HTML & CSS Basics - First Summary

We learned a lot about HTML and CSS - two of the most important languages you need to know as a web developer.

Here's a concise summary of all the core features we learned about thus far. Features which you **need to know** because you will **use them all the time**!

HTML Summary

What & Why?

HTML (HyperText Markup Language) is the heart of every webpage. It defines the structure of the webpage and annotates the content to tell the browser what to display and provide extra meaning. Without HTML, text would always be "just text" - there would be no semantic difference between titles, normal text, subtitles etc. You also wouldn't be able to add images or links.

Even though browsers by default add styles for certain HTML elements (e.g. `<h1>` elements are bold and bigger by default), you **don't use HTML for styling purposes!**

Instead, HTML adds semantic meaning to your content - it provides annotations that describe your content. And in the case of certain elements (e.g. `<img>`, `<a>`) it also tells the browser what to display (e.g. show an image) and what to do (e.g. navigate to another page).

Your website is not just viewed by you but it's also parsed by search engine crawlers or presented by assistive technologies like screen readers. That's why describing your content correctly matters a lot!

HTML Element Anatomy

HTML elements are typically made up of an **opening tag** (e.g. `<h1>`), some **content** (e.g. `Hi there!`) and a **closing tag** (e.g. `</h1>`).

HTML elements can also receive **attributes** that allow you to add extra behavior or configuration to the element (it depends on the attribute).

For example, a link (`<a>`) requires a `href` attribute to tell the browser where it should lead to:

```
<p>This leads <a href="https://www.google.com">to Google</a>.</p>
```

There also are some **void elements** though - e.g. `<img>`. Those elements don't need a closing tag since they don't hold any content.

```
<img src="some-image.jpg" alt="A beautiful image of something">
<!-- or -->
<img src="some-image.jpg" alt="A beautiful image of something" />
```

Nesting HTML Elements

One key characteristic of HTML is that you can nest elements into each other - just as you can see it in this example:

```
<ul>
  <li>This leads to <a href="https://www.google.com">Google</a></li>
  <li>Another item</li>
</ul>
```

Here, the `<li>` element is a so-called **child element** of `<ul>` (which is the **parent**). And it even has a child element itself: The link (`<a>`) element (which is a **descendant** of the `<ul>` element therefore - on the other hand, `<ul>` is an **ancestor** of `<a>`).

The second `<li>` ("Another item") would be a **sibling** to the first `<li>` - together, they form the **children** of `<ul>`.

When writing HTML code, you typically build deeply nested structures because most content can only be described accurately if you do combine HTML elements like this. How else would you build a semantically correct list of links?

HTML Structure & Skeleton

There are two big groups of HTML elements:

1. Elements for presenting and describing your page content (e.g. `h1`, `p`, `ul`, `a`, `img` etc.)
2. Elements for describing your overall page and for linking to other required resources (e.g. `title`, `meta`, `link`, `style`)

That's why a properly formatted HTML document should have a "skeleton" that defines two main areas:

1. `<body>` for page content
2. `<head>` for metadata

Therefore, a correct HTML document skeleton (which you should use in every `.html` file you create) should look like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Page Title</title>
    <!-- Other metadata -->
  </head>
  <body>
    <h1>Hi there!</h1>
    <!-- Other page content -->
  </body>
</html>
```

`<!-- .... -->` marks a comment by the way - this comment will not show up on the page when rendered by the browser.

Inline vs Block Elements

You also learned that are two main kinds of content (`<body>`) elements in HTML:

1. Block elements (e.g. `h1`, `p`, `ul`, ...)
2. Inline elements (e.g. `a`, `span`, `img`)

Block elements always reserve the entire width of the screen for themselves (though that can be changed by CSS).

Inline elements on the other hand are - as the name suggests - there to fit "into a line with other elements". It would, for example, be strange if a link (`<a>`) would force a line break because it reserves the entire screen width for itself.

On the other hand, a new paragraph (`<p>`) or title (`<h1>`, `<h2>` etc.) should probably not be squeezed into the same line as some other text.

You can change that behavior via CSS though, in case you need to.

HTML Elements We Know Thus Far

Up to this point, we learned about a lot of important elements already - here's a summary list:

- `<head>` Elements
  - `<meta>`: Can be used to add extra metadata to your page - e.g. a description that could be picked up by search engine crawlers
  - `<title>`: Allows you to define a page title (will be used by search engines but also shows up in the browser tabs)
  - `<style>`: Can be used to define global CSS styles for the page
  - `<link>`: Allows you to link ("connect") your HTML file to some other resource - typically to a `.css` file which's styles will then be applied as global styles to that HTML document
- `<body>`Elements:
  - `<h1>`, `<h2>`, `<h3>` etc: Headings (titles) which you can add to your page. Should be used in order and only one `<h1>` element should be used per HTML document. A `<h2>` element acts as a subtitle, a `<h3>` element is then another subtitle on an even lower level.
  - `<p>`: Can be used to define a regular paragraph of inline content (typically text)
  - `<em>`: Can be used to add emphesis to a word or phrase
  - `<strong>`: Can be used to add strong emphesis to a word or phrase (as if you would be saying it out aloud, strongly emphasing it verbally)
  - `<img>`: Can be used to display an image. You need to set the image source path via the `src` attribute. And you should set some alternative text via the `alt` attribute - this text is displayed if the image can't be loaded or if a user who uses an assistive technology visits your page
  - `<ul>`/`<ol>`: Can be used to render ordered or unordered lists of data. Use this element if you got list data.
  - `<li>`: Used inside of `<ul>` or `<ol>` to define the individual list items of that list.
  - `<header>`: A block element that defines a header - for the entire page or a subsection of the page (more on that later)

- `<footer>`: A block element that defines a footer - for the entire page or a subsection of the page (more on that later)
- `<main>`: Defines the main content of a page - you should not have more than one `<main>` element per HTML document.
- `<section>`: Defines a new section in your document - typically contains a heading (e.g. `<h2>`), though it's not a "must-have"
- `<span>`: A meaningless inline element which can be used to wrap content that should be targeted with CSS styling (e.g. with help of a CSS class or an id)
- `<div>`: A meaningless block element which can be used to wrap content that should be targeted with CSS styling (e.g. with help of a CSS class or an id)

We'll of course see many more elements throughout this course - you can also explore all available elements in the MDN HTML Elements Reference.

CSS Summary

What & Why?

CSS (Cascading Style Sheets) is a language that allows you to define style definitions for your HTML document. These definitions will be picked up by the browser and control how the content will be displayed (e.g. that some text is red, which font size it should have etc.).

Where HTML defines the structure and meaning of your content (and of course also helps with displaying the content in the first place - e.g. with the `<img>` element), CSS is used to then present the content exactly how you want to present it. It adds **no meaning** or annotation - it's really just about how things look.

If oyu want to change text colors, background colors, sizes, spacings, distances, borders, shadows, positions and all these things, you need CSS.

CSS Syntax

There's a broad variety of available CSS **properties** which you can define. All those properties then take different **values** - the concrete values you can assign depend on the property for which you're assigning it.

```
color: #ccc; /* hexadecimal represention of rgb color */
color: rgb(204, 204, 204); /* same colors as above */
font-size: 18px; /* setting a size in device-independent pixels */
```

CSS styles can be defined in different ways, for example "inline" via the `style` attribute:

```
<h1 style="font-size: 20px">Hi there!</h1>
```

But using such "inline styles" is typically not recommended as it comes with various disadvantages. Most importantly, it clutters your HTML code (hard to maintain!), requires lots of duplication and copy and pasting and it also is hard to change: You often need to visit and adjust multiple elements if you want to adjust one single style.

That's why you typically define **global CSS styles** via the `<style>` element or in an external `.css` file which is then imported via a `<link>` element.

When using such global styles, the CSS syntax stays the same but you group your property-value assignments into a so-called **CSS rule** which also a **CSS selector** that defines for which HTML elements your styles should be applied:

```
h1 { /* is applied to ALL <h1> elements */
  font-size: 20px;
}

#some-id { /* is applied to a single HTML element with id="some-id" */
  color: #ccc;
}

.my-link { /* is applied to ALL HTML elments with class="my-link" */
  text-decoration: none;
}
```

You can also combine CSS selectors:

```
h1, p { /* applies the rule to all <h1> and <p> elements */
  color: rgb(204, 204, 204);
}

p a { /* targets all <a> elements that are descendants of a <p> */
  color: red;
}
```

You can also provide specific styles for specific states of HTML elements - with pseudo-selectors:

```
a:hover { /* targets <a> where the user's mouse is hovering over */
  font-weight: bold;
}
```

What Can You Style?

You can really style (almost) anything with CSS. That's why you should use CSS for styling and not use certain HTML elements because you want a certain look.

HTML is there to define the content, provide the structure and meaning, CSS is then there to present your content however you want to present it.

CSS Values & Units

Since there are dozens of CSS properties you can target (we only saw a small subset thus far), there are also hundreds of possible values you can assign.

The concrete values you can assign always depend on the property you're currently defining: Color properties (e.g. `color`, `background-color`) want color values (e.g. `#fa923f`, `#ccc`, `rgb(204, 204, 204)`, `hsl(180, 20, 75)`, `red`). Dimension properties (e.g. `font-size`, `margin`) want dimension values (e.g. `18px`). The `font-family` property wants a list of font families it should use (e.g. `font-family: 'Oswald', sans-serif`).

Even though the number of available properties and values and units can be intimidating right now, you'll develop a good feeling for the different properties and values over time. In the end, you will always have a couple of properties and values that you use extremely often (and therefore know by heart).

Of course you can also always use the IDE auto-completion support and resources like the MDN CSS Properties List to learn about all possible properties and values.

The Box Model

When working with CSS it's important to understand that HTML elements have a so-called "box model".

This means that all HTML elements do have various "layers" that can be styled:

- The content
- A `padding` around the content (but inside of the border)
- A `border`
- A `margin` around the entire element

You don't need to set any `padding`, `border` or `margin` - you can set only what you need. But with these different "layers", you can add spacing between a visible border and the content and / or around an element.

It is worth noting that block and inline elements behave differently though:

- **Block elements**: It behaves as described above
- **Inline elements**: Vertical `margin` is ignored, vertical `padding` is added but does not push other elements or content away

You can also change if an element behaves like a block or inline element via the `display` property (e.g. `display: block` sets the behavior to "block element" even if it normally is an inline element). There also `inline-block` as a possible `display` value which will basically merge the two behaviors and "unlock" full block behavior whilst keeping the content inline with other content.

Why Is It Called "Cascading"?

Why is it called "Cascading Style Sheets"?

Because a cascade of style definitions can affect one and the same element - i.e. multiple CSS rules can affect the same element.

Here's an example:

```
<style>
  a {
    text-decoration: none;
  }

  .default-link {
    color: red;
  }
</style>

<a class="default-link" href="...">Some linke</a>
```

In this example, the two CSS rules are both affecting the `<a>` element.

Why would you have two instead of just one rule? Because you maybe have different links on your page - some with a `default-link` class, some without it.

Because multiple CSS rules can affect the same HTML element, you can split your CSS definitions across multiple rules. This avoid unnecessary copy & pasting and code duplication!

Specificity

But what happens if different CSS rules target the same CSS property?

Consider this example:

```
<style>
  a {
    color: blue;
  }

  .default-link {
    color: red;
  }
</style>

<a class="default-link" href="...">Some linke</a>
```

In this case, the link would be `red` because the order of CSS rules matters - the later rule (in this case `.default-link` wins).

But actually, it's a bit more complex than that. It's not just about the order.

Instead, there is a concept called **specificity** (yes, it's a horrible word!) involved.

The higher the specificity of a rule, the more important it is. And higher specificity beats lower specificity.

The order of your code influences the specificity but it's not the only factor.

The kind of selector you're using also plays an important role - for example, an id selector (like `#some-id`) beats a regular tag type selector (like `h1`).

And that kind of makes sense: After all, if you target a specific id you have a way more specific selection criteria than if you just target the tag type.

And whilst I could now write a long list of specificity factors and which rule wins over which other rule, the comforting thing is that you can basically trust your own common sense.

Just as it makes sense that an id is more specific than "just the tag", a combined selector (e.g. `p a { ... }`) would win over a "standalone" selector (e.g. `a { ... }`).

So the more specific your CSS selectors get, the higher the specificity. And higher specificity beats lower specificity.

Inheritance

Connected to the "Cascading" and "Specificity" concepts is the concept of "Inheritance".

HTML elements are not just affected by styles that are defined in rules directly targeting those elements but instead they can also be styled by rules that target parent or ancestor elements.

Here's an example:

```
<style>
  body {
    font-family: 'Open Sans', sans-serif;
    text-align: center;
  }

  p {
    margin: 20px;
  }
</style>

<body>
  <h1>I use 'Open Sans' as a font-family!</h1>
  <p>So do I! And we're both aligned to the center.</p>
</body>
```

In this example, both `<h1>` and `<p>` would be affected by the `font-family` and `text-align` CSS definitions in the `body` rule.

Why?

Because of "Inheritance".

They inherit the styles defined in the parent element.

Not all CSS properties are inheritable though - but again, you can trust your common sense.

Text color, most font styles etc. are inherited.

Margins, paddings, borders etc. are not.

Browser Defaults

Browsers also define some default styles for certain elements - for example, they often make the `<h1>` element bigger and give it a `bold font-weight`.

This has a very low specificity though and hence you can easily overwrite these default styles with any CSS selector.

CSS Properties We Know Thus far

- Font-related:
  - `font-family:` Set the font family you want to use - can be a single font or (typically) a list of fonts, where the first font is used and the other fonts act as fallback fonts if the first font can't be loaded
  - `font-size`: Sets the text size - can be set in "device independent" pixels (e.g. `18px`) or other values which we'll discover later
  - `font-weight`: Allows you to set the weight of some text (default is `normal`, you can choose a numeric weight like `700` or an alternative like `bold` instead). Important: Your loaded font needs to support that weight!
- Text:
  - `text-align`: Controls the alignment of the text (e.g. `center`, `left`, `right`)
  - `text-decoration`: Can be used to add extra decoration (or remove it) like underlining (e.g. `text-decoration: underline`)
- Colors:
  - `color`: Sets the text color
  - `background-color`: Sets the background color of an element
  - Colors can be set with differnt kinds of units:
    - Hexadecimal number identifiers (e.g. `#fa923f`, `#ccc`) where you got three two-digit pairs that define the r/g/b (red, green, blue) color parts
    - The `rgb()` "function" that allows you to set a r/g/b color with decimal numbers
    - The `hsl()` function which can be used to define a color as a

combination of hue/saturation/lightness
        - You can use either of these color units - it comes down to your personal preference; every color can be expressed with each of these three methods
        - There also is `rgba()` and `hsla()` in case you also want to add an "alpha channel" (transparency - value between 0 and 1) to your color
- Box Model:
    - `margin`: Sets extra spacing **around an element** - could be set with pixels (e.g. `18px`)
        - `margin` is the shorthand notation that sets spacing in all directions, the long-form would be `margin: <top> <right> <bottom> <left>` (e.g. `margin: 10px 5px 8px 3px`)
        - You also have `margin-left`, `margin-right`, `margin-top` and `margin-bottom` to target specific directions
    - `padding`: Adds extra spacing **inside of an element** - could be set with pixels (e.g. `18px`)
        - Just like `margin`, you can use the shorthand notation `padding: <top> <right> <bottom> <left>` or target specific directions like `padding-bottom`
    - `border`: Can be used to define a visible border around the element content + padding (e.g. `border: 1px solid black`)
    - `border-radius`: Can be used to give a box rounded corners (even if no border is defined)
- Sizes:
    - `width`: Used to define a fixed width for an element - instead of using the full screen width for block elements or the content width for inline elements (could be set in pixels like `18px`)
    - `height`: Used to define a fixed height for an element - instead of inferring it automatically based on the content height in the element
- Other:
    - `box-shadow`: Can be used to define a shadow for an element - is set by defining a x-offset, y-offset, an optional blur radius, an optional spread radius and a color (e.g. `box-shadow: 0 1px 8px rgba(0, 0, 0, 0.2)`)