HTML & CSS

Inhaltsverzeichnis

HTML Basics

A web page is a **document** with text, images, videos, links and other content, **displayed in the browser.** Typically, we create such a document using three languages:

- **HTML** or Hypertext Markup Language, which defines the structure of the page using HTML elements.
- **CSS** or Cascading Style Sheets, which determines how elements on the page will look.
- **JavaScript,** which lets us change HTML and CSS in response to user actions.

In this lesson, we'll focus on HTML Basics, beginning with...

Tags

An HTML page is **made up of elements,** which we insert with tags — e.g.

```
<title>
  This is the browser tab title
</title>
```

- `<title>` is the **opening tag.** A tag always starts with the `<` symbol, followed by a word that denotes the **type of element**, and ends with the `>` symbol.
- `</title>` is the **closing tag**. It always starts with `</`, followed by the same **type of element** as in the opening tag, and ends with the `>` symbol at the end of the line.
- Between the opening and closing tags of an element, there's content, i.a., text, pictures, and videos.

If there's little content, you can write everything in one line:

```
<title>This is the browser tab title</title>
```

Here are examples of typical HTML elements:

```
<h1>This is the main page heading</h1>

<section>
  <h2>This is a section heading</h2>

  The page's section can include many HTML elements.
  For clarity, every such element starts on a new line.
  We indent it by two spaces to the right of the opening tag.

  <p>
    This is a paragraph
    It can contain one or several lines of text
  </p>
</section>
```

Attributes

An attribute is a **modifier,** and as such, it can affect the element's behavior, appearance, and functionality. We'll discuss them in detail later on. For now, here's a closer look at the attribute's structure:

```
<tag attr1="value1" attr2="value2">
  Content
</tag>
```

First comes the element type, second — a space, third – the attribute's name, fourth — the `=` operator, and fifth — the attribute's value in "double quotes". We can add as many attributes as we like, for example:

```
<section class="content">
  <h2 class="title is-4" id="about-us">About us</h2>

  Mate academy ...
</section>
```

Elements without Content

Images, text input fields and a few other elements don't require a closing tag, because the content is specified through attributes. For example, we can add an image using the `<img>` tag with the `src` (image address) and `alt` (image description) attributes:

```
<img src="example.jpg" alt="Example image">
```

Structure of an HTML Page

Here's a basic structure of an HTML page:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Mate academy</title>
  </head>
  <body>

  </body>
</html>
```

Let's break it down, tag by tag:

1. `<!DOCTYPE html>` always comes first. It defines the document type and HTML version.
2. `<html>` is the root element, and its `lang` attribute lets us set the language of the page. Inside the root element, there are two more elements — `head` and `body`.
3. `<head>` contains service information that **isn't** displayed on the page, but *is* important for the page's inner workings. Additional resources, such as styles and scripts, are also connected here.
4. `<meta charset="UTF-8">` sets the character encoding on the page.
5. `<meta name="viewport">` sets the display parameters of the page.

6. `<title>` sets the browser tab title.
7. `<body>` contains the visible content of the page (where most elements are).

Favicon

Favicon is a small image (icon) displayed next to the page title in browser bookmarks. To add a favicon, use the `<link rel="icon">` tag inside the `<head>` tag:

```
<link rel="icon" href="logo.png">
```

💡 The `href` attribute sets the image address.

Headings

In HTML, there are six heading levels. `<h1>` is the most important, `<h6>` — the least:

```
<h1>Heading Level 1</h1>
<h2>Heading Level 2</h2>
<h3>Heading Level 3</h3>
<h4>Heading Level 4</h4>
<h5>Heading Level 5</h5>
<h6>Heading Level 6</h6>
```

A page should have only one `<h1>` heading, but can have many `<h2>` headings — for sections, `<h3>` — for subsections, and so on. **Follow this hierarchy and don't skip levels.** You'll improve page ranking (SEO) and accessibility for screen reader users.

Links

Pages on the Internet often link to each other. For this, we use the `<a>` tag along the `href` attribute, where we specify the URL of the target page or resource:

```
<a href="https://www.google.com/">Google</a>
<a href="index.html">Home</a>
```

By default, when you click on a link, the target page loads in the same browser tab. We can change this behavior with the `target="_blank"` attribute:

```
<a href="https://www.google.com/" target="_blank">
  Google
</a>
```

Text Formatting

Here are the most popular tags used for text formatting:

- `<br>` opens a new line
- `<strong>` makes the text bold
- `<em>` makes the text italicized
- `<span>` lets us style to a part of a text

```
<p>This is a text paragraph.</p>
<p>This is a text paragraph <br>with a line break.</p>
<p>This is a paragraph with <strong>bolded text.</strong></p>
<p>This is a paragraph with <em>italicized text.</em></p>
```

Lists

There are three types of lists in HTML, description, bulleted and numbered. We'll only discuss the latter two, however, since**description lists `<dl>`** are hardly used.
`<ul>` is used to create a **bulleted list,** and `<li>` adds an item:

```
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

`<ol>` is used to create **numbered list:**

```
<ol>
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ol>
```

Lists can be nested:

```
<ol>
  <li>Item 1
    <ul>
      <li>Item 1-1</li>
      <li>Item 1-2</li>
    </ul>
  </li>
  <li>Item 2</li>
  <li>Item 3</li>
</ol>
```

Comments

Comments in HTML are text wrapped in `<!--` and `-->`. **They don't affect the page,** but help programmers better understand the page's logic:

```
<!-- This is a comment. -->
<!--
 <p>
  This paragraph is not shown on the page
 </p>
-->
```

CSS Basics

Cascading Style Sheets, CSS for short, is a language used to describe the layout and styling of website elements. Today, we'll explore its basics.

Inline Styles

Inline styles are a quick way to style elements since **we add them directly to an element** using the `style` attribute. Here's an example:

```
<p style="color: blue; background-color: yellow;">
  This is a paragraph with blue text and a yellow background.
</p>
```

...and here's the breakdown:

1. We added two **CSS declarations,** `color: blue;` and `background-color: yellow;`, to the paragraph.
2. Each of these declarations consists of a **property,** a colon `:`, a **value,** and a semicolon `;`.
3. The `color` property specifices the text color, while `background-color` — the element's background color.

Inline styles might come across as incredibly simple, but don't be fooled. **They crowd the code, making it harder to comprehend,** which is why we mostly use inline styles for specific JavaScript styling (we'll discuss it later on). Other approaches are far more universal, and therefore — popular.

Internal Styles

Internal styles are an alternative way of adding styles to a page with `<style>`. This tag is usually found in the page's `<head>`:

```
<style>
p {
  color: blue;
  background-color: yellow;
}

a {
  color: green;
}
</style>
```

Here, there are two CSS rules, and each begins with a selector (letters `p` and `a`). We can use said selectors — in this case, tag names — to select particular elements and *style them up,* enclosing styles in curly braces `{}`. As a result, for all...

- `<p>` paragraphs, the text is now blue and comes with a yellow background
- `<a>` links, the text is now green

External Styles

Styles can also be moved to a separate CSS file, such as `styles.css`:

```
p {
  background-color: grey;
}

a {
  color: red;
}
```

...and linked to the page using the `<link>` tag inside `<head>`:

```
<head>
  <link rel="stylesheet" href="styles.css">
</head>
```

When a user first visits the page, their browser downloads the stylesheet and caches it (i.e.: saves for later use). So, with next visits, only the HTML is downloaded, while the stylesheet is taken from the cache, which **speeds up page loading.** This continues for as long as the stylesheet's address stays the same.

But doesn't it mean the page looks unstyled until the stylesheet is downloaded? Indeed, but internal styles come with another con — they cannot be cached, meaning they increase the page size and the time required to load it. So, in large projects, a mix of the two approaches is used:

- Critical styles (**Critical CSS**) are added in `<style>`
- All other styles are moved to separate files (**chunks**) and connected to the pages where they're needed

Selectors

When using type selectors (i.e.: tag names), styles apply to all elements of that type. However, it's often necessary to style elements of the same type differently, and in such cases — we need to use `id` and `class` attributes.

`id`

`id` is a **unique identifier** of an element. We often use it for navigation within a page or in forms to link a text field to its label. To demonstrate, here's an element with `id`:

```
<h1 id="main-header">
  Mate academy
</h1>
```

...which we can style by placing `#` before `id`:

```
#main-header {
  color: green;
}
```

But since the identifier is, well, *unique,* we cannot reuse such styles. This makes `id`-bound selectors a rather **unpopular choice** for styling.

`class`

Unlike `id`, the `class` attribute's value doesn't have to be unique. We can also add more than one `class` to a single element:

```
<p class="message">This is a message</p>
<p class="message warning">This is a warning message</p>
```

We can select an element by class adding `.` at the beginning:

```
.message {
  background-color: grey;
}

.warning {
  color: red;
}
```

Here, all elements with the word `message` as their `class` attribute will receive a grey background, regardless of the element type. And the text of all elements with the `warning` class will be red. This makes classes **the most flexible tool for styling,** which is why we'll use them.

Comments

Comments are added to explain why a particular decision was made. In CSS, a comment can be added by wrapping the text in `/*` and `*/`, just like in JavaScript. **Any text between `/*` and `*/` is ignored by the browser.** VSCode lets us comment or uncomment the current line or selected text just by pressing `ctrl + /` (or `cmd + /`). We often use this to temporarily disable some rules and see what the page looks like without them:

```
html {
  /* default text color */
  color: grey;

  /* background-color: light blue; */
}
```

Text Styling

The following CSS properties are most commonly used for text styling:

- **`font-family`** sets the font family.
- **`font-size`** sets the font size.
- **`line-height`** sets the line height of text.
- **`text-align`**: `center` aligns block text to the center.
- **`white-space`**: `nowrap` prevents text from wrapping to the next line.
- **`font-style`**: `normal` for regular font, `italic` for italic font.
- **`font-weight`**: `normal` for regular weight, `bold` for bold font.
- **`text-decoration`**: `none` for regular text, `underline` for underlined text.
- **`text-transform`**: `uppercase` makes all letters uppercase.
- **`color`** sets the text color (more on that in a separate lesson).
- **`background-color`** sets the element's background color.
- **`cursor`**: `default` for the arrow cursor, `pointer` for the pointer cursor, which usually indicates that the element can be interacted with.

Some elements have certain styles by default, e.g.:

- `<a>` underlines the text, colors it blue, and switches the cursor to `pointer` upon hover.
- `<strong>` **bolds** the text.
- `<em>` *italicizes* the text.

Colors and Fonts

Let's learn how to set colors and fonts on a webpage.

Fonts

We can set fonts with the `font-family` property, listing names of font families and font types separated by commas. If the first font isn't available on the user's computer, the browser will fall back on the second font, then — third, and so on:

```
html {
  font-family: Arial, "Helvetica Neue", Helvetica, sans-serif;
}
html {
  font-family: "Times New Roman", Times, Baskerville, Georgia, serif;
}
```

It's a good practice to specify a popular, generic font type at the end, so the browser can always fall back, like `serif` and `sans-serif`. If you'd like to get even more into detail, we recommend **this read**.

Font Properties

The `font-weight` property lets us thicken or thin out the text depending on its value, using keywords like `normal` or `bold`. For a more precise measure, we can use numbers from `100` to `900` in increments of 100. `400` corresponds to `normal`, and `700` — to `bold`.
Here's an example:

```
font-weight: bold;
font-weight: 500;
```

The `font-style`, on the other hand, lets us switch to the italic version of the font. If it exists, that is:

```
font-style: italic;
```

External Fonts

Don't like falling back on the fonts stored locally? Luckily, we can always use an external font with the `@font-face` rule. Just need to specify the font's name and its path:

```
@font-face {
  font-family: CustomFont;
  src: url(fonts/custom_font.woff);
}

html {
  font-family: CustomFont;
}
```

Programmers, like everyone else, like freebies, and the go-to place for free fonts is **Google Fonts**. Here's how to use them:

- Choose the desired font and styles.
- Copy the tags displayed on the right and paste them into the `<head>` of your page:

```
<head>
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link
href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;500;700&display=swap"
rel="stylesheet">
</head>
```

Wohooo! This font is now available in your CSS:

```
body {
  font-family: 'Roboto', sans-serif;
}
```

Relative Units: `em` and `rem`

Font size is typically set in pixels (`px`). Since most font parameters are inherited by nested elements, setting it once for the root element should suffice:

```
html {
  font-size: 16px;
}
```

In such a set up, changing the font size is as simple as... changing the `px` number. For some elements, however, you may want to set a larger or smaller font size, which is where the `em` unit comes in handy. It sets the font size **relative to the parent element's font size:**

```
<section class="section">
  <h2 class="title">Section title</h2>

  <p>Some text</p>
</section>
.section {
  font-size: 16px;
}

.title {
  font-size: 1.5em;  /* 24px = 1.5 * 16px */
}
```

But like most things, the `em` unit has a drawback. Namely, the font size it expresses depends on the element's placement on the page:

```
.section {
  font-size: 16px;
}

.sidebar {
  font-size: 12px;
}

.title {
  font-size: 1.5em;
}
<section class="section">
  <!-- 16px * 1.5 = 24px -->
  <h2 class="title">Section title</h2>
</section>
```

```
<aside class="sidebar">
  <!-- 12px * 1.5 = 18px -->
  <h2 class="title">Sidebar title</h2>
</aside>
```

To make sure the font size stays consistent regardless of the element's position, we can use `rem`. This unit is calculated **relative to the `font-size` of the root element,** which is usually `<html>`:

```
html {
  font-size: 20px;
}

.parent {
  font-size: 16px;
}

.child {
  font-size: 1.5rem; /* 30px = 1.5 * 20px */
}
```

Colors

Several properties let us change the color of elements, i.a.:

- `color` — text color;
- `background-color` — background color;
- `border-color` — border color.

```
color: white;
background-color: black;
```

Colors themselves can be specified in a few formats:

1. **Color names**, such as `white`, `black`, `blue`, `red`, etc.
2. **RGB**, which lets us specify the intesity of red, green and blue, individually. We can use percentages or whole numbers from `0` to `255`:

```
color: rgb(100%, 50%, 0);
color: rgb(255, 128, 0);
```

3. **RGBA**, which combines `rgb` with the color's degree of transparency expressed as a percentage or a number from `0` to `1`:

```
color: rgba(255, 0, 0, 50%);
color: rgba(255, 0, 0, 0.5);
```

4. **Hexadecimal notation** with 3 or 6 characters preceded by `#`:

```
color: #ff0000;
color: #f00; /* same as `#ff0000` */
```

While there are more color formats, they are a rarity — no need to bother.

Box Model Basics

Today, we'll consider a concept *fundamental* to web development — the CSS box model. It incorporates four components that determine the size and position of the page's elements:

- **content**

- **padding**
- **border**
- **margin**.

Box Model Breakdown

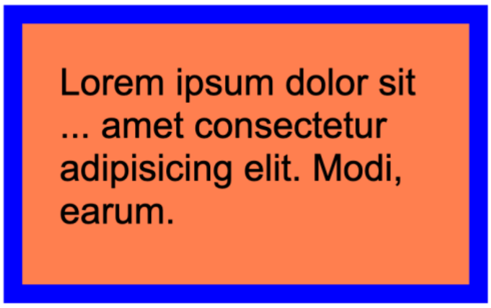
Imagine we have a block element `<div>` with the class `box` and some text:

```
<div class="box">
  Lorem ipsum dolor sit ... amet, consectetur adipisicing elit. Modi, earum.
</div>
```

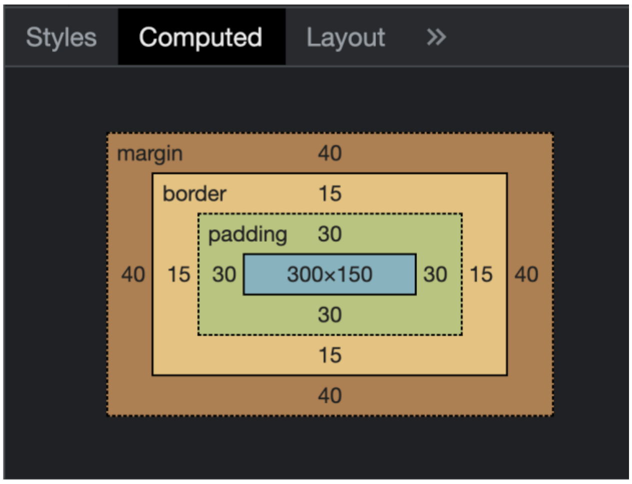
Let's add below styles to **visualize** all the CSS box model components:

```
.box {
  background-color: coral;

  width: 300px;
  height: 150px;

  padding: 30px;
  border: 15px solid blue;
  margin: 40px;
}
```

On the left, we see the element, and on the right — its diagram:



Lorem ipsum dolor sit ... amet consectetur adipisicing elit. Modi, earum.



Here's the component breakdown:

1. **`Content` is the most-inner area that houses content (text, images, etc.).** We set its size to `300px` by `150px` with the **`width`** and **`height`** properties.
2. **`Padding` separates `content` from the element's `border`.** We set its width to `30px` on all sides with the **`padding`** property, but we could differentiate each side with `padding-top`, `padding-right`, `padding-bottom`, and `padding-left`.
3. **`Border` acts as the element's frame.** We set it with the **`border`** property, specfying the border's style (`border-style`), width (`border-width`), and color (`border-color`). It's possible to define border parameters for each side separately (e.g., `border-left-width: 5px`). In case we set the border style to `dashed`, or `dotted`, or make its color `transparent`, the element's background will show from underneath.

4. `Margin` **is the minimum distance from the element's border to adjecent elements.** We set it with with the `margin` property for all sides, but we could also differentiate each side with `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`. The element's background doesn't show in the margin area.

When two elements are close together without any `border`, `padding`, or `content` separating their vertical margins, **these margins can merge,** which is known as **margin collapse.** To mitigate this issue, developers apply padding to the parent element. They also add margins on only one side along each axis: top and bottom (vertical spacing) or left and right (horizontal spacing).

Both margin and padding can be defined with **percentages** rather than pixels. `%` are calculated based on the parent element's width, even for the top and bottom spacings. Using `em` units is an option as well — this way, we tie spacing to the element's `font-size`. It enhances the visual harmony between text and the surrounding space, making for a more appealing design.

box-sizing

The `box-sizing` property affects the content's width and height. We can assign it one of two values:

1. `content-box`, to **exclude** `padding` and `border` from the element's width and height.
2. `border-box`, to **include** `padding` and `border` in the element's width and height.

Here, we set `box-sizing` to `border-box`, which means the content's width will be less than `300px`. That's because `border` and `padding` are subtracted from each side:

```
.content {
  box-sizing: border-box;
  width: 300px;
  height: 150px;

  margin: 40px;
  border: 15px solid blue;
  padding: 30px;
}
```

Block and Inline Elements

Everything we discussed above works for `div` elements, but not for `span`.That's because `div` is a block element (a rectangle with some content), while `span` is an inline element (part of a text, spanning one or more lines). Here's how they differ:

1. **Block elements:**

- Occupy a separate line
- By default, take up the full width of their parent element
- We can set their width and height
- `padding`, `margin`, and `border` increase the width and height occupied by the element
- Examples include `div`, `p`, `ul`, `li`, and `h1`

2. **Inline elements:**

- Occupy the same line as their inline neighbors
- Take up only as much width as necessary for their content
- We cannot set their width and height

- `padding`, `margin`, and `border` do not increase the height occupied by the element (adjacent lines may overlap);
- Examples include `span`, `a`, and `em`

Inline-Block Elements

Sometimes we might need to set custom size and spacing for an element in the text, say, a link. For this, we use **inline-block** elements:

```
.nav__link {
  display: inline-block;
}
```

That's because `inline-block` elements are a combination of `inline` and `block` elements. **`inline`,** because they stretch only to the necessary width, and can be placed in one line with other elements. **`block`,** because their `width` and `height` can be set, and because their `padding`, `border`, and `margin` expand the occupied area in all directions.
By default, `<img>` is an inline-block element. To better understand the difference between them and block/inline elements, check out **this example**.

Centering Content

To align text or an inline element horizontally within its parent, we use the **`text-align`** property:

```
<h1 class="title">
  Mate academy
</h1>
.title {
  text-align: left; // default behavior
  text-align: center; // centers
  text-align: right; // aligns to the right edge
  text-align: justify; // distributes spaces in each line of text evenly
}
```

Vertical centering of a single-line text is the simplest. Just set the element's `heigth` and `line-height` as equal:

```
.title {
  height: 60px;
  line-height: 60px;
}
```

...but if text spans over multiple lines, it will extend beyond the element. We'll consider this later. Back to centering, we can center a **block** element horizontally within its parent automatically, using `margin: 0 auto`:

```
<main class="content">
  <h1 class="title">Mate academy</h1>
</main>
.content {
  width: max-content; /* the width required for the content */
  margin: 0 auto;
}
```

The catch? Such a centering works only if the element has a smaller width than its parent. Though, we'll expand on this — and consider vertical centering — a little later. Now, onto the tasks! :)

Semantic Basics

Some HTML elements are named after their position on the page or their contents. `<p>` stands for paragraph, `<nav>` for navigation (because it contains links that let us navigate the page), `<b>` for bold, etc. That's why we call them **semantic elements.**

Here are a few more **examples** of such elements:

- `<header>` houses the page's opening section with a logo, navigation links, or other elements that typically show on all pages.
- `<footer>` is like the header, but contains less important/repeated information and is shown at the very bottom of the page.
- `<main>` houses the page's main content.
- `<aside>` sports additional content, usually used to wrap a sidebar.
- `<section>` is a page section with its own heading (can be visually hidden though).
- `<article>` wraps a standalone content unit, such as a product card, an article, or a comment.
- `<h1>` through `<h6>` are headings, from the most to the least important.

Using semantic elements has its **advantages:**

1. **Improve page accessibility.** People with disabilities often use screen readers to consume online content, and semantic tags — closely related to their natural origin, English — make the process easier. Why learn new words, if we can quickly decipher what `<ul>` stands for? (That's an "unordered list", by the way!).
2. **Better search engine optimization (SEO).** Google crawlers can't *see* the page, but they can read its structure and content, which determines their position in search results.
3. **Simplified maintenance.** Believe it or not, developers are people, too, and as such, they prefer reading semantic tags.

If you're into supplementary reads, **here's one we highly recommend**.

Typical Page Structure

Usually, pages are composed of:

- Navigation (placed in the `<header>`)
- Two sections (in `<main>`)
- Footer

It's typical to find two comments as well, `<!-- #region HEAD -->` and `<!-- #endregion -->, since they let us quickly collapse code placed in between them (in the code editor). Some elements also have comments added with **emmet queries,** which we can copy and expand using the` tab` key into ready-made markup. Convenient, ain't it? To generate text, we can use **emmet commands** — `lorem20` or `lorem50`. "Lorem" stands for *lorem ipsum,* a latin text used as example text (so that people can design websites even without ready copies). The number specifies how many words we'd like the text to be.

Here’s an example of a HTML page:

```
<!-- #region HEAD -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <link rel="icon" href="https://mate.academy/static/favicon/apple-touch-icon.png">
  <title>mate academy</title>
  <link rel="stylesheet" href="normalize.css">
  <link rel="stylesheet" href="style.css">
</head>

<!-- #endregion -->
<body>
  <!-- header.header>nav.nav>a.nav_link*4 -->
  <header class="header">
    <nav class="nav">
      <a class="nav__link" href="index.html">Home</a>
      <a class="nav__link" href="courses.html">Courses</a>
      <a class="nav__link" href="about.html">About</a>
      <a class="nav__link" href="contacts.html">Contact us</a>
    </nav>
  </header>

  <main class="container">
    <h1 class="title">mate academy</h1>
    <p>Our mission is to help 1M people worldwide build their careers in tech. This takes a dedicated team.</p>

    <!-- section>h2+ul>li*4 -->
    <section>
      <h2>How we teach</h2>

      <ul>
        <li>Employed 3,000+ graduates</li>
        <li>20% theory – 80% practice</li>
        <li>100% online</li>
        <li>1:1 mentor support</li>
      </ul>
    </section>

    <!-- section>h2+article*3>h3+p -->
    <section>
      <h2>Our Courses</h2>

      <article class="course">
        <h3>Front-end Developer</h3>
        <p>HTML/CSS, JavaScript, TypeScript, Web, Git, React/Redux, Vue.js, Angular,
and algorithms</p>
      </article>

      <article class="course">
        <h3>Software Tester (QA)</h3>
        <p>Test documentation, Jira, TestRail, HTTPS, Postman, mobile testing, SQL,
Git, JavaScript, and Cypress</p>
      </article>

      <article class="course">
        <h3>UI/UX Designer</h3>
        <p>Figma, prototyping, customer interviews, mobile apps, CRM, and e-
commerce</p>
      </article>
    </section>
  </main>

  <footer class="footer">
    © Copyright 2023,  Mate academy
  </footer>
</body>
</html>
```

Horizontal Element Placement

Elements in the site's header are often placed horizontally, with the company’s logo on the left, and navigation on the right. Like so:

```
<header class="header">
  <a href="index.html" class="logo">
    <img
      class="logo__img"
      src="https://mate.academy/static/favicon/apple-touch-icon.png"
      alt="mate academy logo"
    />
  </a>

  <nav class="nav">
    <a class="nav__link" href="index.html">Home</a>
    <a class="nav__link" href="courses.html">Courses</a>
    <a class="nav__link" href="about.html">About</a>
    <a class="nav__link" href="contacts.html">Contact us</a>
  </nav>
</header>
```

Using `display: flex` to style the `header` is the most convenient option to align all the elements here, but we’ll explain why exactly in the CSS Advanced module. For now, let’s just take it at face value:

```
html {
  font-family: Arial, Helvetica, sans-serif;
}

body {
  margin: 0;
}

.header {
  display: flex; /* elements are laid out horizontally */
  justify-content: space-between; /* free space will be between elements */
  align-items: center; /* vertically nested elements are centered */

  padding: 10px 20px; /* adds space around the nested elements */
  background-color: #000;
}

.logo__img {
  display: block; /* removes free space from under the image */
  width: 40px;
  height: 40px;
}

.nav__link {
  margin-left: 40px; /* distance between navigation links */
  color: #fff;
  text-decoration: none;
}
```

Navigation List for Improved Accessibility

We can wrap navigation links in an unordered list to let screen readers announce how many elements there are:

```
<nav class="nav">
  <ul class="nav__list">
    <li class="nav__item">
      <a class="nav__link" href="index.html">Home</a>
    </li>
    <li class="nav__item">
      <a class="nav__link" href="courses.html">Courses</a>
    </li>
    <li class="nav__item">
      <a class="nav__link" href="about.html">About</a>
    </li>
    <li class="nav__item">
      <a class="nav__link" href="contacts.html">Contact us</a>
    </li>
  </ul>
</nav>
```

To align the list items horizontally, we'll again use `display: flex`:

```
.nav__list {
  display: flex; /* places items in a row */
  gap: 20px; /* sets a space between items */

  /* reset the default list styles */
  margin: 0;
  padding: 0;
  list-style: none;
}

.nav__link {
  display: block; /* makes it possible to set dimensions for the link */
  padding: 0 10px; /* these paddings are a part of the clickable area */

  /* need to remove padding in .header, so links occupy the full height */
  /* then it's possible to click not only on the text but also above and below it */
  line-height: 60px; /* text will be centered within the line */
  color: #fff;
  text-decoration: none;
  background-color: #444;
}
```

Responsiveness Basics

Websites are displayed on *different screens,* and the thing about screens: they come in very many sizes, from a 5” smartphone to a 100” TV. Your website must display well on them all, so you need to format it accordingly — which is what we’ll discuss today.

Maximum and Minimum Element Sizes

To account for particularly small and large screens, we can use the `max-width`, `min-width`, `max-height`, and `min-height` properties. As names imply, they let us set maximum and minimum dimensions of any element — both in `px` and `%`:

```
.box {
  min-width: 320px;
  max-width: 960px;
  width: 80%;

  margin: 0 auto;
  padding: 20px;
}
```

Here, the minimum width of the `.box` is `320px` + `20px`. It will grow proportionally to the parent element (80% of the parent’s width) until the `.box` reaches its maximum width of `960px` + `20px`. Thanks to `margin: 0 auto`, the element is always centered in the parent container. And importantly: `height` in `% works only if the container's height is set *explicitly.*

Viewport

Screen real estate is inherently limited, not only by the size of the device, but also, for example, by the browser’s UI displayed above your website. If we removed all the surrounding bits — like tabs, the address bar, bookmarks — **we’d be left with the viewport.**

And we have a good reason to mention this. You see, we can actually set the size of elements in relation to the viewport using two units, `vw` (short for the viewport’s `width`) and `vh` (the viewport’s `height`). Both are expressed in `%`:

```
.container {
  width: 100vw;
  height: 100vh;
}
```

CSS Variables

CSS lets us create **custom properties,** otherwise known as CSS variables, to save us from duplicating values over and over again. We create such variables with the -- prefix:

```
body {
  --main-color: #008000;
}
```

…and can use them with the `var()` function:

```
.container {
  color: var(--main-color);
}
```

One particularly good CSS variable application are **global page parameters,** such as sizes and colors, since we can define them once and… proceed. And as a rule of thumb, wherever values are duplicated, they should be made into variables.

⚠ If two values are the same at present, but differ in design logic, you shouldn’t handle them with a single variable.

`calc()`

The `calc()` function lets us calculate the element sizes with arithmetic operations, which makes it particularly useful for creating dynamic, adaptive pages. Especially if different units are at play:

```
.container {
  width: calc(100% - 50px);
  height: calc(50vh - 10px);
}
```

Adapting Content by Height

Consider a page with a fixed-height header and footer:

```
<header class="header"></header>

<main class="main">
  <p>Lorem ipsum dolor sit amet consectetur, adipisicing elit. Sed sequi praesentium facere magnam minus possimus doloremque eum harum placeat quibusdam.</p>
  <p>Voluptate accusamus beatae commodi porro! Non facilis dolor fugit. Eveniet
expedita, maxime natus adipisci harum eius inventore ad iure amet?</p>
</main>

<footer class="footer"></footer>
html {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 24px;
  line-height: 1.5;
}

body {
  margin: 0;
}

.header {
  height: 60px;
  background-color: #0057b7;
}

.main {
  padding: 10px 20px;
}

.footer {
  height: 120px;
  background-color: #ffd700;
}
```

Currently, if there isn't much text on the page, the footer might "detach" from the bottom edge of the page, which looks just… awful. We need to fix it, and now that we know the `calc()` function — we can! Let's calculate the height of `.main` with `vh`:

```
.main {
  box-sizing: border-box; // so padding does not increase the height
  padding: 10px 20px;
  height: calc(100vh - 60px - 120px);
}
```

Since we don't want to duplicate the sizes of `.header` and `.footer`, declaring a CSS variable is a logical next step. Here it is:

```
html {
  --footer-height: 120px;
  --header-height: 60px;
}

.header {
  height: var(--header-height);
  background-color: #0057b7;
}

.main {
  box-sizing: border-box;
  min-height: calc(100vh - var(--footer-height) - var(--header-height));
  padding: 10px 20px;
}

.footer {
  height: var(--footer-height);
  background-color: #ffd700;
}
```

By this point, we’ve standardized the sizes of `.header` and `.footer`. Everything is stored in one place, and our design gets a pass!

CSS Selectors

Selectors are strings that CSS uses to **find elements and apply rules to them,** which makes them *pretty important* for any web developer. So today, we’ll discover the main selector types and ways to combine them.

Simple Selectors

Selecting *all* elements of a specific type is the simplest. Just write its name (tag), e.g.:

- `a` selects all links
- `p` selects all paragraphs
- `section` selects all sections
- `*` selects all, and we mean *all* elements

Class selectors begin with `.` and select all elements with the appropriate CSS class. `.box`, for example, selects all the following elements:

- `<div class="box">`
- `<section class="box active">`
- `<h2 class="title box is-last">`

ID selectors, on the other hand, start with `#` and select all elements with the appropriate `id`. `#about-us`, for example, selects only the element with `id="about-us"`.

💡 We use class selectors the most since they let us select elements of different types.

Attribute Selection

`[]` selects elements by attributes. Here, we select all elements with a `class` attribute:

- `[class]` with any value (`<a class>`, `<p class="">`, `<div class="one two">`, ...)
- `[class="box"]` with an exact value (only `class="box"`, but not `="box1"` or `="box other"`)
- `[class*="box"]` contains a given value, either standalone or as a part of a larger value (`"box"`, `"box1"`, `"12box34"`, `"one box two"`)
- `[class~="box"]` contains a given value, only standalone (`"box"`, `"box other"`, `"one box two"`, but not `"box1"`)
- `[class^="box"]` starts with a given value (`"box"`, `"box1"`, but not `" box"`)
- `[class$="box"]` ends with a given value
- `[class|="box"]` equals `"box"` or starts with `"box-"`

Same can be done for any attribute,
e.g. `[href^="https://"]`, `[alt]`, `[type="checkbox"]`.

Combining Selectors

We can combine selectors in a myriad of ways:

- Writing selectors together, like `.container.block`, picks elements with both `container` and `block` classes.
- Using `,` between selectors, such as `.container, .block`, chooses elements with either `container` or `block` class.
- A space signifies nesting; `.container .block` finds `.block` within `.container`.
- `>` targets direct children; `.container > .block` selects `.block` directly inside `.container`.
- `+` picks the next sibling; `.container + .block` selects the `.block` right after `.container`.
- `~` selects all following siblings; `.container ~ .block` chooses all `.block` after `.container`.
- `:not()` filters out matches; `p:not(.block)` finds paragraphs without `.block`.
- `:has()` finds elements containing others; `section:has(a.nav__link)` selects sections with a `.nav__link` inside.

If desired, we can *combine the combinators* as well:

- `header a[href^="https://"]` selects all external links located in the `header` element.
- `.container > * > p` selects all paragraphs that are second-level descendants in an element with the `container` class.
- `* + *` selects all elements that are not the first children in their container.

Pseudo-Elements and Pseudo-Classes

Sometimes we need to style elements, or element parts, depending on some conditions. The solution? pseudo-elements and pseudo-classes, which we'll discuss today.

Pseudo-Elements

Pseudo-elements, denoted in CSS by two colons `::`, let us add extra styling to specific element parts that we cannot select with regular selectors. For example, we use `::before` to add content *before* an element, while `::after` — *after* an element. Both require the `content` property, even if only an empty string:

```
.box::before {
  content: "🌴";
  color: green;
}

.box::after {
  content: "";
  display: block;
  height: 40px;
  width: 200px;
  background-color: blue;
}
```

`::selection` lets us style text selected by users:

```
.text::selection {
  background-color: yellow;
}
```

`::first-letter` and `::first-line` style the first letter or line of the element’s content, respectively:

```
.content::first-letter {
  font-size: 30px;
}

.content::first-line {
  color: red;
}
```

Pseudo-Classes

Pseudo-classes, denoted by a single colon `:`, target specific states or conditions of an element. In other words, we can apply styles based on the element’s relationship to another element, user interactions, or other conditions. For example, `:hover` lets us style an element in reaction to the user’s mouse hovering over it, which is the usual treatment for links, buttons etc.:

```
.link:hover {
  background-color: blue;
  color: white;
}
```

`:first-child` is used to select and style elements of the element’s first child, while `:last-child` — the last child. If we’d like to select the in-between childs (second, fifth, seventh, etc.), we need to use the `:nth-child(an + b)` formula, which selects all matching elements:

- `:nth-child(2)` — the parent’s second child
- `:nth-child(n+3)` — all children from the third onwards (`3, 4, 5, 6 …`)
- `:nth-child(2n)` — all children at even indexes (`2, 4, 6 …`)
- `:nth-child(3n + 1)` — every third child from the first onwards (`1, 4, 7, 10 …`)
- `:nth-child(-n + 3)` — the first three children (`1, 2, 3 …`)

Alternatively, CSS equips us with three pseudo-classes that select child elements **based on their type,** namely `:first-of-type`, `:last-of-type`, and `:nth-of-type()`. Here they are:

```
/* Makes the first paragraph in any group stand out in red */
p:first-of-type {
  color: red;
}

/* Colors the last of each kind of item (like the last paragraph, link, box, etc.) in blue */
:last-of-type {
  color: blue;
}
```

```css
/* Changes the color of every other link to green, starting with the second one */
a:nth-of-type(2n) {
  color: green;
}
```

`:not()` lets us select elements that don’t match a specific selector:

```css
/* select all elements that AREN’T paragraphs */
:not(p) {
  color: blue;
}

/* in the element with the `container` class, select all paragraphs WITHOUT the `block` class */
.container p:not(.block) {
  color: red;
}
```

🎮 Want to understand selectors and pseudo-classes better, but have fun along the way? Play the **selectors game**!

Specificity

Quite often, a single element on a page matches multiple CSS selectors. If the rules set by these selectors assign different values to the same property, the browser must determine which value to apply. To do this, the browser performs the following steps:

- compares the specificity of the selectors to find the most specific one (we'll discuss how shortly);
- applies the value set by the rule with the most specific selector;
- if there are multiple selectors with the highest specificity, the browser applies the value of the last one (in the order they appear in the code).

To find the most specific selector, the browser breaks down each selector into components, each of which falls into one of three groups:

- ID selectors (`#`);
- class selectors (`.`), attribute selectors (`[]`), and pseudo-classes (`:`);
- element selectors and pseudo-elements (`::`).

Next, the browser counts the number of components in each group. This count is the specificity of the selector. For example:

- the selector `#main-nav [href]:visited` has a specificity of `1-2-0` (`1` for ID, `2` for attribute and pseudo-class, `0` for elements);
- the selector `section a.nav__link` has a specificity of `0-1-2` (`0` for ID, `1` for class, `2` for elements).

```css
#main-nav [href]:visited {
  color: red;
}

section a.nav__link {
  color: blue;
  background-color: yellow;
}
```

In this example, the text will be red because the first selector is more specific, and the background will be yellow since the first rule does not set a background color at all.

Note: The pseudo-classes `:not()`, `:is()`, and `:has()` are not counted. Instead, every selector added to them is counted.

Inline Styles and `!important`

Styles added to an element with the `style` attribute have higher specificity than all styles added from CSS. In the following example, the text will be green:

```
<h1 class="title" style="color: green">
  Mate academy
</h1>
.title {
  color: red;
}
```

However, CSS has the `!important` directive, which ignores specificity and applies the value regardless of specificity:

```
<h1 class="title" style="color: green">
  Mate academy
</h1>
h1 {
  color: blue !important;
}

.title {
  color: red;
}
```

The text in the header will be blue because the `!important` directive has been added to it.

In practice, using the `!important` directive is a very bad idea because it breaks the existing rules for applying styles and creates a lot of problems with maintaining such code. Therefore, it should only be used in extreme cases, clearly understanding why it is necessary at the moment. It is also advisable to add a comment explaining the reason.

URLs and Links

Take the UA version and translate using GPT (I had some troubles with it)

A **URL** (Uniform Resource Locator) is a unique address that identifies a resource on the internet, such as a web page, an image, a video, or any other type of file or resource. URLs are used to locate and access resources on the internet, and they follow a standardized format that consists of several components:

- **Protocol** specifies the resource's access method, such as HTTP, HTTPS, FTP, or others. The protocol is written in lowercase letters and is followed by `://`. For example, `http://` and `https://` are the protocols used for accessing web resources.
- **Domain name** identifies the host server that the resource is located on. It consists of two or more parts separated by dots, such as `example.com`, and ends with a top-level domain (TLD), such as `.com`, `.org`, `.net`, or others.
- **Port** is a number used to identify a specific process or service running on a server. It can be added after a hostname and is preceded by `:`. If not specified,

the web browser will use `80` for HTTP requests and `443` for HTTPS requests (`https://example.com:443`).

- **Path** specifies the location of the resource within the domain. It comes after the **domain name** and is separated by a forward slash `/`. For example, in the `https://example.com:443/users/1/contacts.html`, the path is `/users/1/contacts.html`.
- **Query parameters**, also known as **URL search parameters**, provide additional information about a resource, such as search terms or parameters for dynamic web pages. They come after the path in a URL and are separated from the path by a question mark `?`. Each parameter is made up of a name and a value separated by an equal sign `=`. Multiple parameters are separated by an ampersand `&`. For example, in the `https://example.com/page.html?type=apple&page=2`, `type` and `page` are the parameter names, and `apple` and `2` are the corresponding values.
- **Anchor** or **hash** is the optional last part of a URL that links to a specific section on a page. It is indicated by the hash symbol `#` followed by an identifier for the target section. The anchor is not sent to the server but is used by the client-side browser to navigate to the section on the same page with the ID specified in the hash. For example, in the `https://example.com/page.html#footer`, the browser scrolls to the element with `id="footer"`.

Links in HTML

A **link** is an element on a web page that, when clicked, directs the user to another resource by following the URL specified in the `href` attribute:

```
<a href="https://example.com">Example link</a>
<a href="#section">Go to the section</a>
```

The `href` attribute can also contain a relative path, which is a path relative to the page's current location. For example, if you are currently on the `https://example.com:443/about/contacts/index.html` page, the following links will result in different URLs:

- leading `/` means you want to navigate from the website root. For example, `/page.html` links you to `https://example.com:443/page.html`;
- leading `./` means you navigate from the last `/` in the path. For example, `./page.html` links you to `https://example.com:443/about/contacts/page.html`;
- not having a leading symbol means you navigate from the current location. For example, `page.html` works the same as `./page.html`;
- leading `../` goes up one level (to the parent directory). For example, `../page.html` links you to `https://example.com:443/about/page.html`;
- several leading `../` allow you to go up the same number of levels. For example, `../../page.html` links you to `https://example.com:443/page.html`.

Styling Links

The `:link`, `:hover`, `:active`, and `:visited` pseudo-classes are used to target links in different states:

- `:link` targets links that have not been visited or interacted with in any way;
- `:visited` targets visited links;
- `:hover` targets links that are being hovered by the user;

- `:active` targets links being clicked (for example, when the mouse button is pressed but not released yet).

Please note: using pseudo-classes in CSS, you should follow the `:link` -> `:visited` -> `:hover` -> `:active` order. An easy way to remember this order is with the **LoVe HaTe** acronym. This order is important because it determines how styles are applied to the link when users interact with it:

```
.link:link {
  color: blue;
  text-decoration: none;
}

.link:visited {
  color: purple;
}

.link:hover {
  color: green;
  text-decoration: underline;
}

.link:active {
  color: yellow;
}
```

Images

Until now, we've been focusing only on text, and text-only websites are a little *monotonous.* So today, we'll learn how to add some flair with images ✨

Raster and Vector Images

There are two image types, raster and vector. **Raster images** are most often used for photos and/or complex graphic elements, since they consist of a densely-packed pixel grid, where each pixel is individually colored. Hence, they are best to convey fine details.

Whilst the number of pixels (resolution) isn't the only measure of quality, it's a **limiting factor,** since upscaling a raster image always results in a sharpness loss. And we can't change the number of pixels, because it's predetermined by the original file.

💡 Raster images are often stored as `JPEG`, `PNG`, and `GIF` files.

Vector images, on the other hand, are the better choice for icons, logos, diagrams and the like geometry-based visuals. That's because vectors consist of mathematical equations rather than pixels, which makes them **lightweight and infinitely scalable.**

💡 Vectors are often stored as `SVG`, `AI`, and `EPS` files.

`<img>` VS. `background-image`

When adding images to a webpage, you have two main options: the `<img>` element and the background-image CSS property. Each has its own advantages and applications, which can greatly impact your site's functionality.

Use the `<img>` tag when:

- The image is a key part of the content, like a product photo, a picture of a person, or an item being discussed.
- You want the image to be found by search engines (SEO).
- It's important for the image to be accessible to screen readers and other assistive technologies.

Use the background-image property when:

- The image is meant to be decorative, such as a background texture.
- The image contributes to the site's design and changing it won’t affect the content's meaning.
- You need the image to repeat, tile, or stretch to cover a specific area of the site.

We'll return to background images later in this lesson.

Image Position and Aspect Ratio

The `<img>` tag displays images — as well as videos and iframes — in their original size, which might not always go well with the layout. Adjusting the `width` property alone will automatically change the `height` to maintain the image’s aspect ratio, while adjusting both dimensions can lead to distortion. That’s why we control the size with two CSS properties:

1. `object-fit`, which lets us determine how an image should resize to fit its container:

- `fill` makes the image stretch to fill the container completely, which might distort it.
- `contain` makes the image fit in the container without cropping it, and keeps the proportions intact, which can result in empty space showing.
- `cover` scales the image to cover the container and keeps the original aspect ratio, even if it results in cropping.

2. `object-position`, which lets us position the image within its container. It accepts:

- Horizontal positions: left, center, or right.
- Vertical positions: top, center, or bottom.
- Exact positions using units like `px` or `%`, specifying horizontal and vertical offsets from the container's top left corner.

Below code, for instance, makes the image cover the container and keeps the original aspect ratio (with cropping allowed). It’s positioned `20px` from the left and aligned to the top of the container:

css``` Copy code img { object-fit: cover; object-position: 20px top; }

```
## Background Images

We work with background images using three properties, one of which is
`background-image`, which sets the URL for an image as the background. By default,
the image keeps its original size and repeats across the element's entire area, a
little like bathroom tiles do. We can stop this repetition by setting the
`background-repeat` property to `no-repeat`.

[`background-size`](developer.mozilla.org/en-US/docs/Web/CSS/background-size), on
the other hand, determines the background image's size. We can use options like
```

`cover` or `contain` (both work identically as with the `object-fit` property), or specific numerical values.

Last of the three, the `background-position` property, defines the background image's placement within its container, with options that mirror those of the `object-position` property. For example:

```css
section {
  background-image: url("page-background.jpg");
  background-repeat: no-repeat;
  background-size: cover;
  background-position: center top;
}
```

Here, `page-background.jpg` serves as the section's background, and it doesn’t repeat or scale, covering the element completely. The image is centered horizontally and aligned to the top edge vertically. Should the section's width exceed the image's proportions, the bottom will be trimmed, and if it exceeds vertically beyond the image’s length, the sides will be symmetrically cropped.

Floating Elements and the `clear` Property

Occasionally, we might want to embed an image within a block of text, so the text flows around the image. This effect can be accomplished with the `float` property. It specifies the element’s position — to the left or right of its container — and permits surrounding content to wrap around:

```
<div class="text-with-image">
  <img class="float-left" src="https://placehold.co/150x100" alt="Example Image">


  <p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed eget nulla non eros
    commodo ullamcorper. Fusce nec turpis nec elit interdum hendrerit. Donec vel magna
eros.
  </p>
</div>
.float-left {
  float: left;
  margin: 0 20px 20px 0;
}
```

Here, the image is aligned to the left edge, with margins added to the right and bottom edges of the image to create space between the image and the text. This lets the text wrap around the image smoothly. But *but* **but,** we achieved this goal with `float`, which often causes child elements to fall out of the container. We can fix it with the `clear` property:

```
<div class="text-with-image">
  <img class="float-left" src="https://placehold.co/150x100" alt="Example Image">


  <p>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed eget nulla non eros
    commodo ullamcorper. Fusce nec turpis nec elit interdum hendrerit. Donec vel magna
eros.
  </p>


  <div style="clear: both;"></div>
```

```
</div>
.float-left {
  float: left;
  margin: 0 20px 20px 0;
}

.text-with-image {
  border: 1px solid black;
  padding: 20px;
}
```

If there’s little text, `div` with `clear: both` prevents the image from falling out of the container. Other than adding a separate element, though, we can use the `clearfix` technique and add the following class to the container itself:

```
.clearfix::after {
  content: "";
  display: block;
  clear: both;
}
```

The pseudo-element `::after` ensures that all `float` descendants stay within the container:

```
<p class="clearfix">
  <img class="float-left" src="https://placehold.co/150x100" alt="Example Image">

  Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed eget nulla non eros
  commodo ullamcorper. Fusce nec turpis nec elit interdum hendrerit. Donec vel magna
eros.
</p>
```

Responsive Design Basics

Think of all the devices you own: a laptop or desktop, smartphone, perhaps a TV and tablet, too, each equipped with a different screen (or screens!). As a developer, you’ll have to prove your websites for any screen size — which is why you need to learn responsive design. Let’s cover the basics, first 👦💻

Viewport

Browsing through the Internet, you see the page content itself, then above — bookmarks, then above — the URL address, then above — tabs… Now ignore all those extra elements and focus just on the page. That’s your viewport, where the website is rendered. We can control its size and scale using the `<meta name="viewport">` tag, which is added in the `<head>` of the page:

```
<meta name="viewport" content="width=device-width, initial-scale=1" />
```

…where:

- `width` controls the window width. If you’d like to use the width set by the operating system on the end user’s device, use the value `device-width`.
- `initial-scale` controls the zoom level when the page is first loaded (100% by default).

💡 If the viewport isn’t explicitly set, the default behavior can vary significantly across different devices and browsers.

Media Queries

Media queries let us style the website differently depending on some conditions, most often — screen sizes, which makes for a truly responsive design. To add a query, use this CSS structure:

```
@media [type] ([condition]) {
  /* Your styles! */
}
```

…where:

- `type` is the device type, which can be `screen`, `print`, `speech`, or `all`.
- `condition` is the condition under which the style is applied, inter alia, `min-width`, `orientation`, or `resolution`.

In the example below, padding of `20px` is added to elements with the class `box` only on devices with a screen width of at least `768px`. On devices where the browser window size may differ from the device size (like laptops), we opt for the **window size** instead:

```
@media screen and (min-width: 768px) {
  .box {
    padding: 20px;
  }
}
```

Rules are most often applied to all devices, which lets us omit the `[type]`:

```
@media (max-width: 1440px) {
  .box1 {
    padding: 20px;
  }

  .box2 {
    margin: 30px;
  }
}
```

If we need to apply multiple rules at once, we can combine them with `and`:

```
@media (min-width: 768px) and (max-width: 1023px) {
  /* One or several CSS rules */
}
```

In modern browsers, the same can be expressed like so:

```
@media (width >= 768px) and (width <= 1023px) {
```

Or even — like so:

```
@media (768 <= width <= 1023px) {
```

💡 Media queries don't make the rules inside them more specific; they apply conditions for those rules. These rules then compete based on specificity and order as usual.

Due to the ever-growing popularity of small, mobile devices, the **mobile-first** approach has dominated responsive designs. This means we design websites starting with the smallest screens, where design is usually the simplest — elements are stacked vertically, stretched to full width… etc. Only then do we add media queries for larger screens.

The `<picture>` Element

To speed up page loading, we need to minimize the size of resources loaded, and images are one of the bulkiest components out there. Immediate conclusion: we have to cut the resolution, which will save us a few precious megabytes, right?

Well, yes, but while we can get away with smaller images on smaller screens, it's not the case with 4K TVs, but hey — we'll optimize for some devices at least. To set a different image size depending on the screen size, we use the `<picture>` element:

```
<picture>
  <source
    media="(min-width: 800px)"
    srcset="https://placehold.co/1200x600/ff0/00f.png"
  >
  <source
    media="(min-width: 600px)"
    srcset="https://placehold.co/400x200/ff0/00f.png"
  >
  <img
    class="picture"
    alt="Image description"
    src="https://placehold.co/300x150/ff0/00f.png"
  >
</picture>
.picture {
  box-sizing: border-box;
  width: 100%;
  border: 10px solid #000;
}
```

If the screen is wider than `800px`, only the first image will load; between `600px` and `800px`, only the second image loads; for narrower screens, the image defined in `<img src="">` is used. Any styles applied to the `<img>` element affect images on all screen sizes, as `<source>` elements merely adjust the `src` value based on screen width.

For background images, we apply different settings with `@media` queries to adapt to various screen sizes. The proportions of background images can be maintained across all widths with the `aspect-ratio` property.

Forms

Websites and applications often contain registration forms, login forms, and contact forms, with input fields, sliders, checkboxes and whatnot. Developers use them to obtain information from users, and since you're about to be a developer… we better learn obtaining 👦💻

Adding a Form to a Page

We add a form to a page with the `<form>` element, whose main attributes are:

- `action` with the URL to which obtained data is sent. By default, if the `action` attribute isn't specified, the URL of the current page is user.
- `method`, which specifies the HTTP method used to send the obtained data. By default, the `GET` method is used, meaning the data is added to the URL as search parameters (e.g., `?name1=value1&name2=value2`). If you're working with large and or/confidential information, it's much better to send it in the request body — using the `POST` method instead.

Here's an example:

```
<form action="http://localhost:3000/api" method="post">
  <!-- Form elements are added here -->
</form>
```

Input Fields (`input`)

We add input fields with the tag `<input>`, whose main attributes are:

- **`type`** — the most important attribute by far. It specifies the input field type, thereby allowing only certain data types.
- `name` — the input field's name, by which the server identifies and reads the entered value.
- `id` — a unique identifier, which lets us link a label (`<label>`) to the input field. We'll explore labels in more detail later down the text.
- `value` — sets the field's initial value.
- `placeholder` — displays a hint text if the input field is empty (for instance: "Enter your name").
- `readonly` — makes the field read-only, thereby preventing users from changing its value.
- `disabled` — disables the field, meaning its value can't be changed and isn't sent to the server.
- `required` — forces users to fill in the field, and only then lets them upload the form.

Here's an example:

```
<input type="text" name="firstname" placeholder="Enter your name">
```

Input Type: Text Field

`text` is the default `input` type, which creates a single-line text field. Its attributes include:

- `maxlength`, which sets the field's maximum character count. For example, `maxlength="50"` limits the text input to `50` characters.
- `minlength`, which sets the field's minimum character count. For example, `minlength="2"` forces the user to input at least `2` characters before the form can be submitted.
- `autocomplete`, which instructs the browser whether to suggest filling in the field based on input history. There are only two possible values, `"on"` and `"off"`.

Here's an example:

```
<input
```

```
  type="text"
  name="lastname"
  placeholder="Enter your last name"
  maxlength="50"
  minlength="2"
  autocomplete="off"
>
```

Input Type: Password

`password` shows the entered characters in an anonymized form, as asterisks `*` or dots `•`, depending on the user's OS and browser. Here's an example:

```
<input name="user-password" type="password" placeholder="Enter your password">
```

Input Type: Email

`email` is designed to accept an email address, and since emails are just text input… you could ask "why bother with yet another input field?". We can think of at least three reasons:

- **Format validation.** Browsers automatically check the entered data to ensure it matches the correct email address format (e.g., **something@example.com**).
- **Quick-access keyboard.** On mobile devices, when an email input field is selected, a special keyboard with quick access to `@` and `.` symbols usually appears.
- **Browser hints.** Some browsers may autocomplete the user's email address.

Here's an example:

```
<input type="email" name="email">
```

Input Type: Number

`number` allows entering only numeric values, which is particularly useful for obtaining the user's age, quantity of goods, or amount of money. Its main attributes are:

- `min`, as in the minimum allowed value.
- `max`, as in the maximum allowed value.
- `step`, as in the step by which the value can be decreased or increased (e.g., `1`).
- `value`, as in the initial value of the field.

Here's an example:

```
<input type="number" name="quantity" min="1" max="100">
```

💡 Browsers that support HTML5 usually provide special features for this field type, such as increase and decrease buttons.

Input Type: Range Slider

`range` creates a slider, which lets users select a value from a predefined range. It's a great fit for adjusting audio volume or screen brightness. Here's an example:

```
<input type="range" min="0" max="100" value="50" step="1">
```

Input Type: Checkbox

`checkbox` creates a checkbox with the value specified by the `value` property, and if checked, the `value` is sent to the server under the `name`. Users can select all values of a given `name`, and they all will be sent to the server. Here’s an example:

```
<form action="http://localhost:3000/api" method="post">
  Choose one or more drinks:
  <div>
    Tea
    <input type="checkbox" value="tea" name="drink">
  </div>

  <div>
    Coffee
    <input type="checkbox" value="coffee" name="drink">
  </div>

  <div>
    Water
    <input type="checkbox" value="water" name="drink">
  </div>
</form>
```

If we add the `checked` attribute, the checkbox can be checked by default:

```
<input
  type="checkbox"
  name="drink"
  value="tea"
  checked
>
```

Input Type: Radio Button

`radio` is a lot like `checkbox`, but lets the user select only one input of a given `name`:

```
<form action="http://localhost:3000/api" method="post">
  Choose a drink:
  <div>
    Tea
    <input type="radio" value="tea" name="drink">
  </div>

  <div>
    Coffee
    <input type="radio" value="coffee" name="drink">
  </div>

  <div>
    Water
    <input type="radio" value="water" name="drink">
  </div>
```

```
</form>
```

If we add the `checked` attribute, the radio button can also be checked by default:

```
<input
  type="radio"
  name="drink"
  value="tea"
  checked
>
```

Input Type: Date Selection

`<input>` with the attributes `type="date"`, `month`, and `week` lets users select a date, either from a calendar or by entering the date in a text field. Here's the breakdown:

1. **`date`** lets users specify the entire date, i.e.: **day, month, and year.**
2. `month` lets users select a **month and year,** without specifying the exact day.
3. `week` lets users select a **week and a year,** without specifying the exact day.

Best of all, date selection is supported by browsers out of the box. No need to rely on third-party JavaScript libraries or custom calendar interfaces.

Hidden Fields

`hidden` is a non-interactive field that usually stores operational information, like the user's role or security tokens, or data collected earlier. It **remains invisible to the user,** but the data itself arrives on the server — here's an example:

```
<input type="hidden" name="role" value="student">
```

Labels

What do we need to do to interact with a field, say, input some text or check a box? Click on it, of course! And labels let us place a clickable caption next to the field, essentially **extending the interactive area beyond the field itself.** It's particularly useful on smaller screens, where fields can be very small, and as such — difficult to click.

We link a label to an element by placing the element inside `<label>`:

```
<label>
  <input type="checkbox" name="remember">
  Remember me
</label>
```

Alternatively, we can specify the element's `id` in `<label for="id">`:

```
<input type="checkbox" id="remember" name="remember">
<label for="remember">Remember me</label>
```

💡 Labels let us dress checkboxes as **switches**, too!

Buttons

Buttons let us perform many action, including:

- `submit`, which sends data to the server.
- `reset`, which resets all form fields to their initial values.

- `button`, which is meant for customization with JavaScript (doesn't have any default behavior).

`submit` is the default button type, but since we don't want any unexpected behavior, it's always better to **specify the type explicitly.** Like shown below, with `<button type="submit">`:

```
<form action="http://localhost:3000/api" method="post">
  <input type="text" name="username" placeholder="Enter your username">
  <input type="password" name="password" placeholder="Enter your password">

  <button type="submit">Send data</button>
</form>
```

💡 Using the `disabled` attribute, we can disallow the user from clicking a button. Until, for instance, all required fields are filled.

Dropdown Lists

We can add a dropdown list with the `<select>` element. Options are specified inside, each with a value (seen by developers only) and a name (seen by the users). Here's an example:

```
<form action="http://localhost:3000/api" method="post">
  <select name="drink">
    <option value="">Please select a drink</option>
    <option value="coffee">Coffee</option>
    <option value="tea">Tea</option>
    <option value="juice">Juice</option>
  </select>
</form>
```

The first option is selected by default, but we can use the `selected` attribute to preselect any other option from the list. It's possible to disable an item with the `disabled` attribute, too, or force the user to select an option and stick by it, **disallowing changes.** For example:

```
<option value="" selected disabled>
  Please select a drink
</option>
```

Want users to select many options? Add the `multiple` attribute to the `<select>` tag, and users can select as much as they want with the `Ctrl` key pressed down (or ⌘ on macOS):

```
<select name="drink" multiple>
  <option value="coffee">Coffee</option>
  <option value="tea">Tea</option>
  <option value="juice">Juice</option>
</select>
```

From the data perspective, `<select>` is analogous to radio buttons, while `<select multiple>` is analogous to multiple checkboxes bearing the same name.

Multiline Text Fields

`<textarea>` lets users enter and view text in a multiline format. It's particularly useful in contact forms and other places, where people tend to elaborate:

```
<textarea name="info">Enter your message here...</textarea>
```

The field's size can be set using the `rows` and `cols` attributes, e.g.:

```
<textarea name="info" rows="5" cols="25">Enter your message here...</textarea>
```

In this example, "Enter your message here…" is the default value — and we're free to change it.

`contenteditable`

`contenteditable` is an HTML attribute that lets users edit a given element. Other than text input, this also allows adding tags for formatting.

Form Element Pseudo-Classes

With pseudo-classes, we can style form elements based on their state:

- `:hover` — an element under the mouse cursor
- `:focus` — an element currently interacted with
- `:active` — an element at the moment of clicking
- `:disabled` — a disabled form element
- `:valid` — a field with a valid value
- `:invalid` — a field with an INVALID value
- `:checked` — a selected checkbox or radio button

…while the `::placeholder` pseudo-element lets us style hints shown in empty fields.

Keyboard Navigation

Keyboard navigation lets users navigate through forms, and such, it's important for people with disabilities and a better UX. There's a variety of keys and key combinations to choose from:

- `Tab` moves us between form elements; pressing the `Tab` key allows moving forward, and pressing `Shift+Tab` moves backward.
- `Space` or `Enter` submits the form or activates a specific element.
- `Up` and `Down` arrows move through dropdown lists and options.
- `Esc` cancels or closes a window/popup.

`Tabindex`

`Tabindex` is an HTML attribute. It uses numerical values to define the order in which elements are highlighted when users navigate with `Tab`. Here are all the possible `tabindex` values:

- **Positive integers,** where the element with the highest value is highlighted first, second highest — second, etc.
- **Zero `0`,** used to highlight the elements in the same order as they appear in the document.
- **Negative integers**, where all such elements are ignored during Tab navigation.

Grouping Form Elements

We use `<fieldset>` and `<legend>` tags to organize form elements. These tags help structure the form and give users a clear idea of how elements relate to each other or to a particular category of information:

- **`<fieldset>`** groups related form elements, with the related elements enclosed within it.
- **`<legend>`**, placed at the beginning of a , offers a concise title or description for the group of form elements it encompasses.

Here's an example:

```
<fieldset>
  <legend>Contact Information</legend>
  <!-- Form elements related to contact information go here -->
</fieldset>
```

Positioning

So far, we've learned how to add elements to a page and customize them, but we still don't know how to arrange them; position them. Worry not — we'll fix this oversight right now.

Relative Positioning

By default, elements are positioned in the "document flow" (`position: static`), where the topmost block element pushes the next block element beneath it, the second topmost… and so on. Same with inline elements, just along the horizontal axis, where the leftmost inline element pushes the next inline element to the right… etc. Sometimes, though, we have to diverge from this so-called `static` arrangement and shift an element without affecting the position of other elements. In such cases, we can use `position: relative` with two properties — `left` and `top`. To demonstrate, let's move `.box` by `10px` to the right and `20px` down from its original position:

```
.box {
  position: relative;
  left: 10px;
  top: 20px;
}
```

And just like that, `.box` becomes a **positioned element** and overlays all other elements.

Absolute Positioning

Unlike elements with relative positioning, `position: absolute` removes the element from the document flow and places it in a separate display layer.

The `top` and `left` values are calculated relative to the nearest **positioned** ancestor (which has a `position` different from the default `static`) or the `body` element if such an ancestor is absent.

Typically, we set `position: relative` on the container relative to which we will perform absolute positioning. If `left` or `top` properties are absent, the element with absolute positioning will maintain its initial position along the corresponding axes.

It's important to know that, unlike block elements, elements with absolute positioning do not stretch to fill their container. Instead, their size is determined by the content or explicitly set dimensions.

Overall, `position: absolute` can be a useful tool for precise positioning of elements on a webpage, for example, a modal's close button, or elements that do not fit into the page's grid.

Fixed Positioning

Some blocks, like navigation bars, stay in place while scrolling the page; their position is fixed **in relation to the browser window.** We can achieve this effect with `position: fixed`, e.g.:

```
.nav {
  position: fixed;
  left: 0;
  right: 0;
  top: 0;
  height: 60px;
}
```

Here, the `.nav` element always stays at the top because we set `top: 0`. It also spans the full width of the page, as we specified `left: 0` and `right: 0`.

Sticky Positioning

Elements with `position: sticky` are a little confusing, because they start off acting like their position is relative, i.e., they stay in the original position — no offset. As the user starts scrolling, the distance between the element’s top edge and the website’s top edge decreases. And when it hits a predetermined value (“top value”), the element’s behavior changes to `position: fixed`.

Sticky elements aren’t a part of the normal document flow, meaning their occupied area is preserved, meaning — other elements don’t move around. Simultaneously, they’re confined in their parent container, so once the user scrolls over to the succeeding container, **the sticky element stays behind.** Oh, and sticky elements naturally expand to the container’s full width.

Here’s how we can use `position: sticky;` for sticky subheadings:

```
<main class="main">
  <h1 class="title">Mate academy</h1>

  <section class="section" id="section-1">
    <h2 class="section__title">Section 1</h2>
  </section>

  <section class="section" id="section-2">
    <h2 class="section__title">Section 2</h2>
  </section>

  <section class="section" id="section-3">
    <h2 class="section__title">Section 3</h2>
  </section>

  <section class="section" id="section-4">
    <h2 class="section__title">Section 4</h2>
  </section>
</main>
html {
  font-family: Arial, "Helvetica Neue", Helvetica, sans-serif;
  color: #333a4a;
}

body {
  margin: 0;
  background-color: #f5f6f1;
}

.main {
  padding: 0 24px;
}

.section {
  height: 250px;
  margin: 24px 0;
  padding: 16px;
  background-color: #fff;
  border: 1px solid #ced2ed;
  border-radius: 8px;
}

.section__title {
  position

: sticky;
  top: 10px;
  margin: 0 0 16px;
}
```

In this example, the section’s header will temporarily "stick" to the page’s top, maintaining the distance of `10px`. Next, as the following section approaches, it “unsticks” and disappears.

`z-index`

Elements on a page can overlap, especially when they're moved out of their normal place with the `position` property set to anything other than `static`. That’s because positioned elements are moved to different layers by the browser, which stack above the base content layer.

By default, layers stack in the order they are written in HTML, since their `z-index` is set to `auto`. We can swap this value for any integer, where `z-index: 1` elements belong to the first layer, `z-index: 2` elements belong to the second layer, and so on. If there are four `z` values in our HTML, there are four layers. Elements of `z-index: 3` overlay elements of `2, 1 and 0`; elements of `z-index: 2` overlay elements of `1 and 0`, etc. Negative values, on the other hand, move elements below their parent layer or the base content layer.

💡 `z-index` doesn’t work with the elements of `position: static`, including all children of static elements, since they share the same layer as their parents.

Transparency

The `opacity` property adjusts an element's transparency, ranging from `1` (fully opaque) to `0` (fully transparent), with values in between creating partial transparency. For instance, this code makes the `.box` element semi-transparent:

```
.box {
  opacity: 0.5;
}
```

Opacity affects the whole element and everything inside it, so to change only the background's transparency, we can use RGBA colors. For instance, this makes the `.box` background `20%` transparent, keeping the content fully visible.

```
.box {
  background-color: rgba(255, 255, 255, 0.8);
}
```

Even if an element is transparent, it's still there and can be interacted with, such as clicking or hovering over it. To prevent any interaction, we can always apply `pointer-events: none`.

Extras

There are a few things we still haven’t discussed, so without further ado…

Overflow

Block elements naturally extend to fill the width of their parent container and adjust their height based on the content. However, if the content is too large for the container, it **overflows** or — simply put — spills out either downwards or to the right. We can handle this issue with the `overflow` property, which accepts one of four values:

- `visible` — the default option, where overflown content is shown outside the container. This might cause layout issues if it overlaps with other elements.
- `hidden` — overflow content is cut off and NOT shown outside the container.
- `scroll` — introduces scrollbars to the container, allowing users to scroll to see all content. Scrollbars appear even if there's no overflow.
- `auto` — similar to scroll, but scrollbars appear only if the content actually overflows.

If we want more control, `overflow-x` and `overflow-y` properties are the way to go. They let us manage horizontal and vertical overflow separately:

```
.container {
  overflow-x: auto;
  overflow-y: hidden;
}
```

Visually Hidden Content

`display: none` lets us hide an element three times over: from search engines, assistive technologies and the user's eye (visually). It won't occupy any space on the page, either. But if the "all-in-one" solution is a little too much, we can use four other CSS properties:

- `visibility: hidden` — the element is invisible, but still takes up space.
- `opacity: 0` — the element is transparent, but can still be interacted with and takes up space.
- `pointer-events: none` — the element is transparent to the cursor only, meaning hover effects and clicks don't work.

However, there are cases when you need to hide an element visually but still want it to be accessible to screen readers for enhanced accessibility, such as invisible headings or links for easier navigation. In such instances, the `visually-hidden` class can be applied:

```
<h1 class="visually-hidden">
  Mate academy
</h1>
.visually-hidden {
  position: absolute !important;
  width: 1px !important;
  height: 1px !important;
  margin: -1px !important;
  border: 0 !important;
  padding: 0 !important;
  overflow: hidden !important;
  clip: rect(0,0,0,0) !important;
  white-space: nowrap !important;
}
```

Here, we hid the heading visually. It's not present in the design, but kept in HTML to maintain optimal page structure and better semantics.

Special Characters

Some characters, like `<`, `>`, `"`, and `&`, are reserved in HTML and cannot be directly used in the content of a web page — otherwise, we'd be risking errors. The workaround? Special character sequences, also known as character entities, e.g.:

Sequence	Symbol
`&amp;`	&
`&lt;`	<
`&gt;`	>
`&quot;`	"
`&apos;`	'
`&copy;`	©
`&reg;`	®

These sequences are used to display special characters without causing syntax errors or content display issues. For example, instead of entering `<`, which could be interpreted as the start of an HTML tag, `&lt;` should be used.

Shadows

Shadows can significantly improve the visual appeal of a page. The `box-shadow` property, for instance, adds a shadow to a block element, while `text-shadow` — directly to text:

```
h1 {
  text-shadow: 2px 2px 4px #000000;
}
```

Here, we applied a black shadow (`#000000`) with a horizontal offset of `2px`, a vertical offset of `2px`, and a blur radius of `4px`. Doesn't get simpler than that!

Flexbox

Flexboxes let us control the size, order, direction, and alignment of elements along axes, and the distribution of free space separating the elements. We can turn containers into flexboxes by adding to them the `display: flex;` property.

Flexbox Properties

We can modify flexboxes with a few properties:

- `flex-direction` determines the direction in which elements line up.
- `flex-wrap` decides whether elements can spread across more than one row.
- `justify-content` arranges elements across the main direction they flow.
- `align-items` positions elements across the direction perpendicular to their flow.
- `align-content` organizes rows of elements (when they wrap) perpendicularly.

Flex Child Properties

Flex children take another handful of properties:

- `align-self` positions an element perpendicularly to the main flow.
- `flex-grow` lets elements expand to fill extra space in the container. If set above zero, elements grow in proportion to their `flex-grow` value.
- `flex-shrink` controls how much an element shrinks when there's not enough space. By default, elements shrink at a standard rate (`1`), but setting it to `0` stops them from shrinking.
- `flex-basis` defines an element's initial size along the main direction before adjustments by flexbox. Its default size is to fit its content without breaking lines.
- `order` assigns a number (default is `0`) that determines an element's position. Elements with higher numbers follow those with lower ones. Elements sharing a number stay in their original sequence.

Read the **Complete Guide** for more information on the matter. The useful links are at your disposal, too!

BEM Methodology

Block, Element, Modifier — BEM for short — is a web development methodology, which divides the user interface (UI) into reusable components. Hence, it simplifies the development process of websites and web apps.

Main Concepts

Moving top to bottom, **block** is an independent and self-contained part of the UI, represented by a CSS class that conveys the block's purpose:

```
<nav class="main-navigation">
  Some content
</nav>
```

Element is a part of the block, and as such, **cannot be used outside of said block.** Its `class` must follow the `block-name__element-name` pattern, e.g.:

```
<nav class="main-navigation">
  <ul class="main-navigation__list">
    <li class="main-navigation__item">
      ...
    </li>


    ...
  </ul>


  <input class="main-navigation__search">
</nav>
```

Modifier is a CSS class. It represent the state of a block or an element, it's always used with the main class and should follow one of the two patterns:

- `block-or-element-name--modifier-name`
- `block-or-element-name--modifier-name--modifier-value`

We will use `--` as a separator, but it's a matter of choice, rather than a strict recommendation. There are other **BEM naming conventions**, e.g.:

```
<nav class="main-navigation main-navigation--theme--dark main-navigation--mobile">
  <ul class="main-navigation__list">
    <li class="main-navigation__item main-navigation__item--active">
      ...
    </li>


    ...
  </ul>


  <input class="main-navigation__search main-navigation__search--mode--compact">
</nav>
```

💡 For more, **read the Quick Start Guide**.

Typical BEM Mistakes

Like most things, BEM comes with its own set of pitfalls. We'd like to save you the trouble, so let's put the most frequently made errors on display.

HTML-Bound Errors

1. **Incorrect prefixes in elements.** Instead of assigning an element to a block (as intended), we end up assigning an element to another element (which is incorrect):

```
<div class="example">
  <ul class="example__list">

    <!-- Wrong -->
    <li class="example__list__item">...</li>

    <!-- Correct -->
    <li class="example__item">...</li>

  </ul>
</div>
```

2. **Incorrect element affiliation** — the element's name must contain the name of its block:

```
<!-- Wrong -->
<ul class="menu">
  <li class="item">
    Only if it's not a standalone block
  </li>
</ul>

<!-- Correct -->
<ul class="menu">
  <li class="menu__item">...</li>
</ul>
```

3. **Mistaking underscores for dashes.** While double dash is used to separate a block name from the block's modifier, double underscore separates the block's name from its element:

```
<!-- Wrong -->
<ul class="menu">
  <li class="menu--item">...</li>
  <li class="menu_item">...</li>
</ul>

<!-- Correct -->
<ul class="menu">
  <li class="menu__item">...</li>
  <li class="menu__item">...</li>
</ul>
```

4. Using a modifier **without the belonging class:**

```
<!-- Wrong -->
<ul class="menu--mobile">
```

```
  <li class="menu__item--active">...</li>
</ul>

<!-- Correct -->
<ul class="menu menu--mobile">
  <li class="menu__item menu__item--active">...</li>
</ul>
```

5. **Using a block modifier on an element** rather than a block:

```
<!-- Wrong -->
<ul class="menu">
  <li class="menu__item menu--active">...</li>
</ul>

<!-- Correct -->
<ul class="menu">
  <li class="menu__item menu__item--active">...</li>
</ul>
```

6. **Using a modifier without a prefix.** Said modifier must be preceded by the element's name (and the same is true for block modifiers):

```
<!-- Wrong -->
<nav class="nav fixed">
  <a class="nav__link active" href="#">
    Wrong
  </a>
</nav>

<!-- Correct -->
<nav class="nav nav--fixed">
  <a class="nav__link nav__link--active" href="#">
    Correct
  </a>
</nav>
```

7. **Using one element inside two blocks.** As stated, an element of the parent block cannot be used inside a child block:

```
<div class="parent">
  <!-- Wrong -->
  <div class="child">
    <p class="parent__element">Text</p>
  </div>

  <!-- Correct -->
  <div class="child parent__element">
    <p class="child__element">Text</p>
  </div>
</div>
```

8. **Using an element outside the block** — we can only place elements inside the block they belong to:

```
<!-- Wrong -->
<div class="block">
```

```
  Content
</div>

<p class="block__element">Text</p>

<!-- Correct -->
<div class="block">
  <p class="block__element">Text</p>
</div>
```

9. Using **many naming conventions** in a single project:

```
<!-- Wrong -->
<div class="ParentBlock ParentBlock_mobile">
  <div class="child-block child-block--active ParentBlock-element"></div>
</div>

<!-- Correct -->
<div class="ParentBlock ParentBlock_mobile">
  <div class="ChildBlock ChildBlock--active ParentBlock-element"></div>
</div>

<!-- Correct -->
<div class="parent-block parent-block--mobile">
  <div class="child-block child-block--active parent-block__element"></div>
</div>
```

CSS-Bound Errors

1. Styling an element **in the context of another element:**

```
<ul class="nav__list">
  <li class="nav__item"></li>
</ul>
/* Wrong */
.nav__list .nav__item {
  padding: 0;
}

/* Correct */
.nav__item {
  padding: 0;
}
```

2. While styles of an element can depend on the block’s state, **they can’t depend on the state of another element:**

```
<ul class="nav__list nav__list--active">
  <li class="nav__item"></li>
</ul>
/* Wrong */
.nav__list--active .nav__item {
  padding: 0;
}
```

```
/* Correct */
.nav--active .nav__link { /* Can be styled based on the state of the block */
  padding: 0;
}

.nav:hover .nav__link {
  padding: 0;
}
<nav class="nav nav--active">
  <a class="nav__link" href="#">1</a>
</nav>
```

3. **Increasing element specificity.** Elements must always be placed inside their blocks in HTML, which means adding block selectors to elements is disallowed:

```
<nav class="nav">
  <ul class="nav__list">...</ul>
</nav>
/* Wrong */
.nav .nav__list {
  padding: 0;
}

/* Correct */
.nav__list {
  padding: 0;
}
```

4. **Increasing modifier specificity.** We shouldn't combine the main class with a modifier in a selector; instead, the modifier should always accompany the main class.

```
/* Wrong */
.burger-menu.burger-menu--active {
  background-color: transparent;
}

/* Correct */
.burger-menu--active {
  background-color: transparent;
}
```

5. Styling a block **in the context of another block:**

```
<div class="parent">
  <div class="child"></div>
</div>
/* Wrong */
.parent .child {
  margin-bottom: 10px;
}

/* Correct */
```

```
.parent__element { /* use mix */
  margin-bottom: 10px;
}
<div class="parent">
  <div class="child parent__element"></div>
</div>
```

6. Setting the block's **external geometry or positioning:**

```
<div class="parent">
  <div class="child">...</div>
</div>
/* Wrong */
.child {
  position: absolute;
  top: 0;
  margin: 10px;
  padding: 10px;
}

/* Correct */
.parent__element { /* use mix */
  position: absolute;
  top: 0;
  margin: 10px;
}

.child {
  padding: 10px;
}
<div class="parent">
  <div class="child parent__element">...</div>
</div>
```

Sass

As we grow our CSS code, we're doomed to increase its complexity. That's why developers use **CSS preprocessors**, with a different syntax — which simplifies the development process — and convert the code to CSS automatically. We'll take a look at the popular **Sass** preprocessor and its `scss` syntax, but if you'd like to check out alternatives in your free time, these include **Less** and **Stylus**.

Setting Up a Project Build with Sass Preprocessor

Since preprocessors use a syntax different to that of *pure* CSS code, browsers cannot read their styles directly — hence the need for conversion. And since, during development, we want the page to refresh every time styles change, style compilation needs to happen automatically.

Let's configure your Sass per said requirements. We need to replace the `Live-server` extension with the **parcel** bundler:

1. Run `npm init -y` to create a `package.json` file, where our project settings will be stored.
2. Install `parcel` with the command `npm i -D parcel`.
3. Create a `src` directory with an `index.html` file containing the basic markup.
4. Create a `./src/styles.scss` file and link it to `index.html` with a `link` tag:
   ```
   <link rel="stylesheet" href="styles.scss">
   ```
5. Start the project by adding the `"start"` command in the `"scripts"` section of the `package.json` file:
   ```
   "start": "parcel ./src/index.html --open"
   ```
6. Run `npm start` so that `parcel` installs necessary packages and opens the page in the browser.

Here's an example of the `package.json` file's contents:

```
{
  "scripts": {
    "start": "parcel ./src/index.html --open",
    "build": "parcel build --public-url=/dist/"
  },
  "devDependencies": {
    "@parcel/transformer-sass": "^2.12.0",
    "parcel": "^2.12.0"
  }
}
```

A similar configuration is already used in all tasks, so no need to adjust anything further 😌

Sass Features

The *SCSS* syntax is an extended version of standard CSS, so we can just rename a CSS file, and everything should look as before. Like any extension, though, it does change a thing or two. So below, we'll compare SCSS with CSS to better understand the preprocessing magic.

Nesting and `&`

Sass lets us nest selectors within one another, which improves the style grouping and saves us duplicating the selector when styling nested elements:

SCSS

```
.block {
  font-size: 20px;

  p {
    color: red;
  }
}
```

CSS

```
.block {
  font-size: 20px;
}

.block p {
  color: red;
}
```

When nesting rules within each other, we can use the `&` prefix, which replaces the selector of the nearest container. In this case, the final selector won't have nesting:

SCSS

```
.block {
```

CSS

```
.block {
```

SCSS

```
  font-size: 20px;

  &__element {
    color: red;

    &--modifier {
      color: blue;
    }
  }
}
```

CSS

```
  font-size: 20px;
}

.block__element {
  color: red;
}

.block__element--modifier {
  color: blue;
}
```

Variables

In addition to CSS variables, which we can change directly in the browser — while the page is running — SCSS has its own variables. Their values are substituted at the compilation time, so that browsers can read them. To declare a variable, we precede its name with `$`:

SCSS

```
$bodyFont: Arial, Helvetica, sans-serif;
$bodyColor: white;

body {
  font-family: $bodyFont;
  color: $bodyColor;
}
```

CSS

```
body {
  font-family: Arial, Helvetica, sans-serif;
  color: white;
}
```

The `#{$varName}` syntax lets us also use variables as parts of property names, selectors, and complex expressions. For example:

SCSS

```
$side: 'top';
$size: 10px;

.block--#{$side} {
  margin-#{$side}: calc(10% - #{$size});
}
```

CSS

```
.block--top {
  margin-top: calc(10% - 10px);
}
```

Expressions

With Sass, we can perform calculations with absolute values (like `px`) at the compilation stage. This eliminates the need for relative `calc` and `var`:

SCSS

```
$screenWidth: 600px;
```

CSS

```
.block {
  width: 1200px;
```

SCSS

```
.block {
  width: $screenWidth * 2;
  height: calc(#{$screenWidth + 10px} +
10%);
}
```

CSS

```
  height: calc(610px +
10%);
}
```

Loops

Loops like `@for` are the remedy for code duplication:

SCSS

```
@for $i from 1 through 3 {
  .block-#{$i} {
    height: 100% / $i;
  }
}
```

CSS

```
.block-1 {
  height: 100%;
}

.block-2 {
  height: 50%;
}

.block-3 {
  height: 33.3333333333%;
}
```

`@each` lets us style each element of a list or each pair in a `map`:

SCSS

```
$colors: (
  'error': #f00,
  'notification': #0f0,
  'success': #00f
);

@each $name, $color in $colors {
  .message--#{$name} {
    color: $color;
  }
}
```

CSS

```
.message--error {
  color: #f00;
}

.message--notification {
  color: #0f0;
}

.message--success {
  color: #00f;
}
```

Functions

Functions allow for the reuse of calculation logic:

SCSS

```
$bodyWidth: 600px;

@function size($width) {
```

CSS

```
.container {
  width: 300px;
}
```

SCSS

```scss
  @return $width / 2;
}

.container {
  width: size($bodyWidth);
}
```

CSS

Mixins

Mixins allow for the reuse of CSS property sets and rule sets. We can create one using the `@mixin` directive and include it in our styles with `@include`:

SCSS

```scss
@mixin circle($size) {
  height: $size;
  width: $size;
  border-radius: 50%;
}

.avatar {
  @include circle(50px);

  font-size: 16px;
  color: white;
}
```

CSS

```css
.avatar {
  height: 50px;
  width: 50px;
  border-radius: 50%;

  font-size: 16px;
  color: white;
}
```

File Inclusion

Usually, projects have a main style file `main.scss` or `index.scss`, to which all files connect:

```scss
@import './utils/mixins';

@import './blocks/header';
@import './blocks/page';
@import './blocks/footer';
```

The @use directive lets us import variables, functions, and mixins from one file into another, enabling their use in the Sass file where `@use` is declared:

```scss
// src/_corners.scss
$radius: 3px;

@mixin rounded {
  border-radius: $radius;
}
// style.scss
@use "src/corners" as c;
```

```
.button {
  @include c.rounded;
  padding: 5px + c.$radius;
}
```

💡 You can find more information on Sass in the **official guide**.

Transitions, Animations, and Transformations

In this lesson, we'll learn how to change element styles.

Transition

By default, all changes in CSS are applied immediately. If we modify, say, the element's height on hover, it will "jump". With transitions, we can make this change smooth:

```
.box {
  height: 100px;
  background-color: #f00;
  transition: height 1s;
}

.box:hover {
  height: 200px;
}
```

A transition is set with the `transition` property. It's a shorthand for the rather lengthy `transition-property transition-duration transition-timing-function transition-delay`, where:

- `transition-property` sets properties that will change smoothly
- `transition-duration` sets the time the transition takes to finish
- `transition-timing-function` sets the smoothness of the transition
- `transition-delay` sets the time to wait before the transition starts

For example:

```
.box {
  transition: height 4s ease-in-out 1s;

  /* is the same as */

  transition-property: height;
  transition-duration: 4s;
  transition-timing-function: ease-in-out;
  transition-delay: 1s;
}
```

If we add `transition-` properties to an element in the `:hover` state, they will only apply when the element is hovered over. Once the cursor is gone, however, these styles will disappear instantly — there will be no smooth transition *back.*

Animation

Animations change CSS styles smoothly, too, but in contrast to transitions, they allow for more than two states (initial and final). Moreover, they can repeat several times with no user action. We create an animation using `@keyframes` with a value `from` (`0%`), `to` (`100%`) and — optionally — one or more intermediate offsets (e.g., 50%).

Here’s an example:

```
@keyframes move {
  0% {
    top: 0;
    left: 0;
  }

  50% {
    top: 10px;
    left: 10px;
  }

  100% {
    top: 50px;
    left: 50px;
  }
}
```

💡 `move` is the animation name, which we provided using the `animation-name` property.

`animation` is shorthand for `animation-duration animation-timing-function animation-delay animation-iteration-count animation-direction animation-fill-mode animation-name`, where:

- `animation-name` sets the animation name provided in the `@keyframes`.
- `animation-duration` sets the time the animation takes to complete.
- `animation-timing-function` controls the speed curve of the animation as it moves through each cycle's duration.
- `animation-delay` sets the time to wait before the animation.
- `animation-iteration-count` sets how many times the animation will be executed.
- `animation-direction` sets the direction in which the animation executes — forwards, backwards, etc.
- `animation-fill-mode` sets how the animation applies styles to its target before and after its execution.

For example:

```
.box {
  animation: move 5s ease 2s 3 reverse;

  /* is the same as */

  animation-name: move;
  animation-duration: 5s;
  animation-timing-function: ease;
  animation-delay: 2s;
  animation-iteration-count: 3;
  animation-direction: reverse;
}
```

Importantly, only CSS properties with a set of continuous values — sizes, colors, transparency, etc. — can change smoothly. With fixed values, such as `position`, `display`, or `visibility`, changes occur instantly. **You can find all animatable CSS properties here**

Transformation

Transformations let us rotate, scale, skew, or translate elements:

- `translate(left, top)` shifts the element by specified distances from its initial position.
- `rotate(angle)` rotates the element clockwise by the given angle.
- `scale(times)` increases or decreases the element by the specified number of times.
- `skew(angle)` skews the element by the given angle (or angles).

For example:

```
.box {
  transform: translate(50px, 50px);
  transform: rotate(180deg);
  transform: scale(2);
  transform: skew(25deg);
}
```

Transformations don’t affect neighboring elements and, just like positioned ones, are drawn in a separate layer. Hence, they require fewer checks from the browser during redraw.

CSS Grids

Grids simplify the design of layouts by dividing a page into rows and columns, without the `float` elements and positioning. We can make an element a grid container by setting `display: grid`:

```
.container {
  display: grid;
}
```

Rows and Columns

To set the sizes of grid rows and columns, use the following properties:

```
.container {
  grid-template-rows: 1fr 2fr 1fr;
  grid-template-columns: 60px 60px 60px;

  /* the shorthand for the `grid-template-rows` and `grid-template-columns` */
  grid-template: 1fr 2fr 1fr / 60px 60px 60px;
}
```

`1fr` is short for *one fraction* of the free space remaining in the container. Per analogy, `1fr 2fr 1fr` would mean `1/4 2/4 1/4` of the available free space. We can also use the `auto` value, which lets us define the size of a row or column based on its content, or the `minimax()` function, which limits the size of a column or row — e.g.:

```
grid-template-columns: minmax(150px, 25%) 1fr; // the left column will take up a quarter of the width but no less than 150px
```

We can avoid repeating sizes with the `repeat()` function, which creates several columns or rows of identical dimensions:

```
grid-template-columns: repeat(12, 1fr); // 12 identical columns
```

`auto-fill` and `auto-fit` values allow for flexible adaptation of the column number to the container’s width, depending on the available space:

- `auto-fill` fills the row with as many columns as possible that meet the set minimum width.
- `auto-fit` works similarly, but when there aren’t enough elements to fill all rows, columns stretch to occupy the entire container’s width.

```
.container {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
}
```

Placement of Grid Items

Grid containers treat all their direct children as grid items, which are by default arranged from left to right and top to bottom, each occupying a single cell. This setup works well for straightforward grids like those displaying products in an online shop.

The `grid-auto-flow` property determines the automatic placement of grid items within the container when their positions aren't manually defined. This feature is especially handy for dynamically sized item sets.

- `row` (default) places items across rows.
- `column` arranges items down columns.
- `dense` fills in any available empty cells, attempting to keep items in their sequential order.

Additionally, the Grid system provides ways to manage the alignment of both rows and columns through the `align-content` and `justify-content` properties, sharing the same options as Flexbox. To align items within their respective rows and columns, we use `align-items` and `justify-items` properties, respectively.
For specific placement of items, we can assign them to exact spots with **grid line numbers.** These numbers refer to the lines that separate rows and columns:

```
.item {
  grid-row-start: 1;
  grid-row-end: 2;
  grid-column-start: 1;
  grid-column-end: 3;

  /* the shorthand for the `grid-row-start grid-row-end` */
  grid-row: 1 / 2;
  /* the shorthand for the `grid-column-start grid-column-end` */
  grid-column: 1 / 3;

  /* the shorthand for the `grid-row-start grid-column-start grid-row-end grid-column-end` */
  grid-area: 1 / 1 / 2 / 3;
}
```

We can also name lines, which makes CSS more understandable and easier to read:

```
.container {
  display: grid;
  grid-template-columns: [start] 1fr [middle] 1fr [end];
}

.item {
  grid-column: start / end;
}
```

The space between rows and columns is called **gap,** and we can add it with:

```
.container {
  row-gap: 50px;
  column-gap: 40px;

  /* the shorthand for the `row-gap column-gap` */
  gap: 50px 40px;
}
```

Creating Complex Layouts with `grid-area`

The `grid-area property` lets us specify both an element's location within a grid and define layouts spanning multiple rows and/or columns. Moreover, by using `grid-template-areas` in the container, we can craft a "template" layout using area names, thereby improving the clarity of the layout’s visual representation:

```
.container {
  display: grid;
  grid-template-areas:
    "header header header header"
    "title title title sidebar"
    "menu content content sidebar"
    "footer footer footer footer";
}

.header { grid-area: header; }
.menu { grid-area: menu; }
.content { grid-area: content; }
.sidebar { grid-area: sidebar; }
.footer { grid-area: footer; }
```

Other Possibilities

These are just a few examples of just how complex *yet* flexible grid-based layouts can be. To find out more, **read this guide to CSS Grid**, and take a look at the useful links!