100DaysOfCode

About Headings, paragraphs, lists, and spans => TEXT

*Day 1:*

<h1,2,3,4,5,6>- headings/titlu

<div> -> sectiuni (block-level element). !important spatierea

<span> -> inline-level element. Selectarea specifica a unui text din paragraf (<p>)

parent, child, sibling, grandchild

id ->identificarea sectiunilor

*Day 2:*

<strong> - bold

<em> - italic

Style Sheets - in relation to HTML, dicteaza stilul documentului

<br> - este folosit la spacing (dublu <br> se comporta ca un "enter") ->se foloseste fara </>

<ul> unordered list si <ol> pentru ordered list. Cu child <li>

<li>Limes</li>

<li>Tortillas</li>

<li>Chicken</li>

<img> - pentru imagini. Contine closing "/" <img src="image-location.jpg" /> Unde "src"=image source (URL - Uniform Resource Locator). Se foloseste si alt ca si identificator (pt SEO) alt=""

<video> - cu src="" catre video source (hosted sau extern). Necesite closing tag </video>

Se poate customiza cu width="", height="" si controls. Intre tag-uri se pune (fara alt) un descriptor (ex: Video not supported - cu link extern catre alt video)

Rezumat:

1. HTML stands for HyperText Markup Language and is used to create the structure and content of a webpage.

2. Most HTML elements contain opening and closing tags with raw text or other HTML tags between them.

3. HTML elements can be nested inside other elements. The enclosed element is the child of the enclosing parent element.

4. Any visible content should be placed within the opening and closing <body> tags .

5. Headings and sub-headings, <h1> to <h6> tags, are used to enlarge text.

6.<p>, <span> and <div> tags specify text or blocks.

7. The <em> and <strong> tags are used to emphasize text.

8. Line breaks are created with the <br> tag.

9. Ordered lists (<ol>) are numbered and unordered lists (<ul>) are bulleted.

10. Images (<img>) and videos (<video>) can be added by linking to an existing source.

-About HTML Document Standards

<!DOCTYPE html> declaratie catre browser pentru a interpreta documentul ca si HTML (HTML5). INTODEAUNA SE INCEPE CU DECLARATIA

<html> -creeaza structura si continutul de html dupa declaratie

<head>- contine metadate despre pagina cat si titlul acesteia in browser folosind <title></title>

Rezumat:

1. <!DOCTYPE html>, the declaration specifying the version of HTML for the browser

2. The <html> tags that enclose all of your HTML code

3. The <head> tag that contains the metadata of a webpage, such as its <title>

-About Links

<a>->defineste tag-ul de link iar href defineste un hypelink (ex: <a href="https://en.wikipedia.org/wiki/Brown\_bear"> Learn More</a>) si se inchide cu</a>

target="\_blank" -> in interiorul <a href> deschide link-ul extern intr-un tab nou

"./contact.html" -> in interiorul <a href> dechide link-urile interne (local files) dintr-un root directory

Pentru a insera link pe imagine se foloseste <a href=""> apoi <img src="">

Rezumat

1. The <!DOCTYPE html> declaration should always be the first line of code in your HTML files. This lets the browser know what version of HTML to expect.

2. The <html> element will contain all of your HTML code.

3. Information about the web page, like the title, belongs within the <head> of the page.

4. You can add a title to your web page by using the <title> element, inside of the head.

5. A webpage's title appears in a browser's tab.

6. Anchor tags (<a>) are used to link to internal pages, external pages or content on the same page.

7. You can create sections on a webpage and jump to them using <a> tags and adding ids to the elements you wish to jump to.

8. Whitespace between HTML elements helps make code easier to read while not changing how elements appear in the browser.

9. Indentation also helps make code easier to read. It makes parent-child relationships visible.

10. Comments are written in HTML using the following syntax: <!-- comment -->.

Day 3

<table> -inserarea tabelelor

<thead>-inserarea capetelor de tabel <th> (heading)

<td> - inserarea datelor in tabel

<tbody> pentru tabele mai mari (ca si sectiuni head,body,footer)

<tfoot> pentru footer (ex.totalul)

<tr>-inserarea randurilor

<colspan=""> si <rowspan="" -pentru merge

==>

<table>

<thead>

<tr>

<th>Company Name</th>

<td>Adam's Greenworks</td>

</tr>

</thead>

<tbody>

<tr>

<th>Number of Items to Ship</th>

<td colspan="2">14</td>

</tr>

<tr>

<td>Davie's Burgers</td>

<td>2</td>

<td rowspan="2">Send Invoice</td>

</tr>

<tr>

<td>Baker's Bike Shop</td>

<td>3</td>

<td>Send Invoice</td>

</tr>

<tfoot>

<td>Total</td>

<td>28</td>

</tfoot>

</table>

Day 4

About CSS

To style an HTML element, you can add the style attribute directly to the opening tag. After you add the attribute, you can set it equal to the CSS style(s) you'd like applied to that element. Close it with </style>

<style>

p {

color: red;

font-size: 20px;

}

</style>

Keep it seperate under style.css

"p {

font-family: Arial;

}"

You can use the <link> element to link HTML and CSS files together. The <link> element must be placed within the head of the HTML file.

When linking an HTML file and a CSS file together, the <link> element will look like the following:

<link href="https://www.codecademy.com/stylesheets/style.css" type="text/css" rel="stylesheet">

Note that in the example above the path to the stylesheet is a URL:

https://www.codecademy.com/stylesheets/style.css

Specifying the path to the stylesheet using a URL is one way of linking a stylesheet.

If the CSS file is stored in the same directory as your HTML file, then you can specify a relative path instead of a URL, like so:

<link href="./style.css" type="text/css" rel="stylesheet">

Using a relative path is very common way of linking a stylesheet.

CSS can select HTML elements by using an element's tag name. A tag name is the word (or character) between HTML angle brackets.

For example, in HTML, the tag for a paragraph element is <p>. The CSS syntax for selecting <p> elements is:

p {

}

In the example above, all paragraph elements will be selected using a CSS selector. The selector in the example above is p.

they can also have attributes. One common attribute is the class attribute. It's also possible to select an element by its class attribute.

For example, consider the following HTML:

<p class="brand">Sole Shoe Company</p>

The paragraph element in the example above has a class attribute within the <p> tag. The class attribute is set to "brand". To select this element using CSS, we could use the following CSS selector:

.brand {

}

To select an HTML element by its class using CSS, a period (.) must be prepended to the class's name. In the example above case, the class is brand, so the CSS selector for it is .brand

(styles: color, font-family, uppercase, capitalize...

Luckily, it's possible to add more than one class name to an HTML element's class attribute.

For instance, perhaps there's a heading element that needs to be green and bold. You could write two CSS rules like so:

.green {

color: green;

}

.bold {

font-weight: bold;

If an HTML element needs to be styled uniquely (no matter what classes are applied to the element), we can add an ID to the element. To add an ID to an element, the element needs an id attribute:

<h1 id="large-title"> ... </h1>

Then, CSS can select HTML elements by their id attribute. To select an id element, CSS prepends the id name with a hashtag (#). For instance, if we wanted to select the HTML element in the example above, it would look like this:

#large-title {

}

The id name is large-title, therefore the CSS selector for it is #large-title.

}

Then, you could include both of these classes on one HTML element like this:

<h1 class="green bold"> ... </h1>

We can add multiple classes to an HTML element's class attribute by separating them with a space.

CSS can select HTML elements by their tag, class, and ID. CSS classes and IDs have different purposes, which can affect which one you use to style HTML elements.

CSS classes are meant to be reused over many elements. By writing CSS classes, you can style elements in a variety of ways by mixing classes on HTML elements.

For instance, imagine a page with two headlines. One headline needs to be bold and blue, and the other needs to be bold and green. Instead of writing separate CSS rules for each headline that repeat each other's code, it's better to write a .bold CSS rule, a .green CSS rule, and a .blue CSS rule. Then you can give one headline the bold green classes, and the other the bold blue classes.

While classes are meant to be used many times, an ID is meant to style only one element. As we'll learn in the next exercise, IDs override the styles of tags and classes. Since IDs override class and tag styles, they should be used sparingly and only on elements that need to always appear the same.

LA CLASE SE PUNE ".classname" IAR LA ID SE PUNE "#idname"

TOT TIMPUL SE INCHIDE CU ;}

!Research chaining and nested elements in CSS!

Adding more than one tag, class, or ID to a CSS selector increases the specificity of the CSS selector.

For instance, consider the following CSS:

p {

color: blue;

}

.main p {

color: red;

}

Both of these CSS rules define what a p element should look like. Since .main p has a class and a p tag as its selector, only the p elements inside the .main element will appear red. This occurs despite there being another more general rule that states p elements should be blue.

nstead of writing font-family: Georgia twice for two selectors, we can separate the selectors by a comma to apply the same style to both, like this:

h1,

.menu {

font-family: Georgia;

}

By separating the CSS selectors with a comma, both the h1 and the .menu elements will receive the font-family: Georgia styling.

Day 5

CSS Visual Rules

To change the typeface of text on your web page, you can use the font-family property.

It's a good practice to limit the number of typefaces used on a web page to 2 or 3.

In CSS, the font-weight property controls how bold or thin text appears. The font-weight property has a another value: normal (pentru sectiuni separate, in caz ca toate sunt cu bold).

To align text we can use the text-align property. The text-align property will align text to the element that holds it, otherwise known as its parent. (LEFT,CENTER,RIGHT)

In CSS, these two design aspects can be styled with the following two properties:

color: this property styles an element's foreground color

background-color: this property styles an element's background color.

Opacity can be used to make elements fade into others for a nice overlay effect. To adjust the opacity of an element, the syntax looks like this:

.overlay {

opacity: 0.5;

}

One option is to make the background of an element an image. This is done through the CSS property background-image. Its syntax looks like this:

.main-banner {

background-image: url("https://www.example.com/image.jpg");

}

THE BOX MODEL:

If you have used HTML and CSS, you have unknowingly seen aspects of the box model. For example, if you have set the background color of an element, you may have noticed that the color was applied not only to the area directly behind the element, but also to the area to the right of the element.

Width and height — specifies the width and height of the content area.

Padding — specifies the amount of space between the content area and the border.

Border — specifies the thickness and style of the border surrounding the content area and padding.

Margin — specifies the amount of space between the border and the outside edge of the element.

Borders can be set with a specific width, style, and color.

width — The thickness of the border. A border's thickness can be set in pixels or with one of the following keywords: thin, medium, or thick.

style — The design of the border. Web browsers can render any of 10 different styles. Some of these styles include: none, dotted, and solid.

color — The color of the border. Web browsers can render colors using a few different formats, including 140 built-in color keywords.

p {

border: 3px solid coral;

}

You can modify the corners of an element's border box with the border-radius property.

div.container {

border: 3px solid rgb(22, 77, 100);

border-radius: 5px;

}

THE PERFECT CIRCLE is border-radius:100%

Padding is like the space between a picture and the frame surrounding it. In CSS, you can modify this space with the padding property.

p.content-header {

border: 3px solid coral;

padding: 10px;

}

If you want to be more specific about the amount of padding on each side of a box's content, you can use the following properties:

padding-top

padding-right

padding-bottom

padding-left

Another implementation of the padding property lets you specify exactly how much padding there should be on each side of the content in a single declaration.

p.content-header {

border: 3px solid grey;

padding: 6px 11px 4px 9px;

}

In the example above, the four values 6px 11px 4px 9px correspond to the amount of padding in a clockwise rotation. In order, it specifies the amount of padding on the top (6 pixels), right (11 pixels), bottom (4 pixels), and left (9 pixels) sides of the content.

When using this implementation of the padding property, we must specify a padding value for all four sides of the element.

However, if the top and bottom values for padding will equal each other, and the left and right values for padding will also equal each other, you can use the following shortcut:

p.content-header {

padding: 5px 10px;

}

Margin refers to the space directly outside of the box. The margin property is used to specify the size of this space.

p {

border: 1px solid aquamarine;

margin: 20px;

}

The margin property also lets you center content. However, you must follow a few syntax requirements. Take a look at the following example:

div {

margin: 0 auto;

ONLY WHEN:

In order to center an element, a width must be set for that element

Day5

The overflow property controls what happens to content that spills, or overflows, outside its box. It can be set to one of the following values:

hidden - when set to this value, any content that overflows will be hidden from view.

scroll - when set to this value, a scrollbar will be added to the element's box so that the rest of the content can be viewed by scrolling.

visible - when set to this value, the overflow content will be displayed outside of the containing element. Note, this is the default value.

p {

overflow: scroll;

}

In the example above, if any of the paragraph content overflows (perhaps a user resizes their browser window), a scrollbar will appear so that users can view the rest of the content.

The overflow property is set on a parent element to instruct a web browser how to render child elements. For example, if a div’s overflow property is set to scroll, all children of this div will display overflowing content with a scroll bar.

All major web browsers have a default stylesheet they use in the absence of an external stylesheet. These default stylesheets are known as user agent stylesheets. In this case, the term "user agent" is a technical term for the browser.

User agent stylesheets often have default CSS rules that set default values for padding and margin. This affects how the browser displays HTML elements, which can make it difficult for a developer to design or style a web page.

Many developers choose to reset these default values so that they can truly work with a clean slate.

\* {

margin: 0;

padding: 0;

}

Elements can be hidden from view with the visibility property.

The visibility property can be set to one of the following values:

hidden — hides an element.

visible — displays an element.

<ul>

<li>Explore</li>

<li>Connect</li>

<li class="future">Donate</li>

<ul>

.future {

visibility: hidden;

}

In the example above, the list item with a class of future will be hidden from view in the browser.

Keep in mind, however, that users can still view the contents of the list item (e.g., Donate) by viewing the source code in their browser. Furthermore, the web page will only hide the contents of the element. It will still leave an empty space where the element is intended to display.

Note: What's the difference between display: none and visibility: hidden? An element with display: none will be completely removed from the web page. An element with visibility: hidden, however, will not be visible on the web page, but the space reserved for it will.

Many properties in CSS have a default value and don't have to be explicitly set in the stylesheet.

For example, the default font-weight of text is normal, but this property-value pair is not typically specified in a stylesheet.

The same can be said about the box model that browsers assume. In CSS, the box-sizing property controls the type of box model the browser should use when interpreting a web page.

The default value of this property is content-box. This is the same box model that is affected by border thickness and padding.

Fortunately, we can reset the entire box model and specify a new one: border-box.

\* {

box-sizing: border-box;

}

The code in the example above resets the box model to border-box for all HTML elements. This new box model avoids the dimensional issues that exist in the former box model you learned about.

In addition to the properties that it provides to style HTML elements, CSS includes properties that change how a browser positions elements. These properties specify where an element is located on a page, if the element can share lines with other elements, and other related attributes.

In this lesson, you will learn five properties for adjusting the position of HTML elements in the browser:

position

display

z-index

float

clear

Each of these properties will allow us to position and view elements on a web page. They can be used in conjunction with any other styling properties you may know.

One way to modify the default position of an element is by setting its position property to relative.

This value allows you to position an element relative to its default static position on the web page.

.box-bottom {

background-color: DeepSkyBlue;

position: relative;

}

Although the code in the example above instructs the browser to expect a relative positioning of the div, it does not specify where the div should be positioned on the page.

.box-bottom {

background-color: DeepSkyBlue;

position: relative;

top: 20px;

left: 50px;

}

In the example above, the <div> has been positioned using two of the four offset properties. The valid offset properties are:

top - moves the element down.

bottom - moves the element up.

left - moves the element right.

right - moves the element left.

Another way of modifying the position of an element is by setting its position to absolute.

When an element's position is set to absolute all other elements on the page will ignore the element and act like it is not present on the page. The element will be positioned relative to its closest positioned parent element.

.box-bottom {

background-color: DeepSkyBlue;

position: absolute;

top: 20px;

left: 50px;

}

In the example above, the .box-bottom <div> will be moved down and right from the top left corner of the view. If offset properties weren't specified, the top box would be entirely covered by the bottom box.

hen an element's position is set to absolute, as in the last exercise, the element will scroll with the rest of the document when a user scrolls.

We can fix an element to a specific position on the page (regardless of user scrolling) by setting its position to fixed.

.box-bottom {

background-color: DeepSkyBlue;

position: fixed;

top: 20px;

left: 50px;

}

Day7

Inline Display

The default display for some tags, such as <em>, <strong>, and <a>, is called inline. Inline elements have a box that wraps tightly around their content, only taking up the amount of space necessary to display their content and not requiring a new line after each element. The height and width of these elements cannot be specified in the CSS document. For example, the text of an anchor tag (<a>) will, by default, be displayed on the same line as the surrounding text, and it will only be as wide as necessary to contain its content. inline elements cannot be altered in size with the height or width CSS properties.

To learn more about <em>inline</em> elements, read <a href="#">MDN documentation</a>.

In the example above, the <em> element is inline, because it displays its content on the same line as the content surrounding it, including the anchor tag. This example will display:

"To learn more about inline elements, click MDN documentation."

The CSS display property provides the ability to make any element an inline element. This includes elements that are not inline by default such as paragraphs, divs, and headings.

h1 {

display: inline;

}

The CSS in the example above will change the display of all <h1> elements to inline. The browser will render <h1> elements on the same line as other inline elements immediately before or after them (if there are any).

Some elements are not displayed in the same line as the content around them. These are called block-level elements. These elements fill the entire width of the page by default, but their width property can also be set. Unless otherwise specified, they are the height necessary to accommodate their content.

Elements that are block-level by default include all levels of heading elements (<h1> through <h6>), <p>, <div> and <footer>. For a complete list of block level elements, visit the MDN documentation.

strong {

display: block;

}

In the example above, all <strong> elements will be displayed on their own line, with no content directly on either side of them even though their contents may not fill the width of most computer screens.

The third value for the display property is inline-block. Inline-block display combines features of both inline and block elements. Inline-block elements can appear next to each other and we can specify their dimensions using the width and height properties. Images are the best example of default inline-block elements.

For example, <div>s in the CSS below will be displayed on the same line and with the specified dimensions:

<div class="rectangle">

<p>I’m a rectangle!</p>

</div>

<div class="rectangle">

<p>So am I!</p>

</div>

<div class="rectangle">

<p>Me three!</p>

</div>

.rectangle {

display: inline-block;

width: 200px;

height: 300px;

}

In the example above, there are three rectangular divs that each contain a paragraph of text. The .rectangle <div>s will all appear inline (provided there is enough space from left to right) with a width of 200 pixels and height of 300 pixels, even though the text inside of them may not require 200 pixels by 300 pixels of space.

So far, you've learned how to specify the exact position of an element using offset properties. If you're simply interested in moving an element as far left or as far right as possible on the page, you can use the float property.

The float property can be set to one of two values:

left - this value will move, or float, elements as far left as possible.

right - this value will move elements as far right as possible.

.boxes {

width: 120px;

height: 70px;

}

.box-bottom {

background-color: DeepSkyBlue;

float: right;

}

The float property can also be used to float multiple elements at once. However, when multiple floated elements have different heights, it can affect their layout on the page. Specifically, elements can "bump" into each other and not allow other elements to properly move to the left or right.

The clear property specifies how elements should behave when they bump into each other on the page. It can take on one of the following values:

left — the left side of the element will not touch any other element within the same containing element.

right — the right side of the element will not touch any other element within the same containing element.

both — neither side of the element will touch any other element within the same containing element.

none — the element can touch either side.

div {

width: 200px;

float: left;

}

div.special {

clear: left;

}

In the example above, all <div>s on the page are floated to the left side. The element with class special did not move all the way to the left because a taller <div> blocked its positioning. By setting its clear property to left, the special <div> will be moved all the way to the left side of the page.

Introduction to Color

To use opacity in the HSL color scheme, use hsla instead of hsl, and four values instead of three. For example:

color: hsla(34, 100%, 50%, 0.1);

The first three values work the same as hsl. The fourth value (which we have not seen before) is the alpha. This last value is sometimes called the opacity.

Alpha is a decimal number from zero to one. If alpha is zero, the color will be completely transparent. If alpha is one, the color will be opaque. The value for half transparent would be 0.5.

The RGB color scheme has a similar syntax for opacity, rgba. Again, the first three values work the same as rgb and the last value is the alpha. Here's an example:

color: rgba(234, 45, 98, 0.33);

Alpha can only be used with HSL and RGB colors; we cannot add the alpha value to color: green color: #FFFFF.

There is, however, a named color keyword for zero opacity, transparent. It's equivalent to rgba(0, 0, 0, 0). It's used like any other color keyword:

color: transparent;

There are four ways to represent color in CSS:

Named colors — there are 147 named colors, which you can review here.

Hexadecimal or hex colors

Hexadecimal is a number system with has sixteen digits, 0 to 9 followed by "A" to "F".

Hex values always begin with # and specify values of red, blue and green using hexademical numbers such as #23F41A.

RGB

RGB colors use the rgb() syntax with one value for red, one value for blue and one value for green.

RGB values range from 0 to 255 and look like this: rgb(7, 210, 50).

HSL

HSL stands for hue (the color itself), saturation (the intensity of the color), and lightness (how light or dark a color is).

Hue ranges from 0 to 360 and saturation and lightness are both represented as percentages like this: hsl(200, 20%, 50%).

You can add opacity to color in RGB and HSL by adding a fourth value, a, which is represented as a percentage.

DAY 7

CSS TYPOGRAPHY

What happens when a stylesheet requires a font that is not installed on a user's computer? Most computers have a small set of typefaces pre-installed. This small set includes serif fonts like Times New Roman and sans-serif fonts like Arial.

These pre-installed fonts serve as fallback fonts if the stylesheet specifies a font which is not installed on a user's computer.

To use fallback fonts, the following syntax is required:

h1 {

font-family: "Garamond", "Times", serif;

}

The CSS rule above says:

Use the Garamond font for all <h1> elements on the web page.

If Garamond is not available, use the Times font.

If Garamond and Times are not available, use any serif font pre-installed on the user's computer.

The fonts specified after Garamond are the fallback fonts (Times, serif). Fallback fonts help ensure a consistent experience for the diverse audience of users that visit a site.

New fonts are often centralized in directories made available for public use. We refer to these fonts as non-user fonts.

Google Fonts is one such directory of thousands of open-source fonts, available for free use. Google Fonts gives us a way to retrieve the link for a single font, multiple fonts, or multiple fonts with the font-weight and font-style properties.

When we have the link to the font of our choice, we can add the font to the <head> section of the HTML document, using the <link> tag and the href.

Let's take a look at a few examples:

1. A single linked font, using Droid Serif as an example:

<head>

<link href="https://fonts.googleapis.com/css?family=Droid+Serif" type="text/css" rel="stylesheet">

</head>

2. Multiple linked fonts, using the Droid Serif and Playfair Display fonts as an example:

<head>

<link href="https://fonts.googleapis.com/css?family=Droid+Serif|Playfair+Display" type="text/css" rel="stylesheet">

</head>

3. Multiple linked fonts, along with weights and styles. Here Droid Serif has font weights of 400, 700, and 700i, while Playfair Display has font weights of 400, 700, and 900i:

<head>

<link href="https://fonts.googleapis.com/css?family=Droid+Serif:400,700,700i|Playfair+Display:400,700,900i" rel="stylesheet">

</head>

Once a font is linked, we can create CSS selectors to target elements, just as we do with other fonts.

SUM UP:

Typography is the art of arranging text on a page.

Text can appear in any number of weights, with the font-weight property.

Text can appear in italics with the font-style property.

The vertical spacing between lines of text can be modified with the line-height property.

Serif fonts have extra details on the ends of each letter. Sans-Serif fonts do not.

Fallback fonts are used when a certain font is not installed on a user's computer.

Google Fonts provides free fonts that can be used in an HTML file with the <link> tag or the @font-face property.

Local fonts can be added to a document with the @font-face property and the path to the font's source.

The word-spacing property changes how far apart individual words are.

The letter-spacing property changes how far apart individual letters are.

The text-align property changes where text horizontally on a page.

-CSS GRID ESSENTIALS

Using CSS, you can elegantly lay out elements on a web page. There is no simple answer for how best to do this — depending on what content you are trying to display, multiple different techniques can work well. Codecademy's lessons on the box model, Flexbox, and CSS display and positioning properties explain three possible ways to approach layout.

In this lesson, we introduce a new, powerful tool called CSS Grid. The grid can be used to lay out entire web pages. Whereas Flexbox is mostly useful for positioning items in a one-dimensional layout, CSS grid is most useful for two-dimensional layouts, providing many tools for aligning and moving elements across both rows and columns.

Grid layouts:

grid-template-columns

grid-template-rows

grid-template

grid-template-area

grid-gap

grid-row-start / grid-row-end

grid-column-start / grid-column-end

grid-area

To set up a grid, you need to have both a grid container and grid items. The grid container will be a parent element that contains grid items as children and applies overarching styling and positioning to them.

To turn an HTML element into a grid container, you must set the element's display property to grid (for a block-level grid) or inline-grid (for an inline grid). Then, you can assign other properties to lay out the grid.

By default, grids contain only one column. If you were to start adding items, each item would be put on a new row; that's not much of a grid! To change this, we need to explicitly define the number of rows and columns in our grid.

We can define the columns of our grid by using the CSS property grid-template-columns. Below is an example of this property in action:

.grid {

display: grid;

width: 500px;

grid-template-columns: 100px 200px;

}

This property creates two changes. First, it defines the number of columns in the grid; in this case, there are two. Second, it sets the width of each column. The first column will be 100 pixels wide and the second column will be 200 pixels wide.

We can also define the size of our columns as a percentage of the entire grid's width.

.grid {

display: grid;

width: 1000px;

grid-template-columns: 20% 50%;

}

In this example, the grid is 1000 pixels wide. Therefore, the first column will be 200 pixels wide because it is set to be 20% of the grid's width. The second column will be 500 pixels wide.

We can also mix and match these two units. In the example below, there are three columns of width 20 pixels, 40 pixels, and 60 pixels:

.grid {

display: grid;

width: 100px;

grid-template-columns: 20px 40% 60px;

}

Notice that in this example, the total width of our columns (120 pixels) exceeds the width of the grid (100 pixels). This might make our grid cover other elements on the page! In a later exercise we will discuss how to avoid overflow.

We've learned how to define the number of columns in our grid explicitly. To specify the number and size of the rows, we are going to use the property grid-template-rows.

This property is almost identical to grid-template-columns. Take a look at the code below to see both properties in action.

.grid {

display: grid;

width: 1000px;

height: 500px;

grid-template-columns: 100px 200px;

grid-template-rows: 10% 20% 600px;

}

This grid has two columns and three rows. grid-template-rows defines the number of rows and sets each row's height. In this example, the first row is 50 pixels tall (10% of 500), the second row is 100 pixels tall (20% of 500), and the third row is 600 pixels tall.

When using percentages in these two properties, remember that rows are defined as a percentage of the grid's height, and columns are defined as a percentage of its width.

The property grid-template can replace the previous two CSS properties. Both grid-template-rows and grid-template-columns are nowhere to be found in the following code!

.grid {

display: grid;

width: 1000px;

height: 500px;

grid-template: 200px 300px / 20% 10% 70%;

}

When using grid-template, the values before the slash will determine the size of each row. The values after the slash determine the size of each column. In this example, we've made two rows and three columns of varying sizes.

The same rules from before apply; when using percentages to set rows, each row will be a percentage of the grid's total height. Columns are still a percentage of the total width.

You may already be familiar with several types of responsive units such as percentages (%), ems and rems. CSS Grid introduced a new relative sizing unit — fr, like fraction.

By using the fr unit, we can define the size of columns and rows as a fraction of the grid's length and width. This unit was specifically created for use in CSS Grid. Using fr makes it easier to prevent grid items from overflowing the boundaries of the grid. Consider the code below:

.grid {

display: grid;

width: 1000px;

height: 400px;

grid-template: 2fr 1fr 1fr / 1fr 3fr 1fr;

}

In this example, the grid will have three rows and three columns. The rows are splitting up the available 400 pixels of height into four parts. The first row gets two of those parts, the second column gets one, and the third column gets one. Therefore the first row is 200 pixels tall, and the second and third rows are 100 pixels tall.

Each column's width is a fraction of the available space. In this case, the available space is split into five parts. The first column gets one-fifth of the space, the second column gets three-fifths, and the last column gets one-fifth. Since the total width is 1000 pixels, this means that the columns will have widths of 200 pixels, 600 pixels, and 200 pixels respectively.

It is possible to use fr with other units as well. When this happens, each fr represents a fraction of the available space.

.grid {

display: grid;

width: 100px;

grid-template-columns: 1fr 60px 1fr;

}

In this example, 60 pixels are taken up by the second column. Therefore the first and third columns have 40 available to split between them. Since each gets one fraction of the total, they both end up being 20 pixels wide.

The properties that define the number of rows and columns in a grid can take a function as a value. repeat() is one of these functions. The repeat() function was created specifically for CSS Grid.

.grid {

display: grid;

width: 300px;

grid-template-columns: repeat(3, 100px);

}

The repeat function will duplicate the specifications for rows or columns a given number of times. In the example above, using the repeat function will make the grid have three columns that are each 100 pixels wide. It is the same as writing:

grid-template-columns: 100px 100px 100px;

Repeat is particularly useful with fr. For example, repeat(5, 1fr) would split your table into five equal rows or columns.

Finally, the second parameter of repeat() can have multiple values.

grid-template-columns: repeat(2, 20px 50px)

This code will create four columns where the first and third columns will be 20 pixels wide and the second and fourth will be 50 pixels wide.

But sometimes you might want a grid to resize based on the size of your web browser.

In these situations, you might want to prevent a row or column from getting too big or too small. For example, if you have a 100-pixel wide image in your grid, you probably don't want its column to get thinner than 100 pixels! The minmax() function can help us solve this problem.

.grid {

display: grid;

grid-template-columns: 100px minmax(100px, 500px) 100px;

}

In this example, the first and third columns will always be 100 pixels wide, no matter the size of the grid. The second column, however, will vary in size as the overall grid resizes. The second column will always be between 100 and 500 pixels wide.

In all of our grids so far, there hasn't been any space between the items in our grid. The CSS properties grid-row-gap and grid-column-gap will put blank space between every row and column in the grid.

.grid {

display: grid;

width: 320px;

grid-template-columns: repeat(3, 1fr);

grid-column-gap: 10px;

}

It is important to note that grid-gap does not add space at the beginning or end of the grid. In the example code, our grid will have three columns with two ten-pixel gaps between them.

Finally, there is a CSS property grid-gap that can set the row and column gap at the same time. grid-gap: 20px 10px; will set the distance between rows to 20 pixels and the distance between columns to 10 pixels. Unlike other CSS grid properties, this shorthand does not take a / between values! If only one value is given, it will set the column gap and the row gap to that value.

In this lesson, we have learned how to define a grid container. When explicitly defining a grid, you have to declare the quantity of rows and columns and their respective sizes.

In all of our examples, the items placed in the grid have always taken up exactly one square. This does not always need to be the case; we can drastically change the look of our grid by making grid items take up more than one row and one column. You can see this in the diagram to the right. Items A, B, C, and E span more than one row!

In the following exercises, you will learn CSS properties that will affect the size of the grid items and where they are displayed on the page. By manipulating both the parent and the child elements, we can create beautiful layouts with ease.

Using the CSS properties grid-row-start and grid-row-end, we can make single grid items take up multiple rows. Remember, we are no longer applying CSS to the outer grid container; we're adding CSS to the elements sitting inside the grid!

.item {

grid-row-start: 1;

grid-row-end: 3;

}

In this example, the HTML element of class item will take up two rows in the grid, rows 1 and 2. The values that grid-row-start and grid-row-end accept are grid lines.

Row grid lines and column grid lines start at 1 and end at a value that is 1 greater than the number of rows or columns the grid has. For example, if a grid has 5 rows, the grid row lines range from 1 to 6. If a grid has 8 columns, the grid row lines range from 1 to 9.

The value for grid-row-start should be the row at which you want the grid item to begin. The value for grid-row-end should be one greater than the row at which you want the grid item to end. An element that covers rows 2, 3, and 4 should have these declarations: grid-row-start: 2 and grid-row-end: 5.

It is possible for the value of grid-row-start to be greater than that of grid-row-end.

We can use the property grid-row as shorthand for grid-row-start and grid-row-end. The following two code blocks will produce the same output:

.item {

grid-row-start: 4;

grid-row-end: 6;

}

.item {

grid-row: 4 / 6;

}

This code should look similar to the way grid-template is shorthand for grid-template-rowsand grid-template-columns. In this case, the starting row goes before the "/" and the ending row goes after it. Again, the ending row is exclusive; this grid item will occupy rows four and five.

When an item spans multiple rows or columns using these properties, it will also include the grid-gap if any exists. For example, if an item spans two rows of height 100 pixels and there is a ten-pixel grid-gap, then the item will have a total height of 210 pixels.

he previous three properties also exist for columns. grid-column-start, grid-column-end and grid-column work identically to the row properties. These properties allow a grid item to span multiple columns.

When using these properties, we can use the keyword span to start or end a column or row relative to its other end. Look at how span is used in the code below:

.item {

grid-column: 4 / span 2;

}

This is telling the item element to begin in column four and take up two columns of space. So item would occupy columns four and five. It produces the same result as the following code blocks:

.item {

grid-column: 4 / 6;

}

.item {

grid-column-start: 4;

grid-column-end: span 2;

}

.item {

grid-column-start: span 2;

grid-column-end: 6;

}

span is a useful keyword, because it avoids off-by-one errors (miscalculating the ending grid line) you might make when determining the ending grid line of an element. If you know where you want your grid item to start and how long it should be, use span!

We've already been able to use grid-row and grid-column as shorthand for properties like grid-row-start and grid-row-end. We can refactor even more using the property grid-area. This property will set the starting and ending positions for both the rows and columns of an item.

.item {

grid-area: 2 / 3 / 4 / span 5;

}

grid-area takes four values separated by slashes. The order is important! This is how grid-area will interpret those values.

grid-row-start

grid-column-start

grid-row-end

grid-column-end

In the above example, the item will occupy rows two and three and columns three through eight.

Using grid-area is an easy way to place items exactly where you want them in a grid.

Day9

Advanced CSS GRID

Grid Template Areas

The grid-template-areas property allows you to name sections of your web page to use as values in the grid-row-start, grid-row-end, grid-col-start,grid-col-end, and grid-area properties.

<div class="container">

<header>Welcome!</header>

<nav>Links!</nav>

<section class="info">Info!</section>

<section class="services">Services!</section>

<footer>Contact us!</footer>

</div>

.container {

display: grid;

max-width: 900px;

position: relative;

margin: auto;

grid-template-areas: "head head"

"nav nav"

"info services"

"footer footer";

grid-template-rows: 300px 120px 800px 120px;

grid-template-columns: 1fr 3fr;

}

header {

grid-area: head;

}

nav {

grid-area: nav;

}

.info {

grid-area: info;

}

.services {

grid-area: services;

}

footer {

grid-area: footer;

}

You may want to expand this section of the website to view the code above more clearly.

In the example above, the HTML creates a web page with five distinct parts.

The grid-template-areas declaration in the .container rule set creates a 2-column, 4-row layout.

The grid-template-rows declaration specifies the height of each of the four rows from top to bottom: 300 pixels, 120 pixels, 800 pixels, and 120 pixels.

The grid-template-columns declaration uses the fr value to cause the left column to use one fourth of the available space on the page and the right column to use three-fourths of the available space on the page.

In each rule set below .container, we use the grid-area property to tell that section to cover the portion of the page specified. The header element spans the first row and both columns. The nav element spans the second row and both columns. The element with class .info spans the third row and left column. The element with class .services spans the third row and right column. The footer element spans the bottom row and both columns.

That's it! An entire page laid out in 40 lines of code.

This property is declared on grid containers.

TEMPLATE EXAMPLE:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Grid Stuff</title>

<link rel="stylesheet" type="text/css" href="style.css" />

</head>

<body>

<div class="container">

<header>

<h1>Header</h1>

</header>

<nav>

<h1>Nav</h1>

</nav>

<section class="left">

<h2>Left</h2>

</section>

<section class="right">

<h2>Right</h2>

</section>

<footer>

<h1>Footer</h1>

</footer>

</div>

</body>

</html>

justify-items is a property that positions grid items along the inline, or row, axis. This means that it positions items from left to right across the web page.

justify-items accepts these values:

start — aligns grid items to the left side of the grid area

end — aligns grid items to the right side of the grid area

center — aligns grid items to the center of the grid area

stretch — stretches all items to fill the grid area

We can use justify-content to position the entire grid along the row axis.

It accepts these values:

start — aligns the grid to the left side of the grid container

end — aligns the grid to the right side of the grid container

center — centers the grid horizontally in the grid container

stretch — stretches the grid items to increase the size of the grid to expand horizontally across the container

space-around — includes an equal amount of space on each side of a grid element, resulting in double the amount of space between elements as there is before the first and after the last element

space-between — includes an equal amount of space between grid items and no space at either end

space-evenly — places an even amount of space between grid items and at either end

There are several other values that justify-content accepts, which you can read about on the Mozilla Developer Network. The definitions for these values can also be found in the documentation. It is important to note that the page with the definitions includes some values that are not accepted in CSS Grid layout.

<main>

<div class="left">Left</div>

<div class="right">Right</div>

</main>

main {

display: grid;

width: 1000px;

grid-template-columns: 300px 300px;

grid-template-areas: "left right";

justify-content: center;

}

In the example above, the grid container is 1000 pixels wide, but we only specified two columns that are 300 pixels each. This will leave 400 pixels of unused space in the grid container.

justify-content: center; positions the columns in the center of the grid, leaving 200 pixels on the right and 200 pixels on the left of the grid.

align-items is a property that positions grid items along the block, or column axis. This means that it positions items from top to bottom.

align-items accepts these values:

start — aligns grid items to the top side of the grid area

end — aligns grid items to the bottom side of the grid area

center — aligns grid items to the center of the grid area

stretch — stretches all items to fill the grid area

align-content positions the rows along the column axis, or from top to bottom.

It accepts these positional values:

start — aligns the grid to the top of the grid container

end — aligns the grid to the bottom of the grid container

center — centers the grid vertically in the grid container

stretch — stretches the grid items to increase the size of the grid to expand vertically across the container

space-around — includes an equal amount of space on each side of a grid element, resulting in double the amount of space between elements as there is before the first and after the last element

space-between — includes an equal amount of space between grid items and no space at either end

space-evenly — places an even amount of space between grid items and at either end

There are several other values that align-content accepts, which you can read about on the Mozilla Developer Network. The definitions for these values can also be found in the documentation. It is important to note that the page with the definitions includes some values that are not accepted in CSS Grid layout.

<main>

<div class="top">Top</div>

<div class="bottom">Bottom</div>

</main>

main {

display: grid;

height: 600px;

rows: 200px 200px;

grid-template-areas: "top"

"bottom";

align-content: center;

}

In the example above, the grid container is 600 pixels tall, but we only specified two rows that are 200 pixels each. This will leave 200 pixels of unused space in the grid container.

align-content: center; positions the rows in the center of the grid, leaving 100 pixels at the top and 100 pixels at the bottom of the grid.

The justify-items and align-items properties specify how all grid items contained within a single container will position themselves along the row and column axes, respectively.

justify-self specifies how an individual element should position itself with respect to the row axis. This property will override justify-items for any item on which it is declared.

align-self specifies how an individual element should position itself with respect to the column axis. This property will override align-items for any item on which it is declared.

They both accept these four properties:

start — positions grid items on the left side/top of the grid area

end — positions grid items on the right side/bottom of the grid area

center — positions grid items on the center of the grid area

stretch — positions grid items to fill the grid area (default)

align-self and justify-self accept the same values as align-items and justify-items. You can read about these values on the Mozilla

A strong believer in the power of impact and how that impact can be achieved through the actions of passionate people. Thusly, I always tried to target experiences which could transform me into a change agent for our society, as reflected in my experience.  
  
Passionate about: Community Development, Education and Learning, Advertising, Training&Coaching.  
Specialities: Communications skills, Training skills, Team Management, Social Media, Personal Development