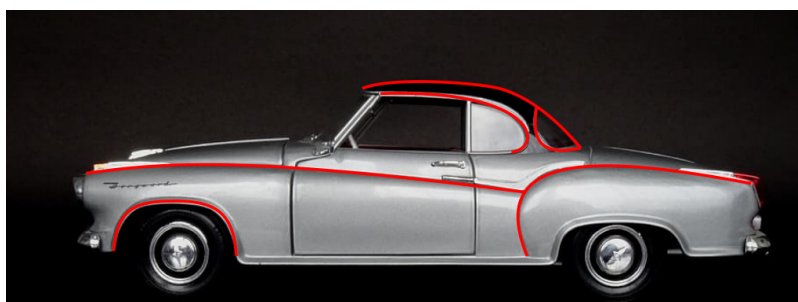


Figure 4.9.2: A car body from the 1960s with some of its lines highlighted in red: the roof, the window, the wheel arches, and the long line along the side. Every one of them is a smooth curve that a designer has to describe exactly.

Today Bezier curves sit everywhere a smooth shape has to be stored exactly. Every letter of every font on this page is built from them, every icon on your phone, every drawing made in a vector graphics program, and many

animations in games. Look at a letter with round shapes, an S or an R, and follow its outline with your eyes. Somebody once placed the points of those curves.

Three points describe your fan's curve, and you can point at all three in your own program:

- It starts at the lower end of the vertical axis.
- It ends at the right end of the horizontal axis.
- It bends toward the corner where the two axes meet, and it never reaches that corner.

The corner is the **control point** of the curve. A control point pulls the curve toward itself, like a magnet, but the curve does not run through it. The straight lines of your fan are the construction lines of exactly this curve, which is why the curve showed up without anybody drawing it.

A curve with a start point, an end point, and one control point is called a **quadratic** Bezier curve. Give it two control points instead, one pulling near the start and one pulling near the end, and it is a **cubic** Bezier curve. Cubic curves can do more, an S shape for example, and most programs use them. The p5.js **bezier** command draws cubic curves, and so does the design program Inkscape.

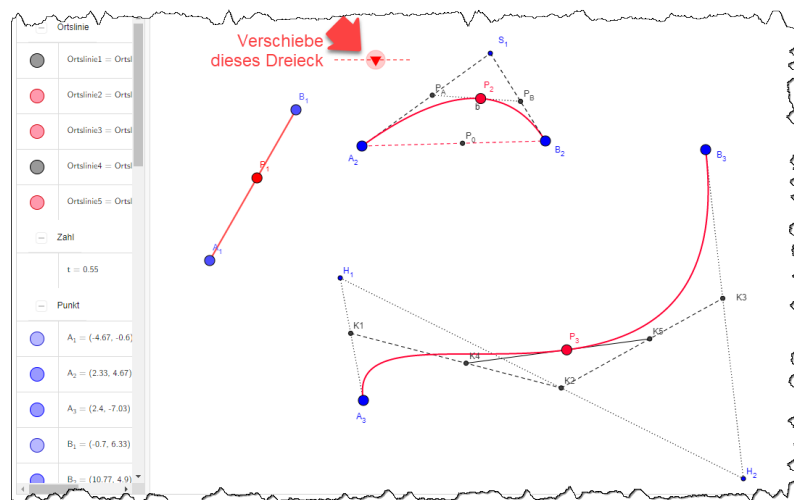


Figure 4.9.3: A geometry program showing three Bezier curves side by side. Top left is a straight line with a moving point on it, top middle is a red curve with a single control point pulling it upward, and the large red S curve below is shaped by two control points. The dashed and dotted lines show how each curve is constructed from its points.

💡 Want to go deeper?

This part is voluntary, and it is the best kind of voluntary. The video [The Beauty of Bézier Curves](#) shows where the curves come from and how fonts use them. On the [interactive GeoGebra page about Bezier curves](#) you can drag the control points yourself and watch the construction lines follow.

4.9.4 Bending curves in Inkscape

Inkscape is a free vector graphics program that designers and software developers use for real work, and every shape you draw in it is made of Bezier curves. Get it from the [Inkscape download page](#) and install it yourself. Installing a program is a skill of its own, so this book gives you no click-by-click instructions for it. If you get stuck, that is what the AI tutor and a search engine are for.

Once Inkscape runs, play with it in four steps.

1. **Pick the pen tool.** It is also called the Bezier tool, and its icon shows a pen drawing a line.

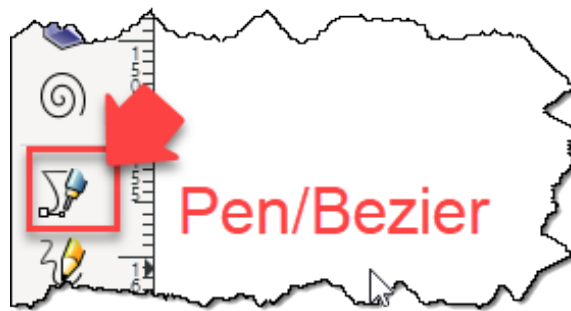


Figure 4.9.4: The Inkscape tool bar with the pen tool, also called the Bezier tool, marked by a red arrow.

2. **Draw a straight line.** Click once for the start point, click a second time somewhere else for the end point, and press **Enter** (a double click on the last point works too). You get a plain straight line.

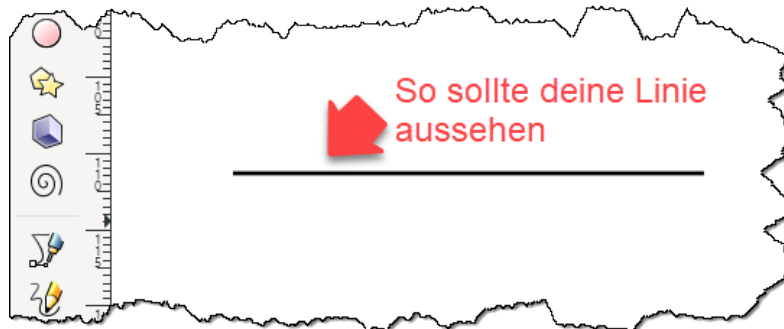


Figure 4.9.5: A single straight black line on the Inkscape canvas, the result of two clicks with the pen tool. The German note says that this is how your line should look.

3. **Bend the line.** Switch to the node tool, the arrow at the top of the tool bar that edits points. Grab the line in the middle and drag it upward. The straight line bends into a curve while you drag.

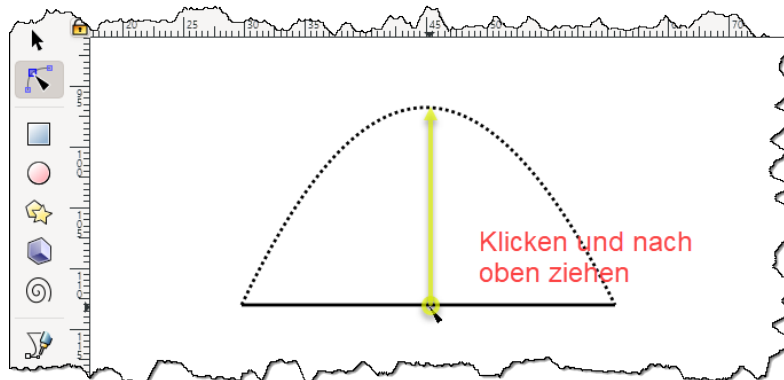


Figure 4.9.6: The node tool is active, and the middle of the line is being dragged upward. A dotted preview shows the arch the line is turning into.

4. **Show the handles.** Click one of the two end points of your curve. Two thin lines pop out with a small circle at the end of each. Drag one of these circles and watch the curve change shape.

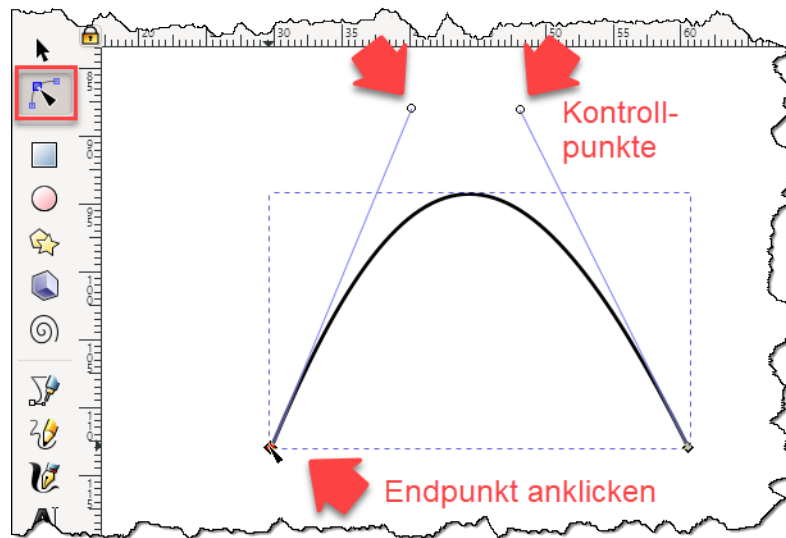


Figure 4.9.7: A finished arch in Inkscape. One end point is selected, and two thin handles reach up to small circles, labeled in German as the control points of the curve.

Those small circles at the ends of the handles are the control points, the very same idea as the corner in your fan. They pull the curve toward themselves, and the curve stays away from them. Two handles, two control points, so an Inkscape segment is a cubic Bezier curve.

Now take about 10 minutes of free play. Draw a heart, a wave, a fish, a mountain range, whatever needs smooth curves. YouTube is full of short tutorials about the Inkscape pen tool, and looking one up is a good move, not cheating.

4.9.5 The bezier command

p5.js draws Bezier curves for you with one command and eight numbers:

```
bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)
```

The eight numbers are four points in a fixed order, namely the start point, the first control point, the second control point, and the end point. Two control points means this is a cubic Bezier curve, the same kind a design program draws.

Two details matter before you type the next program. First, p5.js treats a curve like a shape: with a fill color set, it paints the area the curve encloses. `noFill()` switches that off, and one of the experiments below switches it back on so you can see the difference. Second, the drawing happens inside `draw`, so the picture is repainted every frame and the curve follows your mouse.

```
function setup() {
  createCanvas(510, 500);
}

function draw() {
  background("white");
  noFill();

  const upperY: number = 205 - abs(mouseY - 205);
  const lowerY: number = 205 + abs(mouseY - 205);

  stroke("red");
  strokeWeight(5);
  bezier(5, 205, mouseX, upperY, mouseX, lowerY, 505, 205);

  stroke("blue");
  circle(5, 205, 4);
  circle(505, 205, 4);
  circle(mouseX, upperY, 4);
```

```

    circle(mouseX, lowerY, 4);

    strokeWeight(1);
    line(5, 205, mouseX, upperY);
    line(505, 205, mouseX, lowerY);
}

```

Read the program point by point.

- The start point (5, 205) and the end point (505, 205) never move. Both sit on the same horizontal line, a little above the middle of the canvas, one near the left edge and one near the right edge.
- `abs(mouseY - 205)` is the distance of the mouse from that line. `abs` throws the minus sign away, so the distance is never negative. `upperY` lies exactly that far above the line, `lowerY` the same amount below it.
- Both control points sit at `mouseX`, one at `upperY` and one at `lowerY`. The two mirror each other around the line, one pulling up and one pulling down, and that mirroring is what bends the curve into an S.
- The blue circles mark all four points, and the two thin blue lines connect each end of the curve with the control point that belongs to it. They are the same handles you dragged in the design program.

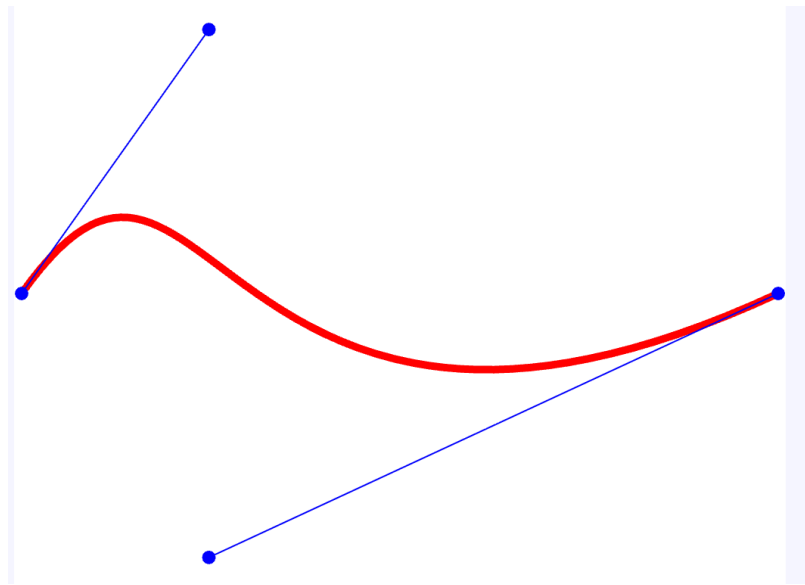


Figure 4.9.8: The finished program: a thick red S curve stretched between two fixed blue points at the left and right edge, with two more blue points above and below the middle line and thin blue handle lines running to them. The mouse cursor sits on the lower control point.

1. **Type it in and move the mouse.** Use the Empty Playground below, either the one with your fan or a fresh one. Format, run, and then move the mouse slowly across the canvas. Watch how the curve chases the two control points without ever touching them.
2. **Aim at the middle line.** Move the mouse until it sits on the height of the two fixed points, at $y = 205$. The distance becomes 0, both control points land on the line, and the curve turns into a straight line.
3. **Experiment.** Swap `upperY` and `lowerY` inside the `bezier` call and you get an arch instead of an S. Then replace both control points with `(mouseX, mouseY)`, so the two become one, and see the simple bow that a single control point produces. Replace `noFill()` with `fill("gold")` and run: the area the curve encloses gets painted, and the four little circles get a gold filling too. Put `noFill()` back, then change colors and stroke weights until it looks the way you want.

💡 Exercise: Empty Playground

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/emptyPlayground.yaml>

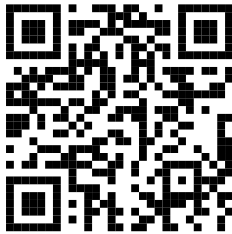
4.9.6 Your exercise: curve patterns with an AI

This is the finale of the Loops part. You take the row recipe from the patterns chapter (Section 4.8.2), where an outer loop notes its position with **push**, an inner loop draws across, and **pop** brings the origin home again, and you combine it with **bezier** into a pattern of your own design. You will not type that program. You describe what you want and an AI writes the code, just like the animal exercise back in the first part (Section 1.5.4). Judging the answer and understanding it is the real work.

💡 Your AI for this exercise

This AI writes code from your prompt. Describe the pattern you want, and paste your working bezier program below your description.

▶ [Open the exercise AI on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/ours6s4x2w>

1. **Design first.** Decide what should repeat before you write a word to the AI. Waves like water, roof tiles, flower petals, woven ribbons, a snake of S curves? Settle on a rough number of rows and columns and on your colors.
2. **Write the prompt.** Describe your pattern precisely, because the AI gives you what you say, not what you mean. Say that you are a beginner and that the code you paste shows everything you have learned. Ask it to keep the structure of your playground, so p5.js and TypeScript, **setup** and **draw**, and explicit type annotations on every declaration. Then paste your working curve program from the section above underneath your description.
3. **Run and iterate.** Put the answer into the playground and run it. If the pattern is not what you pictured, either adjust numbers yourself or send a follow-up prompt saying what should change. Repeat until the result is worth keeping.
4. **Keep only what you understand.** The old rule still decides. Every line you cannot explain gets asked about or thrown out. Format the finished program, and expect to show your pattern in class and explain a line or two of it.

4.9.7 Check your understanding

When your fan and your S curve both run, and you can say in your own words what a control point does, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Curves from straight lines

► Take the quiz on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/upup5ozvg6>

Part 5

Arrays

5.1 Strings have parts: Word Swirrel

Look at the letters below. Which word can you form?

Figure 5.1.1: The Word Swirrel game: the letters of a secret word, scrambled into a new order, wait on a black canvas. Above the canvas, a text box and a Guess button ask: which word can you form?

So far, every variable in your programs has held exactly one thing, one number, one color, one string. This part of the book changes that. You will learn about **arrays**, variables that hold a whole list of things: eleven player names, a hundred bubbles, all the fill levels of a factory. Before the first array appears, you start with something you have used since part 2 without ever looking inside. The string is itself a list, a row of characters, and in this chapter you take one apart and put it back together in a new order. The result is a little game. Word Swirrel scrambles a secret word, and the player has to guess what it was.

5.1.1 AI tutor

Picking a string apart character by character is new territory, and everyone stumbles over the counting once. If a letter comes out that you did not expect, or the word “undefined” shows up on your canvas, tell the tutor what you wrote and what appeared.

Your AI tutor

Hints and questions instead of finished programs, in English or German.

► [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:
<https://app.novedu.at/7xfdgmqh2a>

5.1.2 Look inside a string

Take the string "apple". You know it as one value, but the computer stores it as a row of five characters, and every character has a number, called its **index**:

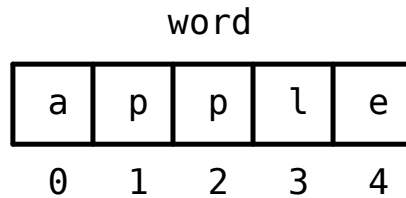


Figure 5.1.2: The string “apple” as a row of five characters. The index of the first character is 0, the index of the last one is 4.

Two new tools let you work with that row. First, `word.length` tells you how many characters the string has. Second, square brackets pick out a single character by its index:

```
const word: string = "apple";
// word.length is 5
// word[0] is "a"
// word[1] is "p"
// word[4] is "e"
```

The index of the first character is 0, not 1. So the last character does not sit at index `word.length` but at `word.length - 1`: five characters, indexes 0 to 4. Read `word[i]` out loud as “the character at position i”.

i Why does counting start at 0?

The index answers the question “how many characters do I have to jump over to reach the one I want?” To reach the first character you jump over none, so its index is 0; to reach the fifth you jump over four, so its index is 4. Programmers count like this everywhere, and from now on you will too.

! One step too far

`word[5]` for “apple” is not an error message, it is worse: the program keeps running and gives you the value `undefined`, which then shows up as the text “undefined” on your canvas or breaks a comparison much later. When a strange `undefined` appears, check your indexes first; the largest valid one is `word.length - 1`.

5.1.3 Growing a string piece by piece

You have built strings before, with template strings like ``Clicks: ${clickCount}``. This chapter adds the second way. You glue strings together with `+`, and you grow a variable with `+=`, exactly like adding numbers to a running total:

```
let scrambledWord: string = "";
scrambledWord += "p"; // scrambledWord is now "p"
scrambledWord += "a"; // scrambledWord is now "pa"
```

The starting point is `""`, the empty string you know as the “nothing yet” value. Each `+=` hangs one more piece on the right end. A variable that collects results round by round is called an **accumulator**, and you have met one before: the running hue of the colored rays worked the same way, only with numbers.

! Course rule update: `+` is now allowed for strings

In the Variables part, the course rule said that strings grow only through template strings, never with `+`. That rule protected you while types were new, because `+` between a number and a string can produce surprises like `"1" + 1` being `"11"`. You know types well enough now: from this part on, `+` and `+=` on strings are allowed and normal. Template strings stay the best choice when you mix text and variables in one sentence.

5.1.4 Cutting pieces out: `substring`

The opposite of gluing is cutting. Every string has the method `substring(start, end)`. It copies the part that begins at index `start` and ends just *before* index `end`. The character at `end` is not included:

```
const flower: string = "flower";
// flower.substring(2, 4) is "ow"
// flower.substring(2) is "ower"
```

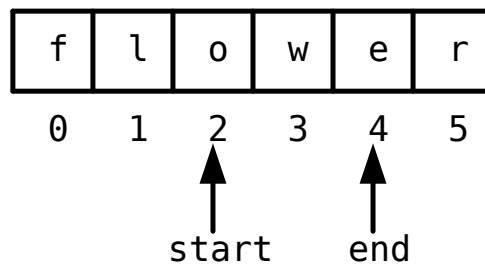



Figure 5.1.3: `substring(2, 4)` copies from the start index up to, but not including, the end index: the characters at indexes 2 and 3.

If you leave out the second number, `substring(start)` copies everything from `start` to the end of the string. And here is the trick this chapter is built on. Combining two cuts removes a single character. Everything before index `i`, glued to everything after index `i`, is the string without the character at `i`:

```
// word is "apple", i is 2
word = word.substring(0, i) + word.substring(i + 1);
// word.substring(0, 2) is "ap", word.substring(3) is "e"
// word is now "ape"
```

One thing `substring` never does is change the string it is called on. It hands you a copy of the piece, and the original stays as it is. To keep the shorter word, you assign the result back to the variable, as in the line above.

5.1.5 The scramble recipe

Now the pieces click together into the heart of Word Swirrel. The secret word waits at the top of the program, in a constant the player never sees on screen, `const wordToGuess = "apple";`. The idea of the scramble is drawing letters out of a bag: pick a random letter of the word, append it to the scrambled result, remove it from the bag, and repeat until the bag is empty.

```
let wordToScramble: string = wordToGuess;
let scrambledWord: string = "";

while (wordToScramble.length > 0) {
  let letterIndex: number = floor(random(wordToScramble.length));
  scrambledWord += wordToScramble[letterIndex];
  wordToScramble = wordToScramble.substring(0, letterIndex)
    + wordToScramble.substring(letterIndex + 1);
}
```

Check the loop against the four steps from the Loops part. The loop variable is unusual: it is the string `wordToScramble` itself.

1. **Initialize.** The bag starts as a copy of the secret word.
2. **Check.** The loop keeps going as long as characters are left.
3. **Do the work.** One random character moves over to `scrambledWord`.
4. **Update.** The substring line makes the bag one character shorter, so `length` sinks by one each round and the loop is guaranteed to end.

Here is one possible run with the word "tea". `random` decides the indexes, so your run will differ:

Table 5.1.1: One possible scramble of the word “tea”.

round	wordToScramble	letterIndex	scrambledWord afterwards
1	"tea"	1	"e"
2	"ta"	1	"ea"
3	"t"	0	"eat"

One small novelty hides in the `random` call: with a single argument, `random(n)` is short for `random(0, n)`. The bag holds `wordToScramble.length` characters, so `floor(random(wordToScramble.length))` is a random valid

index, the dice recipe from the Conditions part with the upper bound taken from the data instead of written as a number.

5.1.6 A function you fill in: guess

Word Swirrel needs one more piece. The player types a guess and clicks a button. The playground page for this exercise has a text box and a Guess button above the canvas, and the starter code contains this:

```
// This method will be called automatically when the user clicks "Guess".
// The guessed text will be in "textInput".
function guess(textInput: string) {
  // <<< Add your code here
}
```

This works like `mouseClicked`. You do not call `guess` yourself; the playground calls it for you when the button is clicked. The part between the parentheses is new. `textInput` is a **parameter**, a variable that the playground fills with the typed text before your code runs. Inside the braces you use `textInput` like any string variable. Treat this as a cliffhanger, left open on purpose. What a function really is, what the line with the parentheses means, and how you write functions of your own is a big enough story that a later part of this book covers it in full. Until then, signatures like this one are always given, and your job is only the body:

```
if (textInput === wordToGuess) {
  background("green");
  text("Correct!", WIDTH / 2, HEIGHT / 2);
} else {
  background("red");
  text(`Wrong!\nIt was "${wordToGuess}"`, WIDTH / 2, HEIGHT / 2);
}
```

You know comparing strings with `===` from rock, paper, scissors, including the catch that capitalization matters. The only new character is `\n` inside the template string: it is not shown as text but stands for a **line break**, so “Wrong!” and the solution appear on two lines.

5.1.7 Your exercise: Word Swirrel

This exercise is a type-in: the code is given, and your job is to type it, run it, and understand every line.

1. **Type in the program.** Use the listings above. The scramble loop goes into `setup`; after it, draw `scrambledWord` centered on the canvas with `text`, and the comparison goes into `guess`. Type, don't paste, because your fingers learn the index and substring syntax now, before the array chapters need it.
2. **Play computer.** Take the word "ape" and scramble it on paper like in the table above (Table 5.1.1), choosing the random indexes yourself. Write down `wordToScramble`, `letterIndex`, and `scrambledWord` for every round.
3. **Experiment.** Change the secret word to your own word. Then print `scrambledWord.length` somewhere on the canvas. Then try `text(wordToGuess[10], ...)` with a short word and watch the `undefined` appear, so you recognize it when it happens by accident.
4. **Read the sample solution.** It is the same program with long comments that explain it, including a picture of how the two substring calls split the word. Read it line by line.

💡 Exercise: Word Swirrel

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfd6fbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0060-Arrays/word-swirrel.yaml>

5.1.8 Check your understanding

When your Word Swirrel runs and the paper scramble of "ape" holds no more questions, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Strings have parts

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/gmqmad45pm>

5.2 Letters criss-cross: rotate

Look at the letters below. Which word can you form?

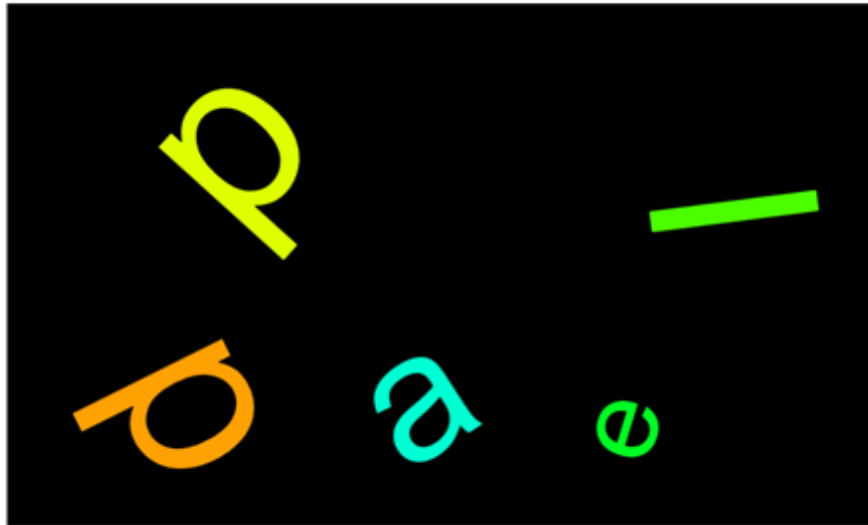


Figure 5.2.1: Word Swirrel, level 2: the letters of the secret word scattered across the canvas, each one at a random position, in a random size, a random color, and tilted at a random angle.

In the first Word Swirrel, the scrambled word sat in the middle of the canvas as one block of text. Level 2 makes it look scrambled too. Every letter gets its own place, its own size, its own color, and its own tilt. Two ideas carry the whole chapter. First, a for loop that visits a string one character at a time; you will write this exact loop shape for the rest of the course. Second, a new transformation: **rotate**, the missing sibling of **translate**, which tilts everything you draw. Scrambling is not even needed anymore; letters thrown all over the canvas are puzzle enough.

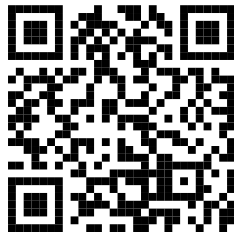
5.2.1 AI tutor

Rotation is famous for surprising beginners: you rotate a letter and it swings to a completely different place. If that happens, tell the tutor which transformation calls you make in which order, and where the letter lands.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:
<https://app.novedu.at/7xfdgmqh2a>

5.2.2 One letter at a time

The scramble loop of the last chapter ate the word up. This chapter needs something gentler. Visit every character once, in order, and draw it. That is a for loop over the indexes:

```
for (let i: number = 0; i < wordToGuess.length; i++) {  
  // wordToGuess[i] is the character of this round  
}
```

Read the header with the indexes in mind: `i` starts at 0, the first index. The loop runs while `i <`

`wordToGuess.length`, and stops exactly when `i` reaches the length, which is one past the last index. For "apple" (length 5), `i` takes the values 0, 1, 2, 3, and 4, every index once and no `undefined`. The `<` does the off-by-one thinking for you; with `<=` the loop would run once too often and read past the end.

Learn this loop by heart. "From 0, while less than length, one step at a time" is *the* pattern for walking through anything that has indexes, and the coming array chapters use it on every page.

5.2.3 rotate: turning the paper

You know `translate`; it moves the origin, and everything drawn afterwards moves with it. `rotate` is its sibling: it turns the whole coordinate system, and everything drawn afterwards is tilted. The angle is in degrees, clockwise, as long as `angleMode(DEGREES)` stands in `setup`, the same setting you used for `arc` in part 1. Every program in this chapter sets it.

The catch is that `rotate` turns the paper **around the current origin**, the point (0, 0). A shape close to the origin tilts almost in place; a shape far from the origin swings around it like a hammer thrower. That is why rotation almost always comes as a team of three:

```
translate(x, y); // 1. move the origin to where the shape belongs
rotate(angle);   // 2. turn the paper around that point
text(letter, 0, 0); // 3. draw AT the origin
```

Move first, then turn, then draw at (0, 0). The letter sits at (x, y) and is tilted by `angle`, and because it sits *on* the pivot point, it tilts in place instead of swinging away. You met the same draw-at-the-origin idea in the pattern recipe (Section 4.7.3); rotation makes it essential.

i Rotate turns the system, not one shape

Like `translate`, `rotate` does not touch any shape that is already drawn. It changes the coordinate system for everything that comes after it, and a second `rotate` adds to the first. That is also why the order matters: translate-then-rotate pins the letter to its spot; rotate-then-translate makes the move itself happen in a tilted direction.

5.2.4 The scatter recipe

Now the loop and the transformation team up. One round of the loop places one letter, and everything about the letter is random:

```
for (let i: number = 0; i < wordToGuess.length; i++) {
  const x: number = random(MARGIN, WIDTH - MARGIN);
  const y: number = random(MARGIN, HEIGHT - MARGIN);
  const charSize: number = random(50, 150);
  const textColor: number = random(0, 360);
  const angle: number = random(-90, 90);

  push();
  translate(x, y);
  rotate(angle);
  fill(textColor, 100, 100);
  textSize(charSize);
  text(wordToGuess[i], 0, 0);
  pop();
}
```

A few details are worth a second look:

- **push and pop wrap every letter.** Translate and rotate calls add up, so without the note-taking, letter 2 would be measured from letter 1's tilted position, and the scatter would drift off the canvas. Each round starts fresh from the saved origin.
- **The random values are decimals.** So far you mostly wrapped `random` in `floor` to get whole numbers, like dice. Positions, sizes, and angles don't need whole numbers: `rotate(23.7)` is a perfectly fine tilt, so the floor stays away.
- **Negative angles are allowed.** `random(-90, 90)` tilts letters up to a quarter turn in either direction; a negative angle turns counterclockwise.

- **The color is HSB.** With `colorMode(HSB)` in `setup`, `fill(textColor, 100, 100)` with a hue from 0 to 360 gives full saturation and brightness, like in the colored rays.
- **MARGIN keeps letters visible.** The random position stays `MARGIN` pixels away from every edge, so even a big rotated letter is not cut off.

One more thing will greet you when you open level 2: the `guess` function comes finished in the starter code. You built it yourself last chapter; this time it is simply there, and your work happens in `setup`.

5.2.5 Your exercise: Word Swirrel (Level 2)

1. **The loop without rotation.** Write the for loop with the random position, size, and color, but leave `rotate` out: `translate`, then draw the letter at (0, 0). Run it; the letters scatter, still standing upright.
2. **Add the tilt.** Put `rotate(angle)` between `translate` and `text`, run, and compare with the goal picture. If a letter seems to vanish, check the order of your three steps (Section 5.2.3).
3. **Break it on purpose.** Comment out `push()` and `pop()` and predict the picture before you run. The transformations pile up, and the letters wander away. Put them back.
4. **Run it a few times.** Every run scatters differently, because `setup` runs once per start and asks `random` for fresh values each time. Play the game with a classmate: they guess, you check.

Exercise: Word Swirrel (Level 2)

► [Open in the playground](#)



Scan the code, or type this address into your browser:

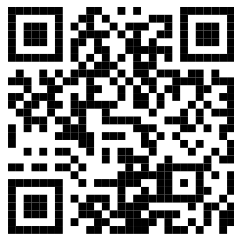
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0060-Arrays/word-swirrel-1.yaml>

5.2.6 Check your understanding

When your letters scatter and tilt like in the goal picture, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

Quiz: Letters criss-cross

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/qodslscj8y>

5.3 Your first array

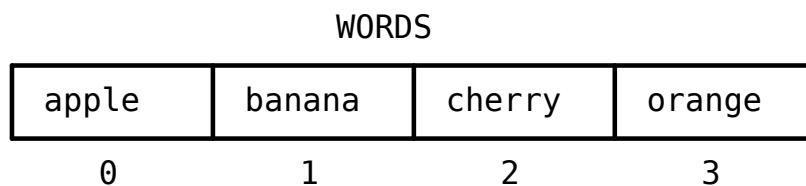


Figure 5.3.1: An array of strings: one variable that holds a whole row of values. Every value in the row has an index, and the counting starts at 0, exactly like in a string.

Word Swirrel can still only play one word; whoever read the code knows the answer forever. Level 3 fixes that with the idea this whole part is named after. An **array** is a variable that holds a whole list of values: twenty words, and the program picks one at random on every start. The change to your code is tiny, three lines, but the idea is one of the biggest in programming. From now on, “many things” no longer means “many variables”.

5.3.1 AI tutor

Arrays bring new syntax: brackets, commas, a strange type annotation. If the red squiggles pile up when you type your first array, show the tutor your lines; the tutor finds a lost comma or bracket in seconds.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/7xfdgmqh2a>

5.3.2 From one string to many

Here is an array that holds four strings:

```
const WORDS: string[] = [  
    "apple",  
    "banana",  
    "cherry",  
    "orange"  
];
```

Read it in three parts. The type annotation `string[]` says that this variable holds an *array of strings*, a row of values where each one is a string. The square brackets on the right are the **array literal**: the elements, separated by commas, between `[` and `]`. Writing each element on its own line is not required, but for long lists it keeps the code readable, and you may leave a comma after the last element too. The name is in ALL_CAPS because the word list is configuration, like `SIZE` and `GRID` were, so the naming rules don't change.

The picture at the top of the chapter shows how to think about it, a row of boxes, each box numbered, counting from 0. That should feel familiar, because a string is exactly the same picture with characters in the boxes. The last chapter's sentence “a string is a row of characters” now gets its general form: an array is a row of anything. There are arrays of numbers too, written `number[]`; they carry the soccer players' shirt numbers in a later chapter.

5.3.3 Picking one at random

You know `random` as a number machine. `random(6)` returns a random decimal number. Hand it an array instead, and it does something much friendlier:

```
wordToGuess = random(WORDS);
```

`random(WORDS)` returns one random **element** of the array, not a number. With the four-word array above, the result is "apple", "banana", "cherry", or "orange", each equally likely. No index arithmetic, no `floor`: `p5.js` picks the element for you. One call per program start, and every run plays a different word.

5.3.4 A variable declared empty

The third change is the sneakiest. In level 2, the secret word was `const wordToGuess = "apple"`. Now the word is only known once `setup` has rolled the dice, so the declaration and the assignment split up:

```
let wordToGuess: string; // declared, no value yet

function setup() {
  // ...
  wordToGuess = random(WORDS);
  // ...
}
```

You have seen this shape once before, with `let message: string;` in the Conditions part: declare with a type, assign before you read. A `const` cannot do this split, because a `const` must get its value at the declaration. Level 3 therefore switches the keyword to `let`, and you will find that change already made when you open the exercise.

The trap: declaring it inside setup

It is tempting to write `const wordToGuess = random(WORDS);` inside `setup`, and the scramble loop even works. But then the variable belongs to `setup` alone, and the `guess` function, which compares the player's input against `wordToGuess`, gets the red squiggle "Cannot find name 'wordToGuess'". The word must live in a global variable, declared outside every function, so that `setup` can write it and `guess` can read it. That is exactly the global- variable rule from the Variables part: global only when two functions need the same value, and here they do.

5.3.5 Your exercise: Word Swirrel (Level 3)

The starter code is the finished level 2. Three changes make it level 3:

1. **Type the array.** Above the constants, declare `const WORDS: string[]` with about 20 words from the exercise page, one per line. Type it; your fingers learn where the commas, quotes, and brackets go.
2. **Split the declaration.** Change `let wordToGuess = "apple";` to the empty declaration `let wordToGuess: string;` and assign `wordToGuess = random(WORDS);` at the top of `setup`, before the letter loop.
3. **Run it a few times.** A different word should scatter on every start. Guess a few yourself; with 20 words, even you won't always know the answer.
4. **Experiment.** Add your own words, short and long; the letter loop follows `wordToGuess.length`, so nothing else needs to change. Then shrink `WORDS` to a single word and run twice, because with one element, `random` has no choice.

💡 Exercise: Word Swirrel (Level 3)

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

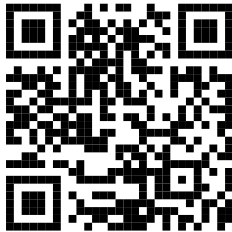
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0060-Arrays/word-swirrel-2.yaml>

5.3.6 Check your understanding

When your game plays a fresh word on every run, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Your first array

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/tvojrl68hj>

5.4 Scale: bigger and smaller

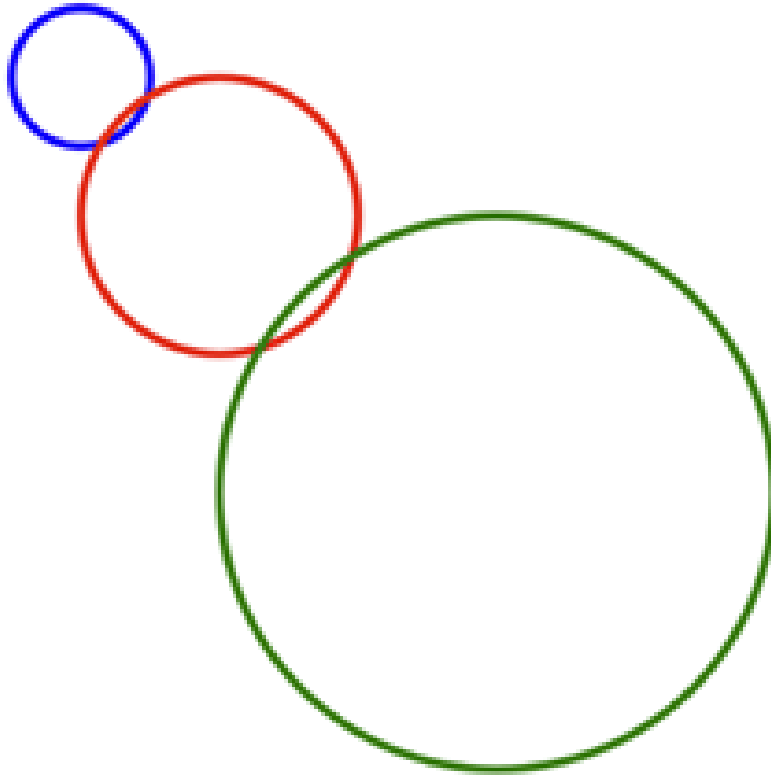


Figure 5.4.1: Three circles drawn by the exact same call, `circle(50, 50, 50)`: blue at scale 1, red at scale 2, green at scale 4. Scaling grows the circles, and it moves them too.

This short chapter takes a break from strings and arrays, on purpose, because the exercises after it need one more drawing tool. Back in the practice projects of part 1, the soccer field recipe contained three magic words: `push`, `translate`, and `scale`. The Loops part explained the first two. `scale` is the last one, and after this chapter the recipe holds no secrets. `scale` makes everything you draw bigger or smaller by a factor, and, as the goal picture shows, it does something to positions that surprises almost everyone.

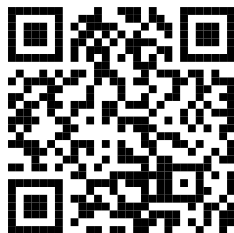
5.4.1 AI tutor

“I scaled my circle and it jumped somewhere else” is one of the most common confusions in this course, and this chapter exists to clear it up. If the pictures still surprise you after reading, ask the tutor to walk through one example with you, step by step.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/7xfdgmqh2a>

5.4.2 Scale multiplies everything

`scale(2)` tells the canvas to multiply everything by 2 from now on. Like `translate` and `rotate`, it changes the coordinate system for everything drawn after it, and `pop` undoes it. But “everything” really means everything:

```
scale(2);
circle(50, 50, 50);
```

The circle’s diameter becomes 100, and its center lands at (100, 100), because the *coordinates* are multiplied too. The call still says (50, 50); the scaled coordinate system puts that point twice as far from the origin.

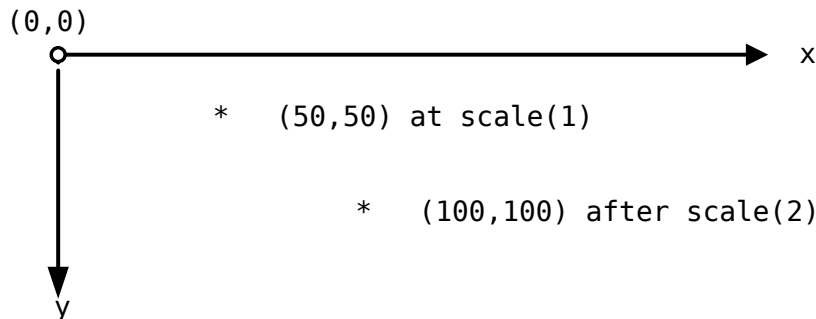


Figure 5.4.2: Scaling multiplies coordinates: the point (50, 50) lands where (100, 100) used to be, twice as far from the origin. Only the origin itself stays put.

Even the stroke is multiplied: a `strokeWeight(3)` line drawn under `scale(2)` comes out 6 pixels wide. The cure is compensation. Set `strokeWeight(3 / 2)` instead, and the scaling doubles the one and a half pixels back to the 3 the eye expects; the exercise code below plays this trick for every scaled circle. Factors below 1 work as well; `scale(0.5)` shrinks everything to half size, and decimals like `scale(2.5)` are fine.

5.4.3 Your exercise: Scaled Circles

The first exercise is a type-in. Here is the code; the three blocks draw the three circles of the goal picture:

```
stroke("blue");
strokeWeight(3);
circle(50, 50, 50);

push();
stroke("red");
scale(2);
strokeWeight(3 / 2);
circle(50, 50, 50);
pop();

push();
stroke("green");
scale(4);
strokeWeight(3 / 4);
circle(50, 50, 50);
pop();
```

1. **Predict first.** Before you run it, write down where the red and green circles will land and how big they will be. Center and diameter, both times.
2. **Type it in and run.** Compare with your prediction and with the goal picture. The first block needs no `push/pop` because it never touches the coordinate system; the other two clean up after themselves, as every transformation should (Section 4.7.4).
3. **Experiment.** Change the factors, try `scale(0.5)`, and remove one `strokeWeight` compensation to see the fat outline. One thing you cannot fix by experimenting yet: making the circles share a center. That is the second exercise.

💡 Exercise: Scaled Circles

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0060-Arrays/scaled-circles.yaml>

5.4.4 The point that cannot move

Why did the circles drift? Because their center (50, 50) is a distance away from the origin, and scaling multiplies that distance. Which point does *not* drift? The origin itself: multiplying 0 by any factor gives 0. The origin is the fixed point of scaling, the one spot that stays exactly where it is.

That observation turns into a recipe you already know from `rotate` (Section 5.2.3): put the origin where the action should happen, then draw at (0, 0).

```
translate(150, 150); // the shared center

circle(0, 0, 50);    // blue, scale 1

push();
scale(2);
circle(0, 0, 50);    // red: bigger, but NOT moved
pop();
```

A circle whose center is (0, 0) sits on the fixed point: `scale` multiplies its diameter, but its center, 0 away from the origin in both directions, stays put. Translate once, draw everything at the origin, and scaling changes only the size. That is precisely how the part 1 soccer field recipe worked; `translate` placed the field, and `scale` turned meters into pixels around it.

5.4.5 Your exercise, part 2: Scaled Circles (Centered)

The starter code is the finished first exercise. The goal picture shows the fix: three concentric rings, blue inside red inside green.

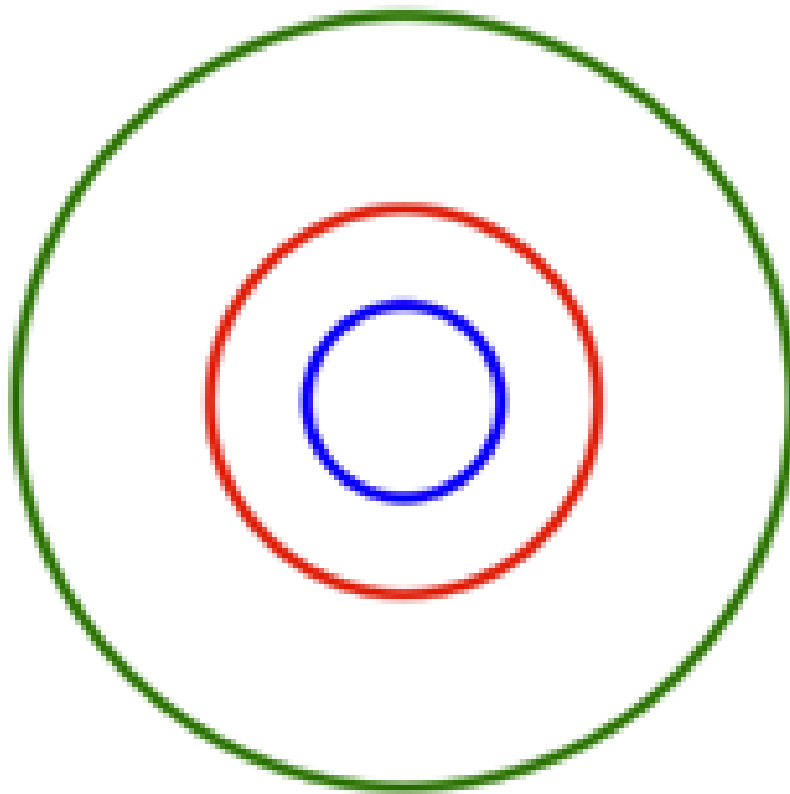


Figure 5.4.3: The centered version: the same three scaled circles, now sharing one center, drawn at the origin after a single translate.

1. **Pick the center.** The biggest circle has diameter 50 times 4, which is 200 pixels, so the shared center must stay at least 100 pixels away from every edge of the 400 by 400 canvas. The goal picture uses (150, 150), but any spot with that safety margin works.
2. **One translate for all.** Move the origin to your center once, at the top, before the blue circle. This translate stays outside every `push/pop` pair on purpose: all three circles should share it.
3. **Draw at the origin.** Change all three `circle` calls to `circle(0, 0, 50)` and run. The scale factors stay as they are; only the coordinates change. If your rings are still not concentric, one of your circles is not being drawn at (0, 0).

💡 Exercise: Scaled Circles (Centered)

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0060-Arrays/scaled-circles-centered.yaml>

5.4.6 Check your understanding

When your rings sit inside each other and you can say *why* the blue circle never moves, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Scale

► Take the quiz on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/7biclsrq9y>

5.5 Parsing: numbers hidden in a string

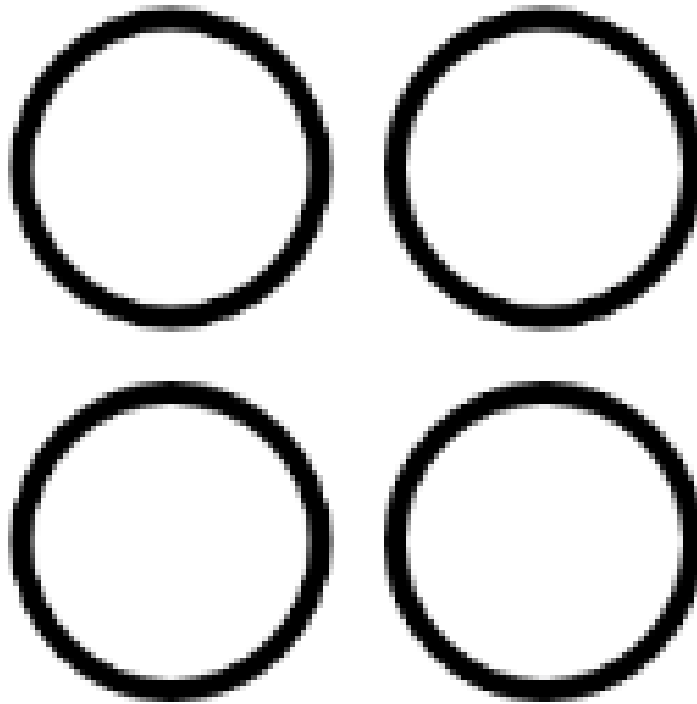


Figure 5.5.1: Four circles in a square arrangement, drawn from the string “50,50 50,100 100,50 100,100”. No coordinate appears in the drawing code; the program reads them all out of the string, character by character.

The string “50,50 50,100 100,50 100,100” describes four circles, and any person can read it. A program can too, but it has to work for it, because a string is just a row of characters, and “5” next to “0” is not yet the number 50. Pulling information out of a string is called **parsing**, and it is everyday programmer work: files, web addresses, and chat messages all arrive as strings. In this chapter you write your first parser, with the tools you already own, one character at a time. The reward is a program that draws whatever the string tells it to.

5.5.1 AI tutor

Parsers are precise little machines, and a single confused index or a forgotten reset produces numbers like 5010 instead of 50 and 10. If your circles land in strange places, tell the tutor your input string and what got drawn; together you can trace where the digits went.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/7xfdmqh2a>

5.5.2 Numbers hidden in characters

The string "42" and the number 42 look alike but are different values with different types, and the difference matters. You can multiply the number, but not the string. The bridge between them is `parseInt`:

```
const n: number = parseInt("42"); // n is the number 42
```

`parseInt` reads a string of digits and returns the whole number they spell. It is the missing half of a pair you already know: a template string turns a number into text for `text(...)`, and `parseInt` turns text back into a number you can calculate and draw with.

5.5.3 The collecting loop

Now to the parser. The plan sounds like something a person would do: walk through the string character by character, and sort every character into one of three cases.

- A **digit** belongs to the number currently being spelled out; remember it.
- A **comma** says that the x coordinate is complete.
- A **space** says that the y coordinate is complete, and with x and y known, a circle can be drawn.

“Remember it” is a job for an accumulator string, called a **buffer**: it collects digits until a separator arrives, then the program converts it and empties it for the next number. The input string itself waits in a constant named `circles`, holding exactly the goal picture’s “50,50 50,100 100,50 100,100”.

```
let x: number = 0;
let y: number = 0;
let coordinates: string = ""; // the buffer

for (let ix: number = 0; ix < circles.length; ix++) {
  switch (circles[ix]) {
    case ",":
      x = parseInt(coordinates);
      coordinates = "";
      break;
    case " ":
      y = parseInt(coordinates);
      circle(x, y, 40);
      coordinates = "";
      break;
    default:
      coordinates += circles[ix];
      break;
  }
}
```

The loop is the letter loop from the criss-cross chapter, and the new part is what the body does with each character. The `switch` compares the character against “,” and “ ”, and its `default` branch catches everything else, which here means the digits. Trace the start of “50,50 ...” by hand: “5” and “0” fall into `default`, so the buffer grows to “50”. The comma converts it into `x = 50` and empties the buffer. Two more digits, then the space completes `y = 50`, and the first circle appears.

5.5.4 The leftover after the loop

Play computer with the loop on the full string and count the circles it draws: three, not four. The last pair “100,100” has no space after it; the string simply ends. The comma still sets `x = 100`, the digits of the y coordinate fill the buffer, and then the loop is over. Nothing ever told the program “the last number is complete”.

The fix is two lines *after* the loop:

```
y = parseInt(coordinates);
circle(x, y, 40);
```

Parsers meet this pattern constantly: separators come *between* values, so the loop handles every value except the last one, and you finish the last one after the loop. When you trace your parser on paper, always check

what the buffer still holds when the loop ends.

5.5.5 Your exercise: Parse Coordinates

1. **Paper first.** Take the short string "7,8 9,4" and play computer: one table row per character, with columns for the character, the case it falls into, the buffer afterwards, and what happens to x and y. Two circles should fall out, at (7, 8) and (9, 4), the second one only in your after-the-loop step.
2. **One pair by hand.** In code, extract the first pair of the real string without a loop: `circles[0]`, `circles[1]`, glue them, `parseInt`, and draw one circle. This warms up the indexing.
3. **The loop.** Replace the hand extraction with the collecting loop and the after-loop flush. Run it; four circles should appear, like the goal picture.
4. **Stress test.** Change the string: five circles, coordinates with one and with three digits. Your parser must not care, because the buffer collects any number of digits.

One rule from the exercise: you must not use the `split` function. `split` exists, and one day it will do this job in a single line for you, but it would also do the thinking for you. The character loop is the skill this chapter is about.

💡 Exercise: Parse Coordinates

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfd6bjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0060-Arrays/parse-coordinates.yaml>

5.5.6 Remembering the previous point

The second exercise feeds your parser a new string, seven points that outline a hexagon, and changes the job. Don't draw a circle per point; connect the points with lines. A line needs *two* points, the current one and the one before it, so the parser must remember the previous point across rounds. The following snippet does that, and it contains one operator you have not met yet, the `!`; its explanation comes right below the code:

```
if (!isFirstPoint) {  
    line(x1, y1, x2, y2);  
}  
  
x1 = x2;    // the current point becomes  
y1 = y2;    // the next line's start point  
isFirstPoint = false;
```

This replaces the `circle` call in the space case: `(x2, y2)` is the point just parsed, `(x1, y1)` the one from the previous round, and after drawing, the current point slides into the “previous” role.

The very first point is special; nothing comes before it, so there is nothing to connect it to yet. A boolean variable `isFirstPoint`, starting as `true`, guards the `line` call, and here you meet one last new operator: `!`, spoken “not”, flips a boolean around. `!isFirstPoint` reads as “not the first point”, so the line is drawn for every point except the first. Until now the course wrote comparisons like `isFirstPoint === false`; from this chapter on, the short `!` form is allowed and preferred.

When you read the sample solution later, notice one more choice: it sorts the characters with `if / else if / else` instead of `switch`. Both fit; comparing a character against a few fixed values is exactly what both were made for. Pick the one you find easier to read, but be able to read both.

5.5.7 Your exercise, part 2: Parse Coordinates 2

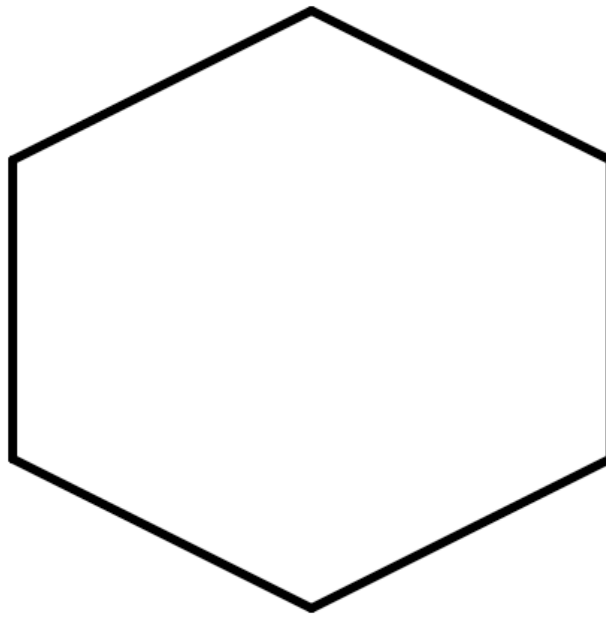


Figure 5.5.2: The hexagon from the second exercise: seven parsed points, connected by six lines. The first point is repeated at the end of the string, which closes the shape.

1. **Look at the data.** The starter string has seven points, and the first and the last are the same: 200,100. Six lines connect seven points, and the repeated point is what closes the ring. No special “close the shape” code needed.
2. **Paper first.** Trace "50,50 150,100 100,200" through the new space case: which line is drawn at each space, what do x1, y1, x2, y2 hold afterwards, and what happens after the loop?
3. **Convert your parser.** Start from your circle parser: the comma case now fills x2, the space case draws the guarded line and shuffles current into previous. Don't forget the flush; the last line is drawn after the loop.
4. **Your own shape.** Write a point string for a triangle or a star, first on graph paper, then in the constant. The parser draws whatever you feed it; that is the whole point of parsing.

Exercise: Parse Coordinates 2

 [Open in the playground](#)



Scan the code, or type this address into your browser:

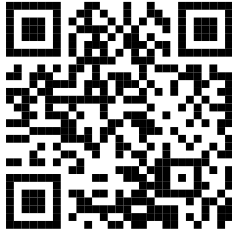
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0060-Arrays/parse-coordinates-2.yaml>

5.5.8 Check your understanding

When your hexagon closes and your paper trace of "7,8 9,4" comes out right, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Parsing

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/oiuzggq1as>

5.6 Parallel arrays: the starting eleven



Figure 5.6.1: The finished dashboard for the TV station: a to-scale soccer field with the starting lineup on a semi-transparent plate. Numbers orange and right-aligned, names yellow and left-aligned, all read from arrays. The screenshot shows the row numbers 1 to 11; your version will show the players' real shirt numbers instead.

A local TV station hires you. Before the match, they want to show the starting lineup on a drawn soccer field. This is the first real job for arrays. Eleven players means eleven numbers, eleven first names, eleven last names, and nobody wants 33 variables. Three arrays carry all of it, and one loop paints the whole list. Along the way, the soccer field itself brings back an old friend: the part 1 practice project drew this very field with a magic recipe, and you now own every word of it.

5.6.1 AI tutor

The field has many small parts, and real-world measurements make every number mean something. If your penalty box sits in the wrong spot or your names refuse to line up, describe to the tutor what you measured and what the canvas shows.



Your AI tutor

Hints and questions instead of finished programs, in English or German.

► [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/7xfdgmqh2a>

5.6.2 Three arrays, one team

The starter code hands you the team as data:

```
const playerNumbers: number[] = [1, 17, 22, 23, 20, 6, 15, 5, 11, 7, 9];
const firstNames: string[] = ["Thibaut", "Lucas", "Antonio", /* ... */];
const lastNames: string[] = ["Courtois", "Vázquez", "Rüdiger", /* ... */];
```

`number[]` is the second array type promised earlier, a row of numbers, here the shirt numbers. And reading an element works exactly like reading a character of a string, with square brackets and a zero-based index: `firstNames[1]` is "Lucas", `playerNumbers[1]` is 17.

The three arrays are a team in a second sense: **the same index belongs to the same player**. Index 1 means Lucas Vázquez with shirt number 17; three arrays, one person per column. Arrays used this way are called **parallel arrays**, and they only work while you keep the promise that all of them have the same length and the same order. Swap two names in `lastNames` without swapping the first names, and player 1 becomes Lucas Rüdiger on live TV.

index	0	1	2
playerNumbers	1	17	22
firstNames	Thibaut	Lucas	Antonio
lastNames	Courtois	Vazquez	Rudiger

Figure 5.6.2: Parallel arrays: one column, one player. The index `i` ties `playerNumbers[i]`, `firstNames[i]`, and `lastNames[i]` together.

5.6.3 The loop over an array

Drawing the list is the letter loop of the criss-cross chapter, applied to an array. From 0, while less than length, one step at a time:

```
fill("yellow");
textAlign(LEFT);
for (let i: number = 0; i < firstNames.length; i++) {
  text(`${firstNames[i]} ${lastNames[i]}`, 33, 12 + i * 5);
}
```

Three details matter here. First, `firstNames.length` is 11, so the loop visits each player once; add a twelfth player to all three arrays and the loop follows without a change. Second, the template string glues the two name parts of the *same index* together; that is a parallel-array read. Third, the y position `12 + i * 5` computes the row from the index, exactly like the zebra stripes computed their y from the loop variable: index 0 draws at 12, index 1 at 17, each player 5 units lower.

The numbers will work the same way in a second loop that you write in the exercise, with `textAlign(RIGHT)` so the digits line up like a table, orange fill and bold `textStyle(BOLD)`, and `text(playerNumbers[i], 30, 12 + i * 5)`. Two loops, same row formula, and the columns will meet neatly at the gap between `x = 30` and `x = 33`.

5.6.4 Meters as pixels

A real soccer field is 100 by 70 meters, and the starter code draws it with those exact numbers:

```
createCanvas((100 + MARGIN * 2) * SCALE, (70 + MARGIN * 2) * SCALE);
// ...
push();
scale(SCALE);
translate(MARGIN, MARGIN);
rect(0, 0, 100, 70); // the field, in meters
```

This is the part 1 soccer recipe, and after the last chapter you can read every line of it: `scale(SCALE)` multiplies everything drawn afterwards by 5, so one “meter” becomes 5 pixels; `translate` shifts the field by the margin, *in meters*, because the translate happens after the scale. From then on, every measurement from the real world goes into the code unchanged; the penalty box is 16.5 deep because it really is 16.5 meters deep. Even `strokeWeight(0.5)` speaks meters, with lines half a meter wide, like real field markings. The canvas size does the same computation in reverse, so the window always fits the field plus its margins.

5.6.5 Field details worth stealing

Two drawing tricks in this exercise deserve a place in your toolbox:

- **The penalty arc without arc math.** The curve at the penalty box is officially a circle of radius 9.15 meters around the penalty spot, but only the part outside the box is drawn. The trick is to draw the *full* circle and then paint the penalty area over it with an opaque green rectangle, so only the arc survives. Painting over is a legal move; you have known that since the very first chapter.
- **See-through color.** The name plate is `fill("#008000d0")`: a hex color with *eight* digits. The first six are the RGB color you know from the Variables part; the extra two are the **alpha value**, the transparency. `d0` is 208 of 255, about 80 percent opaque, so the field shimmers through the plate. Alpha `ff` is fully opaque, `00` fully invisible.

The corners are little quarter circles, drawn with `arc` and degree angles, the same `arc` you learned in part 1. The goal areas, the center circle, and the penalty spots are plain rects, circles, and points measured off the official diagram.

5.6.6 Your exercise: Soccer Field

1. **Real units first.** Look up the field measurements linked in the exercise (goal area 5.5 by 18.32, penalty area 16.5 by 40.32, center circle radius 9.15) and keep the official diagram with its numbers next to you while you code. The code just reads that diagram aloud.
2. **Task 1: the field.** The starter draws the outer rectangle; you add the center line, center circle, goal and penalty areas, penalty spots, penalty arcs (full circle, then paint over), and the corner arcs. Change `SCALE` to 2 and 3 between runs; the whole field must follow.
3. **Task 2: the lineup.** Two loops over the arrays, numbers right-aligned in orange and bold, names left-aligned in yellow (Section 5.6.3). One heads-up: the exercise's screenshot shows the row numbers 1 to 11, but your program must print the real shirt numbers from `playerNumbers`, so your plate will show 1, 17, 22, and so on. Print the numbers, not the index.
4. **Your team.** Replace the eleven with your favorite lineup. All three arrays must change together; that is the parallel- array promise.

💡 Exercise: Soccer Field

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

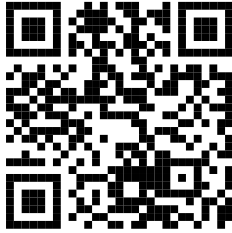
`https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0060-Arrays/soccer-field.yaml`

5.6.7 Check your understanding

When your field survives a `SCALE` change and your own team stands on the plate, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Parallel arrays

► Take the quiz on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/yuvov6jmfj>

5.7 Arrays that grow and shrink

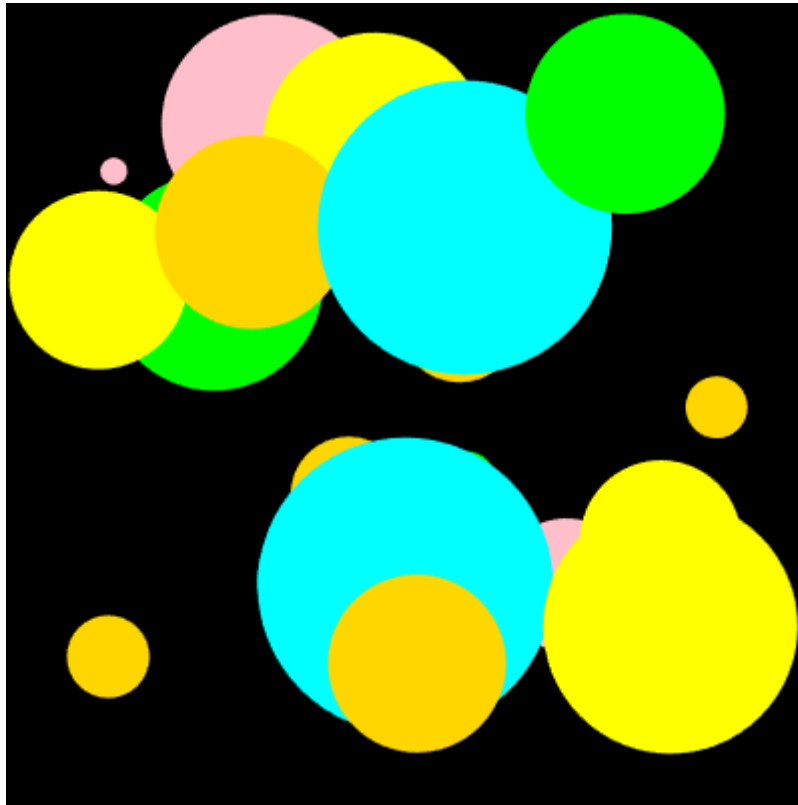


Figure 5.7.1: The bubbles program after a few seconds: colored circles of random sizes cover the canvas, and a new one appears every half second or two. Every bubble lives in four parallel arrays; the canvas only shows what the arrays remember.

The soccer arrays were fixed, eleven players at the start, eleven at the end. This chapter cuts arrays loose. The bubbles program starts with *empty* arrays and grows them while it runs; every second or two, **push** appends a new bubble. Then the follow-up exercise teaches the opposite move, **splice**, which removes a bubble whose time is up. Along the way you meet **millis**, the program's clock, and the biggest idea of the chapter, the one that carries every animated program from now on: the arrays are the program's memory, and the canvas is just a picture of them, repainted sixty times a second.

5.7.1 AI tutor

Growing and shrinking arrays are the first data that changes while you watch. If your bubbles change color mid-flight or vanish in the wrong order, something knocked your arrays out of step; describe to the tutor what you see and paste the lines that change your arrays.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/7xfdmqh2a>

5.7.2 The empty array

An array does not need its elements at the declaration. This is an array of numbers that holds nothing yet:

```
const circlesDiameter: number[] = [];
```

The type annotation `number[]` now really earns its place. With no elements between the brackets, the annotation is the only clue about what the array will hold. From an empty start, `push` appends elements at the end:

```
circlesDiameter.push(80); // length is now 1
circlesDiameter.push(35); // length is now 2, at indexes 0 and 1
```

`push` on an array and `pop` for the drawing settings share a name family by accident; they have nothing to do with each other. The array `push` simply means: hang this value on the end. Each push raises `length` by one, and the new element gets the next free index.

i A `const` that changes?

`const circlesDiameter` and then `push` into it, without a red squiggle? Yes. `const` forbids *reassigning the variable*, like `circlesDiameter = [1, 2, 3];`. What is stored *inside* the array may change freely. The box is nailed down; the content is not. This is normal TypeScript style: an array variable is `const` unless you plan to replace the whole array.

5.7.3 Four arrays, one bubble

A bubble has a position, a size, and a color, so the program keeps four parallel arrays, `circlesCenterX`, `circlesCenterY`, `circlesDiameter`, and `circlesFill`. Creating one bubble means one push into *each* of them; the size comes first, drawn up to a constant `maxDiameter`:

```
const d: number = random(10, maxDiameter);
circlesDiameter.push(d);
circlesCenterX.push(random(d / 2, width - d / 2));
circlesCenterY.push(random(d / 2, height - d / 2));
circlesFill.push(random(availableColors));
```

The position uses the diameter: the center stays at least half a bubble away from every edge, so no bubble is ever cut off. The color comes from `random(availableColors)`, the pick-an-element `random` from Word Swirrel level 3, applied to a little palette array of color names.

! Keep the arrays in step

Parallel arrays that grow have a new way to break: forget one of the four pushes, and from that bubble on, every color belongs to the wrong circle, and the last bubble reads `undefined`. Whenever you push, push into all parallel arrays, in one place, together.

5.7.4 The program's clock: millis

New bubbles should appear every half second to two seconds, not sixty times a second. For that, the program needs a clock. `millis()` returns the number of milliseconds since the program started. The scheduling trick is to store a *deadline*:

```
if (millis() >= nextCircle) {
  // ... push the four values ...
  nextCircle = millis() + random(500, 2000);
}
```

`nextCircle` holds the time at which the next bubble is due. Every frame, `draw` checks the clock against the deadline. As long as the bubble is not due, nothing happens. Once it is due, the code creates the bubble and sets a new deadline half a second to two seconds into the future. The wall clock keeps running; the program just compares against it. The deadline pattern is a good fit whenever the pause changes from round to round and the work belongs to the drawing anyway, and both are true here. A later chapter adds a second way to say “every so often”, for jobs that need a fixed rhythm instead.

5.7.5 The arrays are the truth

Here is `draw`, with the push details folded into a comment, because its shape matters more than any line:

```
function draw() {
  background("black");

  if (millis() >= nextCircle) {
    // push a new bubble, set the next deadline
  }

  noStroke();
  for (let i: number = 0; i < circlesDiameter.length; i++) {
    fill(circlesFill[i]);
    circle(circlesCenterX[i], circlesCenterY[i], circlesDiameter[i]);
  }
}
```

`background("black")` wipes the canvas *every frame*, sixty times a second, and yet no bubble is ever lost. Why not? Because the bubbles don't live on the canvas; they live in the arrays. The loop redraws every stored bubble from scratch each frame, so the picture looks constant even though it is repainted constantly.

Compare that with the colored rays back in the Loops part, where the canvas itself was the memory; the program never repainted the background and the rays piled up. Both styles work, but the array style is stronger: the program can *change* its memory, and the picture follows. That is exactly what the second half of this chapter does, when bubbles start to burst.

5.7.6 Your exercise: Bubbles

The exercise is a type-in of the bubbles program; the complete code is in the exercise description as a picture, so your fingers type every bracket themselves.

1. **Type it in, in three sittings.** First the five global declarations and the palette array, then the deadline `if` in `draw`, then the drawing loop. Run after each sitting; after the second one you'll see nothing yet, and that is fine, because nothing reads the arrays so far.
2. **Watch it grow.** Run the finished program for a minute. New bubbles keep coming, old ones never leave. The arrays only ever grow, and after a while the canvas is packed. Remember that sight; the second exercise fixes it.
3. **Experiment.** Add your favorite colors to the palette, make the deadlines shorter, and try `random(10, 40)` for tiny bubbles only. Each change touches exactly one line, because everything else follows the data.

💡 Exercise: Bubbles

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0060-Arrays/bubbles.yaml>

5.7.7 Removing elements: splice

Bubbles should burst. The removing tool is `splice`. `myArray.splice(i, 1)` removes one element at index `i`. The elements after it slide one index to the left, and `length` shrinks by one:

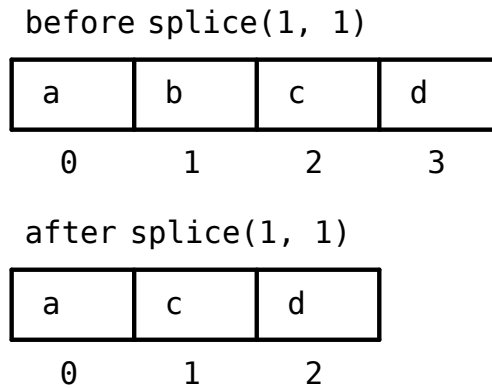


Figure 5.7.2: splice(1, 1) removes the element at index 1. Everything behind it slides one index to the left, and the array gets shorter.

For bursting, every bubble gets a lifetime. A *fifth* parallel array holds it, filled at creation with its personal deadline, the same pattern as `nextCircle`:

```
circlesLifetime.push(millis() + random(2000, 4000));
```

Every frame, a loop checks which bubbles are overdue and splices them out of **all five** arrays, so the columns stay aligned:

```
for (let i: number = circlesLifetime.length - 1; i >= 0; i--) {
  if (millis() >= circlesLifetime[i]) {
    circlesCenterX.splice(i, 1);
    circlesCenterY.splice(i, 1);
    circlesDiameter.splice(i, 1);
    circlesFill.splice(i, 1);
    circlesLifetime.splice(i, 1);
  }
}
```

Look at the loop header: it runs **backwards**, from the last index down to 0. That is not a matter of taste. When `splice` removes index 2, the old index 3 *becomes* index 2. A forward loop would step on to index 3 next and never look at the element that just slid into position 2. It skips a bubble, every time it removes one. Walking backwards, the sliding happens only in the part of the array the loop has already visited, and nothing is skipped.

! Removing while looping: always go backwards

Any loop that may `splice` elements out of the array it is walking must walk from `length - 1` down to 0. Forwards it silently skips the element after each removal; you will not get an error, just bubbles that burst one frame too late, or never. This rule applies to every remove-while-looping situation, not just bubbles.

5.7.8 Your exercise, part 2: Bubbles (Splice)

The starter code is the finished Bubbles program. You add the bursting.

1. **The fifth array.** Declare `circlesLifetime: number[]`, and push a deadline of two to four seconds into it wherever a bubble is created, next to the other four pushes.
2. **Paper first.** Take the array picture above (Section 5.7.7) and walk a forward loop over `[a, b, c, d]` that removes every element: which elements does it actually visit? Then walk it backwards. Now you know why the header looks the way it does.
3. **The removal loop.** Write the backwards loop with the five splice calls and run it; bubbles now appear, live for a few seconds, and pop. The canvas stays alive instead of filling up.
4. **Experiment.** Give bubbles ten seconds to live, then half a second. Then break it on purpose: turn the loop forwards, run it, and watch bubbles overstay. Turn it back.

💡 Exercise: Bubbles (Splice)

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0060-Arrays/bubbles-splice.yaml>

5.7.9 Check your understanding

When your bubbles come and go and you can explain the backwards loop to a classmate, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Arrays that grow and shrink

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/x16kwprm1x>

5.8 Silos: the array runs the show

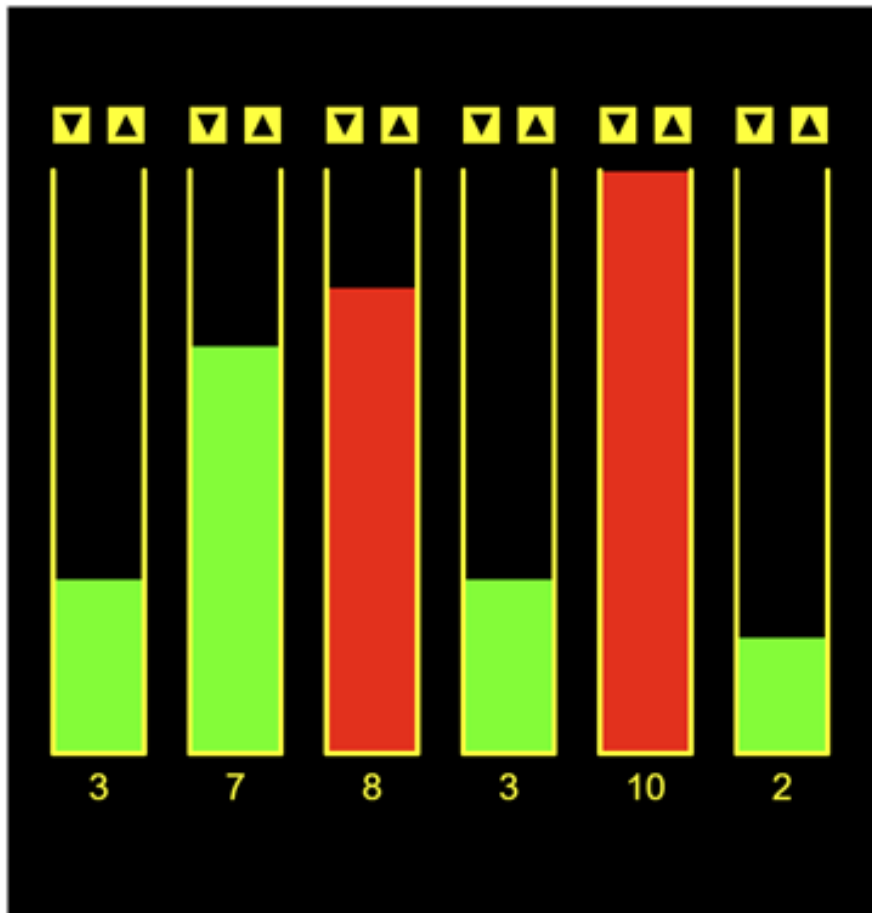


Figure 5.8.1: The silo dashboard: six silos show their fill levels as bars, green when fine, red when critical, with an add and a remove button above each. Everything on this canvas, even the number of silos, comes out of one array.

The last chapter of this part is a real little application. A factory wants a dashboard for its silos. It should show every fill level, warn in red when one runs critical, and let the operator add or remove material with buttons. Every idea of this part comes together here: the fill levels arrive as a string and are parsed into an array, the array decides how many silos exist, the mouse *changes* the array, and the canvas repaints whatever the array says, frame after frame. When this program runs, you have built your first interactive data application.

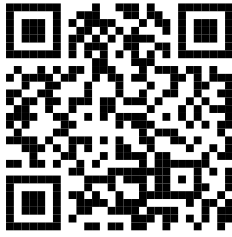
5.8.1 AI tutor

This is the largest program of the course so far, and large programs fail in small places: one wrong constant in a comparison, one forgotten flush in the parser. Build it task by task, and when something misbehaves, tell the tutor which task you are on and what the canvas shows.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/7xfdgmqh2a>

5.8.2 From string to array

The fill levels arrive as configuration text, `const INITIAL_FILL = "3,7,8,3,10,2";`. You wrote a parser like this in the parsing chapter, with one difference; back then, the program drew every parsed value immediately and threw it away. Now the values must be *kept*, and where do parsed values live? In an array:

```
const silos: number[] = [];  
  
let num: number = 0;  
for (let i: number = 0; i < INITIAL_FILL.length; i++) {  
  if (INITIAL_FILL[i] === ",") {  
    silos.push(num);  
    num = 0;  
  } else {  
    num = num * 10 + parseInt(INITIAL_FILL[i]);  
  }  
}  
silos.push(num); // the flush: the last value has no comma
```

The digits grow the number differently this time, with arithmetic instead of a buffer string. `num * 10 + digit` shifts the number one decimal place left and drops the new digit in. For "10", first $0 * 10 + 1 = 1$, then $1 * 10 + 0 = 10$. Recognize it? It is digit extraction from the number systems chapter, run backwards: `% 10` and `floor` took numbers apart, `* 10 +` puts them back together. And the flush after the loop is your old friend from the parsing chapter, in push form.

5.8.3 The array runs the program

After setup, `silos` is `[3, 7, 8, 3, 10, 2]`, and from that moment the array is in charge of everything in the program you are about to build:

- **How many silos?** `silos.length`. Every loop in the program runs to that bound, so adding one number to `INITIAL_FILL` produces a complete extra silo with buttons and label, with no other code change.
- **How full is silo *i*?** `silos[i]`. The bar height is computed from it, and its color too: `silos[i] >= CRITICAL_FILL` paints red, otherwise green.
- **The buttons change the array, not the picture.** A click on add runs `silos[i]++`: writing into a single element, the last array skill this part teaches. Square brackets work on both sides of an assignment; `silos[i]` is a variable like any other.

The drawing itself follows the bubbles rule: `draw` wipes the canvas and repaints every silo from the array, sixty times a second. Nobody redraws “the changed silo”; the click changes the *data*, and the next frame shows the new truth automatically. That split, where events write the state and `draw` paints the state, is how real user interfaces work, from this dashboard up to the apps on your phone.

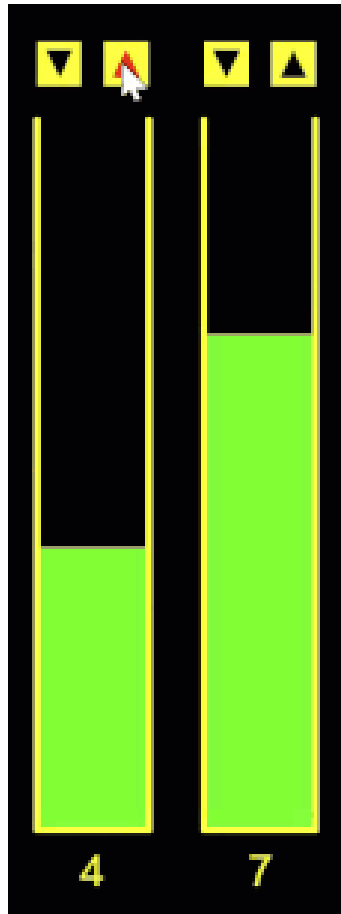


Figure 5.8.2: A close-up of two silos in action: the operator has removed material from the first silo, and the mouse hovers over one of its buttons, the triangle highlighted in red.

5.8.4 Derived arrays

Every silo needs an `x` position, and so do its two buttons. Those positions never change, so the program computes them **once** in setup and stores them in three more parallel arrays:

```
for (let i: number = 0; i < silos.length; i++) {
  const x: number = SILO_GAP + i * (SILO_WIDTH + SILO_GAP);
  silosX.push(x);
  upX.push(x);
  downX.push(x + SILO_WIDTH - BUTTON_SIZE);
}
```

The formula is the grid formula from the Loops part, index times (width plus gap), plus the leading gap. Arrays like these, computed from other data instead of written by hand, are called derived: `silos` is the real data, `silosX`, `upX`, and `downX` are its shadow. Computing them once in setup instead of every frame keeps `draw` simple; wherever a silo or button is drawn or clicked, the position is a lookup, not a calculation. The add button sits at the silo's left edge, so `upX` stores the same values as `silosX`; the remove button leans on the right edge.

5.8.5 Buttons: is the mouse inside?

A button is a rectangle, and “did the click hit it?” is a question you can ask with four comparisons:

```
if (
  mouseX >= upX[i] &&
  mouseX <= upX[i] + BUTTON_SIZE &&
  mouseY >= BUTTON_TOP &&
  mouseY <= BUTTON_TOP + BUTTON_SIZE &&
  silos[i] < SILO_MAX
) {
```

```

    silos[i]++;
    return;
}

```

The first four conditions box the mouse in: right of the left edge, left of the right edge, below the top, above the bottom. All four true means the mouse is inside the button of silo *i*. This **hit test** is the standard recipe for every rectangular button you will ever build. The fifth condition is the guard rail; a full silo ignores its add button, so the level never leaves 0 to `SILO_MAX`.

The same test runs in two places: in `mouseClicked`, inside a loop over all silos, to change the array, and in `draw`, to paint the hover highlight (the triangle turns red while the mouse sits over its button). The last new tool of the part is the bare `return`, which ends `mouseClicked` on the spot after a hit. One click means one button; once it is found and handled, there is nothing left to check, and the `return` says exactly that.

i The mouse ignores translate

If your `draw` translates to each silo before drawing it, remember that `mouseX` and `mouseY` always measure from the canvas corner, untouched by any translate. Hit tests must therefore compare against the stored canvas positions `upX[i]` and `BUTTON_TOP`, not against (0, 0). Mixing the two coordinate worlds is the classic hit-test bug: translate moves the pen, never the mouse.

5.8.6 Your exercise: Silos

The exercise page splits the work into five tasks; take them in order, and run after every one.

1. **Task 1: parse.** Turn `INITIAL_FILL` into the `silos` array in setup (Section 5.8.2), and show each value with `text` in `draw`, spaced with the grid formula. Numbers on the canvas prove the parser works before any drawing begins.
2. **Task 2: the silos.** Sketch one silo first, with the constants labeled (`SILOS_TOP`, `SILO_WIDTH`, `SILO_HEIGHT`) and the bar growing from the bottom; a labeled sketch turns the drawing code into simple reading. Then draw the outline (two sides and the floor), the fill bar with its height computed from `silos[i]`, the red-or-green color, and the centered number below.
3. **Task 3: the buttons.** Two yellow squares above each silo, with black triangles: pointing down into the silo on add, pointing up and away on remove. Push and pop around each button, as around every other drawing block.
4. **Task 4: the clicks.** The hit-test loops in `mouseClicked` (Section 5.8.5), with the guard rails for 0 and `SILO_MAX` and the `return` after a hit. Click yourself through all silos: a red bar must turn green when you empty it below the critical level.
5. **Task 5: the hover.** Reuse the hit test in `draw` to paint the triangle red while the mouse is over its button. Any other highlight you like works too; the sample turns the indicator red.

Exercise: Silos

[▶ Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0060-Arrays/silos.yaml>

5.8.7 Check your understanding

When your dashboard clicks, clamps, and highlights, you have finished the Arrays part; the quiz below closes it. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Silos

► Take the quiz on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/xxumwzr3hy>

5.9 Value or reference: what an assignment copies

This chapter paints no new picture on the canvas. Instead you play detective and look inside the boxes of your variables, with four small experiments that you type yourself. One of them ends with a result that surprises almost everybody the first time. For the detective work you pick up a new tool, `console.log`, a command that lets a program write plain lines of text into an output area of its own. And two loose ends from earlier chapters, the `const` array that happily grows and the `substring` that changes nothing, stop being magic along the way.

5.9.1 AI tutor

Predictions are the whole point of this chapter, and a prediction that misses is worth a question. When the console shows something your paper did not, describe both sides to the tutor: what you expected, what appeared, and which lines you typed. “My two variables changed together and I don’t know why” is exactly the kind of question this tutor is built for.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/7xfdgmqh2a>

5.9.2 A window into the program: the console

Detectives need a way to look at values, and drawing every suspect with `text` on the canvas would be clumsy. The playground has a better tool for this. Below the drawing area sits the **console**, an output area for plain lines of text, and the command `console.log(...)` writes one line into it. Hand it any value you have, a number, a boolean, a string, and the value appears as a line in the console. You can even hand it several values, separated by commas, and they land in one line together:

```
console.log("three times seven is", 3 * 7);
```

Two practical details. The console keeps the lines of earlier runs, and its Clear button wipes it empty again. And `console.log` has two louder siblings, `console.warn` for warnings in orange and `console.error` for errors in red; you will meet them properly in later programs, and for the detective work of this chapter, the plain log is all you need.

5.9.3 Assignment copies the value

Start with the world you have known since the Variables part. Open a fresh sketch in the playground and type this program:

```
function setup() {  
  let a: number = 10;  
  let b: number = a;  
  b = 99;  
  
  console.log(a);  
  console.log(b);  
}
```

There is no `createCanvas` here and nothing is drawn; the program talks only through the console, and the canvas staying empty is fine. Before you press Run, write two numbers on your sheet of paper, the one you expect on the first console line and the one you expect on the second. Every experiment in this chapter starts that way.

The console shows 10 and 99, and that is what everybody predicts. The interesting line is `let b: number = a;`. It reads the value in a's box, which is 10, and puts a copy of that value into b's box. From then on the two boxes have nothing to do with each other, so `b = 99` reaches b's box alone.



Figure 5.9.1: After the copy, a and b are two separate boxes with two separate values. Changing one of them cannot touch the other.

`number` and `boolean` are **value types**: the box of such a variable holds the value itself, and an assignment copies that value. Afterwards there are two independent boxes, always.

Vary it. Replace the two numbers with `let a: boolean = true;` and `let b: boolean = a;`, then set `b = false;` and log both. Booleans behave exactly like numbers here.

Hold on to that “obviously 10” for a minute. The next experiment looks almost the same and does something else.

5.9.4 The array experiment

Now the same shape with an array. Keep the `setup` frame and replace its body:

```
const first: number[] = [10, 20, 30];
const second: number[] = first;
second[0] = 99;

console.log(first[0]);
console.log(second[0]);
```

Write the prediction down before you run it, and be honest with yourself. What does `first[0]` show, and what does `second[0]` show? Most people write 10 and 99. The console shows **99 and 99**.

Here is what really happens. The box of an array variable is far too small for a whole row of values, and it never holds one. It holds a **reference**, an arrow that says where the array lives. The array itself lies somewhere else, on its own.

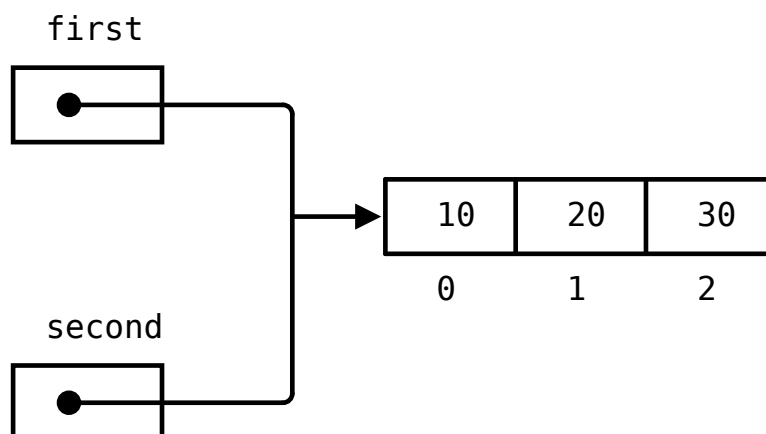


Figure 5.9.2: Two variables named `first` and `second`, and only one array. Each of the two boxes holds an arrow, and both arrows lead to the same row of numbers.

The picture shows the situation after the experiment's assignment. `second = first` copies what sits in `first`'s box, and what sits in `first`'s box is the arrow. So `second` gets an arrow to the very same row. One array, two names for it. Arrays are **reference types**.

The line `second[0] = 99` then touches neither of the two variable boxes. It follows the arrow and changes

element 0 of the one array that exists. `first[0]` reads that same element through the other arrow, so it reads 99 as well.

By the way, `console.log` also takes a whole array. `console.log(first);` prints 99,20,30, every element in one line, which makes it a handy magnifying glass for the experiments below.

Vary it, twice.

1. Add `second.push(40);` after the assignment and log `first.length`. The length is 4, because `push` follows the arrow too. Growth is shared, not just element changes.
2. Change `first[1]` instead of `second[0]`, and read the value back through `second[1]`. It changed the same way. Neither variable is the original and neither is the copy; the two arrows are worth exactly the same.

5.9.5 The const puzzle, solved

The bubbles chapter carried a callout named “A const that changes?” (Section 5.7.2). It said that `const circlesDiameter` can still be pushed into, because `const` nails the box down while the content stays free. That was a rule to accept, and now it has a reason.

`const` forbids exactly one thing, putting something else into the box. The box of an array variable holds the arrow, so `const` means “this arrow never changes again”. And `push` never changes an arrow. It follows the arrow and hangs a value on the end of the row over there, which `const` does not care about.

Add this line to the array experiment and watch the editor:

```
second = [1, 2, 3];
```

The red squiggle appears before you run anything, and the message is worth two seconds of reading. “Cannot assign to ‘second’ because it is a constant.” That is the box complaining, not the array. Delete the line again.

5.9.6 A real copy takes a loop

If an assignment only copies the arrow, how do you get a second array that is truly independent? Three tools you already own are enough, an empty array, a `for` loop, and `push`. Here they copy the array `first` from the array experiment (Section 5.9.4):

```
const copy: number[] = [];
for (let i: number = 0; i < first.length; i++) {
  copy.push(first[i]);
}
```

Read the shape of it. The `[]` creates a brand new array, so `copy` starts with a brand new arrow to a row of its own. The loop then walks the old row and pushes every element into the new one, and each of those elements is a number, a value type, so each `push` copies a value across. Two rows, two arrows, no connection.

Try it. Put the loop into the array experiment, add `copy[0] = 0;` below it, and log `first[0]` and `copy[0]`. This time the two numbers differ, and now you can say exactly why.

i Comparing arrays compares the arrows

`===` between two array variables asks whether the two boxes hold the same arrow, not whether the two rows look alike. For the two variables of the array experiment, `first === second` is `true`, because both arrows lead to one array. But `first === copy` is `false` even when every single element matches, because those arrows lead to two different rows. Comparing contents takes a loop as well.

5.9.7 Strings: a row you can never change

A string looks exactly like the array picture. It is a row of boxes, the indexes start at 0, `word.length` counts them, and `word[i]` reads one of them. So does a string variable hold an arrow too?

Replace the body of `setup` once more, and write your prediction down before you run it:

```
let word: string = "apple";
let copy: string = word;
word = word + "s";
```

```
console.log(word);  
console.log(copy);
```

The console shows “apples” and “apple”, without the quotes, because the console prints a string’s characters and nothing else. The string behaved like the two numbers at the start of this chapter (Section 5.9.3), not like the shared array (Section 5.9.4).

The reason is a rule without exceptions. No operation ever changes an existing string. `word + "s"` does not append an “s” to anything; it builds a completely new string “apples”, and the assignment puts that new string into `word`’s box. What `copy` holds is the old string, untouched, because nothing was ever able to touch it.

You have met that rule once before without a name. Word Swirrel said that **substring** never changes the string it is called on and hands you a copy of the piece instead (Section 5.1.4). Every string method works that way.

Type one more line and let the compiler show you the rule:

```
word[0] = "A";
```

Reading a character with `word[0]` is allowed, writing one is not. The squiggle says “Index signature in type ‘String’ only permits reading”. Delete the line again.

Now the two words for all of this. An array can be changed after it is created, with **push**, **splice**, or a write through an index, so an array is **mutable**. A string can never be changed after it is created, so a string is **immutable**, and every string operation that looks like a change really hands back a new string. Because nothing can ever change a string, sharing one is harmless. A string variable behaves like a value type in every assignment, and JavaScript indeed files strings with the value types, next to **number** and **boolean**.

i Strings in other languages

Other languages model strings differently. In C# and in Java, a string variable does hold an arrow, and the thing at the end of the arrow is immutable. A running program cannot tell the two models apart, and immutability is the reason: when nothing can change the thing the arrow leads to, it makes no difference whether an arrow is there.

5.9.8 Check your understanding

When your predictions and the console agree, and you can say out loud what `second = first` copies, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Value or reference

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/hm2imu0a41>

5.10 Reading MDN: a string method of your own

At the end of part 1 you learned a p5.js command that nobody taught you, straight from the official p5.js reference. That reference has a border, and in this part you crossed it without noticing. Commands like `circle` and `translate` come from p5.js, so the p5.js reference describes them. But `word.length`, `word[i]`, and `substring` are not p5.js at all; they belong to JavaScript, the language itself, and the language has its own documentation. It is called **MDN**, and reading it is the skill this chapter trains. You will get to know the site, and then repeat the research exercise you already know: pick one string method nobody taught you, learn it from its documentation page, build an example, and describe it in your own words.

5.10.1 AI tutor

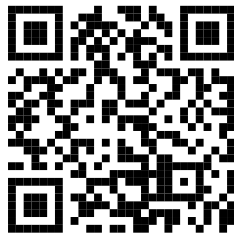
As in the research exercise of part 1, two AIs play different roles here. The AI tutor of this part answers questions about the course material, as in every chapter. The exercise itself sends you to a general AI on the internet on purpose, because handling material nobody prepared for you is what the exercise trains.



Your AI tutor

Hints and questions instead of finished programs, in English or German.

► [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/7xfdgmqh2a>

5.10.2 MDN, the documentation of the web

[MDN Web Docs](#), short for Mozilla Developer Network, is the standard reference for the languages of the web: JavaScript, HTML, and CSS. It is run by Mozilla, the makers of the Firefox browser, and practically every professional web developer has it open in a tab. When they wonder what a method does, they search for “mdn” plus its name, land on the matching page, and read.

Your starting point is the [String reference page](#). The sidebar on the left lists everything a string can do, with all the string methods collected under *Instance methods*; three of them, `length`, `substring`, and the square-bracket access, are old friends by now. And in the upper right corner of every MDN page sits a search box. Type a method name like “repeat” into it and you jump straight to its page, no matter where on the site you are.

There is a trick for getting to know an unfamiliar documentation site: first look up something you already understand, and check the page against your own knowledge. So before you research anything new, open the page for [substring](#) and walk through it with the next section as your guide.

5.10.3 The anatomy of an MDN page

Every method page on MDN follows the same pattern, so once you can read the `substring` page, you can read them all:

- **Try it** sits right at the top: a small editor with a working example and its output below. You can change the code and run it again, directly on the page, which makes it perfect for quick experiments. The examples print their results with `console.log`, the same command you used for the experiments in the value-or-reference chapter (Section 5.9.2); here its lines appear in the output area under the code.
- **Syntax** shows each allowed way to call the method on its own line. For `substring` you see both `substring(indexStart)` and `substring(indexStart, indexEnd)`, which is MDN’s way of saying that the end index is optional. Below that, **Parameters** explains every name from the syntax line (optional ones are marked *Optional*), and **Return value** says what the method hands back.

- **Description** holds the fine print, the details that the short syntax cannot express. For `substring`, this is where MDN tells you what happens when the start index is larger than the end index. Do you know the answer? If not, look it up right now; the chapter about Word Swirrel skipped that detail.
- **Examples** collects longer, copyable examples with explanations.

Further down you find sections like *Specifications* and *Browser compatibility*. They matter to professionals who must support old browsers; you can skip them.

Now do the check. The return value of `substring` is described as a new string containing the specified part of the given string. That matches what you learned: `substring` copies; it never changes the string it is called on. When a documentation page confirms what you already know, you are reading it correctly.

5.10.4 MDN examples are not playground programs

The warning from part 1 applies here too, in a new costume. MDN examples are written in JavaScript, without type annotations and without the rules this course follows. Don't paste examples; read them, understand the idea, and rebuild it in the structure you know. Store the method's result in a typed variable, then look at it with `console.log`, like in the value-or-reference experiments, or draw it with `text(...)`. And when a page mentions something you have never seen (several string methods accept so-called *regular expressions*, a pattern language for text that comes much later), don't push through; pick a different method instead.

5.10.5 Choose your method

Pick **one method** from the *Instance methods* list of the String reference. The firm rules match part 1, with one addition:

- Choose a method you haven't used in this course yet, so `substring` is not an option. So is `charAt`, because it does exactly what the square brackets already do for you.
- The page must make sense without regular expressions. If the syntax section talks about patterns or `RegExp`, choose something else.
- Leave `split` and `join` for later; the course picks them up in a future part.

That still leaves plenty of good candidates. `toUpperCase` and `toLowerCase`, `repeat`, `includes`, `startsWith` and `endsWith`, `indexOf`, `trim`, `padStart` and `padEnd` are all within your reach, and `slice` is a close relative of `substring` with one surprising extra ability. Pick whatever makes you curious.

5.10.6 Your exercise: learn it, build it, describe it

The task has the same three parts as in part 1, so this time you write a page of your class's own string reference:

1. **Learn the method from its MDN page.** Read *Try it*, *Syntax*, and *Parameters*, then experiment in the Try it box the way you learned with `arc`: change one value at a time, run, and watch the output. Keep going until you can predict the result before you press run.
2. **Build your own example.** Write a small playground program that shows your method at work on the canvas: the input string and the result, both drawn with `text`. If you want to shine, combine the method with what this part taught you, for example an array of words and a loop that applies the method to every element. TypeScript, typed variables, formatted before you hand it in.
3. **Describe the method in writing.** Half a page for a classmate who has never seen it: what it does, what each parameter means, and one thing that surprised you while experimenting.

Your teacher will tell you how to hand in your program and your text. Expect a short presentation in class again: everyone picks a different method, so after the presentations your class owns a little string reference of its own, and you wrote one page of it.

5.10.7 AI is allowed, understanding is required

You may use an AI for every part of this exercise, and the rules are the ones from the research exercise in part 1.

1. Use a general AI on the internet, not the tutor from this book, because meeting answers nobody tailored

to this course is part of the research. Paste your own working code into the prompt, so the AI answers in the language and structure your playground expects, and say that you are a beginner whose code shows everything you have learned. And the old test still decides what stays: a line of code or a sentence in your text that you cannot explain in front of the class gets replaced, or you ask until you can.

Part 6

Functions

6.1 Your own commands: functions

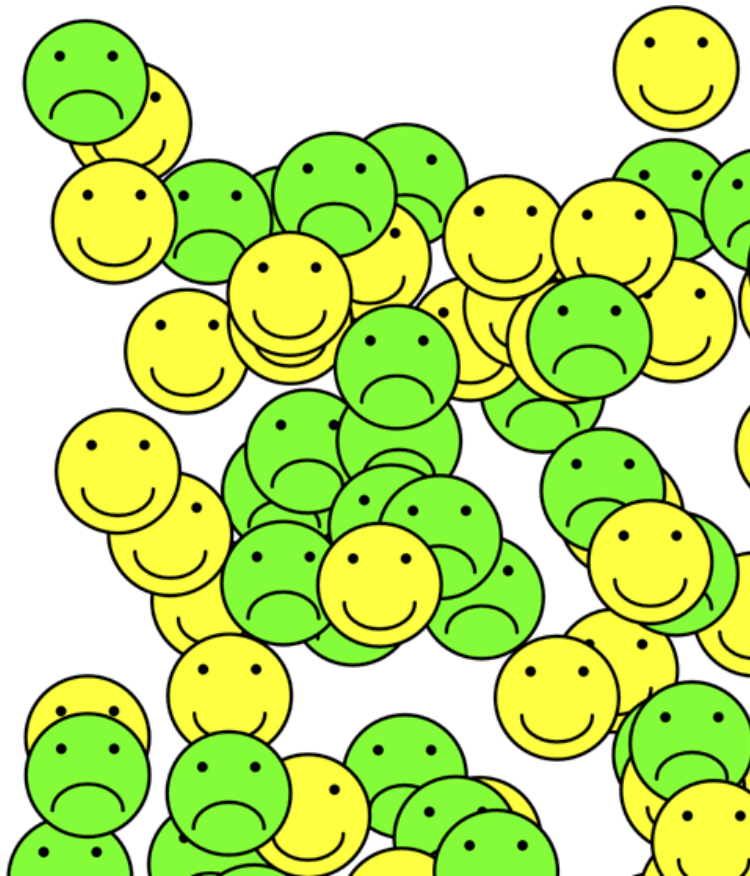


Figure 6.1.1: Face Function after a few dozen mouse clicks. Every click drops one more smiley at a random spot, happy and sad in turn, and all of them come out of a single function that you write once.

You have been writing code inside functions since your very first program. `setup` is a function, `draw` is a function, and `mouseClicked` is one as well. What you have never done is build one of your own. This part of the book changes that. You'll learn to teach the computer new commands, give them names, hand them values to work with, and grow big programs out of small named pieces instead of one long block of statements. The first command you invent draws a smiley face.

In the Arrays part, Word Swirrel handed you a finished line, `function guess(textInput: string)`, and asked you to fill in nothing but the body (Section 5.1.6). The book left that line unexplained on purpose and promised a later part for it. This is that part. Back then you got the signature; today you write your own.

6.1.1 AI tutor

Your first own function brings two new kinds of mistakes. A function that is defined but never called draws nothing at all, and a function called with the wrong value draws the wrong thing. If your canvas stays empty or every face looks the same, paste your definition and your call and tell the tutor what you expected.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgrw3hw>

6.1.2 A function is a command you teach

You already give the computer commands all day. `circle(200, 200, 50)` is one, `fill("yellow")` is another. Those commands come with p5.js, someone else wrote them, and you use them by writing their name and a pair of parentheses.

A function of your own works exactly the same way, with one difference. You write the recipe. You collect a group of statements under a name you invent, once, and from then on that name is a command like `circle` or `fill`. Writing the recipe down is called **defining** the function. Saying the name later to make it run is called **calling** it.

Defining and calling are two separate things, and the order on the page does not matter. A definition anywhere in your file can be called from anywhere else, even from a line above it.

A function also changes what you do when a program needs the same work twice. Back in the cat chapter, copying a line and changing its numbers was the fastest way to draw a second ear (Section 1.4.2), and the clipboard stays a useful typing tool for that. Copying a whole *block* of statements is a different story. Ten pasted lines are ten places to fix when the drawing changes, and the reader of your program has to compare them character by character to see what is different. From this part on, work that repeats gets a name and becomes a function, and the copies become calls.

6.1.3 The anatomy of a function

A function definition and a function call look similar, and they have different jobs. Here are both, with every part named:

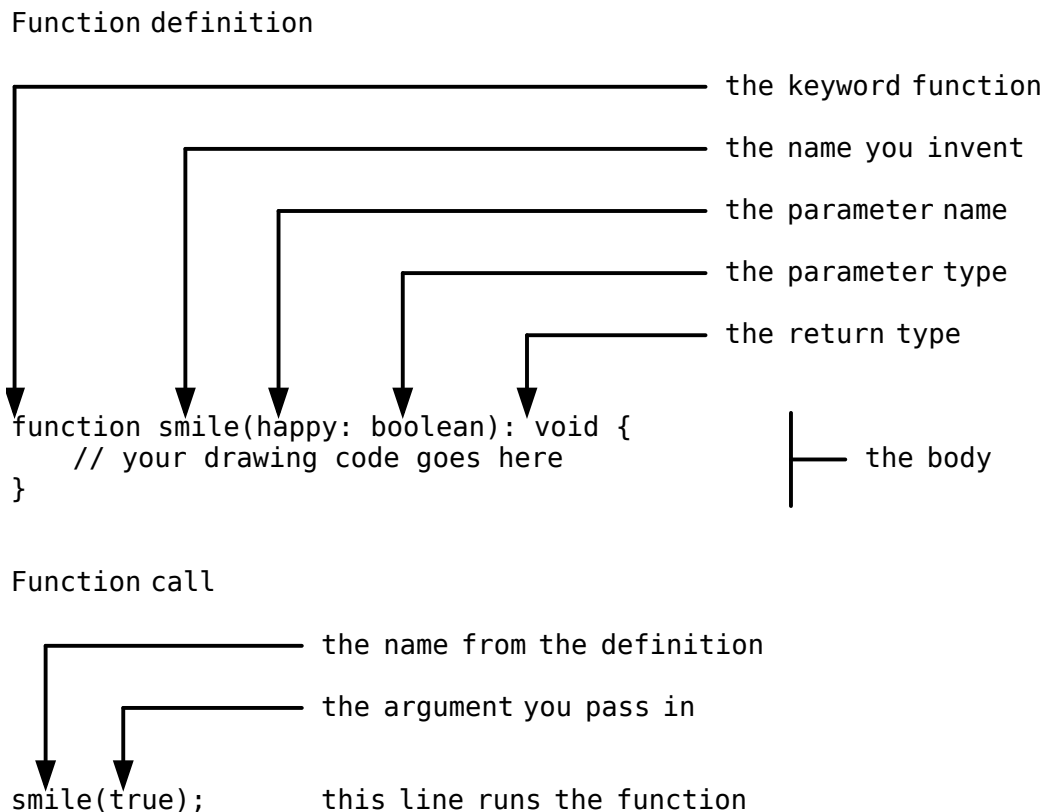


Figure 6.1.2: A function definition teaches the computer a new command, and a function call runs it. The definition names the parameter and its type; the call passes an argument.

The definition starts with the keyword **function**, then the name you invent, then a pair of parentheses, then the body between braces. The body is the recipe, the statements that run whenever somebody calls the function.

Inside the parentheses of the definition sits **happy: boolean**. That is a **parameter**, a variable that exists only inside this function and has no value of its own. Whoever calls the function decides what goes in.

Inside the parentheses of the call sits **true**. That is an **argument**, the value that fills the parameter for this one run. Parameter and argument are two names for the two ends of the same handover. The parameter is written in the definition, the argument is written in the call, and the argument's type has to match what the parameter asks for.

When the computer reaches **smile(true);** it does not simply read on. It remembers where it was, jumps into the body of **smile**, runs the statements there from top to bottom, and then comes back to the line right after the call. Your program continues as if nothing had happened, only a face has appeared on the canvas.

6.1.4 A complete signature, every time

The first line of a function definition is called its **signature**. The signature says the name, what the function needs, and what it gives back. In **function smile(happy: boolean): void**, the part **happy: boolean** is what the function needs, and **: void** is what it gives back.

void means “nothing”. The **smile** function draws on the canvas and then it is done; it hands no value back to whoever called it. Not every function is like that, and a later chapter of this part shows functions that answer with a value. Until then, every function you write ends its signature with **: void**.

! Course rule: every function gets a complete signature

Every function you write in this course names a type for each parameter and a type for the result. That is the “every variable gets a type” rule from the Variables part, applied to functions. Now you can also read the line you have typed a hundred times without understanding it, because **function setup(): void** says exactly that: no parameters, nothing given back. Loop variables carry their type in the same spirit (Section 4.1.3), so a for header reads **for (let i: number = 0; i < 10; i++)**.

6.1.5 One function, two faces

A parameter is worth the trouble because it makes one function do several jobs. The `smile` function takes a single boolean, and that boolean decides between a happy face and a sad one:

```
function smile(happy: boolean): void {
  stroke("black");
  strokeWeight(10);
  if (happy) {
    fill("yellow");
  } else {
    fill("lime");
  }

  circle(200, 200, 350);

  fill("black");
  circle(125, 125, 20);
  circle(275, 125, 20);

  noFill();
  if (happy) {
    arc(200, 250, 200, 150, 0, 180);
  } else {
    arc(200, 300, 200, 150, 180, 360);
  }
}
```

The condition is written as `if (happy)`, without a comparison. A boolean already is the answer to a yes-or-no question, so there is nothing left to compare it against. Writing `if (happy === true)` works too and says the same thing twice.

Two decisions and one function, and the caller picks a face with a single word. Without the parameter you would need two functions, `drawHappyFace` and `drawSadFace`, with almost identical bodies. Every time you changed the eyes you would have to change them twice. That is what parameters are for.

6.1.6 Drawing at a fixed spot, appearing anywhere

Look at the numbers in the face above. The eyes sit at 125 and 275, the head is a circle of diameter 350 around (200, 200). Those numbers never change, so on its own this face would always land in the same corner of the canvas at the same size. The move that frees it is one you know from the Loops part, where the origin started to wander (Section 4.7.2):

```
function smile(happy: boolean): void {
  push();
  translate(random(0, width), random(0, height));
  scale(0.2);

  // ... the drawing code from above ...

  pop();
}
```

`translate` moves the origin to a random point of the canvas, so the fixed coordinate (200, 200) now means “200 to the right and 200 down from there”. `scale(0.2)` multiplies every following coordinate and length by 0.2 (Section 5.4.2), which turns a 350 pixel head into a 70 pixel one. The `push` and `pop` around them put the coordinate system back the way it was (Section 4.7.4), so the next call starts from the same place as the first one.

Here is the habit worth taking from this chapter. A function that draws with fixed numbers can still draw anywhere, because a `translate` in front of the drawing decides where those numbers land. Every drawing function in this part is built like that.

6.1.7 Flipping a boolean back and forth

The finished program should alternate, so a happy face, then a sad one, then a happy one again. Alternating means the program has to remember what it drew last, and a variable that survives between two mouse clicks lives outside every function:

```
let nextSmiling: boolean = true;

function mouseClicked(): void {
  nextSmiling = !nextSmiling;
  smile(nextSmiling);
}
```

`nextSmiling` is declared outside the braces of any function, so `setup` and `mouseClicked` both see the same variable. The line `nextSmiling = !nextSmiling;` flips it with the `not` operator you met in the parsing chapter (Section 5.5.6). A `true` becomes `false`, a `false` becomes `true`, and the value goes straight back into the variable it came from. That single line is the standard way to toggle a boolean, and you will write it often.

6.1.8 Your exercise: Face Function

The starter code contains the signature of `smile` and one call in `setup`, both with the parts labeled in comments. Your job is the body and the clicking.

1. **Read the starter code.** Find the definition, find the call, and say out loud which is which. The comment arrows point at the same parts as the anatomy figure above (Section 6.1.3).
2. **Draw one happy face.** Fill the body of `smile` with the head, the two eyes, and the smiling arc, using the fixed coordinates from this chapter. Ignore the `happy` parameter for a moment and run the program; one face has to appear before anything else is worth trying.
3. **Wrap it in push, translate, and scale,** so the face lands at a random position and comes out small. Run it a few times and watch the face jump around.
4. **Use the parameter.** Add the two `if (happy)` decisions for the fill color and the mouth arc. Test both faces by calling `smile(true);` and then `smile(false);` in `setup`.
5. **Add the clicking.** Write `mouseClicked` and call `smile` from it. Only then add `nextSmiling` above all functions, flip it in `mouseClicked`, and hand it to both calls, the one in `setup` and the one in `mouseClicked`. One change, one run, exactness first.
6. **Experiment.** Give the sad face a different color, change `scale(0.2)` to `scale(0.5)`, or call `smile` three times in `setup` and see three faces from one function.

💡 Exercise: Face Function

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfd6fbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/face-function.yaml>

6.1.9 Check your understanding

When your canvas fills with faces and you can point at a definition and a call and name them correctly, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Your own commands

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/6zf737logv>

6.2 Refactoring: from dice to domino

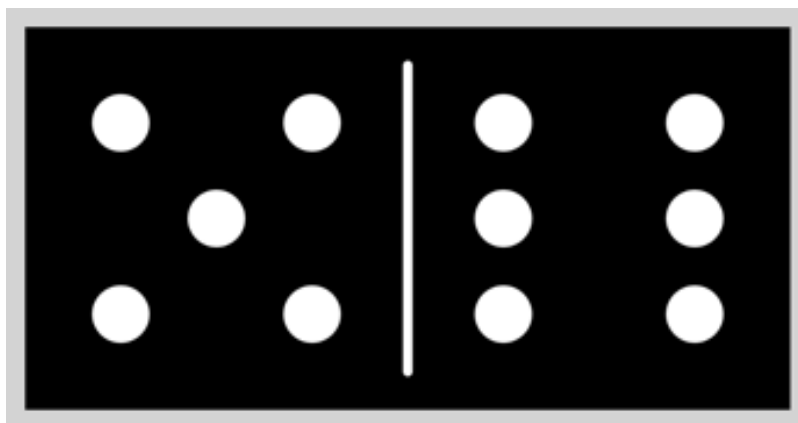


Figure 6.2.1: One domino piece drawn by the finished program. Two dice faces share a black tile, a thin white line separates them, and both numbers are rolled at random.

Not every program starts as an empty file. This one starts finished. The starter code draws a single dice face with the right number of dots, and it works. Your job is to change how the program is *built* while the picture stays the same, and only then to grow it into a domino piece with two faces. Programmers do this every day and they have a name for it, refactoring. It is the move that turns one long program into named building blocks.

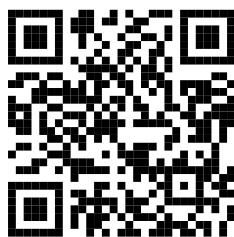
6.2.1 AI tutor

Moving code into a function is where “it worked a minute ago” happens for the first time. A dot lands in the wrong half, or the canvas comes out empty because the new function is defined but never called. Tell the tutor which step you were on and what changed in the picture.

Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.2.2 Refactoring means restructuring, not rewriting

Refactoring means changing the structure of working code while what it does stays exactly the same. Nothing new appears on the canvas, nothing disappears, nothing moves by a pixel. You are not fixing a bug and you are not adding a feature, you are making the code easier to read and easier to reuse.

That constraint sounds limiting, and it is the best part of the deal. Because the result has to look identical, you always have a test in front of you. Run the program, look, change, run again, compare. If the two pictures differ, your refactoring broke something and you know it within seconds.

The picture is your test

Refactor in small steps and run after every single one. One extracted function, one run, one look. Refactorings that are checked only at the end are how programmers lose whole afternoons.

6.2.3 The dice, all in one block

The starter code draws a black square and puts white dots on it, all of it inside `setup`. The dots are the interesting part:

```
fill("white");
if (dice === 1 || dice === 3 || dice === 5) {
  circle(diceSize / 2, diceSize / 2, dicePointDiameter);
}
if (dice !== 1) {
  circle(diceSize / 4, diceSize / 4, dicePointDiameter);
  circle(3 * diceSize / 4, 3 * diceSize / 4, dicePointDiameter);
}
// ... two more if blocks, for the 4 and up and for the 6 ...
```

You built these six faces once already, in the dice exercise of the Conditions part (Section 3.3.6). Only the arithmetic changed. Positions are fractions of `diceSize` instead of fixed pixel numbers, so `diceSize / 4` is the quarter line and `3 * diceSize / 4` the three quarter line. Each `if` adds one group of dots, and the four of them together cover all six faces.

Four `if` statements, ten `circle` calls, and a comment on top saying what the block is about. That comment is a hint. Whenever a group of statements needs a comment to explain it, it wants to be a function with that comment as its name.

6.2.4 Extracting a function

Pulling a block of statements out of its surroundings and giving it a name is called **extracting a function**. It always runs through the same five steps.

1. **Choose the statements** that do one job together. Here, the four `if` blocks with the dots.
2. **Write a new function** whose name says that job, `drawDice`.
3. **Make sure the moved code still reaches everything it needs**. This is the only step that requires thinking.
4. **Put a call** where the statements used to be.
5. **Run and compare** with the picture from before.

Step 3 splits the values the block uses into two groups. `dice` is different every time somebody wants a dice face, so it becomes a **parameter**. `diceSize` and `dicePointDiameter` are the same for the whole program, so they move to the top of the file, above all functions, where everything can read them.

```
const dicePointDiameter: number = 30;
const diceSize: number = 200;

function drawDice(dice: number): void {
  fill("white");
  // ... the four if blocks with the dots, unchanged ...
}
```

That split is a rule you can reuse forever. A value that changes from call to call is a parameter, and a value that is the same everywhere is a constant at the top of the program. Guessing wrong is harmless, because turning one into the other later is refactoring too.

6.2.5 Two dice make a domino

A domino piece is two dice faces on one tile, so with `drawDice` in place it needs one more function around it:

```
function drawDomino(dice1: number, dice2: number): void {
  push();

  fill("black");
  noStroke();
  rect(0, 0, diceSize * 2, diceSize);

  drawDice(dice1);
```

```

    translate(diceSize, 0);
    drawDice(dice2);

    stroke("white");
    strokeWeight(5);
    line(0, 20, 0, diceSize - 20);

    pop();
}

```

Read it from the top. The black tile is drawn once and is twice as wide as a single face. The first `drawDice` paints the left half, because the origin still sits in the tile's top left corner. Then `translate(diceSize, 0)` moves the origin one face width to the right, and the second `drawDice` runs with the very same fixed numbers, which now land in the right half.

You met this habit in the `face` function chapter, and here it does real work. A function that draws with fixed numbers can draw anywhere, because the `translate` in front of it decides where those numbers land. The dividing line proves it from the other side. It is drawn at `x = 0` and still lands in the middle of the tile, because after the `translate` the origin *is* the middle. The `push` and `pop` around everything put the coordinate system back, so whoever called `drawDomino` never notices that the origin moved.

6.2.6 Values flow down the call chain

Somebody has to decide the two numbers, and that somebody is `setup`:

```

function setup(): void {
    createCanvas(diceSize * 2 + 20, diceSize + 20);
    background("lightgray");
    translate(10, 10);

    const dice1: number = floor(random(1, 7));
    const dice2: number = floor(random(1, 7));

    drawDomino(dice1, dice2);
}

```

The two rolls happen in `setup`, travel into `drawDomino` as arguments, and travel one step further from `drawDomino` into `drawDice`. Values flow *down* the chain of calls, from the function that decides them to the function that finally uses them.

You could roll the dice inside `drawDice` instead and save yourself two lines. Don't. A function that receives its numbers can draw any piece you ask for, including a fixed 3:5 for testing, while a function that invents its own numbers can only ever do one thing.

One detail is easy to trip over. The constant in `setup` is called `dice1`, the parameter of `drawDomino` is also called `dice1`, and the parameter of `drawDice` is called `dice`. Those names have nothing to do with each other, because only the *value* travels.

6.2.7 Your exercise: Domino

The starter code is the working dice program, everything inside `setup`. Refactor it in two steps and keep looking at the picture.

1. **Run the starter code first** a few times and look at the dice. That picture is your test for the next step.
2. **Extract `drawDice`.** Follow the five steps from the extraction section above (Section 6.2.4), move the two constants to the top of the file, and give the function its parameter and its `: void`. Run it. The picture has to be the same dice as before.
3. **Add `drawDomino`.** Write the function with its two parameters, let it draw the wide black tile, call `drawDice` twice with a `translate` in between, and add the white dividing line. Enlarge the canvas in `setup` so the wider tile fits.
4. **Roll in `setup`.** Move the two `random` rolls into `setup` and pass them down as arguments. Test with `drawDomino(3, 5)`; first and check that the left half shows three dots and the right half five, then put the rolls back.

5. **Experiment.** Change `diceSize` at the top of the file and run again. One number, and the whole piece scales.

💡 Exercise: Domino With Functions

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfd6bjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/domino-1.yaml>

6.2.8 Five pieces from one loop

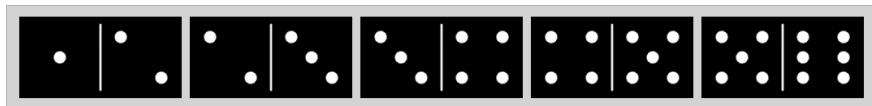


Figure 6.2.2: Five domino pieces in a row, showing 1:2, 2:3, 3:4, 4:5, and 5:6. One `drawDomino` function drew all of them, called five times by a `for` loop.

Once `drawDomino` exists, a row of pieces costs almost nothing. The five pieces 1:2, 2:3, 3:4, 4:5, and 5:6 have a pattern, since the right number is always the left number plus one. A `for` loop counting from 1 to 5 already holds both numbers you need, and a new constant `gap` at the top of the file holds the distance between two pieces:

```
for (let i: number = 1; i <= 5; i++) {  
  drawDomino(i, i + 1);  
  translate(diceSize * 2 + gap, 0);  
}
```

The loop variable becomes an argument. In the first round `i` is 1 and the piece is 1:2, in the last round `i` is 5 and the piece is 5:6. After each piece, `translate` moves the origin right by one whole tile plus a gap, so the next call starts where the previous piece ended.

Five tiles of 400 pixels are far wider than any comfortable canvas, so the row is shrunk with `scale(0.2)` before the loop starts. Everything inside the loop then draws at a fifth of its size, including the `translate` steps, so the spacing shrinks along.

6.2.9 Your exercise, part 2: Domino (Advanced)

The starter code is your finished domino program, with `drawDomino` and `drawDice` already in place. Only `setup` changes.

1. **Add a gap constant** at the top of the file, next to the other two, with a value of about 20.
2. **Make the canvas fit.** Five pieces are ten face widths plus four gaps, shrunk to 20 percent, plus a small margin. Work the number out on paper before you type it.
3. **Write the loop** with the `translate` at its end, as in the section above (Section 6.2.8). Start with two pieces if that makes the first run easier to read.
4. **Experiment.** Let the loop count down from 5 to 1 and watch the row reverse. Then try `drawDomino(i, i);` for five doubles.

💡 Exercise: Domino (Advanced)

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

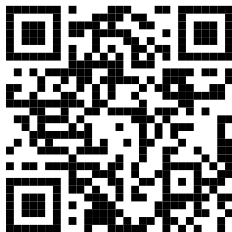
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/domino-2.yaml>

6.2.10 Check your understanding

When your row of five dominoes stands and you can explain why `drawDice` needs no idea where it is drawing, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Refactoring with functions

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/jrtrx3pzig>

6.3 Switch again: a domino of symbols

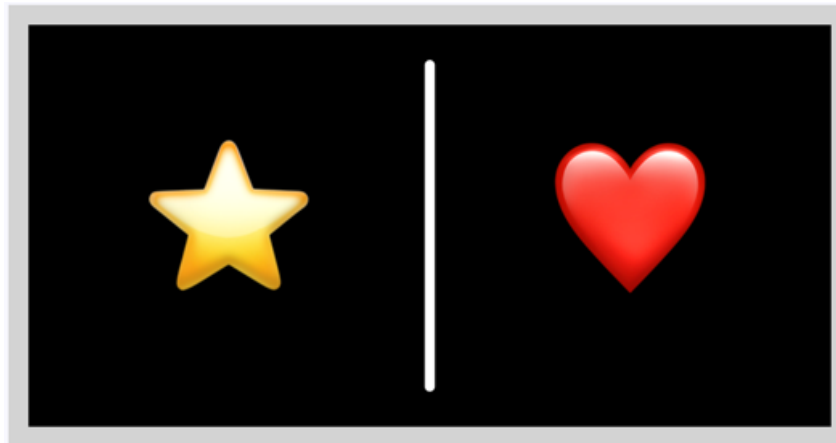


Figure 6.3.1: The finished symbols domino. The black tile, the white dividing line, and the two equal halves are the same as on a dice domino, and only the content of the halves changed. Each side shows one random symbol out of six.

Take a domino tile and replace the dots with a star, a heart, or a triangle. The tile is still a tile, the dividing line is still in the middle, and the two halves are still filled by the same code called twice. That is the quiet lesson of this chapter, and it is worth more than the drawing. The structure of a program does not depend on what its functions paint. Before you get there, you need a way to pick one of six symbols, and the tool for that job is the **switch** statement from the Conditions part.

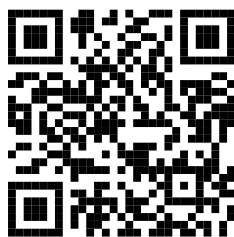
6.3.1 AI tutor

With six cases in a row, a forgotten **break** is the mistake to expect. The usual sign is a symbol drawn on top of another one, so a star with a triangle through it. The other classic here is a drawing setting that one case turns on and never turns off again. Describe what you see and paste the case you suspect.

Your AI tutor

Hints and questions instead of finished programs, in English or German.

► [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.3.2 The switch statement, in short

A **switch** statement compares one value against a list of exact values. It replaces a chain of **else if** lines that all ask about the same variable, and the Conditions part introduced it with a dice program (Section 3.8.2). The shape looks like this:

```
switch (symbol) {  
    case 1:  
        // what to do when symbol is exactly 1  
        break;  
    case 2:  
        // what to do when symbol is exactly 2  
        break;  
}
```

Keep three things in your head before you write six cases in a row.

- `switch (symbol)` names the value once, and every `case` below asks about that same value. The line `case 3:` means the same as `symbol === 3`.
- Every case ends with `break`. Without it the computer runs straight into the statements of the next case without checking whether that case matches, which is how one symbol ends up drawn on top of another.
- A `default` case runs when nothing matched, and it is optional. The switch in this chapter has none, because the value is a roll from 1 to 6 and every one of those has its own case.

6.3.3 Drawing one symbol out of six

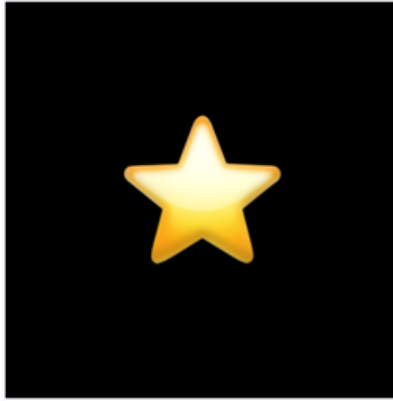


Figure 6.3.2: One symbol on one black square, drawn by a switch with six cases. This run picked the star.

The six symbols are a square, a circle, a triangle, a star, a heart, and a four leaf clover, numbered 1 to 6. Each one is drawn white and centered on the point `(centerX, centerY)`, at a width of `size` pixels. All three of those values are prepared for you in the starter code.

The first two symbols are shapes you know:

```
switch (symbol) {
  case 1:
    // square
    fill("white");
    stroke("white");
    strokeWeight(2);
    rectMode(CENTER);
    rect(centerX, centerY, size, size);
    rectMode(CORNER);
    break;
  case 2:
    // circle
    fill("white");
    noStroke();
    circle(centerX, centerY, size);
    break;
}
```

One command in there is new. `rect(x, y, w, h)` normally reads `x` and `y` as the *top left corner* of the rectangle, and `rectMode(CENTER)` switches that to the *center*, which is what a symbol sitting on `(centerX, centerY)` needs. `rectMode(CORNER)` switches it back afterwards.

The triangle is case 3, and `triangle` takes three corner points, so six numbers. Centered on the point and `size` pixels wide and high, it has its tip at `(centerX, centerY - size / 2)` and its two base corners half a `size` to the left and to the right, both at height `centerY + size / 2`.

Cases 4, 5, and 6 work differently. A star, a heart, and a four leaf clover are not shape commands; they are characters, so you print them with `text`. Set `textAlign(CENTER, CENTER)` so the character sits centered on the point instead of starting there, set `textSize(size)` so it comes out as large as the other symbols, and then call `text` with the character and the center point. The three characters are hard to type, so copy them out of the exercise description.

⚠ Put the drawing settings back

`rectMode(CENTER)` and `stroke("white")` stay in effect after the case that set them, and they follow the program into everything drawn afterwards. Switch them back in the same case, with `rectMode(CORNER)` and `noStroke()`, and every case stays independent of which one ran before it.

6.3.4 Your exercise: symbols with switch

The starter code draws a black square and rolls a number from 1 to 6 into the variable `symbol`. Your job is the switch that turns that number into a picture.

1. **Write the switch with one case.** Start with `switch (symbol) {` and the square, and replace the roll with `const symbol: number = 1;` for the moment so you can actually see it.
2. **Add the cases one at a time**, each with its `break`, forcing the matching number before every run. One case, one run.
3. **Work out the triangle on paper.** Write down the three corner points as offsets from (`centerX`, `centerY`) before you type the `triangle` call, because six numbers in one line are hard to debug by staring.
4. **Put the roll back** and run the program ten times. Every symbol has to show up, and none of them may be drawn on top of another.

💡 Exercise: Drawing Symbols with Switch

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/domino-symbols-0.yaml>

6.3.5 The same skeleton, a different body

With a working symbol switch, the domino is the refactoring you have already done once. First extract the drawing into a function of its own, `drawSymbol`, then wrap a `drawDomino` around it that calls it twice.

```
function drawSymbol(symbol: number): void {
  const centerX: number = diceSize / 2;
  const centerY: number = diceSize / 2;
  const size: number = 80;

  switch (symbol) {
    // ... the six cases, unchanged ...
  }
}
```

The three constants move *into* `drawSymbol` instead of to the top of the file, because no other function needs them. A value that only one function uses belongs inside that function, and only values that several functions share, like `diceSize`, live at the top.

Now the interesting part:

```
function drawDomino(symbol1: number, symbol2: number): void {
  push();

  fill("black");
  noStroke();
  rect(0, 0, diceSize * 2, diceSize);

  drawSymbol(symbol1);

  translate(diceSize, 0);
  drawSymbol(symbol2);
}
```

```

stroke("white");
strokeWeight(5);
line(0, 20, 0, diceSize - 20);

pop();
}

```

Hold that function next to the `drawDomino` of the dice domino (Section 6.2.5). The tile, the `translate` between the halves, the dividing line at `x = 0`, the `push` and `pop` around everything, all identical. Two lines in the body differ, and in both of them only a name changed from `drawDice` to `drawSymbol`.

That is the point of this chapter. `drawDomino` never knew what a dice face is, and it does not know what a star is either. All it knows is that a tile has two halves and that something gets drawn in each of them. You build programs out of such skeletons, and once you have one, you can hang a new body on it in minutes. Look for that shape whenever two of your programs feel similar.

6.3.6 Your exercise, part 2: Domino With Symbols

The starter code is the finished symbol switch, all of it inside `setup`. Refactor it into the domino, in the same two moves as the dice chapter.

1. **Extract `drawSymbol`.** Move the switch and its three constants into the new function, move `diceSize` to the top of the file, give the function its parameter and its `: void`, and call it from `setup`. Run it. The picture has to be the same single symbol as before.
2. **Add `drawDomino`** with its two parameters, the wide black tile, two `drawSymbol` calls with a `translate` in between, and the white dividing line. Make the canvas wide enough for the whole tile.
3. **Roll in `setup`.** Pick the two random numbers in `setup` and pass them down as arguments. Test with `drawDomino(1, 4)`; first, so you know which half is which, and then put the rolls back.
4. **Check the cleanup.** Run the program twenty times. If a square ever comes out with a stroke it should not have, or a symbol comes out in the wrong size, a case forgot to put a drawing setting back.

💡 Exercise: Domino With Symbols

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/domino-symbols-1.yaml>

6.3.7 Practice on your own: a row of symbol pieces



Figure 6.3.3: Five symbol dominoes in a row, from square and circle up to heart and four leaf clover. One `drawDomino` function, called five times.

There is one more exercise for this chapter, and this time there is no walkthrough. Domino With Symbols (Advanced) asks for the five pieces 1:2, 2:3, 3:4, 4:5, and 5:6 in a row, drawn with a `for` loop that calls your `drawDomino`. You have done exactly this with the dice domino (Section 6.2.8), and everything you need is either in your own code or in that section.

Work it out yourself, and use the tutor when you get stuck rather than looking for a finished answer. Independent practice is what turns a chapter you understood into something you can do.

💡 Exercise: Domino With Symbols (Advanced)

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

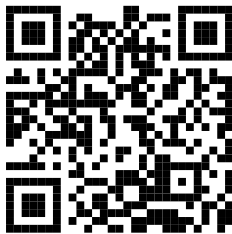
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/domino-symbols-2.yaml>

6.3.8 Check your understanding

When your symbol domino runs and you can name the two lines that differ between the two `drawDomino` functions, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Switch and the symbols domino

▶ [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/2sv5ps1a3g>

6.4 Decomposition: four functions, one chart

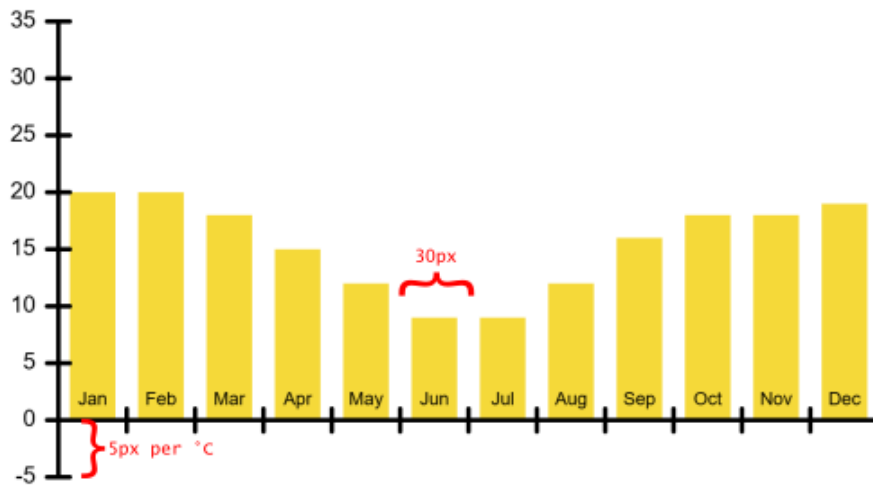


Figure 6.4.1: The finished temperature chart: one yellow bar per month, a temperature scale on the left, month names under the bars. The red notes are measurements of the finished picture, not part of your program. Every degree is five pixels tall, and every month owns a column 30 pixels wide.

Twelve numbers, one for each month, hold the average temperature of a city. The starter code brings two such rows of real climate data, one for Linz and one for Johannesburg, and your job is to turn them into a chart that anybody can read at a glance. That is more work than a domino, and it is the point where a new habit starts to pay. Before you write a single line, you split the picture into jobs, give every job a name, and only then start coding.

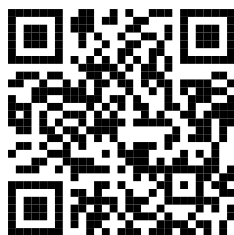
6.4.1 AI tutor

Charts fail quietly. The bars grow downward, the labels are shifted by one month, or everything is squeezed into the top left corner, and the program never complains. When your picture disagrees with the one above, tell the tutor which formula you used for the position and what you see on the canvas.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.4.2 Two roads to the same functions

There are two ways to end up with a program made of functions. Refactoring starts from code that already runs, and you pull the repeated pieces out into functions afterwards, like the dice that became `drawDice` (Section 6.2.7). **Decomposition** works the other way around. You start from the problem, split it into jobs, name the jobs, and write the code job by job. The split happens on paper, before the first line of code.

Splitting a picture into simpler parts is not new to you. Your first own program did it with a house made of rectangles and a triangle (Section 1.3.2). What is new is that each part now gets a name and a body of its own, so the split survives in the code instead of only in your head.

The temperature chart splits into four jobs, and the starter code already carries their names:

```

function drawAxes(): void {
  // <<< Add code to draw X and Y axes here
}

function drawYLabels(): void {
  // <<< Add code to draw labels for the Y axis here
}

function drawXLabels(): void {
  // <<< Add code to draw labels for the X axis here
}

function drawTemperatures(temperatures: number[]): void {
  // <<< Add code to draw the temperatures here
}

```

Three of them need nothing from the outside and always paint the same thing, so they have no parameters. The fourth one is different. `drawTemperatures` takes an array of numbers, which means the same function draws the bars for any city you hand it.

6.4.3 The array travels as a reference

A parameter is a variable, and handing an argument over to a function fills that variable exactly the way an assignment fills one. So everything the arrays part said about assignments applies to function calls too (Section 5.9.4).

For numbers and booleans, the parameter gets a copy of the value. The function may do whatever it likes with that copy, and the caller never notices. The `happy` parameter of the `smile` function (Section 6.1.5) was such a copy. This kind of handover is called **call by value**.

`drawTemperatures(AVG_TEMP_JOHANNESBURG)` behaves differently. It hands over the arrow, not twelve copied numbers, so the parameter `temperatures` points at the very same array as the constant. That is efficient, and it has a consequence. If the function wrote `temperatures[0] = 0;`, Johannesburg’s real data would be changed for the rest of the program. This kind of handover is called **call by reference**.

`drawTemperatures` only reads its array, so nothing goes wrong here. From now on, though, remember that a function which receives an array holds the caller’s data and not a private copy. A function that really has to change things builds a real copy first, with the empty array and the loop from the arrays part (Section 5.9.6), and works on that one.

6.4.4 Plan it on paper

Take your sheet of paper and write the plan down before you open the playground.

1. Write the four function names below each other.
2. Next to each name, write one sentence about what it paints. Be concrete, “the two black lines the chart hangs on” is a plan, “the axes” is just the name again.
3. Sketch the chart roughly and mark two things in it: where the line for 0 degrees runs, and how wide one month is.
4. Under the sketch, write the four names again, this time in the order in which they must be called.

Step 4 is the easiest one to skip, and it is the one that decides whether your chart looks right.

6.4.5 Draw order decides what you see

Look at the finished chart at the top of this chapter. The month names sit *on* the yellow bars, and the zero line runs across their feet. Every shape is painted on top of what is already there (Section 1.3.4), so this picture only comes out if the bars go down first and the axes and labels follow:

```

drawTemperatures(AVG_TEMP_JOHANNESBURG);
drawAxes();
drawYLabels();
drawXLabels();

```

Turn those four calls around and the chart is still there, but the bars cover the zero line and swallow the month names. The picture is the test, and the order of the calls is part of your program's design.

6.4.6 From degrees to pixels

A chart is a translation. The data is in degrees Celsius and in month numbers, the canvas wants pixels, and two formulas do the translating:

```
let x: number = 50 + i * 30;
let y: number = 225 - temperatures[i] * 5;
```

The x formula starts at 50, which is where the vertical axis stands, and moves 30 pixels to the right for each month. Month 0 starts at 50, month 1 at 80, and so on until month 11 at 380. Each bar is 20 pixels wide and gets five pixels of air on both sides, so it is drawn five pixels right of the column start.

The y formula is the more interesting one. 225 is the height of the zero line on the canvas, and every degree is worth five pixels. The minus sign is there because y grows downward, so warm months must land *above* the zero line. 20 degrees becomes $225 - 100$, which is 125, and the bar is 100 pixels tall.

i What happens below zero

January in Linz is -0.7 degrees. Put that into the formula on paper. The top edge lands at $225 - (-0.7 * 5)$, which is 228.5, below the zero line, and the height $-0.7 * 5$ comes out negative. A rectangle with a negative height is drawn upward, so the bar hangs below the line, exactly where a reader expects a freezing month. One formula, both directions, no if needed.

6.4.7 Ticks and labels come from loops

The exercise has one hard rule. Every tick mark and every label is placed by a loop, never by a hand-written position. Twelve hand-typed x values would work today and break the moment you change the width of a column.

For the temperature scale, the loop counts in degrees, not in pixels:

```
for (let t: number = -5; t <= 35; t += 5) {
  let y: number = 225 - t * 5;
  // a short line across the axis, and the number t next to it
}
```

The loop variable `t` is a temperature, and the same formula as for the bars turns it into a y position. Check the two ends: -5 becomes 250 and 35 becomes 50, and those are exactly the two ends of the vertical axis line. The scale and the axis fit because they are computed from the same numbers.

The month loop works the same way, with `x` computed from the loop variable. Its labels use the text box form of `text` with five parameters (Section 2.7.4), one box per column, so each month name is centered under its own bar.

6.4.8 The month names come from a switch

The loop counts 1, 2, 3, and the chart needs "Jan", "Feb", "Mar". Mapping a handful of fixed values to fixed results is what `switch` is for, the same tool you used for the domino symbols (Section 6.3.4):

```
let month: string = "";
switch (i) {
  case 1:
    month = "Jan";
    break;
  case 2:
    month = "Feb";
    break;
  // ... ten more cases ...
}
```

The variable is declared before the `switch` and filled inside it, so it still exists when the `text` call below needs it. Do not forget a single `break`, or your December will be called something else entirely.

6.4.9 Implement one, test one

Four empty functions are four chances to make a mistake, and if you write all four before you press Run, you will not know which one is wrong. The exercise asks for a different rhythm.

Write **one** function. Call it from **setup**. Run the program and look at the canvas. Only when that part is right do you start the next function. Your first run shows two bare lines, your second one adds the temperature scale, and the picture grows toward the goal with every step.

One detail catches everyone. The order in which you *write* the functions is not the order in which you *call* them. You implement **drawTemperatures** last, and its call still has to move to the first line of **setup**, because the bars belong under everything else.

6.4.10 Your exercise: Temperature Chart

1. **Paper first.** Do the four planning steps from the section above. Keep the sheet next to the keyboard while you code.
2. **The axes.** Write **drawAxes**, call it from **setup**, run it. The vertical line carries the temperatures, the horizontal one is the zero line, not the bottom edge of the canvas.
3. **The temperature scale.** Write **drawYLabels** with the degree loop from -5 to 35. Tick marks and numbers come from the same loop.
4. **The months.** Write **drawXLabels** with a loop over the twelve months and the **switch** for the names.
5. **The bars.** Write **drawTemperatures** with the two formulas, then move its call to the top of **setup** so the bars sit under the axes and labels.
6. **Experiment.** Swap the argument to **AVG_TEMP_LINZ**. The data is real, and the chart tells you something true. Linz is warm in July and freezing in January, while Johannesburg does the opposite, because it lies south of the equator where the seasons are turned around. Then find the one Linz month that dips below the zero line and compare it with the calculation you did on paper.

Exercise: Temperature Chart

► [Open in the playground](#)



Scan the code, or type this address into your browser:

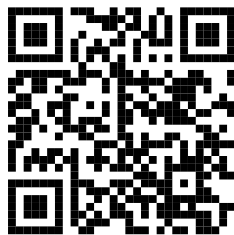
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/temperature-chart.yaml>

6.4.11 Check your understanding

When both cities draw correctly and you can say out loud why the bars are painted before the axes, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

Quiz: Decomposition

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/i6dy55ik07>

6.5 Functions that answer: return values

```
function setup(): void {
  createCanvas(300, 300);
  background("black");
  strokeWeight(2);
  stroke("lime");
  noFill();
}

function mouseClicked(): void {
  const centerX: number = getRandomInt(0, width);
  const centerY: number = getRandomInt(0, height);
  const diameter: number = getRandomInt(10, 50);
  circle(centerX, centerY, diameter);
}

/**
 * Helper function to get an integer random number between min and max
 * @param min Minimum value (inclusive)
 * @param max Maximum value (exclusive)
 * @returns Random integer between min and max
 */
function getRandomInt(min: number, max: number): number {
  return floor(random(min, max));
}
```

Figure 6.5.1: The Return Values program as the exercise shows it, as a picture, so that your fingers type every line themselves. Data travels through `getRandomInt` in two directions: the parameters carry values in, and the return value carries one value back out.

Every function you have written so far was a painter. You handed it a few values, it drew something, and that was the end of the story. This chapter adds the missing direction. A function can also work out an answer and hand it back to the code that called it, and once it can do that, you can wrap any calculation you use often into a tool with a name.

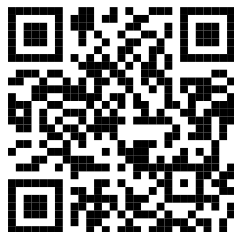
6.5.1 AI tutor

The word `return` looks harmless and hides two effects at once, so if a line after your `return` never seems to run, that is worth asking about. Describe to the tutor what your function should hand back, what type you wrote after the parentheses, and what the editor underlines.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.5.2 A function that hands a value back

Up to now, every function you wrote ended its signature with `: void`, which is the promise “I give nothing back”. A function that answers replaces `void` with the type of its answer:

```
function getRandomInt(min: number, max: number): number {
  return floor(random(min, max));
}
```

Read the head from left to right. The function is called `getRandomInt`, it wants two numbers, and after the closing parenthesis stands `number`, so it hands a number back. The course rule for signatures has not changed (Section 6.1.4), every parameter and the result still get a type. Only the result type is no longer `void`.

The word `return` does two things in one move. It works out the value on its right and hands that value to whoever called the function, and it ends the function immediately. You have seen the second half of that before, as a bare `return`; that left `mouseClicked` early (Section 5.8.5). The version with a value behind it is the same jump, only with luggage.

⚠ Nothing after the return runs

Statements below a `return` in the same block are never executed. If you write a `return` in the middle of a function and wonder why the drawing below it stays invisible, that is why. Compute first, return last.

6.5.3 The call jumps away and comes back

You know the jump from the definition chapter (Section 6.1.3). The computer remembers where it was, runs the body of the function, and comes back to the line after the call. A function with a return value makes that trip in both directions. The arguments travel over, the function works, and the answer travels back to exactly the spot where the call stood:

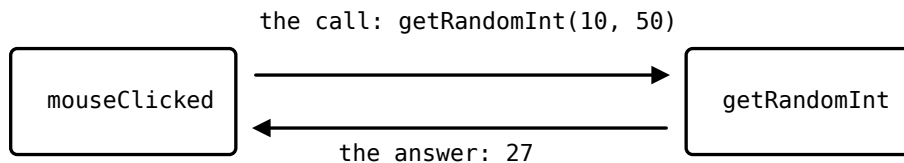


Figure 6.5.2: A call with a return value travels in both directions. `mouseClicked` hands 10 and 50 over to `getRandomInt`, and `getRandomInt` hands one number back to the place where the call stands.

That is why a call to such a function can stand wherever a value is allowed. It behaves like a number:

```
const diameter: number = getRandomInt(10, 50);
```

The right side of the assignment is not “a function”; it is whatever `getRandomInt` handed back, so `diameter` ends up holding a plain number like 27. The same call could sit inside a `circle` call or inside a calculation, in every place where you could have typed 27 yourself.

6.5.4 One recipe, one name

Look at what the body of `getRandomInt` actually does. Nothing in it is new, because `floor(random(min, max))` is the dice recipe from the Conditions part (Section 3.3.3), which turns a random decimal into a whole number. What is new is that the recipe now has a name.

Naming an idea is what makes it reusable. The program of this chapter needs random whole numbers three times, and thanks to the name, all three read like plain sentences:

```
function mouseClicked(): void {
  const centerX: number = getRandomInt(0, width);
  const centerY: number = getRandomInt(0, height);
  const diameter: number = getRandomInt(10, 50);
  circle(centerX, centerY, diameter);
}
```

Three calls, one shared body. If you ever decide that the numbers should be rounded differently, you change one line in one place and all three calls follow. And a reader who has never seen your program understands `getRandomInt(10, 50)` at a glance, while `floor(random(10, 50))` always needs a moment of thought.

The bounds are not symmetric, and that is worth knowing before you use the tool. `random(10, 50)` produces a decimal that is at least 10 and always stays below 50, and `floor` then cuts the decimals away. So `getRandomInt(10, 50)` gives you 10 at the lowest and 49 at the highest. The lower bound is included, the upper bound is not, exactly like the last index of an array is `length - 1`.

6.5.5 Doc comments explain a function to its user

A function you want to reuse deserves one more thing, a comment that describes it from the outside. TypeScript has a special comment form for that, written above the function and starting with `/**`:

```
/**
 * Helper function to get an integer random number between min and max
 * @param min Minimum value (inclusive)
 * @param max Maximum value (exclusive)
 * @returns Random integer between min and max
 */
function getRandomInt(min: number, max: number): number {
    return floor(random(min, max));
}
```

The first line says what the function is for. Each `@param` line explains one parameter by name, and `@returns` describes the answer. Notice that the comment is the place where “inclusive” and “exclusive” get written down, because the signature alone cannot say it.

The payoff is immediate. Type `getRandomInt` somewhere in the playground editor and stop, or point the mouse at a call you already wrote. The editor shows your own text in a small box, right where you need it. This is where all those helpful tooltips for `random` and `circle` come from, and from this chapter on your own functions get them too.

6.5.6 Your exercise: Return Values

This exercise is a type-in. The complete program is in the exercise description as a picture, and the pieces are all in the sections above.

1. **Type in the program.** The starter code already contains `setup` with the black background and the line outline. Add `getRandomInt` with its doc comment, then `mouseClicked` with the three calls and the `circle`. Type it, don't paste it.
2. **Hover over your own call.** Point the mouse at `getRandomInt` inside `mouseClicked` and read the box that pops up. Then change one word in the doc comment and hover again.
3. **Play computer.** On paper, answer these three questions for `getRandomInt(10, 50)`. What is the smallest number it can hand back, what is the largest, and what does `random` produce if the answer turns out to be 27?
4. **Watch the circles pile up.** Click ten times. Nothing repaints the background after `setup`, and the program has no `draw`, so every circle stays where it was painted.
5. **Experiment.** Give the circles a random line color by writing a second function that returns a color name, or make the diameter depend on the position by passing `centerY` into a function of your own. Whatever you build, keep the rule: one function, one job, one clear answer.

Exercise: Return Values

 [Open in the playground](#)



Scan the code, or type this address into your browser:

https://ts-web-playground-server-bfd6fbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/return_values.yaml

6.5.7 Check your understanding

When your circles appear on every click and you can explain what `return` hands back and where it hands it to, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Return values

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/yfyga16359>

6.6 Callbacks: a function you hand over

```

let interval: number;

function setup(): void {
  createCanvas(300, 300);
  background("black");
  strokeWeight(2);
  stroke("lime");
  noFill();

  // setInterval calls a given function at a specified interval.
  //
  // +----- ID of the created interval (required to stop it)
  // |         +----- Function name
  // |         |         +----- Function to call regularly
  // |         |         |         +----- Interval in milliseconds
  // v         v         v         v
  interval = setInterval(drawCircle, 1000);
}

function mouseClicked(): void {
  // _clearInterval_ stops a previously created interval.
  // Note that the interval ID is required to stop it. We got
  // this ID when we created the interval with setInterval.
  clearInterval(interval);
  console.log("Interval cleared");
}

/**
 * Helper function drawing a circle at a random position with a random diameter.
 */
function drawCircle(): void {
  const centerX: number = random(width);
  const centerY: number = random(height);
  const diameter: number = random(20, 80);
  circle(centerX, centerY, diameter);
}

```

Figure 6.6.1: The Intervals program as the exercise shows it, as a picture, so that you type every line yourself. The comment with the arrows explains the two arguments of `setInterval` and the value it hands back.

A circle appears every second, without a draw loop and without anybody clicking. The program behind it fits on one screen, and it rests on an idea that changes how you see functions. A function is not only something you call. You can also hand it to somebody else as a value, so that they call it for you, at a moment you never have to think about again.

6.6.1 AI tutor

The classic mistake in this chapter is a pair of parentheses too many, and it produces no error message, just a canvas that stays empty or fills up once and never again. If your circles do not appear, or all of them appear at the same moment, show the tutor the line where you set the interval up.



Hints and questions instead of finished programs, in English or German.

► Ask your AI tutor on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.6.2 You have used callbacks since your first program

Look at `setup` and `draw` with fresh eyes. You write both of them, and you have never called either of them. `mouseClicked` is the same story, and so is `guess` from Word Swirrel (Section 5.1.6). Someone else calls those functions. The p5.js library knows the names, calls `setup` once when the program starts, calls `draw` about 60 times a second, and calls `mouseClicked` when the mouse is clicked (Section 3.4.2).

A function that you write and that somebody else calls for you is called a **callback function**. Every event function you have written so far is a callback where the name did the work. p5.js looks for a function named `draw`, so a function with that exact name gets called.

This chapter turns that around. Instead of using a name that somebody already listens for, you hand your function over yourself, and you can name it whatever you like.

6.6.3 Handing a function over: `setInterval`

`setInterval` is a command of the browser, not of p5.js, and it means “call this function again and again, with this much time in between”:

```
interval = setInterval(drawCircle, 1000);
```

The second argument is the pause in milliseconds, so 1000 stands for one second. The first argument is the interesting one. It is the function `drawCircle`, written by its bare name, with no parentheses behind it. That name is a value now, like a number or a string, and it travels into `setInterval` as an argument. `setInterval` stores it and calls it every second from then on.

Your job stops after that single line. You never call `drawCircle` anywhere in the program, and it still runs once per second, forever, until somebody stops it.

⚠ No parentheses behind the function name

`setInterval(drawCircle, 1000)` hands the function over. `setInterval(drawCircle(), 1000)` does something completely different. The parentheses mean “call it now”, so your program calls `drawCircle` itself, once, right there, and then hands the result of that call over to `setInterval`. The result is nothing, because `drawCircle` returns nothing, and `setInterval` sits there with nothing to call. You get one circle and never another one, and the editor may underline the line in red, because the result of a call is not something `setInterval` can call later. Parentheses call a function. The bare name is the function itself.

6.6.4 Stopping it again: `clearInterval`

`setInterval` answers too, so it is a function with a return value like the ones from the previous chapter (Section 6.5.2). Its answer is a number, an identity card for the interval it has just started. You need that number later, so it goes into a global variable:

```
let interval: number;
```

The variable is declared outside every function, because `setup` stores the number and `mouseClicked` reads it, and a variable declared inside `setup` would be gone the moment `setup` ends. Stopping the interval then takes one line:

```
function mouseClicked(): void {
  clearInterval(interval);
  console.log("Interval cleared");
}
```

`clearInterval` needs the identity card, because a program can have several intervals running at the same time and the browser has to know which one you mean. After that call, the circles stop appearing.

The second line is an old friend with a new job. `console.log` writes its message into the playground’s console, as in the value-or-reference experiments (Section 5.9.2), and nothing of it appears on the canvas. Here it works as a debugging tool: the message proves that this piece of code really ran, and you will reach for that trick whenever a program does nothing and you want to know how far it got.

6.6.5 The function that gets called

The callback itself holds no surprises. It is a normal function with a normal signature, and it draws one circle at a random place:

```
/**
 * Helper function drawing a circle at a random position with a random diameter.
 */
function drawCircle(): void {
  const centerX: number = random(width);
  const centerY: number = random(height);
  const diameter: number = random(20, 80);
  circle(centerX, centerY, diameter);
}
```

Nothing in this function knows that it is a callback. It has no parameters, returns nothing, and would work exactly the same if you called it yourself from `mouseClicked`. Being a callback is not a property of the function; it is a property of the way the function is used.

i Whole pixels are not needed here

`random(width)` hands back a decimal, and this program keeps it that way, because a circle at $x = 143.7$ looks no different from one at $x = 144$. Rounding with `floor` matters when a number is used as an index or shown as text, not when it is a position on the canvas.

6.6.6 Two ways to say “every so often”

You now know two ways to make something happen at regular times, and they are worth comparing, because you will choose between them in every animated program from here on.

The **deadline pattern** stores the next due time and checks it in `draw` (Section 5.7.4). The check happens 60 times a second, and the work runs when the clock has passed the deadline. It fits when each round should have a different length, like bubbles that appear after half a second to two seconds, and when the work belongs to the drawing anyway.

`setInterval` hands the job to the browser and keeps `draw` free. It fits a fixed rhythm, and it comes with an off switch, because `clearInterval` stops the whole thing without a flag variable and without an `if`. The program of this chapter has no `draw` function at all. Between two calls of `drawCircle` nothing in your code runs.

6.6.7 The canvas keeps everything

`background("black")` runs once, in `setup`, and nothing repaints it afterwards. So every circle that `drawCircle` paints stays on the canvas, and after a minute the canvas is full of overlapping lime rings.

That is the painting style of the early parts of this book, where the canvas itself was the memory. The bubbles program showed the other one, with the arrays as the truth and the whole picture repainted from them every frame (Section 5.7.5). The intervals program has no memory of its circles at all, because the paint on the canvas is all there is. Both styles are correct, and the choice follows from the question “can this picture change?”. A circle that will never move again is fine as paint. Anything that moves, grows, or disappears needs to live in variables.

6.6.8 Your exercise: Intervals

This exercise is a type-in. The whole program is short, and every line of it appeared in the sections above.

1. **Type in the program.** The starter code gives you `setup` with the black background and the lime stroke. Add the global `interval` variable, the `setInterval` line at the end of `setup`, the `drawCircle` function with its doc comment, and `mouseClicked` with `clearInterval`. Type it, don't paste it.
2. **Watch and stop.** Run it and count the circles for ten seconds. Then click once. The canvas keeps what is there, and nothing new arrives.
3. **Break it on purpose.** Change the line to `setInterval(drawCircle(), 1000)` and look at what the editor says about it. Run it anyway. One circle appears right away and no other one follows. Read the warning about parentheses again, then undo the change.

4. **Experiment.** Try 100 milliseconds instead of 1000. Then write a second function `drawBigCircle` and start a second, slower interval for it, so two intervals run side by side. Each one hands back its own identity card, so you need a second global variable to keep it.

 Exercise: Intervals

► [Open in the playground](#)



Scan the code, or type this address into your browser:

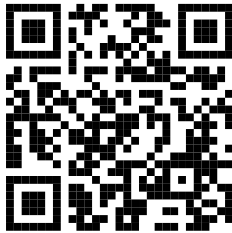
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/intervals.yaml>

6.6.9 Check your understanding

When your circles arrive once a second and stop on a click, and you can say what the parentheses would change, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

 Quiz: Callbacks and intervals

► [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/fhgc5lht0q>

6.7 A game from functions: Bubble Buster

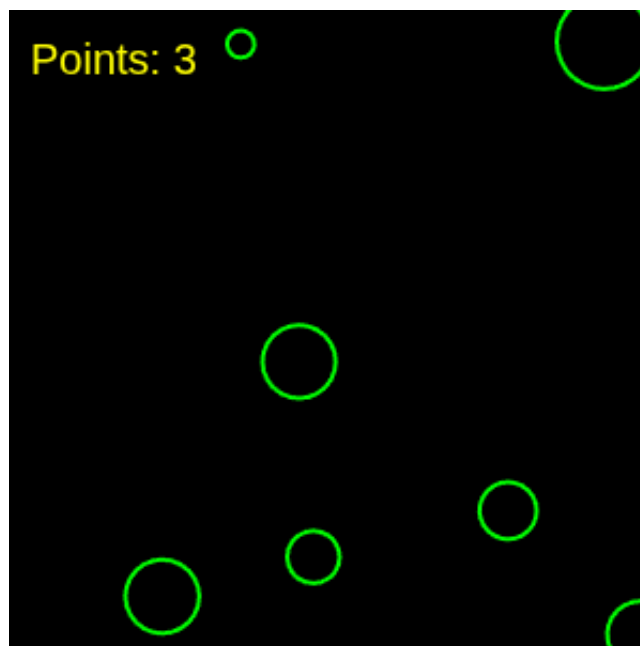


Figure 6.7.1: Bubble Buster in play: lime outlined bubbles float on a black canvas, and a yellow points counter in the top left corner counts every bubble the player has popped.

Bubbles are back. In the Arrays part you built a program where bubbles appeared, waited, and burst on their own (Section 5.7.6); this chapter turns the same bubbles into a game you play with the mouse. The bubbles have not changed; the engine underneath them has. Everything the program does now lives in a function with a name, and each of those functions has exactly one job.

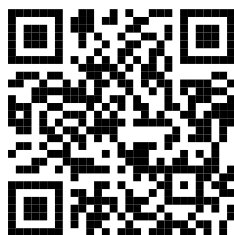
6.7.1 AI tutor

Bubble Buster is the first program in this course where five or six of your own functions work together, so a bug can hide in the cooperation instead of in a single line. If a click pops the wrong bubble or nothing happens at all, tell the tutor which function you think is responsible and paste it.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.7.2 The same bubbles, a new engine

Bubble Buster keeps its bubbles in three parallel arrays, one for each thing a bubble needs:

```
const circlesX: number[] = [];  
const circlesY: number[] = [];  
const circlesDiameter: number[] = [];
```

Three arrays instead of the five from the Arrays part, because a bubble in this game has no color of its own and no lifetime. Every bubble is drawn with the same lime outline, and bubbles disappear when the player pops them, not when a clock says so.

The clock changes too. In the Arrays part, `draw` compared `millis()` against a deadline sixty times a second (Section 5.7.4). Bubble Buster hands that job to `setInterval` instead (Section 6.6.8), so `draw` never thinks about time at all:

```
circleInterval = setInterval(addRandomCircle, waitingTime);
```

Remember the rule for handing a function over. `addRandomCircle` without parentheses passes the function itself; with parentheses you would call it once yourself and hand over its result.

6.7.3 One function, one job

The guiding idea of this chapter is small enough to write on one line. Every function you write does one job, and its name says which one. When you can describe a function’s job without the word “and”, the split is right. Look at the first job, creating a bubble. It needs nothing from the caller, and it gives nothing back; it just adds one bubble to the arrays:

```
/**
 * Helper method that adds a random circle to the screen.
 */
function addRandomCircle(): void {
  circlesX.push(random(width));
  circlesY.push(random(height));
  circlesDiameter.push(random(10, 50));
}
```

No parameters between the parentheses, and `: void` as the return type, because there is no answer to hand back. The function reaches straight into the global arrays, which is exactly why it works both as an interval callback and as a normal call from `setup`.

Drawing the bubbles is a different job, and it stays in `draw`, which wipes the canvas each frame and repaints every bubble the arrays remember.

6.7.4 Answering a question: `isInside`

The second function has the opposite shape. It takes three values in, it draws nothing, and it hands back an answer. The question it answers is whether a point lies inside a bubble, and the answer is a `boolean`.

A point is inside a circle when its distance from the center is smaller than the radius, so the function needs that distance. The click and the center form a right triangle, with one horizontal leg and one vertical leg:

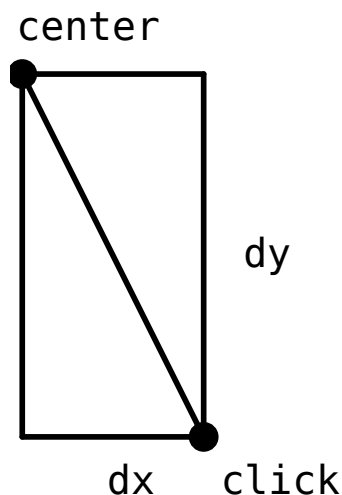


Figure 6.7.2: The bubble center and the click point are two corners of a right triangle. `dx` is the difference of the x values, `dy` the difference of the y values, and the diagonal between the two corners is the distance you need.

The Pythagorean theorem from math class turns the two legs into the diagonal, so `a * a + b * b` is the square of the distance and `sqrt` brings it back down to a length:

```

/**
 * Helper method that checks if a point is inside a circle.
 * @param x The x-coordinate of the point.
 * @param y The y-coordinate of the point.
 * @param circleIndex The index of the circle in the circles array.
 * @returns True if the point is inside the circle, false otherwise.
 */
function isInside(x: number, y: number, circleIndex: number): boolean {
  const dx: number = x - circlesX[circleIndex];
  const dy: number = y - circlesY[circleIndex];
  const distance: number = sqrt(dx * dx + dy * dy);
  return distance < circlesDiameter[circleIndex] / 2;
}

```

The last line deserves a slow read. The comparison `distance < circlesDiameter[circleIndex] / 2` produces `true` or `false`, and `return` hands that value straight to whoever asked. The diameter is halved because the radius is what a distance has to beat.

Note the third parameter. `x` and `y` are a point anywhere on the canvas, while `circleIndex` says which bubble to compare against, so the bubble data stays in the arrays where it belongs.

i p5.js could do this for you

p5.js has a built-in function `dist(x1, y1, x2, y2)` that computes the distance between two points in one call. Bubble Buster does the math by hand on purpose, because the Pythagorean theorem shows up in graphics programming again and again. Once you have seen it work, `dist` is a fair shortcut in your own projects.

6.7.5 Popping bubbles, backwards

Clicking is handled by `mouseClicked`, the callback p5.js calls for you. It walks all bubbles, asks `isInside` about each one, and removes the ones that were hit:

```

function mouseClicked(): void {
  for (let i: number = circlesX.length - 1; i >= 0; i--) {
    if (isInside(mouseX, mouseY, i)) {
      circlesX.splice(i, 1);
      circlesY.splice(i, 1);
      circlesDiameter.splice(i, 1);
      points++;
    }
  }
}

```

The loop header runs from `length - 1` down to 0, and that is the same rule you met with the bursting bubbles (Section 5.7.7). Any loop that may `splice` elements out of the array it is walking has to walk backwards, or it silently skips the element that slid into the freed index.

Two overlapping bubbles under one click are both removed, because the loop keeps going after a hit. Every removal also raises `points`, which `draw` prints in the top left corner.

6.7.6 Stopping the game

Bubbles that nobody pops pile up, and when ten of them are on the canvas the game is over. The check belongs at the very top of `draw`, before any drawing happens:

```

function draw(): void {
  background("black");

  // If more than 10 circles are on the screen, stop the game.
  if (circlesX.length >= 10) {
    stopGame();

    // Note that the return statement stops the execution of the function.
    return;
  }
}

```



```

    // ... draw the bubbles and the score ...
}

```

A bare `return;` inside a `void` function means “I am done here”. The rest of `draw` never runs, so the game over screen that `stopGame` painted stays on the canvas instead of being covered by bubbles.

`stopGame` clears both intervals, paints the final message, and ends with a `p5.js` function that is new here, `noLoop()`. A call to `noLoop()` tells `p5.js` to stop calling `draw` altogether, so `draw` stops running sixty times a second just to check the same `if` forever. Its partner is `loop()`, which starts the calls again, for example when you add a restart to the game.

6.7.7 Leveling up by replacing an interval

The game gets harder over time, and the trick is that an interval can be thrown away and replaced. A second interval fires every ten seconds and does exactly that:

```

/**
 * Helper method that is called when the player advances to the next level.
 */
function nextLevel(): void {
  clearInterval(circleInterval);
  waitingTime /= 2;
  circleInterval = setInterval(addRandomCircle, waitingTime);
}

```

Three lines, one job. `clearInterval` stops the old bubble timer, `waitingTime /= 2` halves the pause, and `setInterval` starts a new timer with the shorter pause. Storing the new id back into `circleInterval` matters, because the next level has to be able to stop the timer that is actually running.

6.7.8 Your exercise: Bubble Buster

The starter code gives you the three arrays, the timer variables, the score, and empty `setup` and `draw` functions. Everything else is yours, and the exercise description lists five tasks. Work through them in order and run the program after each one.

1. **Random bubbles.** Write `addRandomCircle` as described in Section 6.7.3, call it once in `setup` so the first bubble is there immediately, and start the interval that calls it every three seconds. Draw all bubbles in a loop in `draw`.
2. **Popping.** Write `isInside` with its three parameters and its `boolean` return value, then write `mouseClicked` with the backwards loop that splices out every bubble under the pointer.
3. **Points.** Count each popped bubble and print the score in the top left corner with `text`.
4. **Levels (advanced).** Add the second interval that calls `nextLevel`, so the bubbles come twice as fast every ten seconds.
5. **Game over (advanced).** Add the early `return;` in `draw` and the `stopGame` function that clears both intervals, shows the final score, and calls `noLoop()`.

💡 Exercise: Bubbles With Functions

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/bubbles_functions.yaml

When your game runs, read the sample solution and compare its decomposition with yours. That comparison is the real lesson here.

6.7.9 Check your understanding

When your bubbles pop under the mouse and you can say out loud what each of your functions is responsible for, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: A game from functions

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/5xhezrj2tx>

6.8 Many files, one program



Figure 6.8.1: The finished snowman on a white 800 by 500 canvas: two stacked body circles in a pale ice blue, a black hat, coal eyes, an orange carrot nose, a smiling mouth of coal dots, and six buttons down the front.

Every program you have written so far lived in a single file called `index.ts`. That stops here. Melting Snowman is a guessing game in the style of Hangman, and its code is spread over two files, one for the game and one for the drawing of the snowman. This chapter builds the drawing half and, with it, the skill of finding your way around a program that no longer fits on one screen.

6.8.1 AI tutor

Two files means a new kind of confusion, because a mistake can look like it happens in the file you are reading while it really comes from the other one. When something does not appear on the canvas, tell the tutor which file you edited and which one you expected to react.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgrmw3hw>

6.8.2 Why programs get split into files

Splitting a program into several files is called **modularity**, and each file is a module. It costs a little order and pays back three things.

- **You find things faster.** A snowman drawing bug sends you to `snowman.ts`, not into a scroll through 300 mixed lines.
- **Several people can work at once.** One person writes the game rules while another person improves the snowman, and nobody overwrites anybody.
- **A file can be reused.** A module that only draws a snowman can be copied into a Christmas card program without dragging the guessing game along.

The rule of thumb is the same one you know from functions. One function has one job, and one file holds the functions that belong to one job. Melting Snowman uses `index.ts` for the game itself and `snowman.ts` for everything that draws the snowman.

6.8.3 One program on several sheets of paper

In the playground, the files of a program are not separate worlds. They are one program written on several sheets of paper, and the computer reads all sheets before it starts. A function defined in `snowman.ts` can be called from `index.ts` with no extra ceremony, exactly as if both lines stood in the same file:

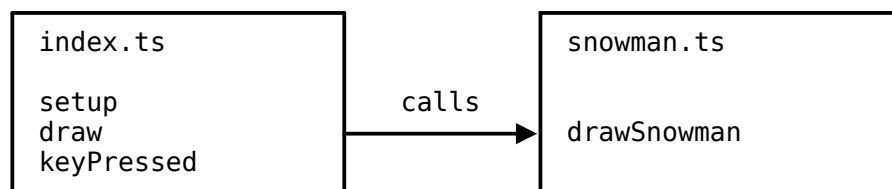


Figure 6.8.2: The call needs no ceremony: `draw` lives in `index.ts` and calls `drawSnowman`, which is defined in `snowman.ts`. Both files together form one program.

The same is true for global variables. A `let` at the top of one file is visible in all of them, which is convenient and also a reason to give globals clear names.

i Real projects say where things come from

Bigger TypeScript projects do not share everything automatically. There, each file lists what it needs from other files with `import`, and says what it offers with `export`. The playground leaves that machinery out so you can concentrate on the split itself, and later courses pick `import` and `export` up properly.

6.8.4 Drawing once instead of sixty times a second

A snowman does not move. Nothing in this program changes unless the player presses a key, so repainting the canvas sixty times a second would burn power to produce the very same picture. Two p5.js functions give you a program that draws only when there is a reason:

```
function setup(): void {
  createCanvas(800, 500);
  angleMode(DEGREES);

  // Draw the screen one time
  redraw();
  noLoop(); // Stop calling draw() automatically
}
```

`noLoop()` switches the automatic calls to `draw` off. `redraw()` asks for exactly one call, right now. Together they turn `draw` from a loop that runs forever into a command you trigger when something has actually changed. `draw` itself stays a normal `draw`, and you still never call it directly; you call `redraw()` and p5.js calls `draw` for you.

6.8.5 Reacting to the keyboard

Melting Snowman is played on the keyboard, so the program needs a third callback next to `setup` and `draw`. You have met `keyPressed` and the variable `key` once before, in the research exercise at the end of the Variables part (Section 2.9.4). There you looked them up in the p5.js reference yourself; from here on they are a regular part of the course:

```
function keyPressed(): void {
  numberOfKeypresses++;

  // Refresh the screen one time
  redraw();
}
```

p5.js calls `keyPressed` once for every key the player presses, and inside it the global variable `key` holds the character that was pressed, as a string. Pressing the A key makes `key` the string "a", and holding shift makes it "A". This first part of Melting Snowman ignores which key it was and only counts the presses, so every key melts the snowman a bit further. The letters start to matter when the guessing logic arrives.

What is new is the partner that `keyPressed` works with here. The pairing of `keyPressed` and `redraw()` is the whole event loop of this program. A key changes a variable, `redraw()` repaints the canvas once with the new value, and then the program waits again.

6.8.6 A doc comment as a specification

The file `snowman.ts` already contains a complete `drawSnowman` function that draws the snowman with all its parts, and above it sits a doc comment. That comment is not decoration; it is the specification of what the function is supposed to do, and your task is to make the code match it:

```
* * 1 wrong: Lower three buttons
* * 2 wrong: Upper three buttons
* * 3 wrong: Left half of mouth
* * 4 wrong: Nose
* * 5 wrong: Right half of mouth
* * 6 wrong: Left eye
* * 7 wrong: Right eye
* * 8 wrong: Hat
* * 9 wrong: Top body part
* * 10 wrong: GAME OVER
```

Reading a specification carefully is a skill of its own, and one word in this one carries most of the meaning. The parts left out are **additive**. Three wrong guesses do not hide the mouth half alone; they hide the parts for one, two, and three wrong guesses together, so the snowman keeps melting instead of growing back.

Additive is exactly what a “less than” comparison gives you. The nose belongs to the number 4, so it is drawn as long as the count has not reached 4:

```
// Nose
if (numberOfWrongGuesses < 4) {
  push();
  noStroke();
  fill("orange");
  triangle(0, 195, 0, 165, 40, 180);
  pop();
}
```

At zero, one, two, or three wrong guesses the nose appears. From four on it is gone and stays gone, because no larger number is ever less than 4. Wrap each part in the comparison that belongs to its line in the specification and the melting takes care of itself.

Two groups of parts need a different move. The buttons and the mouth dots are not drawn one by one but by **for** loops that repeat a shape six times, moving or rotating the origin between rounds. Hiding half of them means running the loop fewer times rather than putting an **if** around a single shape, so the number the loop starts at is what your condition should change.

6.8.7 Your exercise: Melting Snowman (1)

The starter code is a running program. `index.ts` counts key presses and hands the count to `drawSnowman`, and `snowman.ts` draws the complete snowman and ignores the count so far. You change that.

1. **Run it first.** Press a few keys and watch nothing happen. The count rises, the picture stays, and that is the gap you close.
2. **Read both files.** Find the call in `index.ts` and the definition in `snowman.ts`, and follow the value from the key press to the parameter `numberOfWrongGuesses`.
3. **One part at a time.** Start with the nose from Section 6.8.6, run, and press four keys to check it disappears at the right moment. Then do the hat, the eyes, and the body parts.
4. **Then the loops.** The mouth and the buttons come last, because they need the loop start instead of a plain **if**. Test each of them with the exact number from the specification.
5. **Play the whole melt.** Press ten keys in a row and compare every step with the list in the doc comment.

! No sample solution for this exercise

This exercise comes without a sample solution on purpose, so the specification in the doc comment is your only reference. The next chapter's exercise gives you a finished `snowman.ts` as part of its starter code, and comparing it with your version then is worth much more than reading it now.

💡 Exercise: Melting Snowman (1)

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/melting-snowman-1.yaml>

6.8.8 Check your understanding

When your snowman melts key press by key press in the order the specification demands, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Many files, one program

► Take the quiz on app.novedu.at



Scan the code, or type this address into your browser:

<https://app.novedu.at/o5ur816cnx>

6.9 Finish the game: logic functions



Figure 6.9.1: Melting Snowman in the middle of a round: the snowman has already lost its buttons and its nose, and under the hat the word to guess stands in blue monospace letters, some revealed, the rest still underscores.

The snowman can melt, but nothing yet decides when it should. This chapter adds the guessing to Melting Snowman, and the interesting part is where that guessing lives. The functions that work out what a keystroke means take strings in and hand strings back without touching the canvas, which makes them the first functions in this course you can check without running the program.

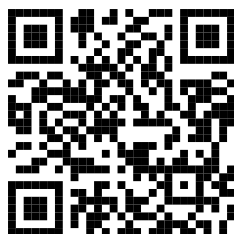
6.9.1 AI tutor

The bugs in this exercise are quiet ones. A letter that refuses to show up, or a game that counts a correct guess as wrong, almost always comes from a comparison rather than from the drawing. Paste the function you suspect together with the word you tested it with, and say what you expected the return value to be.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.9.2 Functions that think and functions that draw

Melting Snowman is now spread over three files. `index.ts` holds the game state and the callbacks, `snowman.ts` draws the snowman, and `textGuesser.ts` holds everything about the word. Inside that third file, two very different kinds of function sit next to each other.

Logic functions take values and return values. They compute an answer and change nothing you can see. `getInitialCurrentWord` and `guessKey` are of this kind, and both of them take strings and return a string.

Drawing functions put pixels on the canvas and return nothing. `drawResult` and `drawSnowman` are of this kind, and their return type is `: void`.

Keeping the two apart pays off immediately, because a logic function can be checked on paper. You write down what you pass in, you follow the code line by line, and you compare the result with what you expected. No canvas, no timing, no guessing. A drawing function can only be judged by looking at the picture.

6.9.3 The word status string

The game shows the player how much of the word is already known, and it keeps that picture in a single string called `currentWordStatus`. The string is exactly as long as the word to guess. Every letter that is still secret stands there as an underscore, every letter already guessed stands there as itself, and a space stays a space so the player can see where one word ends and the next begins.

For the word "ice cream" the game starts with "___ _", which is three underscores, a space, and five more underscores.

Building that first string is the job of one function, and its signature is given in the starter code:

```
function getInitialCurrentWord(wordToGuess: string): string {
```

One string goes in, one string comes out. The way there is the accumulator you know from Word Swirrel (Section 5.1.3). Start with the empty string, walk the characters of `wordToGuess` from index 0 to `length - 1`, and hang one character onto the result each round, an underscore for a letter and a space for a space.

6.9.4 Guessing a letter

The second logic function is the heart of the game. It gets the key the player pressed, the secret word, and the current status, and it returns what the status should look like afterwards:

```
function guessKey(key: string, wordToGuess: string, currentWordStatus: string): string {
```

The function does not change `currentWordStatus` and it does not know whether the guess was good or bad. It builds a new string of the same length, position by position. Where the secret word has the guessed letter, the new string shows that letter; everywhere else it keeps whatever the old status had.

Follow four rounds with the short word "ice". The status starts as three underscores, and every row shows one key press:

Table 6.9.1: Four guesses on the word "ice". The second row is a wrong guess.

key pressed	status before	returned status
i	---	i__
x	i__	i__
e	i__	i_e
c	i_e	ice

Look at the second row. The function does not report a wrong guess; it simply produces a string identical to the one that went in. The caller in `index.ts` uses precisely that:

```
const newCurrentWordStatus: string = guessKey(key, wordToGuess, currentWordStatus);
if (currentWordStatus === newCurrentWordStatus) {
    wrongGuesses++;
}
currentWordStatus = newCurrentWordStatus;
```

Nothing changed means nothing was found, so the guess was wrong. One small function, one clear answer, and the decision stays with the caller.

6.9.4.1 Capital letters must not matter

The word to guess in the starter code is "Winterwald", with a capital W, and the player will type a small w. Comparing the two with `===` fails, and the player would never be able to reveal the first letter.

Every string has a method `toLowerCase()` that returns a copy in small letters and leaves the original alone, the same way `substring` hands you a copy. Comparing `key.toLowerCase()` with `wordToGuess[i].toLowerCase()` makes the guess work no matter which shift key was involved.

⚠️ Reveal the original character, not the typed one

When the comparison matches, put the character from `wordToGuess` into the new status, not the key the player pressed. The game recognizes a win by testing `currentWordStatus === wordToGuess`, and a status that reads "winterwald" never equals a word that reads "Winterwald".

6.9.5 Waiting for the font

The word status is drawn in a monospace font that the program loads from the internet, and that download takes time. A program cannot draw with a font that has not arrived yet, so `setup` waits for it:

```
async function setup(): Promise<void> {
    font = await loadFont("https://cdataexchange.blob.core.windows.net/images/SyneMono-Regular.ttf");

    currentWordStatus = getInitialCurrentWord(wordToGuess);

    createCanvas(800, 500);
    angleMode(DEGREES);

    redraw();
    noLoop();
}
```

Read `await` as “wait here until it has arrived”. The line pauses until `loadFont` is finished, and the font lands in the global variable `font` before any of the following lines run. A function that contains an `await` must be marked `async`, and its return type becomes `Promise<void>` instead of `void`, which is TypeScript’s way of writing “returns nothing, but not immediately”. You do not need the theory behind promises today. The next chapter uses the very same move for loading images.

6.9.6 Who decides that the game is over

All the state of the game sits in global variables in `index.ts`, and `draw` reads them to pick one of three pictures:

```
function draw(): void {
    background("white");

    if (currentWordStatus === wordToGuess) {
        acceptKeys = false;
        drawResult(true, wrongGuesses);
    } else if (wrongGuesses === MAX_WRONG_GUESSES) {
        acceptKeys = false;
        drawResult(false, wrongGuesses);
    } else {
        drawSnowman(wrongGuesses);
        drawCurrentWordStatus(font, currentWordStatus);
    }
}
```

The status equal to the secret word means the player has won. Ten wrong guesses mean the snowman has melted away. Anything else means the round continues, so the snowman and the word appear. In the two end states `acceptKeys` becomes `false`, and `keyPressed` leaves immediately with a bare `return`; while that flag is down, so a finished game ignores the keyboard.

`drawResult` is the drawing counterpart of the two logic functions. It gets a `boolean` saying whether the player won and the number of wrong guesses, and it writes one sentence in the middle of the canvas. A loss reads “Game Over” in red. A win reads “No wrong guesses!”, “One wrong guess!”, or something like “4 wrong guesses.”, depending on the count. Choosing between those three sentences is a job for the `switch` statement you know from the Conditions part.

6.9.7 Your exercise: Melting Snowman (2)

The starter code contains the finished `index.ts`, a complete `snowman.ts` that melts correctly, and a `textGuesser.ts` with three empty function bodies. If your snowman from the previous exercise never quite melted in the right order, compare it with the one you get here before you start.

1. **The initial status.** Implement `getInitialCurrentWord` as described in Section 6.9.3, then run the program. The underscores appear next to the snowman. Change the word in `index.ts` to one with a space in it and check that the space stays a space.
2. **The guess.** Implement `guessKey`. Work it through on paper first with the table above (Table 6.9.1), then write the loop. Correct letters appear, wrong ones melt the snowman.
3. **The result text.** Implement `drawResult` following the requirements in its doc comment.

Test the finished game in all four states before you call it done.

- Win without a single wrong guess.
- Win with exactly one wrong guess, so the text reads “One wrong guess!”.
- Win with several wrong guesses, so the number appears in the text.
- Lose by making ten wrong guesses on purpose and watch the snowman disappear completely.

💡 Exercise: Melting Snowman (2)

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/melting-snowman-2.yaml>

6.9.8 Check your understanding

When your game can be won and lost and you can explain why a wrong guess is recognized by comparing two strings, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Finish the game

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/rbzd15az6>

6.10 Images and orthographic worlds

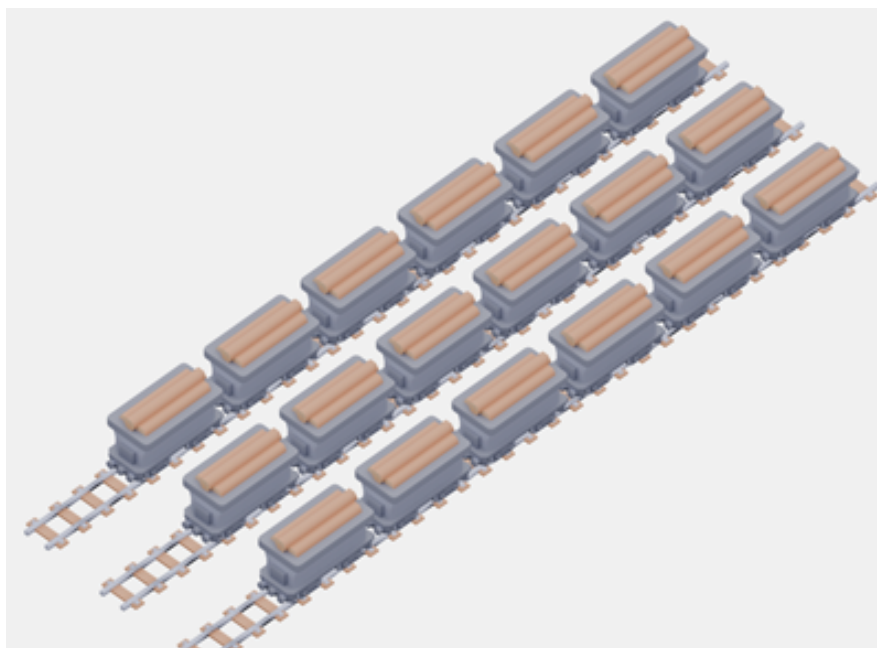


Figure 6.10.1: Three railroad tracks, each carrying a train of six wagons. Every tile in this picture is the same railroad image and the same wagon image, drawn again and again at shifted positions.

Until now, everything on your canvas came out of p5 drawing commands: circles, rectangles, lines, text. This chapter hands you a new kind of building block, the picture file. You load ready-made images into your program and place them on the canvas, and while you do that you learn the design idea behind games like the one in the goal picture above. The tiles fit together so perfectly because of the way they were rendered, and the second half of this chapter explains exactly why.

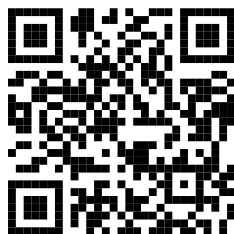
6.10.1 AI tutor

Two things go wrong in every first image program. The picture does not appear at all, or it appears squashed. If your canvas stays empty, tell the tutor the URL you used and where your `loadImage` call sits; if the picture looks wrong, describe what you passed to `image`.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.10.2 Pictures your program loads

An image is not part of your code. The picture file lives somewhere on the internet, under an address, and your program has to fetch it before it can draw it. Fetching takes time, a few milliseconds on a good connection and much longer on a bad one, so p5 gives you a command that waits for the picture and hands it over when it has arrived.

```
const BASE_URL: string = "https://cddataexchange.blob.core.windows.net/images/trains";
```

```
let railroad: p5.Image;

async function setup(): Promise<void> {
  railroad = await loadImage(`${BASE_URL}/railroad-straight.png`);

  createCanvas(800, 550);
}
```

You met this shape in the melting snowman game, where `setup` loaded a font the same way. Three parts work together here.

- `loadImage(url)` starts fetching the picture at that address.
- `await` means “wait here until it has arrived”. Without it, the next line would run while the picture is still on its way, and you would draw nothing.
- A function that contains `await` has to be marked `async`, and its return type becomes `Promise<void>`. Read that as “this function finishes later, and when it does, it returns nothing”.

The variable `railroad` has the type `p5.Image`, the type of a loaded picture. It stands outside `setup` because `draw` needs it too, and it is declared with `let` because it only gets its value once the image has arrived.

6.10.3 Drawing a picture: `image`

The command that puts a loaded picture on the canvas is `image`, and the version this course uses takes five arguments.

```
image(railroad, 0, 0, railroad.width, railroad.height);
```

The first argument is the picture. The next two are the position of its top left corner, exactly like the top left corner of a rectangle. The last two are the width and the height the picture gets on the canvas.

Every `p5.Image` knows its own size, so `railroad.width` and `railroad.height` are the real pixel size of the loaded file, and passing those two values draws the picture at its natural size. Any other pair of numbers stretches or squeezes it. That is a real tool when you want it (half size, double size) and a common accident when you don’t, because a picture with the wrong proportions looks subtly broken. When you change one of the two, change the other by the same factor.

6.10.4 Two ways to look at the world

Now to the design half of this chapter. Every 3D picture, whether it comes from a camera or from a program, is a flat image of a world that has depth, and there are two ways to flatten it.



Figure 6.10.2: A cube rendered with a perspective camera. The top face is visible and the vertical edges lean toward a vanishing point, so the back of the cube is smaller than the front.

Perspective works like your eyes and like a photo camera. Objects that are farther away appear smaller, and parallel lines (think of railroad rails running to the horizon) come closer together in the distance.



Figure 6.10.3: The same cube from the same angle, rendered with an orthographic camera. All edges stay parallel and distance changes no size, so the cube reads as flat bands of color.

Orthographic ignores distance. An object keeps its size no matter how far away it is, and parallel lines stay parallel. Nobody sees the world this way, which is the point. Technical drawings use it because you can measure a length directly in the drawing, and 3D printing software uses it for the same reason.

6.10.5 Why orthographic tiles can repeat

A tile-based world is built from a few pictures used over and over. The goal picture of this chapter is drawn from exactly two images, one railroad segment and one wagon. Whether that works at all is decided by the projection.

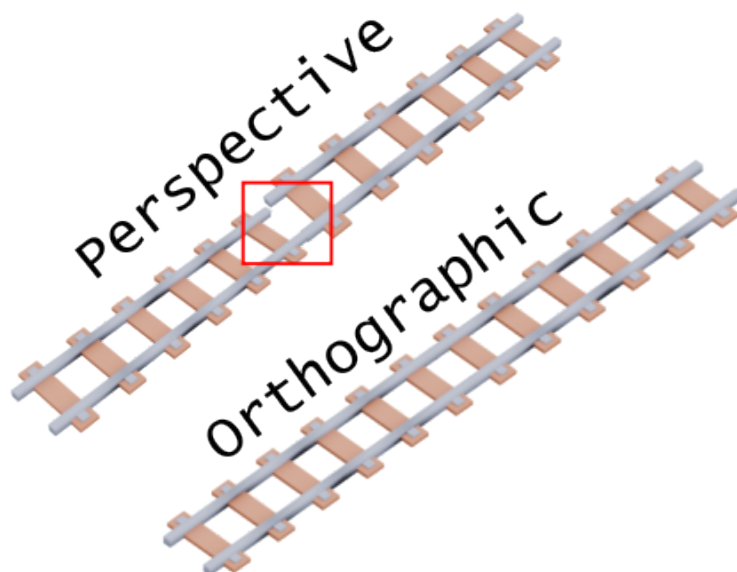


Figure 6.10.4: The same railroad segment repeated several times, rendered both ways. In the perspective version the near end of a segment is wider than the far end of its neighbor, so the rails jump at the joint marked in red. In the orthographic version the segments fit exactly.

In the perspective rendering, the segment is wider at the near end than at the far end, because that is what perspective does. Put a copy next to it and the wide end meets a narrow end. The rails do not line up, and no amount of shifting fixes it, because the two ends have different sizes.

In the orthographic rendering, both ends are equally wide. One copy ends where the next one begins, so you can build a track of any length out of a single image file. Classic tile games are built on this. In the strategy game SimCity 2000, the whole city is assembled from small orthographic tiles, which is why streets and buildings snap together across the entire map. You can see a screenshot of it in the [Wikipedia article about SimCity 2000](#).

6.10.6 Drawing from right to left

Pictures with a transparent background overlap, and the rule for overlapping is simple: whatever you draw later covers what is already on the canvas. In a world seen from an angle, the tile that is nearer to the viewer has to be drawn later than the tile behind it.

In this railroad world, nearer means further to the left and further down. So the drawing starts at the top right, at the farthest segment, and walks left and down toward the viewer.

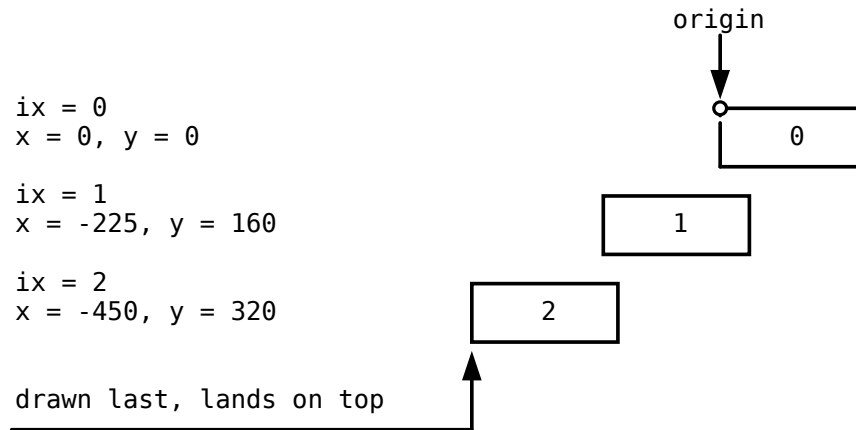


Figure 6.10.5: Drawing order for the railroad segments. The origin sits at the top right, and each segment with a higher index is drawn further left and further down, on top of the segment before it.

The starter code sets this up in three lines at the top of `draw`.

```
scale(0.5, 0.5);

translate(850, -100);

for (let i: number = 0; i < 5; i++) {
  drawRailroad(i);
}
```

`scale(0.5, 0.5)` halves everything, because the rendered images are large and the canvas is 800 by 550. `translate(850, -100)` moves the origin to the upper right corner, so that negative `x` values still land on the canvas. Both work on the coordinate system, exactly as in the patterns chapter (Section 4.8.4), and both apply to the numbers you write afterwards.

The function that draws one segment turns an index into a position.

```
/** Draw a railroad segment */
function drawRailroad(ix: number): void {
  image(railroad, -RAILROAD_WIDTH * ix, RAILROAD_HEIGHT * ix, railroad.width, railroad.height);
}
```

Segment 0 lands at the origin. Segment 1 lands one width to the left (a negative `x`) and one height down. The loop counts up, so each new segment appears nearer to the viewer than the one before, which is exactly the order the overlap needs.

6.10.7 Your exercise: Orthographic Rendering

The exercise starts with a working track of five railroad segments and asks you to put a train on it.

1. **Read the starter code.** It contains four constants with the sizes of the two images, the `drawRailroad` function explained in Section 6.10.6, and comments starting with `// <<<` at every place where your code goes.
2. **Load the wagon.** Add a second `p5.Image` variable and load `train-carriage-wood.png` in `setup`, with `await`, like the railroad image.
3. **One wagon function.** Write `drawTrainWagon(ix: number): void` in analogy to `drawRailroad`, using `WAGON_WIDTH` and `WAGON_HEIGHT`. The wagons are smaller than the segments, so they need their own step sizes.

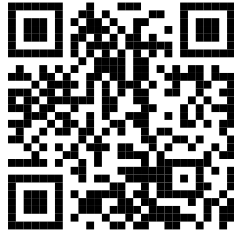
4. **Six wagons.** Call it in a loop in `draw`. Six wagons on five segments is correct; the wagon step is smaller than the segment step.
5. **Three tracks (task 2).** Wrap both loops in an outer loop that runs three times and ends with `translate(125, 100);`. Each round shifts the origin to the next track, and the two inner loops draw the same thing again in the new place.

Images bring failure modes that shapes never had. A typo in the URL, a forgotten `await`, a size that squashes the picture, and the canvas stays empty or looks wrong with no error message to read. That is worth its own helper.

💡 Your image helper

Images bring new failure modes: a typo in the URL, a forgotten `await`, a distorted size. This AI helper specializes in images and helps you in every exercise from now on.

▶ [Open the image helper on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/u1s11rxld1>

💡 Exercise: Orthographic Rendering

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

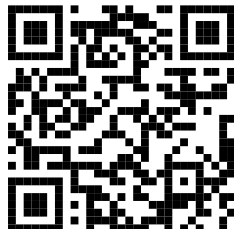
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/orthographic.yaml>

6.10.8 Check your understanding

When your three tracks stand on the canvas and you can say why the wagons had to be drawn from right to left, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Images and orthographic worlds

▶ [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/z6eb02cby1>

6.11 Trains from strings: parsing functions

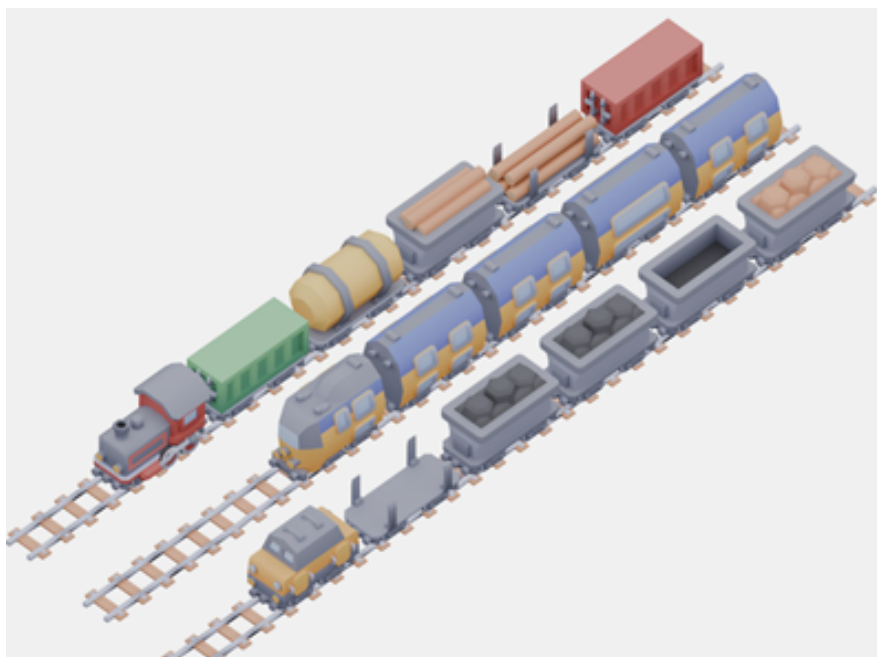


Figure 6.11.1: Three tracks in a train station, each with a different train on it. A steam locomotive with freight wagons, a blue and yellow city train, and a short yellow train at the front. Not one of these trains is written in the code; they all come out of a single string.

A real train station gets its data from another computer system, as text. A line like `TLB,TCCG,TCTL,TCW,TCL,TCCR` says which wagons stand on a track, one abbreviation per wagon. You know how to pick a string apart character by character, and you know how to draw an image. This chapter packages both into functions that hand back arrays.

6.11.1 AI tutor

A parser that returns pictures fails quietly, because a wrong index gives you the wrong wagon instead of an error message. If a wagon looks unexpected, tell the tutor the input string and which abbreviation should have produced it. If the canvas stays empty, say how many images your loading loop pushed into the array.

Your AI tutor

Hints and questions instead of finished programs, in English or German.

► [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.11.2 Loading many images with `for...of`

This exercise needs 41 wagon images. Three things come with the starter code: the array `imageUrls` with the file names, the constant `BASE_URL` with the address they hang on, and the empty array `trainCars` for the loaded pictures. A loop fills the last one. With the counting loop you already know, it looks like this.

```
for (let i: number = 0; i < imageUrls.length; i++) {  
  const wagonImage: p5.Image = await loadImage(`${BASE_URL}/${imageUrls[i]}`);  
  trainCars.push(wagonImage);  
}
```

```
}
```

Look at what `i` is used for. It appears once, to get `imageUrls[i]` back out of the array. That is pure bookkeeping, and TypeScript has a loop that does it for you.

```
for (const imageUrl of imageUrls) {  
    const wagonImage: p5.Image = await loadImage(`${BASE_URL}/${imageUrl}`);  
    trainCars.push(wagonImage);  
}
```

`for...of` walks through an array and hands you one element per round, directly in a constant you name yourself. No counter, no `length`, no square brackets, and no chance of an off-by-one mistake. Read the head out loud as “for each `imageUrl` of `imageUrls`”.

Use `for...of` when you need every element and no index arithmetic. Use the classic `for` when the position itself matters, for example to count, to walk backwards, or to compare two arrays position by position.

i The one declaration that stays without a type

Every declaration in this course carries its data type, and the head of a `for...of` loop is the single place where TypeScript refuses one. `for (const imageUrl: string of imageUrls)` is a syntax error, so the head keeps the short form you see above. TypeScript still knows the type, because it reads it off the array. The counter of a classic `for` loop is a normal declaration and keeps its `: number` (Section 4.1.3).

Inside the loop, `await` waits for one image before the next one starts. When the loop ends, `trainCars` holds as many images as `imageUrls` holds names, in the same order.

6.11.3 The abbreviation table

The file `wagons.ts` contains two arrays that belong together. `imageUrls` holds the 41 file names, `abbreviations` holds the 41 short codes that arrive in the data, and both are in the same order. They are parallel arrays, the technique from the soccer field chapter (Section 5.6.2), so `abbreviations[1]` is “TCC” and `imageUrls[1]` is the coal wagon. Because `trainCars` was filled in the order of `imageUrls`, the index carries over one step further, from `abbreviations` straight into `trainCars`. That gives you the whole job in one sentence. Find the position of an abbreviation, then take the image at the same position.

6.11.4 A search you write yourself

Searching an array is one of the oldest jobs in programming, and the simplest method is what a person does with a printed list. Start at the top, compare, and stop at the first match.

```
function getWagonIndex(wagonAbbreviation: string): number {  
    for (let i: number = 0; i < abbreviations.length; i++) {  
        if (wagonAbbreviation === abbreviations[i]) {  
            return i;  
        }  
    }  
    // This should NEVER happen!  
    return -1;  
}
```

This is called a **linear search**, because it walks the array in a line from front to back. Two details make it work. `return` inside a loop leaves the whole function immediately, so the remaining rounds never run and no “found it” variable is needed. And the second `return` is reached only when the loop ran through without a match. It answers with `-1`, the usual convention for “not found”, because no array position is ever negative.

The exercise forbids the built-in search function `indexOf` on purpose. Searching is a skill, not a button, and later courses compare this simple search against faster ones.

6.11.5 A function that returns an array

Now the parser. `parseTrain` takes one train string and hands back the images of its wagons in drawing order, which its return type `p5.Image[]` says exactly. The loop inside is the collecting loop from the parsing chapter

(Section 5.5.3). A buffer variable collects characters until a separator arrives, the separator here is the comma, and the last abbreviation is still in the buffer when the loop ends (Section 5.5.4).

One line at the end is new. `reverse()` turns an array around, so the first element becomes the last one. The train needs that, because the wagons are drawn from right to left and the locomotive, first in the string, has to end up at the front of the train.

```
function parseTrain(trainString: string): p5.Image[] {
  const result: p5.Image[] = [];

  let wagon: string = "";
  for (let i: number = 0; i < trainString.length; i++) {
    if (trainString[i] === ",") {
      result.push(trainCars[getWagonIndex(wagon)]);
      wagon = "";
    } else {
      wagon += trainString[i];
    }
  }

  result.push(trainCars[getWagonIndex(wagon)]);

  // The locomotive stands first in the string and must be drawn last.
  return result.reverse();
}
```

Read `trainCars[getWagonIndex(wagon)]` from the inside out. The call `getWagonIndex(wagon)` returns a number, and that number is used straight away as the index into `trainCars`. One function feeds the next, and no extra variable is needed. Notice also that `result` is declared with `const` and still grows, because `const` only forbids pointing the variable at a different array.

6.11.6 Drawing the parsed train

Once `setup` has called the parser, the array holds ready-to-draw images and `draw` only walks through it. That loop is already in the starter code.

```
for (let ix: number = 0; ix < train.length; ix++) {
  drawTrainWagon(train[ix], ix);
}
```

The given function `drawTrainWagon(wagon: p5.Image, ix: number): void` wants two things, the picture and the position number that decides where the wagon lands, and it places the image like the railroad segments in the orthographic chapter (Section 6.10.6). The wagon arrives as a parameter, so this one function draws all 41 wagon types.

6.11.7 An array of arrays

A station has more than one track, and the data says so. In the advanced part of the exercise, one string describes all tracks, with semicolons between the trains and commas between the wagons.

```
const TRACKS: string = "TLB,TCCG,TCTL,TCW,TCL,TCCR;TEDA,TEDB,TEDB,TEDC,TEDB;TDA,TCF,TCC,TCC,TCB,TCD";
```

Three trains means three arrays of images, kept in one variable of the type `p5.Image[] []`. Read the two pairs of brackets as “an array whose elements are arrays”. This is your first **two-dimensional array**.

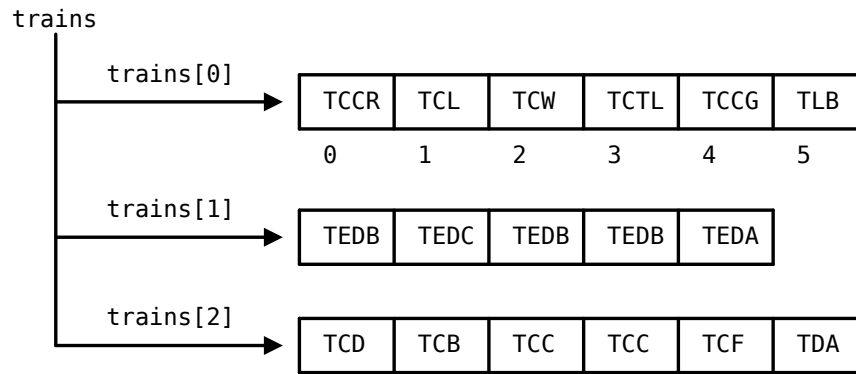


Figure 6.11.2: The variable `trains` holds three arrays, one per track. Each inner array holds the wagon images of one train, already reversed. Reading `trains[1][2]` gives the third image of the second train.

Indexing works in two steps. `trains[1]` is a whole train, an array of images, and a second pair of brackets lands you inside that train, so `trains[1][2]` is one single image. First index picks the track, second index picks the wagon.

The parser for the whole station has the same shape as `parseTrain`, one level up. It cuts at semicolons, and instead of an image it pushes the result of `parseTrain`.

```

function parseTracks(tracksString: string): p5.Image[][] {
  const result: p5.Image[][] = [];

  let track: string = "";
  for (let i: number = 0; i < tracksString.length; i++) {
    if (tracksString[i] === ";") {
      result.push(parseTrain(track));
      track = "";
    } else {
      track += tracksString[i];
    }
  }

  result.push(parseTrain(track));
  return result;
}

```

Drawing then needs two loops, one over the tracks and one over the wagons of the current track. The `translate(125, 100)` at the end of each round moves the origin to the next track, as in the orthographic exercise.

```

for (const track of trains) {
  for (let i: number = 0; i < 5; i++) {
    drawRailroad(i);
  }

  let ix: number = 0;
  for (const wagon of track) {
    drawTrainWagon(wagon, ix);
    ix++;
  }

  translate(125, 100);
}

```

The wagon loop shows an honest compromise. `for...of` gives the picture, but `drawTrainWagon` also wants a position number, so a counter runs alongside. A classic `for` does the same job here, so pick whichever you find easier to read.

6.11.8 Your exercise: Train Station

The starter code gives you the constants, the two drawing functions, and `wagons.ts` with the parallel arrays. The three base tasks build one train.

1. **Load all wagon images.** In `setup`, loop over `imageUrls` and `await loadImage` for each name, pushing every image into `trainCars`. Afterwards `trainCars.length` must equal `imageUrls.length`. Draw both numbers with `text` to check.
2. **Write `getWagonIndex`.** A linear search over `abbreviations`, with `-1` when nothing matches. Test it before you go on by drawing the result of `getWagonIndex("TCW")` on the canvas.
3. **Write `parseTrain`.** The collecting loop, the push after the loop, and the `reverse` at the end. Call it in `setup` with the `TRAIN` constant and store the result in `train`.

The advanced tasks add the other two tracks.

4. **Write `parseTracks`.** Same pattern one level up, semicolon instead of comma, and `parseTrain` for each piece. Store the result in `trains`.
5. **Draw all tracks.** Replace the drawing code in `draw` with the two nested loops, and end each round with `translate(125, 100)`.

Paper helps in step 3. Take the short string "TLB,TCW", write one row per character, and note what the buffer holds, when a push happens, and what the array looks like before and after `reverse`.

💡 Exercise: Train Station

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

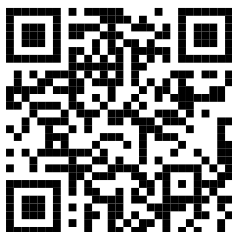
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/train-station.yaml>

6.11.9 Check your understanding

When your three trains stand in the station and you can explain what `trains[1][2]` is, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Trains from strings

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/uvsddvycpo>

6.12 The earned shortcut: split

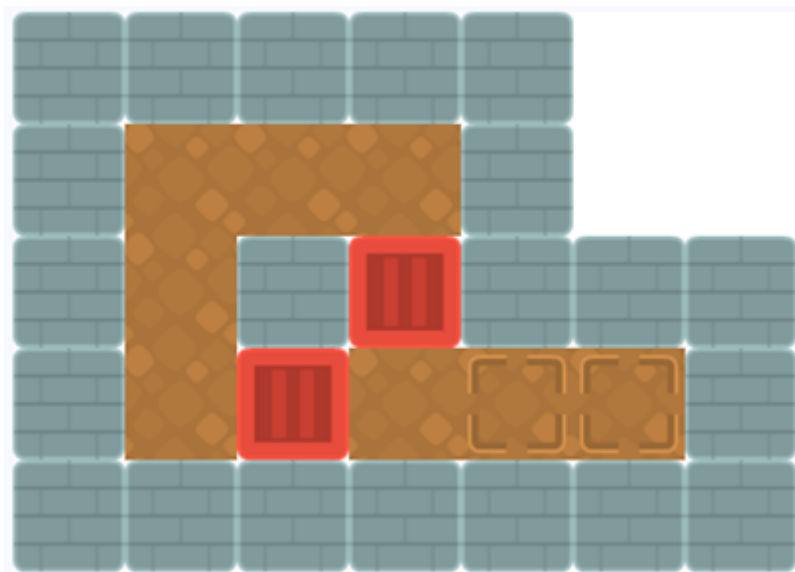


Figure 6.12.1: A Sokoban level on the canvas: a wall of gray blocks around a brown floor, two red crates, and two target markers. The whole picture is built from five image files and one string of characters.

Sokoban is a puzzle game from the early 1980s. You push crates through a warehouse until every crate stands on a marked spot, and you can only push, never pull, which is what makes it hard. This chapter builds the board, not the game. A level arrives as one long string, your program turns it into a grid, and the grid turns into images. Along the way you finally get the tool the Arrays part kept from you.

6.12.1 AI tutor

Grid programs fail in visible ways, which is good news. A row too far right, a level squeezed into a corner, or one wrong tile type all point straight at the code that placed them. Tell the tutor which level string you used and describe what the canvas shows, and check your loop order first when rows and columns look swapped.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.12.2 A shortcut you have earned

In the parsing chapter of the Arrays part, one rule was strict. You had to walk through the string character by character, collect characters in a buffer, and handle the leftover after the loop (Section 5.5.3 and Section 5.5.4). The Train Station chapter made you do it twice more, for commas and for semicolons. That was on purpose, because parsing by hand is the skill, and now you get the tool.

❗ Course rule update: split is now allowed

Until now, cutting a string into pieces meant writing the collecting loop yourself. From this chapter on, `split` is allowed for simple separators, and you should use it. You have built that loop by hand several times and you can explain what it does, so the shortcut hides nothing from you any more. Two things stay

as they are. When the data needs more than “cut at this character”, the character loop is still the right answer. And in an exam, a task may forbid `split`, exactly like the exercises in the Arrays part did.

`split` takes a separator and returns an array of the pieces. The separator itself is thrown away. And if you pass the empty string as the separator, `split` cuts between every pair of characters, so you get an array of single characters.

```
const line: string = "TLB,TCCG,TCW";
const parts: string[] = line.split(",");
// parts is ["TLB", "TCCG", "TCW"]

const word: string = "box";
const letters: string[] = word.split("");
// letters is ["b", "o", "x"]
```

6.12.3 A level made of characters

A Sokoban level is a small map, and the file `levels.ts` writes each map as a picture made of characters.

```
const levels: string[] = [
  `XXXXX__
X  X__
X@XbXXX
X b ..X
XXXXXXX`,
  // ... four more levels
];
```

Two things about that string. It is written with backticks, and a template string may run over several lines. Every line break you type inside it becomes one `\n` character in the value, the line break character you met in Word Swirrel (Section 5.1.6). So the level above is a single string, and `\n` sits between its rows.

Each character says what stands on one square of the map.

Table 6.12.1: The seven symbols of a Sokoban level.

symbol	meaning
X	wall
space	floor you can walk on
@	the player
b	a box
.	a target spot
B	a box that already stands on a target
-	nothing at all, outside the warehouse

In the goal picture at the top of this chapter, the white corner is where the `_` characters sit. Nothing is drawn there.

6.12.4 From one string to a grid

The map has rows and columns, but the string that holds it is one long row of characters. The starter code puts one map into the constant `levelString`, and making its rows real means cutting twice, once at the line breaks and once between the characters. The result is a grid you can index with two numbers.

```
let level: string[][] = [];
let maxWidth: number = 0;

for (const line of levelString.split('\n')) {
  const chars: string[] = line.split('');
  // Track the maximum width to properly size the canvas
  if (chars.length > maxWidth) {
    maxWidth = chars.length;
  }
}
```

```

    level.push(chars);
}

```

The type `string[] []` is the two-dimensional array from the train station chapter (Section 6.11.7), filled with characters this time. `level[2]` is the third row, and `level[2][3]` is the character in its fourth column. Row first, column second, because that is the order the rows were pushed.

`split('\n')` returns the rows and `split('')` cuts a row into its characters. The code writes these two strings with single quotes; single and double quotes make exactly the same string in TypeScript, and both are common in real code.

`maxWidth` grows to the length of the longest row, because rows may differ in length. With that number the canvas fits the level exactly.

```

createCanvas(maxWidth * cellSize, level.length * cellSize);

```

`cellSize` is 64, the pixel size of one square, and `level.length` is the number of rows. So the canvas is as wide as the widest row and as tall as the level has rows.

6.12.5 From a symbol to an image

Five image files cover all the tiles. Their addresses are given in the array `imageNames`, and the loaded pictures go into the array `images`, with the same loop shape as the train wagons.

```

for (const imageName of imageNames) {
    const loadedImage: p5.Image = await loadImage(imageName);
    images.push(loadedImage);
}

```

The order in `imageNames` decides the position of every picture in `images`, so index 0 is the wall, 1 the floor, 2 the target, 3 the box, and 4 a box on a target. Translating a character into one of them is a job for `switch`, the tool from the Conditions part.

```

function getBlockImageBySymbol(type: string): p5.Image {
    switch (type) {
        case "X": // Wall
            return images[0];
        case ".": // Target spot
            return images[2];
        case "b": // Box/Crate
            return images[3];
        case "B": // Box on target
            return images[4];
        default: // Floor or player (currently rendered as floor)
            return images[1];
    }
}

```

Each `case` ends with `return` instead of `break`, because returning leaves the function immediately. The `default` branch catches everything not listed, which here means the space and the `@` of the player. Both are drawn as floor, so the player is invisible in this exercise.

6.12.6 Walking the grid with the origin

Every tile could be placed by calculating its `x` and `y` from the row and column number. The sample solution takes the other road from the move-the-origin chapter and walks the origin instead (Section 4.7.4).

```

for (const row of level) {
    push();
    for (const cell of row) {
        if (cell !== ' ') {
            const img: p5.Image = getBlockImageBySymbol(cell);
            image(img, 0, 0, cellSize, cellSize);
        }

        translate(cellSize, 0);
    }
}

```



```

    pop();
    translate(0, cellSize);
}

```

Every tile is drawn at 0, 0, and the origin does the walking. Inside a row, each round moves it one cell to the right. `push` saves the position at the start of the row and `pop` brings it back, so the next row starts at the left edge again, one `translate(0, cellSize)` lower. The `if` skips the `_` characters, which leaves the canvas white and produces the empty corner of the goal picture.

One thing is unusual in this exercise. All of this runs in `setup`, and there is no `draw` function at all. A level does not move, so it is drawn once and stays on the canvas.

6.12.7 Your exercise: Sokoban Levels

The starter code gives you the image names, the empty arrays, and an unfinished `getBlockImageBySymbol`. Build it in the order below and run after every step.

1. **Load the images.** Loop over `imageNames`, `await loadImage` for each one, and push the result into `images`.
2. **Parse the level.** Cut `levelString` into rows and each row into characters, push each row into `level`, and remember the longest row in `maxWidth`.
3. **Size the canvas.** Create the canvas from `maxWidth`, `level.length`, and `cellSize`, and paint a white background.
4. **Finish `getBlockImageBySymbol`.** Replace the placeholder body with the `switch` over the symbols.
5. **Draw the grid.** The two loops with `push`, `pop`, and the two `translate` calls.
6. **Try other levels.** `levels` holds five maps. Change the index in `levels[0]` and run again. The last map is much wider than the others, which is where your `maxWidth` line proves itself.

💡 Exercise: Sokoban Levels

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

https://ts-web-playground-server-bfd6fbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/sokoban_levels.yaml

6.12.8 Build the whole game

The big optional challenge of this part is the game itself. You have a board on the screen, and the game is what happens when the player presses a key. Nobody walks you through this one, which is the point. You know every construct it needs, so the steps below are a sketch and no more.

- **Read the keys.** `keyPressed` and the variable `key` tell you which key was pressed. Turn a key into a direction, one step in `x` and one step in `y`.
- **Find the player.** The `@` is still in your grid. Two loops over rows and columns find its row and column.
- **Look at the target square.** Player position plus direction gives the square the player wants to enter, and what stands there decides everything. An `X` blocks the move, so nothing happens at all.
- **Pushing moves two things.** If a box stands on the target square, look one more square in the same direction. Only if that one is free do both the box and the player move, and both grid cells change.
- **Redraw after a change.** Put the drawing code into its own function and call it again after every move, or move it into `draw` and use the `noLoop` and `redraw` pattern from the melting snowman.
- **Check for the win.** The player has won when no target spot is left uncovered. Walk the grid and count.

One hint on symbols. A box that moves onto a target becomes a `B`, and a box that leaves a target becomes a `b` again, so you have to remember which squares were targets. The simplest fix is a second grid that never changes and holds only the targets. Give each of these jobs its own function with a clear name, and you get a game you can show to anyone.

6.12.9 Check your understanding

When your level stands on the canvas and you can say why `split` is allowed now and what it does with the separator, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: The earned shortcut

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/jnvxytlo9>

6.13 Design your own functions: Crossword

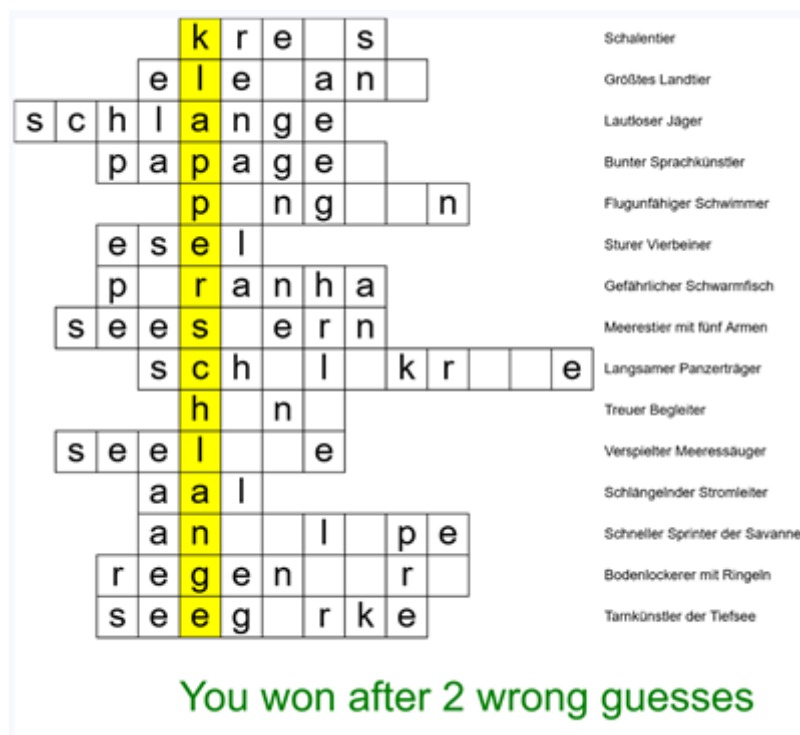


Figure 6.13.1: The finished crossword: 15 animal names, each shifted so that one of its letters lands in the yellow column, hints on the right, and a message below that counts the wrong guesses.

Somebody has always told you which functions to write. The domino exercise named `drawDice` and `drawDomino`, the temperature chart listed its four functions, and the train station told you what `parseTrain` had to return. The crossword task says one thing about functions and nothing more: do not write all the code in a single function. Which functions exist and what each does is your decision now, and that decision *is* the exercise.

6.13.1 AI tutor

Designing your own functions feels wobbly the first time, because there is no single right answer and every exercise before this one had one. Describe your planned function names to the tutor before you write code and ask what it thinks of the split. It will not design the program for you, but it says so when one of your functions is doing three jobs at once.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.13.2 From jobs to functions

The method has four steps, and the first one happens on paper.

1. **Say what the program does**, in plain sentences, one per job. Not code, not names, just the work.
2. **Turn every sentence into a function**. One job, one function.

3. **Name each function after its job**, so the name answers “what does this do?” before anyone reads the body.
4. **Build one function at a time**, run it, look at the canvas.

For the crossword, the sentences come straight out of the task. The program takes the data apart, paints the yellow column, draws the cells with the known letters and the hints, reacts to a key, decides whether that key was a good guess, and decides whether the player has won. Six sentences, six functions:

Table 6.13.1: One possible decomposition of the crossword. Your names and your split may differ, and that is allowed.

The job	A function for it
Take the data apart into arrays	<code>function splitInput(): void</code>
Paint the yellow solution column	<code>function drawSolutionHighlight(): void</code>
Draw cells, known letters, and hints	<code>function drawCrossword(): void</code>
Show wrong guesses or the win message	<code>function drawResult(): void</code>
Decide whether a typed key counts	<code>function isValidGuess(key: string): boolean</code>
Decide whether the game is over	<code>function hasWon(): boolean</code>

Two kinds of function sit in that table. Four of them *do* something and return nothing, so their return type is `void`. Two *answer a question* and return `boolean`, and you can hear it in the names `isValidGuess` and `hasWon`. `draw` becomes a short list of orders:

```
function draw(): void {
    background('white');

    drawSolutionHighlight();
    drawCrossword();
    drawResult();
}
```

The order of the three calls is not decoration. The yellow rectangle goes down first and the cells are painted on top of it, like the bars-then-axes rule of the temperature chart. Turn the first two around and the grid vanishes under a yellow block.

Now step 4 pays off. Write `splitInput` alone, draw the parsed values as text, look, then write the next function. Six checked steps rarely need a long bug hunt, because you always know which function you touched last. That is the temperature chart workflow (Section 6.4.10), applied to your own plan.

6.13.3 The data: two splits

The starter code hands you the crossword as one long string with one animal per line, here shortened to its first three of 15 lines:

```
const crossword: string = `krebs,0,Schalentier
elefant,-1,Größtes Landtier
schlange,-4,Lautloser Jäger`;
```

Each line holds three fields separated by commas, the animal name, a start position, and a hint. (The animals and hints are German words.) Two levels of separators mean two levels of `split`, the tool from the Sokoban chapter (Section 6.12.7). Cut the text into lines, then each line into fields:

```
function splitInput(): void {
    const words: string[] = crossword.split('\n');
    for (let i: number = 0; i < words.length; i++) {
        const parts: string[] = words[i].split(',');
        animals.push(parts[0]);
        startPos.push(parseInt(parts[1]));
        hints.push(parts[2]);
    }
}
```

The three parallel arrays `animals`, `startPos`, and `hints` are globals, and after one call of `splitInput` they hold 15 entries each. `parseInt` turns the middle field into a number, because the start position is used for arithmetic, not for text.

6.13.4 Where does each letter go?

The whole picture hangs on one number. The solution word `klapperschlange` stands vertically in **column 4**, and every animal name contains one letter of it. The start position says how far a word is shifted so that its solution letter lands in that column. `krebs` starts with its `k` and is not shifted, so its start position is 0. In `schlange` the `a` sits at index 4, so that word starts four columns further left, at -4.

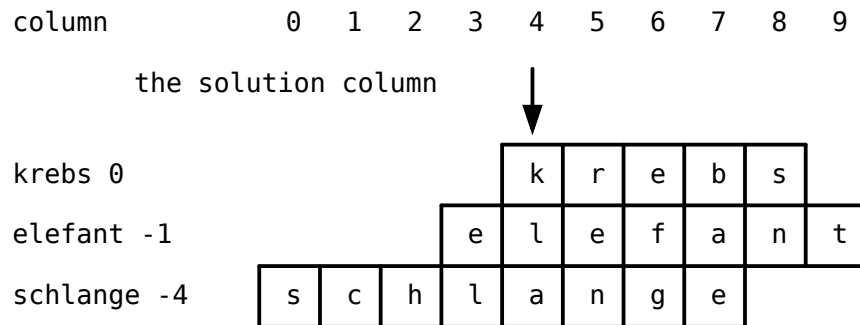


Figure 6.13.2: The first three animal names of the crossword. Every word is shifted by its start position, so that its letter of the solution word ends up in column 4.

That gives you the formula for the whole grid. Letter `j` of animal `i` sits in column `4 + startPos[i] + j` and in row `i`. Multiply the column by the cell width and the row by the cell height for the position on the canvas, a job `translate` can do for you:

```
for (let i: number = 0; i < animals.length; i++) {
  push();
  translate(0, i * letterHeight);
  for (let j: number = 0; j < animals[i].length; j++) {
    push();
    translate((4 + startPos[i] + j) * letterWidth, 0);
    // draw the cell border, and the letter if it is known
    pop();
  }
  pop();
}
```

Inside the inner `push`, the origin sits in the top left corner of one cell, so the cell is a plain `rect(0, 0, letterWidth, letterHeight)` and its letter goes to the same spot. A function that draws at the origin can draw anywhere, the rule from the first chapter of this part.

6.13.5 Does it contain that? includes

The guessing logic asks the same question over and over. Is this character inside that word, and is it already in the list of guessed characters? One tool answers both, `includes`, on strings and on arrays:

```
// 'krebs'.includes('e') is true, 'krebs'.includes('x') is false
// guessedCharacters.includes('e') is true once 'e' was guessed
```

For a string, `includes` asks whether the text contains that piece of text, and for an array whether one of the elements is that value. Both answer with a `boolean`, so both fit into an `if`. A guess is worth something when the character appears in at least one animal name and has not been guessed before:

```
function isValidGuess(key: string): boolean {
  for (let i: number = 0; i < animals.length; i++) {
    if (animals[i].includes(key) && !guessedCharacters.includes(key)) {
      return true;
    }
  }
}
```

```

    return false;
}

```

The loop stops at the first animal that contains the character, because `return true` leaves the function immediately, and only when no animal matched does the program reach the last line and answer `false`. Return early on a hit, return the opposite after the loop. That shape fits every “is there any element with ...” question. The win check is the same shape upside down, because one missing letter is enough to answer `false`:

```

function hasWon(): boolean {
  for (const letter of solution) {
    if (!guessedCharacters.includes(letter)) {
      return false;
    }
  }
  return true;
}

```

`for...of` walks arrays, and it walks strings too, one character per round. It beats a counting loop here, because the function needs every letter but never an index.

6.13.6 One key, one guess

The player types, so the program listens with `keyPressed` and reads the character out of `key`, the pair from the melting snowman (Section 6.8.7). The handler stays tiny, because both decisions already live in functions:

```

function keyPressed(): void {
  if (!hasWon()) {
    if (isValidGuess(key)) {
      guessedCharacters.push(key);
    } else {
      wrongGuesses++;
    }
  }
}

```

Read it as a sentence. If the game is still running and the guess is valid, remember the character, otherwise count a wrong guess. Both ways of guessing wrong, a character in no animal name and one that was already used, end in the `else` branch, because `isValidGuess` answers `false` for both. The typed character travels into `isValidGuess` as an argument, and the parameter there is called `key` as well, which is allowed and common. Notice that nothing here draws. The handler changes the data, and the next frame of `draw` paints the new truth.

6.13.7 Your exercise: Crossword

Levels 1 and 2 build the picture, levels 3 and 4 are the advanced ones that make it a game.

1. **Plan first, on paper.** Write your list of jobs in plain sentences, then a function name for each, with its parameters and its return type. Five minutes here save an hour later.
2. **Check the columns on paper.** Take `schlange` with start position -4 and write down the column of each of its eight letters using `4 + startPos[i] + j` (Section 6.13.4). Does the `a` land in column 4? A wrong idea of the start position puts every cell in the wrong place.
3. **Level 1: the empty grid.** Parse the data, then draw the yellow column, the empty cells at their computed positions, and the hints on the right. Run right after the parser and show `animals.length` on the canvas, so you know the arrays are full.
4. **Level 2: all letters visible.** Draw every letter into its cell, with no guessing logic yet. Now you can check the alignment, because the solution word must read downwards in the yellow column.
5. **Level 3 (advanced): guessing.** Start with an empty `guessedCharacters` array, show a letter only when the array contains it, and add `keyPressed` and `isValidGuess`.
6. **Level 4 (advanced): counting and winning.** Count the wrong guesses, show the number below the grid, and write `hasWon` so the message becomes a victory line.

💡 Exercise: Crossword

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

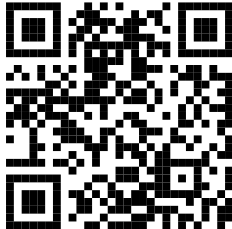
<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/crossword.yaml>

6.13.8 Check your understanding

When your crossword fills up letter by letter, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Design your own functions

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/evgrs823kr>

6.14 Drag and drop: Target Game



Figure 6.14.1: The end of a round of Target Game: when the 60 seconds are over, the canvas shows nothing but “Game Over!” and the final score.

You grab things with the mouse every day. You move a file into a folder, pull a photo into a chat, drag a window across the screen. That move is called **drag and drop**, and the Target Game builds it from scratch. A filled circle waits somewhere on the canvas, an empty target circle waits somewhere else, and you have 60 seconds to drag the filled circle into the target as often as you can.

6.14.1 AI tutor

Drag and drop breaks in ways that are hard to read from the canvas, because a circle that does not move looks exactly like a circle whose hit test is wrong. Tell the tutor which of the three mouse functions you are working on and what happens when you press, move, and release. This exercise is an exam task, so the tutor gives hints instead of finished code, which is what you want here anyway.

Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.14.2 Three moments and one variable

Dragging looks like one smooth movement, but for the program it is three separate moments, and p5 has a function for each of them. `mousePressed` runs the moment a button goes down, `mouseDragged` runs on every mouse movement while the button stays down, and `mouseReleased` runs when the button comes back up. None of them knows anything about the others, and that is the problem to solve.

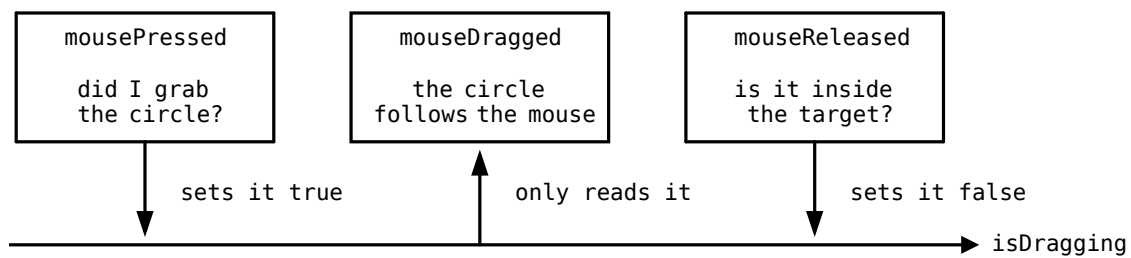


Figure 6.14.2: The three moments of a drag, and the variable that connects them. Press decides whether a drag starts, drag moves the circle only while the variable is true, and release ends the drag and evaluates the drop.

The connection is a single global variable, a `boolean` that remembers whether a drag is going on right now. The circle's own position lives in two more globals, `circleX` and `circleY`, and `grabbedTheCircle` is the hit test you build in the next section:

```
let isDragging: boolean = false;

function mousePressed(): void {
  if (grabbedTheCircle(mouseX, mouseY)) {
    isDragging = true;
  }
}

function mouseDragged(): void {
  if (isDragging) {
    circleX = mouseX;
    circleY = mouseY;
  }
}

function mouseReleased(): void {
  isDragging = false;
}
```

Read the three bodies as one sentence. Pressing on the circle turns the drag on, moving the mouse takes the circle along as long as the drag is on, and releasing turns it off again. Without the variable, `mouseDragged` would have no way of telling a real drag from a mouse that was pressed somewhere else on the canvas, and the circle would jump under the pointer whenever you swipe across the canvas.

State that lives between events is the idea worth keeping from this chapter. Every drag and drop you have ever used works this way, and so does the rest of the game. The score, the remaining time, and the positions of both circles are variables, and `draw` paints whatever they currently say.

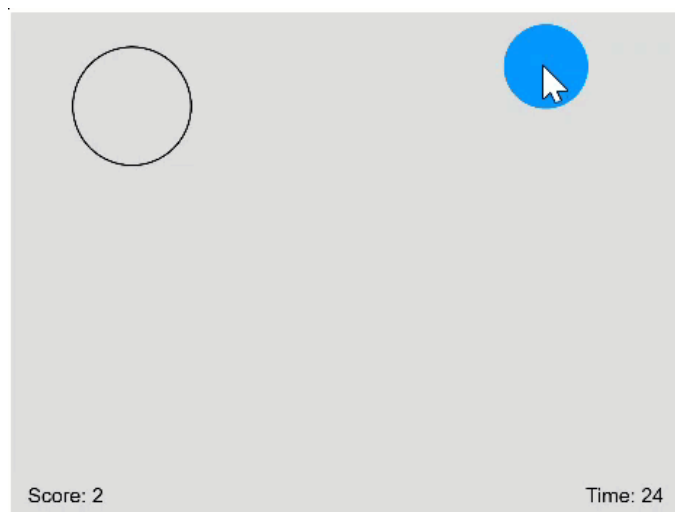


Figure 6.14.3: A round in progress: the filled circle hangs on the mouse pointer, the empty target waits at the top left, the score sits in the lower left corner, and the remaining time in the lower right.

6.14.3 Did I grab the circle?

`mousePressed` has to answer one question before anything moves. Is the mouse pointer inside the filled circle? That is the hit test from Bubble Buster (Section 6.7.8), the point in a circle. Measure the distance between the pointer and the center of the circle, and compare it to the radius. Anything shorter than the radius lies inside. The distance comes from the Pythagorean theorem, and it is worth its own function, because the game needs it three times:

```
function distance(x1: number, y1: number, x2: number, y2: number): number {
  return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2));
}
```

The two differences are the legs of a right triangle and the distance is its hypotenuse. With that tool in hand, the grab test is one comparison:

```
distance(mouseX, mouseY, circleX, circleY) <= CIRCLE_RADIUS
```

Compare this with the button hit test of the silo dashboard (Section 5.8.5), where four comparisons boxed the mouse into a rectangle. Round shapes are easier, because a circle asks only one question, and that is how far away the point is.

6.14.4 Fully inside: a circle in a circle

Dropping is a harder question than grabbing. The player scores only when the filled circle lies **completely** inside the target, so a circle that pokes over the outline is not enough. One line decides it:

```
distance(circleX, circleY, targetX, targetY) + CIRCLE_RADIUS <= TARGET_RADIUS
```

Think about the point of the filled circle that sits farthest away from the target's center. It lies one radius behind the center of the filled circle, so its distance from the target center is the distance between the two centers plus `CIRCLE_RADIUS`. When even that point stays within `TARGET_RADIUS`, every other point does too.

Two tests, two formulas. A *point* is inside a circle when its distance is at most the radius. A *circle* is inside a circle when the centers' distance plus the small radius is at most the big radius. Mixing them up gives a game that scores far too easily.

⚠ Radius in the task, diameter in the code

The task speaks about radii, 50 pixels for the filled circle and 70 for the target, but p5's `circle(x, y, d)` wants a **diameter** as its third number. Write `circle(circleX, circleY, CIRCLE_RADIUS * 2)`, and keep the constants as radii, because every distance comparison in the game needs radii. A circle that looks half the size of the picture is almost always this mistake.

6.14.5 Random positions that behave

Both circles start at random positions, and both have rules. A circle must be fully visible, so its center keeps one radius of distance from every edge, which `random` with a range takes care of:

```
let x: number = random(TARGET_RADIUS, width - TARGET_RADIUS);
let y: number = random(TARGET_RADIUS, height - TARGET_RADIUS);
```

The target also must not overlap the filled circle, at the start and after every point. Two circles stay apart when the distance between their centers is larger than the two radii together, so draw a random position and check it, and if it fails, draw another one:

```
while (distance(x, y, circleX, circleY) < CIRCLE_RADIUS + TARGET_RADIUS) {
  x = random(TARGET_RADIUS, width - TARGET_RADIUS);
  y = random(TARGET_RADIUS, height - TARGET_RADIUS);
}
```

Keep drawing until the position is good: that is the loop pattern for any random value with a condition attached. One warning about packing this into a function. A function returns one value, not two, so a position finder either returns a `number[]` with the x and the y in it, like the array-returning functions of the train station, or you write two smaller functions instead.

6.14.6 The clock: 60 seconds

The game ends after 60 seconds, and the remaining time counts down on the canvas. That is a job for `setInterval` from the callbacks chapter (Section 6.6.8), which calls a function of yours once per second:

```
const GAME_SECONDS: number = 60;
let remainingSeconds: number = GAME_SECONDS;
let timer: number;

function setup(): void {
  createCanvas(800, 600);
  timer = setInterval(countDown, 1000);
}

function countDown(): void {
  remainingSeconds--;
  if (remainingSeconds <= 0) {
    clearInterval(timer);
  }
}
```

`setInterval(countDown, 1000)` hands the function over **by name, without parentheses**, because p5 has to call it later. And notice what `countDown` does not do. It draws nothing. It changes one number, and `draw` decides every frame whether to paint the game or the “Game Over!” message, based on that number.

6.14.7 What the grading looks at

The Target Game is an exam task, so the exercise page lists quality criteria next to the game rules. All four are course rules you have been following for a while.

- **All drawing happens in `draw`.** Mouse handlers and the timer change data, `draw` paints it.
- **Repeated logic becomes its own function.** The distance, the inside test, and the search for a free position are the obvious ones.
- **Your functions take parameters instead of reading globals.** A distance on four numbers is useful everywhere; one that secretly reads `circleX` is useful once.
- **No magic numbers.** `CIRCLE_RADIUS`, `TARGET_RADIUS`, and `GAME_SECONDS` at the top of the file, not 50, 70, and 60 spread through the code.

6.14.8 Your exercise: Target Game

This exercise is an exam from a previous year, and the exercise page shows it twice, in English and in German, so you can read the tasks in whichever language is faster for you. Tasks 1 and 2 are the ones you must pass, tasks 3 and 4 make it a real game. There is no sample solution here, and none is needed, because a finished Target Game proves itself when you play it.

1. **Plan your functions first.** Write down the jobs before the code, with the method from the crossword chapter (Section 6.13.2). The task text even names three good candidates for you.
2. **Task 1: the draggable circle.** Draw the filled circle at a random position, fully visible, then add `mousePressed`, `mouseDragged`, and `mouseReleased` with the `isDragging` variable between them (Section 6.14.2).
3. **Task 2: the target.** An outlined circle with radius 70 at a random position that does not overlap the filled circle. Use `noFill()` so only the outline shows.
4. **Task 3: scoring.** On release, check whether the filled circle lies completely inside the target (Section 6.14.4). If so, count a point, move the target to a new free position, and show the score in the lower left corner.
5. **Task 4: the clock.** Count 60 seconds down with `setInterval`, show the remaining time in the lower right corner, and paint the game over message with the final score when the time runs out.
6. **Play it, then read your own code.** Does every drawing call sit in `draw`? Does any of your functions read a global it could have taken as a parameter? Fixing that now is exam training.

💡 Exercise: Target Game

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/target_game.yaml

6.14.9 Check your understanding

When your circle follows the mouse, lands in the target, and the clock runs out on you, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Drag and drop

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/a44iqu53n1>

6.15 Compound types: values that belong together

Three arrays for one soccer team, four for one screen full of bubbles. Since the Arrays part you have been keeping several rows in step by hand, and you have read the warnings that come with that job. The soccer field chapter said that swapping two entries in `lastNames` without swapping the first names sends player 1 onto the field as Lucas Rüdiger (Section 5.6.2). The bubbles chapter added a rule in a callout, push into all parallel arrays together, because one forgotten push moves every color onto the wrong circle from there on (Section 5.7.3).

Both warnings exist for the same reason. Nothing you have written so far can say “these values belong to one thing”, so the belonging lives in your discipline instead of in your code. This chapter closes that gap. Afterwards a position is one value, a bubble is one value, and that whole class of bug becomes impossible to write. You also pick up something bigger than a bug fix along the way, because you learn how to invent your own types.

6.15.1 AI tutor

Rewriting a working program to use compound types feels different from writing new code. You change one line, a dozen red squiggles appear at once, and the program is broken for a few minutes on purpose. That is normal, and the tutor is a good partner while it lasts. Paste one squiggle together with the function it sits in and ask what the type of that value should be now.



Your AI tutor

Hints and questions instead of finished programs, in English or German.

► [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.15.2 Why compound types

Take a position on the canvas. It is two numbers, an `x` and a `y`, and neither of them says anything on its own. An `x` of 10 fits every point on a vertical line through the canvas, and only the `y` next to it picks one of them. The two numbers describe one thing and are useful only together.

Parallel arrays store such a thing by taking it apart. The `x` values go into one row, the `y` values into another, and the only thing tying them back together is the index. Nothing in the program states that `circlesX[3]` and `circlesY[3]` describe the same bubble. You know it, the compiler does not, and a rule the compiler cannot check is a rule that breaks sooner or later.

A **compound type** is a type built out of several values, each with a name of its own. Values that describe one thing then sit in one place and travel together, so one variable is one position and one element is one bubble. The connection between the `x` and the `y` stops being a promise you keep and becomes part of the value.

6.15.3 An object in one box

TypeScript writes a compound value as an **object**, a set of named parts inside one pair of braces. Here is a position:

```
let position: { x: number, y: number } = { x: 10, y: 20 };
```

Read the line in two halves. Everything between the colon and the `=` is the type, and it lists which names the object has and what each of them holds. Everything after the `=` is the value, and it fills those names with 10 and 20. The named parts are called **properties**, so this position has a property `x` and a property `y`.

You reach a property with a dot. `position.x` is 10 and behaves like any other number, so `circle(position.x, position.y, 20)` draws a circle where the position says. A dot on the left of an `=` writes instead of reads, as in `position.y = 50`;

The console you met in the value-or-reference chapter (Section 5.9.2) is the magnifying glass for all of this. Open a fresh sketch, type this program, and write on paper what you expect on each of the three output lines before you press Run:

```
function setup(): void {
  let position: { x: number, y: number } = { x: 10, y: 20 };
  position.y = 50;

  console.log(position.x);
  console.log(position.y);
  console.log(position);
}
```

Nothing is drawn here and the canvas stays empty, which is fine. The first two lines show 10 and 50. The third one shows `[object Object]`, and knowing that now saves you confusion later. The output area turns every value it gets into text, and an object has no useful text form of its own. An array does have one, which is why logging a whole array printed 99,20,30 in the array experiment (Section 5.9.4). So log an object's properties by name, and the magnifying glass works again.

Vary it. Delete `y: 20` from the value, so the braces hold only `x: 10`, and watch the editor before you run anything. The red squiggle says that the property `y` is missing. An object either has everything its type promises or the program does not compile, and that check is exactly what parallel arrays could never give you. Put the `y` back.

6.15.4 Objects are references

This section teaches no new rule. Objects behave exactly like the arrays of the array experiment (Section 5.9.4), and running that experiment once more with an object is the fastest way to see it. Replace the body of `setup`, and write your prediction down first:

```
const a: { x: number, y: number } = { x: 10, y: 20 };
const b: { x: number, y: number } = a;
b.x = 99;

console.log(a.x);
console.log(b.x);
```

The console shows 99 and 99. The box of `b` never held the object, because an object is far too big for a variable box. The box holds an arrow that says where the object lives. `const b = a` copies what sits in `a`'s box, which is the arrow, so afterwards two boxes hold two arrows to one single object. `b.x = 99` follows an arrow and changes the one object that exists, and `a.x` reads that same property through the other arrow. Objects are **reference types**.

The `const` rule from the const puzzle (Section 5.9.5) carries over word for word. `const` nails down the box, and the box holds the arrow, so `const` means that this arrow never changes again. Writing `b.x = 99` is allowed, because it never touches the arrow. Adding this line is not:

```
b = { x: 0, y: 0 };
```

The squiggle appears before you run anything, and it complains about the variable, not about the object. Delete the line again.

A real copy of an object needs a brand new object, and here objects are easier than arrays. Copying an array took a loop (Section 5.9.6), because a loop is the only way to move an unknown number of elements. An object has a fixed set of properties whose names you know, so one line is enough:

```
const copy: { x: number, y: number } = { x: a.x, y: a.y };
```

The braces build a new object with an arrow of its own, and each property is filled with a number, a value type, so both values are copied across. Two objects, two arrows, no connection. Add `copy.x = 0`; below the line and log `a.x` and `copy.x` to watch the two numbers go their separate ways.

i Comparing objects compares the arrows

`===` between two object variables asks whether the two boxes hold the same arrow, just as it does for two array variables. Two separate objects with the same property names and the same values are still **false** under `===`, because they are two objects. Comparing the contents means comparing the properties one by one, for example with `a.x === copy.x && a.y === copy.y`.

6.15.5 Invent your own types: the `type` keyword

Writing `{ x: number, y: number }` once is fine. Writing it as a variable type, a parameter type, and a return type in the same program is three chances to write it slightly differently, and the better way carries a bigger idea with it.

Until this chapter, the set of types was fixed. `number`, `boolean`, and `string` came with the language, arrays of them came with the square brackets, and that was the whole list. The `type` keyword lets you add to the list. You give a shape a name, and from then on that name is a type like any other:

```
type Position = { x: number, y: number };
```

The line belongs at the top of your file, next to the global variables and outside every function. Type names start with a capital letter, which is how a reader tells `Position` the type from `position` the variable.

Your new type works in all three places a type can appear. In a variable declaration:

```
let target: Position = { x: 400, y: 300 };
```

As a parameter type, so a function takes a whole position instead of two loose numbers:

```
function drawMarker(p: Position): void {  
    circle(p.x, p.y, 20);  
}
```

And as a return type, which closes a loose end from the target game. That chapter had to warn you that a function returns one value and not two, so a position finder either hands back a `number[]` or gets split into two functions (Section 6.14.5). With `Position` in the language, a random position is one value:

```
function randomPosition(): Position {  
    return { x: random(width), y: random(height) };  
}
```

The `return` builds a fresh object right there and hands it back, so the caller keeps a whole position in one variable with `const p: Position = randomPosition();`. The rule that a function returns one value has not changed at all. You widened what one value can be.

6.15.6 Arrays of compound types

A bubble in Bubble Buster is a position and a size, so it takes three numbers, and a type of its own says so:

```
type Bubble = { x: number, y: number, diameter: number };
```

Once `Bubble` exists, `Bubble[]` is a type too, a row of bubbles, and the whole game state becomes one array:

```
const bubbles: Bubble[] = [];
```

index	0	1
bubbles	x: 120 y: 80 diameter: 40	x: 260 y: 140 diameter: 25

Figure 6.15.1: One array where every element is one complete bubble, with a named compartment for `x`, one for `y`, and one for the diameter, so index `i` points at a bubble instead of tying three separate rows together.

Every operation the game needs gets shorter. Creating a bubble is one push of one object instead of three pushes into three arrays:

```
bubbles.push({
  x: random(width),
  y: random(height),
  diameter: random(10, 50)
});
```

Removing a bubble is one `splice` instead of three, because `bubbles.splice(i, 1);` takes the whole bubble out. Drawing walks the single array, with `bubbles[i]` as the whole bubble and a dot for the part you want:

```
for (let i: number = 0; i < bubbles.length; i++) {
  circle(bubbles[i].x, bubbles[i].y, bubbles[i].diameter);
}
```

Read `bubbles[i].x` from left to right, element `i` of `bubbles`, then its `x`. And `bubbles.length` counts bubbles now, not numbers. No second array can fall out of step with the first, because there is no second array.

6.15.7 Your exercise: Bubble Buster, rebuilt

This chapter brings no new exercise. You reopen Bubble Buster (Section 6.7.8) and rebuild its engine on one array of objects, starting from your own working solution. If your solution never got finished, paste the sample solution of that exercise into the playground and work on that instead.

What you are doing has a name. A **refactoring** is a rewrite that changes how a program is built without changing what it does. Work through the tasks in order and run the game after each one.

1. **Task 1: the type.** Add `type Bubble` at the top of the file, with the three properties `x`, `y`, and `diameter`, all of them `number`.
2. **Task 2: one array.** Replace `circlesX`, `circlesY`, and `circlesDiameter` with `const bubbles: Bubble[] = [];`. Every line that used the old arrays now shows a squiggle, so fix them one at a time, including the drawing loop in `draw` and the game over check, which becomes `bubbles.length >= 10`.
3. **Task 3: one push.** Rewrite `addRandomCircle` so it pushes a single object with all three properties filled in, instead of pushing three numbers into three separate arrays.
4. **Task 4: one splice.** In `mouseClicked`, the backwards loop stays exactly as it is, because a loop that removes elements from the array it walks always has to go backwards (Section 5.7.7). Only the removal changes, from three `splice` calls to one.
5. **Task 5: a bubble instead of an index.** Change `isInside` so its third parameter is a `Bubble` instead of an index, and read the center and the diameter from that parameter. The Bubble Buster chapter had to explain that index parameter and why the bubble data stayed behind in the arrays (Section 6.7.4), and with a compound type the explanation disappears. The call turns into `isInside(mouseX, mouseY, bubbles[i])`.

You are done when the game plays exactly as before. Same bubbles, same popping, same score, same game over screen, and far fewer ways for the program to break.

Exercise: Bubbles With Functions

[▶ Open in the playground](#)



Scan the code, or type this address into your browser:

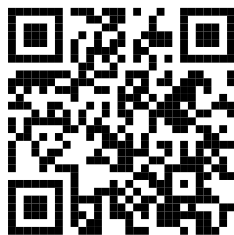
https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/bubbles_functions.yaml

6.15.8 Check your understanding

When your Bubble Buster runs on one array of bubbles and you can say out loud what `type Bubble` gave the program, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Compound types

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/zs3lx6py0a>

6.16 Practice: Ratio Visualizer

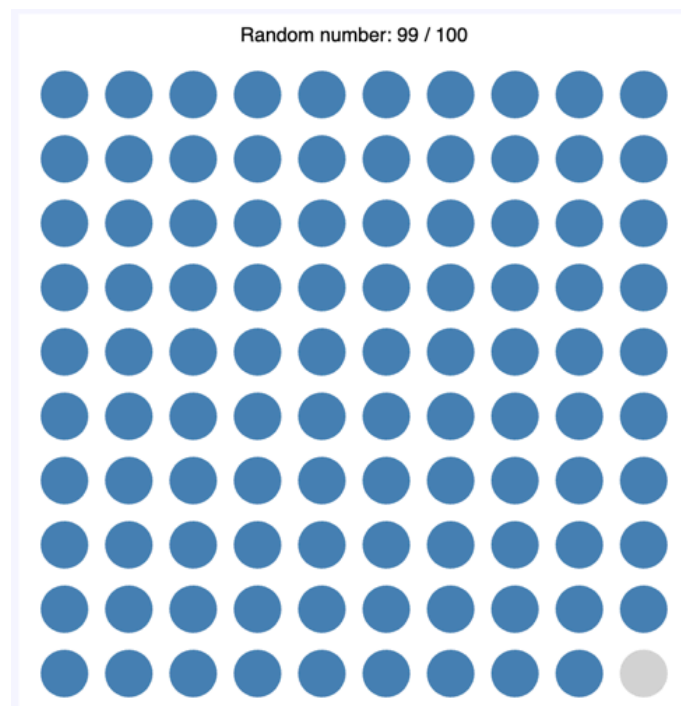


Figure 6.16.1: The Ratio Visualizer: 100 circles in a 10 by 10 grid show a random number as a share of 100. Here the number is 99, so exactly one circle stays gray.

A progress bar shows a share as a single filled strip. The Ratio Visualizer shows the same idea in two dimensions. A random number between 0 and 100 decides how many of 100 circles are colored, and you build the picture out of two small functions that you design yourself. Nothing in this chapter is new, which is exactly the point. This is practice, and the only question is whether the tools of this part are already yours.

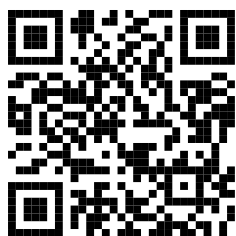
6.16.1 AI tutor

You have everything you need for this chapter, so work on your own first and give yourself time to be stuck. If a circle ends up in the wrong place or the wrong ones turn blue, describe to the tutor what your two functions do and what the canvas shows, and ask for a hint rather than for code.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.16.2 A small team of two functions

You now design small teams of functions, and the Ratio Visualizer needs a team of two. The jobs split along a line you have seen in several chapters of this part, because one function decides and the other draws.

- The **deciding** function answers one yes-or-no question. Given a circle's place in the grid and the random number, should this circle be colored? A question like that has a **boolean** return type, and the function draws nothing at all.

- The **drawing** function paints one single circle at the current origin, in one style when the circle is colored and in another style when it is not. It decides nothing. Somebody hands it the answer.

Two jobs, two functions. Keeping them apart pays off immediately, because you can look at a wrong picture and know where to search. If the wrong circles are blue, the deciding function is wrong. If the circles look wrong or sit in the wrong place, the drawing function or the loop around it is wrong.

The deciding function is where the real thinking happens. The circles are counted left to right and top to bottom, so the circle in the top left corner is number 0, the one next to it is number 1, and the last circle of the first row is number 9. The first circle of the second row continues at number 10. Work out on paper how a row number and a column number turn into that single counting number, then compare it with the random number. Two lines of code, one good idea.

6.16.3 You choose the names and parameters

The exercise does not tell you what to call your functions or which parameters they take. That freedom is the task. You met the same kind of open design in the Crossword exercise (Section 6.13.7), and the method that worked there works here as well. Name the job first, in plain words, then turn the name into a signature.

Ask two questions about each function. What does it need to know, and what does it hand back? Everything a function needs to know becomes a parameter, and what it hands back becomes the return type. A function that decides whether a circle is colored needs to know where the circle sits and which number it is compared against, so it takes numbers and returns a **boolean**. A function that draws one circle needs to know only whether that circle is colored, so it takes a **boolean** and returns nothing, which is : **void**.

Write both signatures on paper before you type them. If a signature needs five parameters, the function is probably doing two jobs and wants to be split.

6.16.4 Walking a grid with translate

Drawing 100 circles does not mean computing 100 pairs of coordinates. The drawing function paints at the origin, at (0, 0), and **translate** moves the origin between calls. A function that draws at the origin can draw anywhere, and a grid is the clearest example of that.

One row is a loop that draws and then steps to the right. The two sketches below use made-up names, **drawTile** for your drawing function and **CELL** for the distance between two neighbors:

```
for (let col: number = 0; col < 10; col++) {
  drawTile();
  translate(CELL, 0);
}
```

After that loop, the origin has wandered far to the right, which is the wrong starting point for the next row. **push** and **pop** solve it. **push** remembers where the origin is, and **pop** puts it back, so the pattern for a full grid is to push before a row, draw the row, pop after it, and then step down by one cell height:

```
push();
// ... the row loop from above ...
pop();
translate(0, CELL);
```

The outer loop repeats those four steps once per row. No coordinate arithmetic appears inside the loops, and every circle lands where it belongs.

6.16.5 Your exercise: Ratio Visualizer

The starter code gives you the canvas, the layout constants for the grid, the colors, and the random number. Your job is the two functions and the drawing logic in **setup**.

1. **Plan on paper.** Write down the two function signatures with their names, their parameters, and their return types. Then work out the counting rule for the deciding function with a small example, for instance which circles are colored when the random number is 23.

2. **Write the deciding function first.** It is the shorter one, and you can check it in your head before any circle exists on the canvas.
3. **Write the drawing function.** It paints one circle at the origin and uses the given color constants. Wrap its style changes in `push` and `pop` so they do not leak into the rest of your program.
4. **Fill in setup.** Show the random number as a title near the top, move the origin to the center of the first circle with `translate`, and then draw all 100 circles with the nested loop pattern from the section above.
5. **Run it several times.** The random number changes on every run, so check the border between the blue circles and the gray ones against the title. With the number 23 you should see two full blue rows and three blue circles in the third row. An off-by-one mistake shows up exactly there.

💡 Exercise: Ratio Visualizer

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/ratio-visualizer.yaml>

6.16.6 Practice on your own: Tally Visualizer

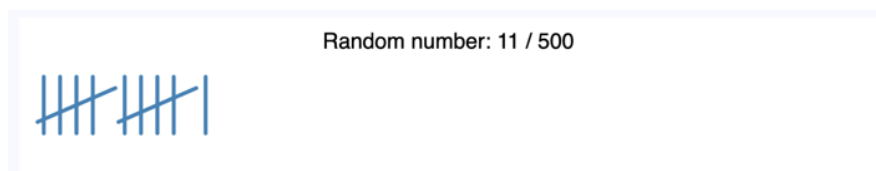


Figure 6.16.2: The Tally Visualizer: a random number out of 500, drawn as tally groups of five strokes. Here the number is 11, so two full groups and one single stroke appear.

The Tally Visualizer is the same idea in a different picture, and this one you do entirely on your own. A tally chart counts in groups of five, four vertical strokes with a diagonal stroke across them, and the grid of 10 rows with 10 groups stands for 500. The two functions split the work exactly like in the Ratio Visualizer, only the deciding function now returns a number instead of a boolean, because a group can be empty, full, or anything in between. The exercise text comes in English and in German, and it offers a smaller minimum version if the full grid feels too big at first. It was a real exam in an earlier year of this course.

There is no walkthrough for this one, and there will not be one. You have everything you need. Read the task, plan your two functions on paper, and build it.

💡 Exercise: Tally Visualizer

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/tally-visualizer.yaml>

6.17 Training: functions and arrays

This chapter has no picture to draw. It is a training session, five small functions that take something in and hand something back, and a program that tells you straight away whether each one is right. Musicians practice scales, and this is the programming version of that. The functions are short on purpose, so all your attention goes to the two skills this part is built on, writing a clean signature and getting the logic inside it correct.

6.17.1 AI tutor

The exercise in this chapter tells you exactly which function is failing, so bring that information along when you ask. Name the function, say what the check expected and what your version returned, and the tutor can point you at the line that is wrong without handing you the solution.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfghmw3hw>

6.17.2 The program checks itself

The training exercise comes with a `setup` that is already written, and that `setup` is a test bench. It calls each of your functions with inputs whose correct answers are known, compares what came back against those answers, and paints one line per check on the canvas. A check that passes shows a green `✓`, a check that fails shows a red `✗` next to the value your function actually returned.

Checks written as code and run by the program itself are called **automated tests**, and professional programmers write them for exactly this reason, because a machine repeats 100 checks in a second without ever getting bored or sloppy.

Notice that the checks in `setup` call functions that are written further down in the same file. That is allowed. Inside one file, the order of your function definitions does not matter, so you can put the test bench first and the work below it.

i Given code: read it, don't copy it

The `setup` of this exercise uses two tools from later courses. One is `?:`, a compact way of choosing between two values in a single expression, which is how the `✓` and `✗` get picked. The other is `indexOf`, a ready-made search in an array, which one of the checks uses to look through your result. Read over them, get a rough sense of what they do, and leave them alone. You write only the functions below `setup`, and none of them needs either tool.

6.17.3 Work one function at a time

Every function in the file starts with a placeholder line that returns a fixed, wrong value, so the whole program runs from the first second. That is a good starting point, because the canvas stays readable no matter how much is still missing.

Work from top to bottom. Replace the placeholder in the first function, run the program, and look at the two lines that belong to it. Green means done, so move on. Red means you found a bug now, while your head is still inside that one function and the other four have not distracted you. Never write three functions and run once.

Read the doc comment above each function before you touch it. It is the specification, and it states what each

parameter holds, what the return value must be, and what should happen in the awkward cases, for example when the input string is empty. Two of the checks feed in an empty string on purpose.

6.17.4 Good to know: those comments are JSDoc

The comments above the five functions are not ordinary comments. A comment that starts with `/**` is a **doc comment**, written in a format called **JSDoc**, and editors read it. You saw one in the Return Values chapter (Section 6.5.5), and the ones in this exercise follow the same three-part pattern.

```
/**
 * Find the index of a number in an array
 * @param array - An array of numbers (e.g. [1, 2, 30, 4, 5])
 * @param number - The number to find the index of (e.g. 30)
 * @returns The index of the number, -1 if the number is not found
 */
```

A summary line first, then one `@param` per parameter using the parameter's exact name, then `@returns` for the answer. A function that ends in `: void` gets no `@returns`, because it hands nothing back.

Two things make JSDoc worth the typing. It holds the part of the truth that types cannot say, such as the format of the string or what comes back when nothing is found, and the editor shows it to you in a small box the moment you type the function's name and the opening parenthesis. Those boxes you have been reading all course for **random** and **circle** come from doc comments too. Write them for every function that might not be obvious to a reader.

6.17.5 What the five functions do

The five functions come in a deliberate order, from a plain loop to a function that builds an array.

- **getSum** takes a string of numbers separated by commas, such as `"1,2,30,4,5"`, and returns their sum. Turning that text into numbers is the parsing job from the Arrays part, with the collecting loop from Section 5.5.3 and the leftover number that has to be handled after the loop ends (Section 5.5.4). Since the Sokoban chapter you also own `split`, so either road is open, but whatever you pick has to survive the empty string, which must give 0.
- **getIndexOf** takes the same kind of string plus a number, and returns the position **in the string** where that number starts. For `"1,2,30,4,5"` and the number 30 the answer is 4, because the characters of `"30"` begin at index 4 of the text. When the number does not occur at all, the answer is -1.
- **findIndexInArray** takes an array of numbers and a number, and returns the position **in the array**, so for `[1, 2, 30, 4, 5]` and the number 30 the answer is 2. Compare that with **getIndexOf** above, same question, different counting, and reading the specification carefully is the only way to see it. Write the search yourself with a loop, as you did in the train station chapter.
- **decodeCard** takes a two-character card code and returns a readable description, so `"S5"` becomes `"5 of Spades"` and `"HJ"` becomes `"Jack of Hearts"`. The first character is the suit (the doc comment spells it *suite*), the second is the rank, and the comment lists every letter and what it stands for. A pair of `switch` statements fits this job better than a stack of `ifs`.
- **findAllEvenNumbers** takes a string of numbers separated by commas and returns an array holding the even ones. It combines everything before it, because the walk through the string is the one from **getSum** and the result is built with `push`. When no even number shows up, the function returns an empty array, not nothing.

6.17.6 Your exercise: functions and arrays training

1. **Open the exercise and run it once** before you change anything. Most of the 11 checks show a red , and a few show a green by accident, because a placeholder happens to return the value that one check expects. A check that a placeholder passes proves nothing, so do not count those yet.
2. **Implement `getSum`**, run, and look at its two lines. Keep going only when both are green.
3. **Implement `getIndexOf`, then `findIndexInArray`**, running after each one. If the two of them return the same number for 30, read their doc comments again.

4. **Implement `decodeCard`**, run, and check the exact wording of the result. "5 of Spades" passes, "5 of spades" does not, because the comparison in the check is exact.
5. **Implement `findAllEvenNumbers`**, run, and confirm the last three checks. All 11 green means you are done.
6. **Format your code** and read your five signatures once more. Every parameter typed, every return type written out.

💡 Exercise: Functions and Arrays Training

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/function-array-training.yaml>

6.18 Test your knowledge: Poker Cards



Figure 6.18.1: Five poker cards on the canvas, and below them the verdict the program worked out from the hand. Here no combination matched, so the program names the highest card, the 9 of diamonds.

This chapter is an exam, and not a pretend one. Students in earlier years of this course sat down with exactly this task and were graded on it. You get it as practice, with the same text, the same starter code, and the same requirements. Your job is a program that looks at five playing cards and works out what the hand is worth, built from a set of small functions that hand their results to each other.

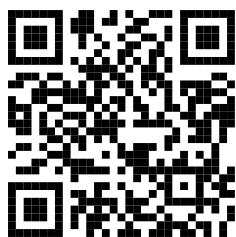
6.18.1 AI tutor

In a real exam you sit alone, so treat the tutor as a last resort here and get as far as you can by yourself first. When you do ask, ask about one function, describe the rule you are trying to implement and what your version returns, and use the answer as a hint to keep going.

Your AI tutor

Hints and questions instead of finished programs, in English or German.

► [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/xjvfgmw3hw>

6.18.2 How to take an exam like this

An exam of this size is won before the first line of code. Six habits decide how far you get.

1. **Read everything first.** Read the whole task and every doc comment in the starter code before you write anything. Ten minutes of reading are cheap, and finding out after 40 minutes that a function returns something different from what you assumed is expensive.
2. **Implement in the order the task gives.** The order is not decoration. Each of the first four functions uses the one before it, so a working `getCardValue` makes `getCardDescription` short, and both together make `getHighestCard` almost easy.
3. **Minimum requirements before anything advanced.** Finish all four functions that the task lists as the minimum before you even look at the advanced ones. Four finished functions beat eight half-finished ones in every grading scheme.

4. **Run after every single function.** The program runs from the start, and after each function you should see one more thing on the canvas behave correctly. That is your safety net, and it also tells you how much time one function really costs you.
5. **Quality over quantity.** The exam text says so explicitly, and it means what it says. A correct, readable, formatted function is worth more than two rushed ones that almost work.
6. **Format your code before you hand it in.** That rule has been in force since part 1 of this course, and in an exam it is free points. Readable code is easier to grade and easier to defend.

6.18.3 Reading the card representation

Understanding the specification is part of this exam, and the card representation is the piece worth reading twice.

The deck holds 40 cards. There are four suits (the exam text spells the word *suite*), hearts, diamonds, clubs, and spades, written as the letters H, D, C, and S. There are ten values, 1 to 9 and 10. No jacks, queens, kings, or aces appear, and red and black play no role in these simplified rules.

One card is a string of exactly two characters. The first character is the value, the second is the suit.

Table 6.18.1: How a two-character card code is read.

card	meaning
"5H"	5 of hearts
"3D"	3 of diamonds
"0S"	10 of spades

The last row of the table holds the trap. The character '0' does not mean zero, it means 10, because a value has to fit into a single character. Every function that works with values has to handle it, and forgetting it is the single most common way to lose points in this exam. Watch the order of the two characters as well, because a card code puts the value first and the suit second.

6.18.4 Given code you do not write

The exercise comes with a second file, `helpers.ts`. It draws the cards, deals a random hand, and shuffles the deck, so that your program has something to work with from the first minute. You never edit it.

The file `index.ts` also contains a `draw` function that is already finished and marked as untouchable. It asks your functions, in order from the strongest hand to the weakest, what this hand is, and it prints the verdict under the cards. That is why every function you have to write starts with a placeholder line that returns a wrong value. Without those lines, `draw` could not run at all. Delete a placeholder when you implement its function.

i The helpers file is given code

`helpers.ts` uses several tools from later courses, among them sorting with a comparison function, arrow functions, and `join`. Open the file and read it if you are curious about how the cards get onto the canvas, because reading code you did not write is a normal part of the job. Do not copy those constructs into your own functions. Everything the exam asks for can be built with what this course has taught you.

6.18.5 Your exam: Poker Cards

Work through the exam in the order below, with the exam habits from Section 6.18.2 next to you.

1. **Read the task and all doc comments**, then write down in one sentence per function what it takes and what it gives back.
2. `splitCardsString` turns the string "5H,6H,7H,8H,9H" into an array of five card strings. Since the Sokoban chapter you have `split` for separators like this one.
3. `getCardValue` turns one card into its number, and this is where '0' becomes 10.
4. `getCardDescription` turns one card into readable text such as "5 of hearts". Call `getCardValue` instead of repeating its logic.

5. `getHighestCard` walks the hand and returns the description of the card with the largest value. With those four functions done, the minimum requirements are complete, and the canvas already names the highest card of every hand you deal.
6. **Then the advanced functions**, as many as you can finish properly. `getCounts`, `isFlush`, and `isStraight` are the ones the exam text singles out, and the checks for pairs, three of a kind, and a full house build on `getCounts`.
7. **Test with hands you choose.** The starter code holds named example hands for every combination, from a royal flush down to a high card, and one marked line decides which of them the program uses. Put a hand there whose answer you know, and once it is correct, switch that line back to the random hand and click the canvas to deal new ones.
8. **Format your code and read it once from top to bottom** before you call yourself finished.

The exam is written in English in the playground. The original German version of the same instructions is kept next to the exercise, in [the German exam text on GitHub](#), so you can read the requirements in whichever language makes them clearer.

💡 Exercise: Poker Cards

▶ [Open in the playground](#)



Scan the code, or type this address into your browser:

<https://ts-web-playground-server-bfdfbjcthubcardq.westeurope-01.azurewebsites.net/playground?exerciseUrl=https://raw.githubusercontent.com/rstropek/ts-web-playground/main/exercises/0070-Functions/poker-cards.yaml>

6.18.6 Where this leaves you

With functions in your hands, you now hold the whole toolbox that a programmer picks up in a first year, sequence, data, decisions, loops, arrays, and functions. Everything from here on is built out of those six ideas, in bigger and bigger arrangements. What changes next is not the language but the workshop: in the next part you'll leave the web playground and set up the tools professional developers use every day, on your own computer. Take a moment to notice how far you have come, and then turn the page.

Part 7

Visual Studio Code

7.1 A new workshop: Visual Studio Code, Node.js, and npm

Back in the first part, this book made you a promise (Section 1.2.2). Professional developers write their code in tools like Visual Studio Code, and this course started you in the web playground because those tools need a lot of setup first. Six parts later, you've earned that setup. When you're done with this chapter, Visual Studio Code and Node.js will be installed on your own computer and checked, and you'll know what each of these tools is for.

7.1.1 AI tutor

This part of the book has its own AI tutor, and it knows the tools you're about to install. The exercise below sends you to a search engine or to a general AI on the internet on purpose, because finding out how to install software is exactly the skill it trains. The tutor of this part helps you make sense of what you find out there and explains the pieces that belong to this course.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/v06yhwe08>

7.1.2 What Visual Studio Code is

Visual Studio Code, or VS Code for short, is a code editor made by Microsoft. It's free, and among professional developers it's the most widely used editor there is. VS Code runs on your own computer and edits the files that lie on your own hard drive, which is the big difference from the web playground.

A lot of it will feel familiar right away. VS Code puts red squiggly lines under mistakes while you type, and when you hover your mouse over a command, a small tooltip shows its documentation. The playground editor behaves the same way, so you already know how to work with those two helpers.

One thing is new. VS Code knows nothing about exercises, tasks, or the description of what you're supposed to build. It's a general tool that edits any code for any purpose, from a school program to the software of a bank. Your programs are yours now, and nobody prepared them for you.

7.1.3 Why you need Node.js

A browser can run programs written in JavaScript, the close relative of TypeScript, but only inside a web page. Developer tools are programs too, and many of them are written in JavaScript as well. Two of them are waiting for you in the next chapters, a small web server that shows your program in the browser and the TypeScript compiler that translates your TypeScript code into JavaScript. Both have to run directly on your computer, outside any web page, and a browser can't do that.

Node.js is the program that makes it possible. It runs JavaScript directly on your computer, with no browser around it. You won't write Node.js programs yourself in this course. You install Node.js because the tools of the next chapters need it to run at all.

7.1.4 What npm is

npm is short for Node Package Manager. You don't install it separately, because it comes along automatically when you install Node.js.

The Welcome chapter called p5.js a **library**, a collection of ready-made code that you use in your own programs. In the web playground, p5.js was simply there, prepared for you. On your own computer nothing is prepared, so something has to fetch that ready-made code from the internet. npm is that something.

npm downloads **packages** from a huge public catalog on the internet. A package is a library or a tool, bundled up so that a computer can download and install it. The p5.js library arrives on your computer as a package, and so do the developer tools of the next chapters. How you actually run npm comes later, once you have a project that needs it.

7.1.5 The terminal

A terminal is a window in which you control your computer by typing commands instead of clicking. You type one line, press Enter, and the computer answers in text. That sounds old-fashioned, but professional developers use a terminal every day, and npm is a tool you give commands to exactly this way.

VS Code has a terminal built into it, called the **integrated terminal**, so you don't need a separate program for it. In this chapter you need the terminal for one single job, the two commands that tell you whether your installation worked.

7.1.6 Research like a professional

This book gives you no click-by-click installation instructions, and that's a deliberate decision. Every operating system installs software differently, and download pages, buttons, and version numbers change every few months, so printed steps get old fast. Setting up software is also a skill you'll need for your whole working life. What the book gives you instead is the goal, the concepts behind the tools, the official sources, and a way to check that everything worked.

Find the steps for your computer with a search engine, with a general AI like ChatGPT or Google Gemini, or with both. When you ask an AI, say which operating system you use (for example Windows 11, macOS, or Ubuntu Linux) and what you already tried. Without that, the answer is a guess about a computer the AI has never seen.

Download the installers only from the official websites, never from a random download portal. You need the [official Visual Studio Code download page](#) and the [official Node.js download page](#). The Node.js page offers more than one version. Choose the **LTS** version, short for long-term support, which is the one the page marks as recommended.

7.1.7 Your exercise: set up your workshop

Your task is the research itself, and there's no playground exercise for it. Work through these three steps on the computer you'll use for the rest of the course:

1. **Install VS Code.** Find out how Visual Studio Code is installed on your operating system, and install it.
2. **Install Node.js.** Do the same for Node.js, and take the LTS version.
3. **Check that both work.** Open VS Code, find out how to open its integrated terminal, and type `node --version` there. Press Enter, then type `npm --version` and press Enter again. Each command answers with a version number. A `v` followed by numbers means the installation worked.

If a command answers with an error message instead, don't worry. The most common trap is a terminal that was already open before Node.js was installed, because such a terminal never noticed the new program. Close VS Code completely, open it again, and try both commands once more.

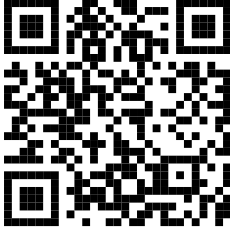
If the error is still there, treat it as research material. Copy the exact message into the search engine or into the AI, together with your operating system, and read what comes back. Reading error messages is what the rest of this part builds on.

7.1.8 Check your understanding

When your workshop is set up, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: A new workshop: Visual Studio Code, Node.js, and npm

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/iojypytr5i>

7.2 Getting the starter code

Your workshop has tools now, and it still has no workbench. This chapter builds one. When you're finished, a tidy folder for the whole course sits on your computer, the course's starter project sits inside it, and a first copy of that project is ready to become your next exercise. Nothing about programming changes here, only the place where your programs live.

7.2.1 AI tutor

A general AI on the internet knows GitHub, ZIP files, and git very well, but it has never seen this course's starter project and it doesn't know the folder rules of this chapter. The AI tutor of this part knows both. Downloading and unpacking is research territory again, so look up the steps for your computer with a search engine or a general AI, and bring your questions to the tutor when you're unsure whether what you found fits this course.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/v06yhwe08>

7.2.2 Why you start from a starter project

In the web playground, every exercise was prepared for you. You opened a link, and the editor already held a working program with the right structure around it. On your own computer nothing is prepared, and a project needs several files before you can write a single line of code. It needs a web page for the browser to open, configuration for TypeScript and the developer tools, and a list of the packages the project needs.

Nobody types those files by hand for every new idea. Professional developers keep a prepared project, a **template**, and start every new project as a copy of it. This course has a template too, and it's called the **starter code**.

The starter code is published on GitHub, a website where programmers store their code projects and share them with other people. You have seen links to it in this book before. One project on GitHub is called a **repository**, and the course's template lives in the [vite-p5-starter repository](#).

7.2.3 What's in the box

The starter code is a small folder, and here is the whole project seen from above:

```
vite-p5-starter/  
  index.html  
  package.json  
  tsconfig.json  
  vite.config.ts  
  biome.json  
  readme.md  
  public/  
  src/  
    index.ts
```

- `src/index.ts` is the file where your program will live. Everything you write in the coming chapters goes in there.
- `index.html` is the small web page that the browser opens to show your canvas.
- `package.json` is the project's ID card. It holds the project's name and the list of packages the project needs, and `p5.js` is on that list.
- `tsconfig.json`, `vite.config.ts`, and `biome.json` are configuration files for TypeScript and the developer tools. Leave them alone for now.
- `readme.md` is the starter's own description, written by your teachers. Reading it is never wrong.
- `public/` is the folder for pictures and other files your program may load later. In a fresh starter it's still empty.

One thing is missing on purpose. The packages themselves are not in the box, and nowhere in the starter is there a folder with the `p5.js` code in it. `package.json` only writes down *which* packages are needed. Fetching them is a job for `npm`, and that happens in the next chapter.

7.2.4 Two ways to get the starter code

There are two ways to get the starter code onto your computer, and both are fine for this course.

The straightforward way is a download. On the repository page, the green **Code** button offers **Download ZIP**. You get one compressed file that holds the whole project, and you unpack it into a folder of your choice. No extra software needed.

The professional way is **git**, a version-control program that developers use to keep track of every change in a project. `git` has to be installed on your computer first, and two of its commands matter here. The command `git clone` copies a repository from GitHub to your computer, and `git pull` refreshes that copy later, so improvements your teachers make to the starter arrive on your machine with one command. Installing `git` and typing those commands is research territory, exactly like the installations in the previous chapter, so look it up with a search engine or a general AI.

Choose honestly. The ZIP download is completely fine for the whole course. `git` is the tool you'll meet in every programming job. It's worth the research if you're curious, and it pays off every time the starter gets an update. If you took the ZIP, an update simply means downloading the ZIP again.

7.2.5 One folder per exercise

From now on you decide where your files live, so this course gives you one rule for it. Follow it from the first exercise on, and your work stays findable and runnable for the rest of the year.

! The folder rule of this course

1. **One folder for the course.** Create a single folder, for example `creative-coding`, in a place you'll find again next week. Not in the Downloads folder, and not loose on your desktop among everything else.
2. **The starter stays clean.** Put the starter code (the unpacked ZIP or the cloned repository) inside it as `creative-coding/vite-p5-starter`. This is your master copy, and you never write program code in it. Before a new exercise, refresh it with `git pull`, or by downloading the ZIP again.
3. **One copy per exercise.** For every exercise, copy the whole starter folder and rename the copy after the exercise, for example `creative-coding/shapes`. All your coding happens in the copy.

! Choose the place for your course folder carefully

Two kinds of places cause trouble that has nothing to do with your code:

- **Folders that sync to a cloud.** Don't put your course folder into OneDrive, SharePoint, Dropbox, or any other folder that syncs your files to the internet. In the next chapter, `npm` will create tens of thousands of small files in every exercise folder, and the sync tool tries to upload every single one of them. That makes your computer slow, and it can block files at the exact moment `npm` needs them, which produces error messages that look like your fault but aren't. Watch out: on many school computers the Desktop and the Documents folder sync to OneDrive automatically, so those two are exactly the places to avoid.

- **Long or unusual paths (Windows).** Windows limits how long the path to a file may be, and npm builds folders inside folders many levels deep. If your course folder already sits deep in the folder tree, that limit is suddenly within reach. Special characters in folder names, like spaces or the letters ä, ö, and ü, confuse some developer tools too.

The safe choice is short and plain. On Windows, a folder like `C:\Code` is ideal, and your course folder becomes `C:\Code\creative-coding`. On macOS or Linux, a simple folder directly in your home directory does the same job.

After your first few exercises, your course folder looks like this:

```
creative-coding/  
  vite-p5-starter/  the clean master copy, never edited  
  shapes/          exercise 1, a copy of the starter  
  smiley/          exercise 2, a copy of the starter  
  dice/            exercise 3, a copy of the starter
```

Three good things follow from the folder rule. Every finished exercise stays as you left it and still runs weeks later. A broken experiment can only damage the one folder you're experimenting in. And when your teachers improve the starter code, your next copy picks up the improvement automatically, while your older exercises keep working with the version they were built from.

7.2.6 Your exercise: build your folder structure

There's no playground exercise this time. You build the folder structure on the computer you'll use for the rest of the course:

1. **Create your course folder.** Make a folder named `creative-coding` in a short, simple place, for example inside `C:\Code` on Windows. Keep it out of OneDrive, SharePoint, and Dropbox.
2. **Get the starter code into it.** Download the ZIP from the [vite-p5-starter repository](#) and unpack it there, or research git and use `git clone`. Both are fine, and the ZIP is the quicker way.
3. **Check the starter folder.** Open it and make sure you see `package.json` and a folder `src` with `index.ts` in it. Unpacking a ZIP sometimes creates a folder inside a folder with almost the same name, and if the files are one level too deep, move the inner folder up so the files sit directly in your starter folder. The unpacked folder may also carry a longer name like `vite-p5-starter-main`, and renaming it to `vite-p5-starter` keeps your course folder tidy.
4. **Make your first exercise copy.** Copy the whole starter folder and name the copy `shapes`. The next chapter starts exactly there.
5. **Verify.** Your `shapes` folder contains `package.json` and `src/index.ts`. When both are there, you're done.

If a step goes wrong, ask a search engine or a general AI, and say which operating system you use and what you already tried.

7.2.7 Check your understanding

When your folder structure stands, take the short quiz below. You answer six questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Getting the starter code

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/tirdgr8fk6>

7.3 Your first program, again

Weeks ago you typed your very first program into the web playground, and a pink canvas with a line, a rectangle, a circle, and a triangle appeared. Today that same picture comes back, served by your own computer, out of a folder that belongs to you. Same language, same library, only the workshop has changed, and by the end of this chapter you'll have the proof on your screen.

7.3.1 AI tutor

Error messages from npm and from the terminal are new territory. They talk about folders, packages, and servers instead of a missing semicolon. The AI tutor of this part knows the starter project of this course and helps you read them. For the coding part it works as always and gives you hints, not finished solutions.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/v06yhwe08>

7.3.2 Open your exercise folder in VS Code

Start VS Code and choose **File > Open Folder**. That menu path is the same on Windows, macOS, and Linux. Open the copy you made for this exercise, the folder called **shapes**, and not the course folder around it or the untouched master starter next to it.

The first time you open a folder, VS Code asks whether you trust its authors. You do, because this code came from your course. The Explorer sidebar then lists the files you know from the starter tour, among them **src**, **index.html**, and **package.json**.

7.3.3 npm install brings the packages

Your folder holds a list of the packages the project needs, and no packages yet. **package.json** is that list, and npm turns it into real code on your hard drive.

Open VS Code's integrated terminal. It opens directly inside the folder you have open, which is exactly why opening the right folder matters. Then type:

```
npm install
```

npm reads **package.json**, downloads every listed package from the internet, and puts all of it into a new folder called **node_modules**. Expect a progress display and a short wait, because p5.js and the developer tools together are a lot of files.

Run **npm install** once per exercise copy. **node_modules** is big, it's generated, and you never edit anything in it by hand. You can delete the whole folder and get it back with another **npm install**. That's also why your download was so small. A list of package names takes far less space than the packages themselves.

7.3.4 npm start gives you your own web server

A p5.js program lives in a web page, and something has to serve that page to your browser. In the playground a server on the internet did that for you. On your own computer you start your own:

```
npm start
```

The terminal answers with an address, usually `http://localhost:5173`. Open it in your browser. If port 5173 was already taken, the number in the terminal is different, so read the address the terminal actually printed.

`localhost` always means *this computer*, the one you're sitting at. The page never travels over the internet and nobody else can open it, which is why this kind of server is called a **development server**. It exists so you can look at your own work while you build it.

i localhost has a number: 127.0.0.1

Computers address each other with numbers, so behind the name `localhost` there is a numeric address, 127.0.0.1. Both mean the same thing, this computer. A web page or an AI will sometimes write `http://127.0.0.1:5173` instead of `http://localhost:5173`, and both open the same page.

Programmers even wear this as a joke. A popular T-shirt says "There's no place like 127.0.0.1", playing on the English saying "there's no place like home", because that's what the number is, your computer's own home address.

What appears is a white canvas, the starter's own little program, already running before you changed a line. The server also watches your files, so when you save a change, the browser reloads by itself and you never press a run button. That's called **hot reload**, the modern version of the run experience you know from the playground.

While the development server runs, the terminal is busy and takes no new commands. To stop the server, click into the terminal and press **Ctrl+C** (on a German keyboard the key is labeled Strg). Until you stop it, the browser tab keeps working.

7.3.5 A look at `src/index.ts`

Open `src/index.ts`. The file starts with a block that looks unfamiliar:

```
import p5 from "p5";

const p = new p5((sketch: p5): void => {
  sketch.setup = setup;
  sketch.draw = draw;
  sketch.keyPressed = keyPressed;
});
```

Those lines are a **fixed frame**. They connect your functions to `p5.js`, and you never change them. Some of the syntax in there belongs to a later course, so treat the frame the way you treated `push` for a long time. It's a recipe that works, and the details come when you need them.

Below the frame, familiar friends are waiting:

```
function setup(): void {
  p.createCanvas(800, 400);
}

function draw(): void {
  p.background("white");
}

function keyPressed(): void {}
```

`setup` and `draw` do exactly what they did in the playground, and `keyPressed` is a new slot that runs whenever a key is pressed. You don't need it here, so leave it empty.

7.3.5.1 Every `p5.js` name gets a `p`.

One real difference separates the starter from the playground. The starter uses `p5.js` in **instance mode**, which means all `p5.js` names live on an object called `p` instead of standing on their own. Every name that came from `p5.js` gets `p.` in front of it, commands and constants alike:

```
// in the web playground
createCanvas(400, 500);
background("pink");
angleMode(DEGREES);

// in the starter project
```

```
p.createCanvas(400, 500);
p.background("pink");
p.angleMode(p.DEGREES);
```

Your own variables and your own functions keep their names unchanged. Only what p5.js gave you needs the prefix.

7.3.5.2 Which statement goes into setup, and which into draw

You met the split between `setup` and `draw` in the Conditions part, and the starter makes it impossible to ignore. `setup` runs once at the start. `draw` runs about 60 times per second, and it begins with `p.background("white");`, which paints the whole canvas white before anything else happens.

So anything you draw in `setup` gets painted over a fraction of a second later, and drawing statements belong in `draw`. Only the `p.createCanvas` call stays in `setup`, because the canvas is created once, not 60 times a second. Your own background statement takes the place of the white one at the top of `draw`.

A picture that never changes gets redrawn 60 times per second and still looks perfectly still, because every frame is identical to the one before.

7.3.6 When something goes wrong

New tools bring new error messages, and every one of them is research material. Copy the exact text into a search engine, into a general AI together with your operating system, or into the tutor of this part. These four are the ones almost everybody meets:

- **The terminal says `npm` is not recognized.** Either Node.js isn't installed, or the terminal was already open before you installed it and never noticed the new program. Close VS Code completely and open it again.
- **`npm install` finds no `package.json`.** The wrong folder is open in VS Code. Open the exercise copy `shapes` itself.
- **The browser shows nothing.** Look at the terminal. Is the development server still running? If it stopped, start it again with `npm start` and use the address it prints.
- **A red squiggle sits under a drawing command.** That's almost always a missing `p.` in front of a p5.js name.

7.3.7 Your exercise: your first program, again

Bring your very first program back to life, this time in your own workshop:

1. **Get your old code.** Open the Shapes exercise in the web playground and copy your code from the first week. If it's gone, type the example from the task description again; it shows the same program.
2. **Sort the statements.** In `src/index.ts`, put the `createCanvas` call with your sizes into `setup`. Put every drawing statement into `draw`, and let your own background statement replace the white one.
3. **Add the prefix.** Put `p.` in front of every p5.js name. You don't have to find them all by reading, because the red squiggles point at every one you miss. They're the same squiggles you met in your very first chapter, so let the editor guide you.
4. **Save the file and watch the browser.** The page reloads by itself. Saving also formats your code automatically, because the starter brings the formatter's settings along, and handing in formatted code is a rule of this course anyway.
5. **Play.** Change a color or a coordinate, save, and watch how fast the picture follows.

You're done when the browser at your `localhost` address shows the same picture as in your first week.

Bonus for the fast ones: bring the smiley from the arcs chapter over too. It needs `p.angleMode(p.DEGREES);` in `setup`, and the constant needs the prefix just as much as the command does.

That's the proof you came for. Same language, same library, same picture, and a workshop that can build any exercise in this book. The playground carried you through six parts and did its job well. From here on, the tools on your own computer take over.

7.3.8 Your coding assistant

The welcome chapter promised you a coding assistant for the last part of the book (Section 1.1.5), and this is where you get it. It works inside a **coding agent**, a program that reads and writes the files of your project while you chat with it. Your teacher decides which agent your class uses. Two free ones are [pi.dev](#) and [OpenCode](#), and both connect the same way.

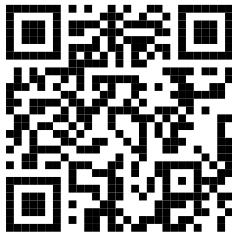
The box below is where you connect. Open the link, sign in with your school account, and the page shows a server address and a model name, plus a personal key of your own. Copy all three into your coding agent, and it talks to the assistant of this course instead of a general AI. The key doesn't change, so you do this once per computer.

🔥 Your coding assistant

Sign in with your school account and the page gives you a key of your own. That key is what your coding tool needs. The code below opens the page and nothing else.

Your teacher sees that you asked for a key, never what you ask the assistant.

▶ [Get your coding key on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/boh73jhiaiv>

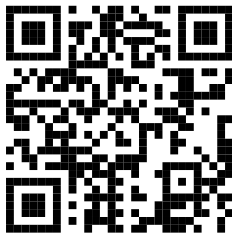
The assistant knows this course, so it writes TypeScript with p5.js in the style you've been taught and stays with the commands you already have. That doesn't move the responsibility. You have to be able to explain every line you hand in, and that counts double for a line an AI wrote for you.

7.3.9 Check your understanding

When your picture is back on the screen, take the short quiz below. You answer seven questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Your first program, again

▶ [Take the quiz on app.novedu.at](#)



Scan the code, or type this address into your browser:

<https://app.novedu.at/7kau29o1bi>

7.4 Pride in your craft

Your first program runs in your own workshop, and that's a real result. Running is not the same as finished, though. This chapter adds the last layer that turns a working program into work you can hand over, and it gives you two helper tools plus the two commands that run them.

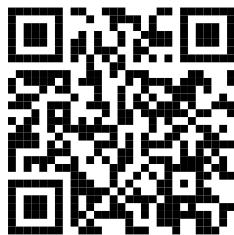
7.4.1 AI tutor

The tools in this chapter talk to you in a new kind of text. A linter message names a file, a line, a rule, and a problem, all in a few dense lines, and that takes practice to read. The AI tutor of this part knows the starter project and its rules, so paste a message you don't understand into the chat and ask what it wants from you. You'll get the message explained, not your program rewritten.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/v06yhwe08>

7.4.2 Working code is not finished code

A carpenter who builds a chair doesn't stop when the chair holds your weight. The joints still get cleaned up, the surface gets sanded, and the edges get smoothed, because handing over a rough chair says something about the person who built it. None of that sanding makes the chair carry more weight. It makes it a piece of work somebody is proud to deliver.

Code works the same way. Your program is read far more often than it is written, by you next week when you want to change something, by classmates when you build something together, and by your teacher when it gets graded. Messy code hides mistakes, and tidy code shows them.

That's why this course has demanded formatted code before you hand anything in since the very first weeks, and why unformatted submissions cost you points. It's not pedantry but the same care the carpenter puts into the surface. In your new workshop you have tools that do most of that work for you.

7.4.3 The formatter

A **formatter** is a tool that rewrites the layout of your code. It fixes indentation, spacing between words and symbols, line breaks, and it adds end-of-statement semicolons where they're missing. What a formatter never touches is the meaning. The same commands run in the same order with the same values, so a formatted program draws exactly the picture it drew before.

You met a formatter long before this chapter. Back in the chapter about the cat you right-clicked your code in the playground and chose **Format Document**, and the indentation and the spacing of the whole file snapped into shape (Section 1.4.3). That helper came with the playground website, so you never had to think about where it lived. In your own workshop it's a program of its own that sits in your project folder.

You've been using that program since the last chapter without being told its name. Every time you saved `src/index.ts` and the indentation jumped into place, that was the formatter, started automatically by the starter's VS Code settings.

The starter also gives you a command that does the whole project at once:

```
npm run format
```

Save-formatting only touches the file you're editing right now. `npm run format` walks through every file of the project and formats all of them. On a project that's already tidy, the answer looks like this:

```
Formatted 8 files in 2ms. No fixes applied.
```

This is the command to run before you hand anything in.

7.4.4 The linter

A **linter** is a tool that reads your code and points at things the language allows but that are suspicious or against the rules of your project. A linter doesn't care how your code looks. It cares about what your code says.

Here is the kind of thing a linter reports:

- a variable you declare and then never use
- a variable declared with `var`
- a comparison written with `==`
- an `if` whose branch has no curly braces around it
- a function without explicit type annotations

That list should sound familiar, because it's the list of rules this course has been drilling into you since the parts about variables and conditions. Those were never school rules invented to make your life harder. They're the rules professional programmers work by, and in this project a machine checks them for you. In every professional team, a linter does exactly the same job.

A linter reports and explains, and fixing is mostly your job. A message looks like this:

```
src/index.ts:3:5 lint/correctness/noUnusedVariables  FIXABLE
  ! This variable speed is unused.
  i Unused variables are often the result of typos, incomplete
    refactors, or other sources of bugs.
```

Read it from the front. `src/index.ts` is the file, `3:5` is line 3 and column 5, and `lint/correctness/noUnusedVariables` is the name of the rule you broke. Below that come a few sentences that explain the problem and often suggest a fix. Here the linter noticed a variable nobody uses, which is very often the leftover of a typo or of code you started and never finished.

7.4.5 Biome, and the two commands

Biome is both the formatter and the linter for TypeScript in this course. One program, two jobs, and it's already installed in your exercise folder because it came along with the starter and `npm install`. Remember that name, because from now on Biome is the tool behind every formatting and every rule message you see in this course.

Biome is also the successor of **Format Document** from the playground (Section 1.4.3). What the playground website quietly did for you, a tool in your own project does now, and it does more than the website ever did, because the playground never told you about `var`, `==`, or a variable nobody uses.

The rules Biome applies in this project live in the file `biome.json`, which you saw during the starter tour. Leave that file alone. Every project decides for itself which rules it wants, and in this course your teachers made that decision for you.

Two commands start Biome, and they do different amounts of work:

```
npm run format
npm run check
```

`npm run format` runs the formatter only. It fixes layout and semicolons in every file, and it says nothing about `var`, `==`, or unused variables.

`npm run check` runs the formatter, the linter, and the sorting of your imports in one pass. It applies the fixes it can apply safely, so formatting and missing semicolons are repaired without asking you. Everything else it reports, and those reports are yours to fix by hand.

7.4.6 Your exercise: polish your shapes

Open the `shapes` folder from the last chapter in VS Code, open the integrated terminal, and work through these five steps:

1. **Format the whole project.** Type `npm run format`. Most likely you'll see "No fixes applied", and that's success, not a failure. Your files were already formatted, because saving formats them for you. Clean is the normal state here.
2. **Run the full check.** Type `npm run check`. On an untouched starter with your shapes program in it, this is probably quiet too.
3. **Provoke the linter on purpose.** Now break the rules deliberately. Add these two lines inside your `draw` function, exactly as they are, with the missing semicolon and the missing braces:

```
var speed = 5
```

```
if (speed == 5) p.circle(100, 100, 50);
```

Save the file, then type `npm run check` again.

4. **Read every message.** For each one, find the file and the line, the name of the rule, and what the rule wants from you. Compare the file with what you typed: the semicolon after 5 is already back, because the formatter fixed that by itself. The `var`, the `==`, and the missing braces are only reported, because changing them is a decision only you can make. Fix all three by hand, the way this course writes them: a typed declaration like `const speed: number = 5;`, a comparison with `===`, and curly braces around the branch. Run `npm run check` again, and repeat until the check is quiet.
5. **Remove the experiment.** Delete the lines you added in step 3, save, and run `npm run check` one last time. Your program is clean again.

From now on this is a habit, not an exercise. Before you hand something in and before you show it to anybody, run `npm run format` or `npm run check` over your project. That includes code a coding assistant wrote for you, because the linter judges the file, not the person who typed it.

That completes your workshop. You have the editor, Node.js, and npm, you have the starter project and a folder rule that keeps a whole year of exercises findable, and now you have the tools that finish the surface of your work. The picture on the canvas is the part everybody sees, and the clean code behind it is the part programmers see. Caring about both is what turns programming into a craft.

7.4.7 Check your understanding

When your check runs quiet, take the short quiz below. You answer seven questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: Pride in your craft

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/p1wspqpjr5>

7.5 One file, many pictures: the slot machine



Figure 7.5.1: One picture file with 24 icons in it. Six columns and four rows of equally sized tiles: apples, cherries, a pear, grapes, bananas, and more fruit on green tiles in the first three rows, and hearts, a leaf, and a drop on teal tiles in the last row.

Twenty-four pictures, one file. That's the idea this chapter is built on, and it's the last new thing you learn in this part. Everything else you already have, because the program you build here is a slot machine, and it's the first program in this course that you design and write completely in your own workshop. No playground link, no starter code with your places marked, no sample solution. Your folder, your editor, your program.

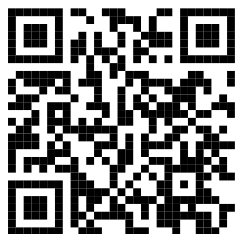
7.5.1 AI tutor

The slot machine is the graduation piece of this part, so treat it the way you would treat a graded task. The AI tutor of this part knows the starter project, the tools, and the sprite sheet of this chapter, and it answers with hints, questions, and explanations instead of finished code. That's exactly what you need here, because the point of this exercise is that you can build it.

💡 Your AI tutor

Hints and questions instead of finished programs, in English or German.

▶ [Ask your AI tutor on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/v06yhwe08>

7.5.2 Many pictures in one file

Think back to the Sokoban board (Section 6.12.5). Five tile types meant five image files, five addresses in an array, and five loads in `setup`. The board looked good, and it was still a very small game with a wall, a floor, a target, and two kinds of crates.

A real game has hundreds of tiles. Every wall corner, every floor variation, every frame of a walking figure is a small picture, and loading them one by one would mean hundreds of files and hundreds of downloads. Game programmers solved that in the arcade era and have done it ever since. All the small pictures are packed into one big picture file, and the program cuts out the piece it needs at the moment it draws.

Such a file is called a **sprite sheet**, and the small pictures inside it are **sprites**. One file means one download, one `loadImage`, and one variable in your program, no matter how many sprites are in it.

The trick that makes cutting easy is the grid. All sprites have the same size and sit in rows and columns without gaps, so their positions follow simple arithmetic. With tiles that are 96 pixels wide and 96 pixels tall, the tile in column `col` and row `row` starts at pixel `col * 96` from the left and `row * 96` from the top. That's the same grid arithmetic that placed the squares of the chessboard (Section 4.5.2), only the numbers are pixels inside a picture file now.

The sheet at the top of this chapter is exactly such a grid. The file is 576 pixels wide and 384 pixels tall, and it holds 6 columns and 4 rows of 96 by 96 tiles, 24 icons in total. The first three rows are fruit on green tiles, among them apples, cherries, a pear, grapes, bananas, and a strawberry. The fourth row is different: its tiles are teal, and it holds special symbols like hearts, a leaf, and a drop. Counting starts at zero here, as always, so the top left tile is column 0, row 0.

7.5.3 A picture inside your own project

Every image in this course so far came from an address on the internet. Your program loaded it from a web server somewhere far away, which works, but it also means your program only runs while that server does.

In your own workshop you have a better option. Pictures, sounds, and other ready-made files belong to the project, and the starter has a folder waiting for exactly that. During the starter tour it was empty and its description said “for pictures and other files your program may load later” (Section 7.2.3). That folder is `public/`, and its moment has come.

Getting the sprite sheet into your project takes two steps:

1. Download the file [Sheet_Fruits_Panel.png](#). Your browser may show the picture instead of saving it, in which case a right-click and **Save image as** does the job.
2. Put the file into the `public/` folder of your exercise copy, so that `public/Sheet_Fruits_Panel.png` exists.

The development server takes care of the rest. Everything in `public/` is served at the root of your page's address, so the file that sits in `public/Sheet_Fruits_Panel.png` on your hard drive is reachable as `/Sheet_Fruits_Panel.png` in your program. The leading slash means “start at the root of this address”, and nothing about the address mentions the folder `public` itself.

Loading works exactly as it did with pictures from the internet (Section 6.10.2), with the `p.` prefix in front of the `p5.js` names (Section 7.3.5.1):

```
let sheet: p5.Image;

async function setup(): Promise<void> {
  sheet = await p.loadImage("/Sheet_Fruits_Panel.png");

  p.createCanvas(800, 400);
}
```

Two details matter in your own file. The starter's `setup` is written as `function setup(): void`, and `await` only works inside an `async` function, so you change the line to `async function setup(): Promise<void>`. The starter even has a comment above `setup` that says so. And the variable `sheet` stands outside `setup` with `let`, because `draw` needs it too and it gets its value only when the picture has arrived.

7.5.4 Cutting one picture out: image with nine arguments

The `image` command you know takes five arguments, the picture plus where and how big it goes on the canvas (Section 6.10.3):

```
p.image(sheet, 0, 0, sheet.width, sheet.height);
```

Run that with the sprite sheet and you get all 24 icons at once, because the five-argument version always draws the whole file. For a slot machine you need one icon, so `image` has a longer version that adds four more numbers.

```
p.image(sheet, 0, 0, 96, 96, 192, 0, 96, 96);
```

Nine arguments, and they read in two groups of four. First comes the picture. Then four numbers for the **destination**, the place on the canvas, so x and y and the width and height it gets there, exactly the four you already know. Then four numbers for the **source**, the window inside the file, again x and y and a width and a height, this time measured in pixels of the picture file.

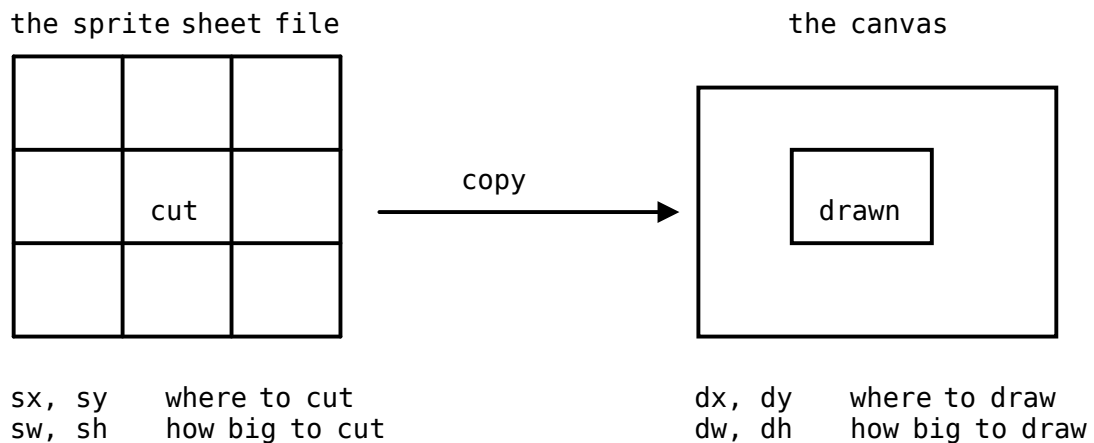


Figure 7.5.2: The nine-argument `image` command cuts a window out of the picture file and copies it onto the canvas. The four source numbers say where to cut and how big the window is, the four destination numbers say where the copy lands on the canvas and how big it is drawn.

The line above says: take the 96 by 96 window that starts at pixel 192 from the left and 0 from the top, which is the third tile of the first row, and draw it into the top left corner of the canvas at its natural size.

Written with literal numbers this is hard to read, and the slot machine needs it four times per frame, so it belongs in a function. Two constants and one function are the whole building block:

```
const SPRITE_SIZE: number = 96;
const SYMBOL_ROW: number = 0;

/** Draw one symbol of the chosen row at the position x, y */
function drawSymbol(symbolIndex: number, x: number, y: number): void {
  const sx: number = symbolIndex * SPRITE_SIZE;
  const sy: number = SYMBOL_ROW * SPRITE_SIZE;

  p.image(
    sheet,
    x,
    y,
    SPRITE_SIZE,
    SPRITE_SIZE,
    sx,
    sy,
    SPRITE_SIZE,
    SPRITE_SIZE,
  );
}
```

`sx` and `sy` are the grid arithmetic of the sprite sheet, one multiplication each. `SYMBOL_ROW` picks the row of the sheet you want to work with, so changing that single number swaps the whole set of symbols. The formatter puts every argument on its own line here because the call is too long for one line, and that's normal. Read it as the picture, then the four destination numbers, then the four source numbers.

With that function in place, four symbols side by side are a loop:

```
function draw(): void {
  p.background("white");

  for (let i: number = 0; i < 4; i++) {
    drawSymbol(i, i * SPRITE_SIZE, 20);
  }
}
```

That is not a slot machine yet. It's the building block the slot machine is made of, and everything else in this

chapter's challenge is built from things you already know.

⚠ Wrong numbers, no error message

Nothing in the nine-argument `image` can produce a red squiggle or a message in the terminal, because every argument is simply a number. A mistake shows up on the canvas, so learn to read it there.

- **The whole sheet, shrunk into one tile.** Your call has no source window, which happens when you use the five-argument version, so p5 draws the entire file into your small destination rectangle. You see all 24 icons at once, tiny.
- **The right icon in the wrong place, or the wrong icon.** The two groups are swapped. Destination comes first, source second, and both groups look the same, so the swap runs happily and draws nonsense.
- **Parts of two neighboring icons in one tile.** The source position is off. Check that you multiplied the column and the row by 96 instead of using them as pixel values.

7.5.5 Your coding assistant

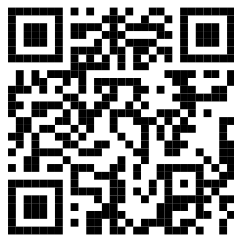
The slot machine is the biggest program in this book, and the coding assistant may help you build it. Connect it the way you did in Section 7.3.8, with the key you already have. Then read what it gives you before you keep it, because the craftsmanship requirements below apply to every line in your file, whoever typed it first.

🔥 Your coding assistant

Sign in with your school account and the page gives you a key of your own. That key is what your coding tool needs. The code below opens the page and nothing else.

Your teacher sees that you asked for a key, never what you ask the assistant.

▶ [Get your coding key on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/boh73jhiav>

7.5.6 Your challenge: build the slot machine

A slot machine is a gambling device. Reels with symbols spin, they stop one after the other, and matching symbols pay out. Your program is the harmless version of it, with credits that exist only on your canvas.

This is the graduation piece of the part, and it's bigger than anything you have written so far. It uses drawing commands, colors, constants, arrays, loops, conditions, your own functions, the mouse, `setInterval`, text on the canvas, and the sprite sheet of this chapter. Nobody hands you a prepared project with `// <<<` comments this time. You get requirements, the way you will get them in every project from now on, and you decide what the functions are called and how the program is put together.

7.5.6.1 Set up your project

Follow the folder rule of this course (Section 7.2.5):

1. Copy your clean master starter folder and name the copy `slot-machine`.
2. Open the copy in VS Code, run `npm install`, and start it with `npm start`.
3. Put `Sheet_Fruits_Panel.png` into the `public/` folder of the copy, as described in Section 7.5.3.

7.5.6.2 Basic requirements

Everything in this list has to work.

Pick one row. Choose any row of the sprite sheet and use its six icons as the symbols of your machine. Ignore the other three rows completely.

Four reels. Show four symbols side by side. When the program starts, each one shows a random symbol of your row, and `p.floor(p.random(6))` gives you a random index from 0 to 5.



Figure 7.5.3: Four reels side by side, each showing one icon from the same row of the sprite sheet: an apple, a pear, cherries, and an orange.

Two buttons, drawn with p5. Your program needs a “Spin” button and a “Stop” button, and you draw both yourself with a rectangle and a text label. HTML buttons are not allowed here, because this course has never taught them.



Figure 7.5.4: Two buttons drawn with p5: a green button labeled Spin, and a gray button labeled Stop.

The click. Clicks arrive in a function called `mouseClicked`, and that function needs one connection to `p5.js` before it runs. The frame at the top of `src/index.ts` lists the functions `p5.js` calls for you, and the starter connects three of them. Add a fourth line, and write the matching function below in the file:

```
const p = new p5((sketch: p5): void => {
  sketch.setup = setup;
  sketch.draw = draw;
  sketch.keyPressed = keyPressed;
  sketch.mouseClicked = mouseClicked;
});
```

That single line is the only change the frame ever needs from you. `mousePressed` works exactly the same way if you prefer it. Which button was hit is then the rectangle hit test from the silo dashboard, four comparisons that box `p.mouseX` and `p.mouseY` into the button (Section 5.8.5).

Spin. A click on “Spin” starts the machine. All four reels change their symbols quickly and randomly, which is a job for `setInterval` (Section 6.6.3). Store the id it returns in a variable, because stopping the machine needs that id.



Figure 7.5.5: The four reels while the machine is spinning. The symbols change several times per second, and all four keep changing until the machine is stopped.

Stop. A click on “Stop” ends the spinning with `clearInterval` (Section 6.6.4), and the four symbols stay as they are. A click on “Spin” starts a new round.

The result. After stopping, your program says what the player got, as text on the canvas:

Table 7.5.1: The four possible results of a round.

what the reels show	text
four different symbols	No match
two equal symbols	Two of a kind
three equal symbols	Three of a kind
four equal symbols	Jackpot!

The text disappears when the next spin starts. Finding the result means counting, and you have counted with arrays before. Make an array with one counter per symbol of your row, walk over your four reels, and count up the counter of each symbol you see. The largest counter in that array is your answer.

7.5.6.3 Advanced requirements

When the basic machine runs, make it a real game.

Buttons that show their state. Only one of the two buttons does anything at any moment. While the reels stand still, that's "Spin", and while they spin, it's "Stop". Show the difference, for example by drawing the button that does nothing in gray and the active one in a strong color. A `boolean` that remembers whether the machine is spinning is enough to decide it.

Credits. The player starts with 10 credits, and the number is visible on the canvas at all times.

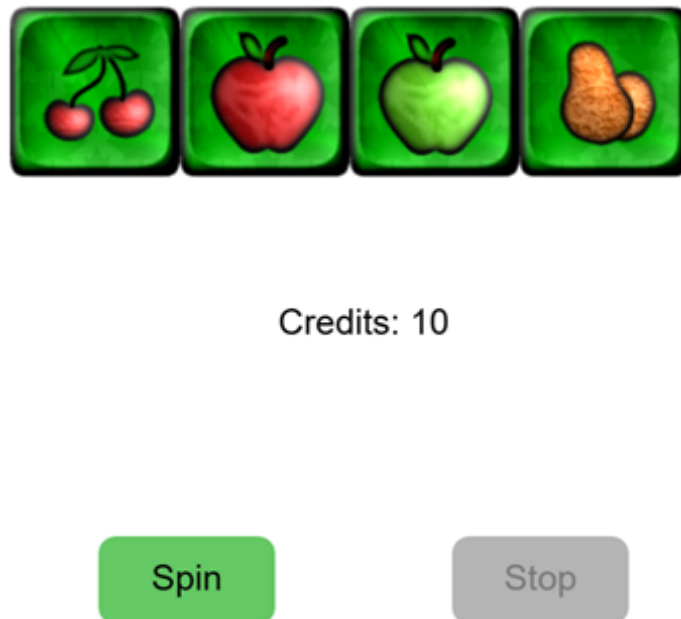


Figure 7.5.6: The finished machine: four reels at the top, the text "Credits: 10" in the middle, and the Spin and Stop buttons at the bottom.

Paying and winning. Every spin costs one credit, taken when the player presses "Spin". At zero credits no spin starts, and the canvas says why. Stopping pays out, depending on the result of the round:

Table 7.5.2: What a round pays back.

result	payout
No match	nothing
Two of a kind	1 credit, the spin was free
Three of a kind	2 credits
Jackpot!	4 credits

7.5.6.4 Craftsmanship requirements

The slot machine is judged as a piece of work, not only as a running program.

- **Hand in clean code.** Run `npm run format` and `npm run check` before you show your program, and keep going until the check is quiet (Section 7.4.6).
- **No copy and paste.** Drawing a button, drawing a symbol, and rolling a random symbol happen more than once, so each of them is a function with parameters. If you find yourself copying a block and changing two numbers in the copy, you found a function.
- **Meaningful names.** `drawButton` and `spinning` say what they are. `doIt` and `flag2` say nothing.
- **Few global variables.** Some state has to be global, like the loaded sheet, the four current symbols, and the credits. Everything a function can take as a parameter should be a parameter.

- **Only what you can explain.** This course has one standing rule about language and library features it hasn't covered. You may look things up and you may ask an AI, and whatever ends up in your file, you have to be able to explain line by line. Code you cannot explain counts as wrong, no matter who wrote it. You are the programmer here, and you are responsible for the result.

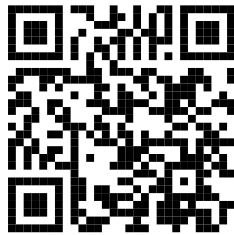
When your machine spins, stops, pays out, and passes `npm run check`, you have finished the first program of this course that is completely yours. That is the end of this part, and the workshop it built is the one you'll use for the rest of the book.

7.5.7 Check your understanding

When your slot machine runs, take the short quiz below. You answer seven questions about this chapter in your own words, and an AI reads your answers and tells you what you already understand and what you should read again. The quiz is anonymous, and answering in German is fine too.

i Quiz: One file, many pictures: the slot machine

► [Take the quiz on app.novedu.at](https://app.novedu.at)



Scan the code, or type this address into your browser:

<https://app.novedu.at/vh2ueq5lwe>

7.6 Where you are now

Your first program was one line. You typed `circle(200, 260, 360);` into a playground in your browser, pressed Run, and a circle appeared. The same chapter showed you the line with a capital C, which draws nothing, and that was the lesson every other chapter has quietly relied on since: the computer takes you literally. Your last program was a slot machine with a picture sheet, a spinning animation, credits, and a check that passes, written in a folder of your own with nothing prepared for you. This short chapter is the pause between those two programs and whatever you build next.

7.6.1 What you carry out of this book

Look at the chapter list once more, from the top. Every part added one idea, and each idea only made sense because the one before it was already yours.

- **Values with a name.** A variable is a box with a name and a type, and a program is a story of values moving between boxes.
- **Decisions.** An if statement asks a question and runs one branch or the other. A switch asks the same question six times at once.
- **Repetition.** A loop lets you say “again” without typing it again, and a loop inside a loop gave you grids, meshes, and chessboards.
- **Many values under one name.** An array holds a whole row, and you know the difference between copying a number and copying the arrow to an array, which many people who program for a living never quite learned.
- **Commands of your own.** A function takes parameters, may answer with a return value, and can even be handed to somebody else to call later.
- **A workshop of your own.** VS Code, Node.js, npm, and Biome. You can unpack a project, install its packages, format it, check it, and run it without a playground preparing any of it for you.

None of these are toy versions. `let`, `if`, `for`, arrays, functions, and `npm install` are the same words a professional types on a Monday morning. The programs were small so that you could see every part of them, and that is the only difference.

7.6.2 Two habits that matter more than any command

The command list above is easy to look up again. Two things from this book are harder to find in a reference, and they are the ones you should keep.

The first is **pen and paper before the keyboard**. You played computer with a table of `mouseX` values, you drew a silo with its constants labeled before you drew it in code, and you predicted what `console.log` would print before you ran it. Every time your prediction was wrong, you learned something the running program alone would have hidden. Keep doing that. A sketch on paper is the cheapest debugging tool you’ll ever own.

The second is **only what you can explain**. You may look anything up, and you may ask an AI for help, and that stays true next year and in every job after it. The rule is that whatever ends up in your file, you can walk through line by line and say why it’s there. Code you can explain is yours. Code you can’t explain is borrowed, and borrowed code fails you at the worst moment, usually in an exam or in front of a customer.

7.6.3 Keep making things

The folder on your computer doesn’t go away when the course ends, and neither does what you know. Until the next part of this course starts, make something. It doesn’t have to be big, and it doesn’t have to be good. Take the bouncing ball and give it gravity. Give the slot machine a sound, or a second player. Draw the flag of a country you’d like to visit, with a loop for the stripes. Redo Word Swirrel with a word that means something to you. The point isn’t the result; the point is that you open VS Code without anyone telling you to, and that your fingers remember `npm start`.

When something breaks, and it will, read the error message from the beginning, the way you learned in the very first part. Then sketch. Then ask. That order has not changed since your first circle, and it won't change next year either.

7.6.4 What comes next

This book drew on a canvas. The next book, *Creative Coding 2*, keeps the language and the tools you have now and changes the world they work in. Instead of painting pixels, your program builds a web page out of elements, styles them with CSS, draws with SVG, and reacts to clicks and keystrokes on that page. A web page turns out to be full of **objects**, things that have properties and can do things, and by working with the page's objects you learn what an object is. Later you write your own, with classes, and you test them so that you can prove they work instead of hoping. Every one of those chapters stands on a variable, a loop, an array, and a function, so you already have the ground it's built on.

That is the whole plan. You came in with no programming experience, and you leave with a workshop, a slot machine, and the habit of predicting before you run. Close the laptop, take a walk, and be a little proud of that. See you in the next book.