**Responsive web design course Notes**

* The lower the number, the higher the importance, so h2 elements have less importance than h1 elements.
* HTML5 has some elements that identify different content areas. These elements make your HTML easier to read and help with Search Engine Optimization (SEO) and accessibility.
* The strong element is used to indicate that some text is of strong importance or urgent.
* The em element is used to indicate that emphasis should be put on the object text
* he div element is used mainly for design layout purposes unlike the other content elements you have used so far
* article elements commonly contain multiple elements that have related information.
* You can add a fallback value for the font-family by adding another font name separated by a comma. Fallbacks are used in instances where the initial is not found/available.
* #FF0000 and #FF0000CC . The second one defines the alpha value of the color..

**Registration Form**

* The vh unit stands for viewport height, and is relative to 1% of the height of the viewport.
* The method attribute specifies how to send form-data to the URL specified in the action attribute. The form-data can be sent via a GET request as URL parameters (with method="get") or via a POST request as data in the request body (with method="post").
* The rem unit stands for root em, and is relative to the font size of the html element.
* Minlength = 8 for min length
* With type="password" you can use the pattern attribute to define a regular expression that the password must match to be considered valid.

**[a-z0-5]{8,}**

* The above is a regular expression which matches eight or more lowercase letters or the digits 0 to 5
* To relate the radio inputs, give them the same name attribute with a value of account-type. Now, it is not possible to select both radio inputs at the same time.
* Select only the .inline elements, and give them width of unset. This will remove the earlier rule which set all the input elements to width: 100%. Unset is used to remove any previously applied rule. In this case the width was unsetted
* Color by default is used to designate text.

**Box Model**

* margin: 0 auto 20px; This will remove its top margin, horizontally center it, and set its bottom margin to 20 pixels.
* border-radius: 8px 10px; Set’s its top-left radius and bottom-right radius to 8px, and top-right radius and bottom-left radius to 10px
* padding: 8px 1px 3px 5px; Respectively goes for Top Right Bottom Left;

**Flex Box**

* Giving an element a display property of flex will make it a flex container. Any direct children of a flex container are called **flex items.**
* **Flexbox** has a main and cross axis. The main axis is defined by the flex-direction property, which has four possible values:
* row (default): horizontal axis with flex items from left to right
* row-reverse: horizontal axis with flex items from right to left
* column: vertical axis with flex items from top to bottom
* column-reverse: vertical axis with flex items from bottom to top

**Note**: The axes and directions will be different depending on the text direction. The values shown are for a left-to-right text direction.

**flex-wrap**

property determines how your flex items behave when the flex container is too small. Setting it to wrap will allow the items to wrap to the next row or column. nowrap (default) will prevent your items from wrapping and shrink them if needed.

**Flex-flow**

* Shorthand for flex-direction and flex-wrap.  
  flex-flow: row wrap;

**justify-content**

determines how the items inside a flex container are positioned along the main axis, affecting their position and the space around them. Consider the snippet below

*Flex-direction : row;  
justify-content : center;*

It will imply that elements being aligned horizontally ( row) and being centered **horizontally** (row)

**align-items**

positions the flex content along the cross axis. In this case, with your flex-direction set to row, your cross axis would be vertical

**align-content**

Aligns a flex container’s lines within the flex container when there is extra space in the cross-axis, similar to how [justify-content](https://tympanus.net/codrops/css_reference/flexbox/#section_justify-content) aligns individual items within the main-axis. This property has no effect when the flex container has only a single line.

**gap**

The **gap**CSS shorthand property sets the gaps, also known as gutters, between rows and columns. The gap property and its row-gap and column-gap sub-properties provide this functionality for flex, grid, and multi-column layout. You apply the property to the container element.

Be carefull with the **Gap** attribute as it may deform flex or grid items. Make sure to always pick or assign a gap that won’t disturb the layout

* A flex container refers to an HTML element (usually a div) whose CSS display property is set toflex or flex-inline.This element serves as a container for other elements that we seek to align. The elements to be aligned, which are direct children of the flex container are the flex items. Each row or column of flex items in a flex container is a flex line.
* **Stretch is is the initial value.** The flex items are stretched out from the cross-start to the cross-end, while still respecting the constraints imposed by [min-height](http://tympanus.net/codrops/css_reference/min-height)/[min-width](http://tympanus.net/codrops/css_reference/min-width)/[max-height](http://tympanus.net/codrops/css_reference/max-height)/[max-width](http://tympanus.net/codrops/css_reference/max-width). Note that if the flex container’s height is constrained this value may cause the contents of the flex items to overflow the items.
* The**::after**pseudo-element creates an element that is the last child of the selected element. You can use it to add an empty element after the last image. If you give it the same width as the images it will push the last image to the left when the gallery is in a two-column layout.

**Flex-grow**

* Defines how element grow after his flex-basis size is traversed.  
  flex-grow: 2;

**Flex-shrink**

* Defines how element shrinks after his flex-basis size is traversed.  
  flex-shrink: 1;

**Flex**

* Short hand for flex grow, shrink and basis.

Flex: 2 1 250px; // does the samething as above.

The **difference** between align-items and align-content

The align Items takes only the following properties ( flex-start, flex-end, center, baseline, stretch) . It aligns the whole block not the lines (due to it’s properties).

On the other hand align-content accepts two additional values which are the game which make the difference (space-around, space-between).

- Rather than setting each aspect ratio individually, you can use the object-fit property to determine how images should behave.

Give your .gallery img selector the object-fit property and set it to cover. This will tell the image to fill the img container while maintaining aspect ratio, resulting in cropping to fit.

- **Span**  is highly used to group elements (especially text ) for alignment. It’s like grouping in Adobe

**Typography**

* The **rem** unit stands for root em, and is relative to the font size of the html element. In other words setting an element’s font size to 1em is setting it to the font-size assigned in the html selector, or if not precised, the font-size of the browser. Like the word indicates “**Root** element(rem)”
* The **:not** pseudo-selector can be used to select all elements that do not match the given CSS rule.

div:not(#example) {

color: red;

}

ie all the divs except that with the ‘example’ id

.daily-value p:not(.no-divider) {

   border-bottom: 1px solid #888989;

}

* **50vw** means 50% of the view width
* Even though you added a placeholder to the first input element in the previous lesson, this is actually not a best-practice for accessibility; too often, users confuse the placeholder text with an actual input value - they think there is already a value in the input.
* Remove the placeholder text from the first input element, relying on the label being the best-practice.
* For every required field just add the required before closing the “>” . Example : <input type=”number” required/>, <select required>. Notice that for non self closing elements the required keyword is added on the first ”<>”. Okay good.
* For self closing elements the / comes after the element name e.g <input/>. On the other hand non self closing elements have the / before the name. e.g <div></div> , <textarea> </textarea>
* The br tags will allow each part of the address to be on its own line and are useful for presenting address elements properly.
* Address element gives information about the page author . (by default puts text in italics)
* The child combinator (>) is useful because it ensures that only the immediate children (li elements directly under ul) are selected and styled.

It is useful to see the default border around the fieldset elements, during development. However, it might not be the style you want.

border: none;removes the field borders.

When we use media queries, we are getting information from the device or media through which the webpage is been displayed to have a personalized or unique render. We can be querying browser settings , screen size , preferences etc. It’s all about knowing the client to feed him like he wants.

- Navigation accessibility can be improved by providing keyboard shortcuts.

The accesskey attribute accepts a space-separated list of access keys. For example:

<button type="submit" accesskey="s">Submit</button>

HEAD

BODY

FOOTER

Translucent background images implementation:

background-image: linear-gradient(90deg, rgba(2, 87, 154,0.8), rgba(74, 10, 35,0.5)),

   url("https://cdn.pixabay.com/photo/2020/01/27/19/22/piano-4798138\_640.jpg") ;

Notice background-image has the **linear-gradient** property we used to define overlapping color. Wow

You should give **priority to screen readers** when coding the crud html. You may even change the order of elements in the crud html just for it to make sense and then put them back in right other using html or CSS.

**Aria-hidden** Is used to hide content from screen readers. Aria-hidden=true

HTML tables use the **caption** element to describe what the table is about. The **caption** element should always be the first child of a table, but can be positioned with the caption-side CSS property.

td: Table data Cell

th: Table header Cell

tr: Table Row

**span[class~="sr-only"]** aNotice the ”~=” which stands for includes . That is , we are picking all spans who have sr-only in their list of classes.

The **Clip** function is used to actually clip or crop ( just like in illustrator ) elements. It can be images, text etc

clip: rect(1px , 1px, 1px, 1px);

The **Clip-path** property is used to define the visible area of an element. Using **Inset(50)** as value for instance implies setting the only 50 percent (from sides and from top and bottom)of the element visible.

The **:first-of-type** pseudo-selector is used to target the first element that matches the selector.

Use the **max-width, max-height , max-**.. for responsiveness

The **calc()**function is a CSS function that allows you to calculate a value based on other values.

.example {

margin: 10px;

width: calc(100% - 20px);

}

The **span[class]** syntax targets any span element that has a class attribute set, regardless of the attribute's value.

For any property you want to use , you can only write it once per element. That’s cannot define say transform or color property twice. Yeah.

**span[class~="sr-only"] << INVERSE OF >> span:not(.sr-only)**

**span[class~=”sr-only”]**  
That is all spans that have ‘sr-only’ in their class lists or even that just have **word ‘sr-only’** in the class list. In other words if there’s an element with the class ‘bsr-only’ this selector will select it.

**VS  
span:not(.sr-only)** all spans that do not have .sr-only in their class lists.

you can use the **!important** keyword to ensure these properties are always applied, regardless of order or specificity.

border-collapse: collapse; border collapse is used to merge cells so they share a common border.

The key difference between **tr[class="total"]** and **tr.total** is that the first will select tr elements where the only class is total. The second will select tr element that have total class.

The point here is that the ‘tr.total’ approach helps us to select specific total class elements. Let’s say in a website we assign the class “hover\_animate” to buttons, links and call to actions. For sure we have to give a hover animation to all of them.. but we may not want buttons and links to have the same animation . In such a case instead of assigning a new class say

“button\_hover” and “link\_hover” , we can just use this nice technique and this is what it will look like :

button.hover\_animate : selects buttons that have the hover\_animate class  
a.hover\_animate : selects anchor elements that have the hover\_animate class.

|  |  |
| --- | --- |
| **tr .data**  **class1.class2** | Selects all elements with the data class indented in tr. |
| **tr[class=”data”]** | Selects all tr elements with .data being their only class. It is a fancy way of saying tr.data |
|  |  |
| **tr[class\*=”data”]** | Selects all tr elements who the ‘data’ in one of their class . it will select <tr class=”ourdata” ></tr> for instance. Insane right?! |
| **tr.data** | Selects the .data class but only for tr elements. Also, they should have no other class |
| **tr[class~=”data”]** | This is the extended version of the tr.data selector. Selects the .data class but only for tr elements. Also, there can be as many other classes as possible on the tr element but also ‘data’ can be the only class. |

background: linear-gradient(#5e5e5e 85%,#45454f 100%)

**Linear gradient** stuff

**Positioning**

**Static** is the default positioning for all elements. If you assign it to an element, you won't be able to move it around with top, right, left, or bottom.

**Sticky** positioning allows an element to stick to a specific position within its containing element or viewport, based on the scroll position.

**Absolute** positioning means being positioned **relative** to a parent. If there’s no parent or ancestor having it’s position set to relative then the positioning will be done with the browser page simply.

**Fixed** positioning as the word says defines or set’s an element’s position to fix and it isn’t relative to anything. The only thing it’s relative to is the web browser page.

border-bottom-right-radius: 40%;

  border-bottom-left-radius: 40%;

  border-top-left-radius: 90px;

  border-top-right-radius: 10p

Always start with ‘border’

The content property is used to set or override the content of the element. By default, the pseudo-elements created by the ::before and ::after pseudo-selectors are empty, and the elements will not be rendered to the page. Setting the content property to an empty string "" will ensure the element is rendered to the page while still being empty.

Elements created by ::before and ::after selectors if set to absolute will have their original element ( from which the ::after or ::before was set) as relative for positioning .. kind of an ancestorial relationship or something.

At times you may do the following to totally reset box-sizing in your styling.

\*, \*::before, \*::after {

  box-sizing: inherit;

}

Notice we are using **inherit**  here which simply means that \* will inherit from his direct ancestor which is no one else than html. (you should have set box-sizing to inherit in html selector). Inherit can be used in so many other situations.. looks interesting.

When **Hidden** value is given to the overflow attribute of an element, that element will hide any subelement incase of an overflow. Overflowing here is synonym of going beyond the defined width, height or just simply the size of that parent object.

**#** is used to for id! And **.**  for classes 😂😂

**overflow-y: auto** this line of code is golden! It enables vertical scrolling. For say a sidebar or some sort.

Keep in mind that whenever you want to create a navbar, the nav links should be <ul> list elements.. It is an approach which help handle and style the nav links in a more convinient and comprehensive way.

You can create variables in class styles. All is make sure the places you’ll apply the styling cascade the element you declared the variables in. In orther words they should be children of that element. also, when using variables you can give a fallback value to variable like so:

  background-color: var(--building-color2, green);

In this case green is the fallback of –building-color2. Perfect.

You can specify where you want a gradient transition to complete by adding it to the color like this:

gradient-type(

color1,

color2 20%,

color3

);

Here, it will transition from color1 to color2 between 0% and 20% of the element and then transition to color3 for the rest.

Cool, you can use **repeating-linear-gradient**  instead of  **linear-gradient** when you want the gradient pattern to repeat again and again.

You can use the bottom of a square to form a triangle.



Just style that bottom box well and yeah.! You can see even now it alreay ressemble triangle.

You can add multiple gradients to an element by separating them with a comma (,) like this:

gradient1(

colors

),

gradient2(

colors

);

  flex-wrap: wrap;

Above is a useful guy.

**Grid**

The **loading** attribute on an img element can be set to**lazy** to tell the browser not to fetch the image resource until it is needed. This lazy guy is very interesting.. you can define the guys who should load first, give priority..! that’s huge. Also when an element’s loading property is set to lazy , it won’t load until it’s necessary .

The Referer HTTP header contains information about the address or URL of a page that a user might be visiting from. This information can be used in analytics to track how many users from your page visit freecodecamp.org, for example. Setting the **rel attribute to noreferrer** omits this information from the HTTP request. : **rel=”noreferrer”**

**h1, h2 , h3 , h4 etc..** are elements use most often in articles, new papers and that kind of stuff.

height="400" != height= 400 : always put values in the “ “

**<blockquote>** should be used for long , block level or standalone quotations. On the other hand  **<q>** should not be used for short, inline quotations. Eg she said “I am sick” the I am sick here could be put in a <q> element.

<hr> are cool too. (horizontal rules)

**The CSS Grid**

Setting the content to have a three-column layout by adding a grid-template-columns property with a value of 1fr 94rem 1fr will create three columns where the middle column is 94rem wide, and the first and last columns are both 1 fraction of the remaining space in the grid container.

**Grid-template-columns**

In other words you’re defining the number of columns directly with their sizes in just one move:   grid-template-columns: 1fr 94rem 1fr;

This is suitable for setting page margins,then content size or something.. Power of css grid!

Or you could use

*Grid-auto-columns****: 150px*** whichwill simply define each new columns to be 150px .

Or you could use this shorthand

  grid-template-columns: repeat(5, 1fr);

Which simply means five columns each taking one fraction of space.

When a page or element is separated like this , if there where 5 elements they’ll be fixed each in a fraction automatically

**Important**   
**Grid-template-columns**: repeat (auto-fill, 300px) ;

Notice here we are using auto-fill instead of the number of columns. This means that each column will occupy 300px and the number of columns will increase and the screen grows.

The number of rows is therefore defined as:

**Number\_of\_columns = ( *screen\_size* / 300px )**

The number of columns will depend and defined according to the screen size. For me this is the best **responsive** feature of the css grid.

.

Or even better

Grid-template-columns(auto-fill, **minmax(250px, 400px)**;

Here min max is used instead of the hard 300px . Meaning that the grid items will stretch up to 400px before a new column is creating (still respecting the formula above )

**Grid-template-rows**

Does same thing as grid-template-columns but for rows

*Grid-auto-rows****: 150px*** will simply define each new row to be 150px regardless of the number.Pixels was used there as unit but any other unit could be used.

The same above goes for the rest.

**Grid-column**

Use grid-column to define the column span a grid item will occupy in the grid. By default it is 1 column with.

You can use the following approaches:

*Grid-column* = “span 3” // which expands the item on 3 columns

*Grid-column=* 1 / 3 //expands from column 1 to column 3 (1 and 3 included)

*Grid-column= 2*  // *specifies the column to occupy*

*Grid-column= 1 / -1* // from the first to the last.

Note that “*span [number-of-columns]*” defines the number of columns to span on but does not defines which one to start on particularly. Therefore it should be used when defining things in order..

**Grid-row**

Does the samething as column but for rows.

**Grid-template-areas**

Gives averall control and permits to draw the grid , defining areas. This really should be used when you kind of have chaos in your layouts

**Absolute**

After grid and flex ways of a arranging things there absolute which should come as a third option. That’s done by setting a parent as relative and then using *left , right , top, buttom* and stuff to position the child accordingly

**Row-gap**  to add space between rows in a grid.

**fr** standands for fraction

One option is the grid-column property, which is shorthand for grid-column-start and grid-column-end. The grid-column property tells the grid item which grid line to start and end at.

Create a .heading rule and set the grid-column property to 2 / 3. This will tell the .heading element to start at grid line 2 and end at grid line 3.

  grid-column: 1/3;

1 says the column number the element is gon’ start on and the 3 were it will end. In this case it will start at column 1 and end at column 3. Also in case you not certain of how many columns there will be but want an element to extend to the last column or something, use -1 which stands for last. That way the above will be ‘grid-column: 1/-1;’

**Object-fit**  does not only go for images . It tells the browser how to position elements in general with their container. Eg of a value to assign is **cover.**

mmm.. we manipulate vectors almost just like text dude.   
**grid** defaults to column alignment.

  grid-template-columns: repeat(5, 1fr);

The above

That’s how grid functions. You may instead want decide on how much space each element will occupy using the grid-column: 1/3; or grid-row: 1/3; depending on which layout you working with.. it’s fabulous!

The **grid-auto-flow** property permits you to add more content or elements and expect them being on the same row. ( Creating a new column or row (depending on what was specified) for every new element). This property takes either row or column as the first value, with an optional second value of dense.

    grid-auto-flow: column;

However, the algorithm defaults the new column width to be auto, which may not always match your current columns.  
You can override this with the **grid-auto-columns** property.   
eg:   grid-auto-columns: 1fr;

Or grid-auto-rows: 1fr; if you where dealing with rows. Now you’ve defined the size of the future automatically created rows or columns.

Much like Flexbox, with CSS Grid you can align the content of grid items with align-items and justify-items. align-items will align child elements along the column axis, and justify-items will align child elements along the row axis.

 You can create columns within an element without using Grid by using the **column-width** property. This works best with text.. anyways.

To make text blocks start and finish on same line.. that’s no line longer than

the other ( all the lines have the same length) use:

text-align: justify;

The CSS property **float: left**; is used to align an element to the left within its containing element. When applied to a text element or any other block-level or inline-block element, it causes the element to move to the left of its containing element, allowing content to flow around it on the right side. It’s like setting it to absolute or you know making it **float** litterally like the property name says. Float: left is therefore read as let this element float to the left.

Just like in python when you want to put double quotation marks to a string you use single quotations and vice versa, the same goes for css. So property : **“**’**”** or property :**‘**”**’**;

The value  **min-content** generally used in cases like   grid-template-rows: repeat(3, min-content); is used to tell css that each row size (in this case ) will be set to the minimum size required to contain it’s content. In short, the section size will grow with the content. Nice.

In css grid The **gap**property is a shorthand way to set the value of**column-gap** and **row-gap** at the same time. If given one value, it sets the column-gap and row-gap both to that value. If given two values, it sets the row-gap to the first value and the column-gap to the second.

The place-items property can be used to set the align-items and justify-items values at the same time. The place-items property takes one or two values. If one value is provided, it is used for both the align-items and justify-items properties. If two values are provided, the first value is used for the align-items property and the second value is used for the justify-items property.

In modern web development it is we use **<i></i>** for icons.. but span is a better alternative.

Obersve normal youtube watch link:

<https://www.youtube.com/watch?v=3cEQfNZ_F1w>

Vs

Youtube embed link:

<https://www.youtube.com/embed/3cEQfNZ_F1w>

The second one is used to embed videos in html. I think with js or next you’ll have an algorithm that will convert normal watch to embed video links.

This is the standard size for youtube videos: width="427" height="240"

Nice!.

It is normal for the scroll bar to be above the header navbar.

Papa, the html skeleton of a standard webpage is the following

Html  
 head  
 body  
  **header  
 main  
 footer**  
 body  
Html

Notice how header main and footer are found in the body element. Okay.

Jazz logo Image link: https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTL08X3MKVfHgOwi88MQ\_wKtm3RTY4Euc\_qyA&usqp=CAU

It is very, very important to place your media queries at the bottom of your CSS, to avoid media query effects from being overwritten by other syles .

To draw a circle , simply width and height should be equal then set border radius to be 50%;

**Very important ( key to responsiveness )**

Looks like setting the whole page container to relative is a good idea 🤔🤔. Apparently that way your max-width and min-width stuff will obey

The **Transform-origin** property is used to set the point around which a CSS transform is applied. Eg : around which point will an object rotate.

For instance doing this:   transform-origin: 0% 0%;

Will imply setting the transform point( eg : rotation axis) of the object 0% from the left and 0% from the top respectively. Defining the center will generally by 50% 50%

The **animation-name** property is used to link a @keyframes rule to a CSS selector. The value of this property should match the name of the @keyframes rule. Give your .wheel selector an animation-name property set to wheel.

The **animation-iteration-count** property sets how many times your animation should repeat. This can be set to a number, or to infinite to indefinitely repeat the animation.

The **animation-timing-function** property sets how the animation should progress over time.

Use the **skew** transform function, which takes two arguments. The first being an angle to shear the x-axis by, and the second being an angle to shear the y-axis by.

This is how to target all descendents of an element:

**element-name \* {**

**}**

Notice the asterisk.

::before and ::after stuffs when mastered and well manipulated can really serve 🤔

cursor: not-allowed 🤔.. nice. That’s the red crossed circle.

This is what we use for those inline minor transitions:

**transition:** property parameter1 parameter2 … ;

example:

btn{

transition: transform 1s ease-in-out 0s;

}

You could also do this :

Transition : all 1s ease-in-out 0s;

// notice we selected all the transitions instead of just one. Perfect!

**BEM Naming convention**

**B**lock **E**lement **M**odifier

**Blocks**

are main elements of your website design the could be *nav, footer, card , carosel etc)* . We write them just like they appear. That’s eg .carrosefl {} .navbar{}

**Elements**

They are sub entities or sub bodies of blocks. They are written like this: “block**\_\_element**” . Notice the underscore between block and element.

Something is set to be a block element when it a block of element . Let’s take an example , let’s say we want buttons found on a carossel have their special kind of styling . We will call those buttons or give them a class of “carossel**\_\_button**” . They are buttons yes, but they are carossel buttons. If the buttons on carossels where the same used everywhere in the page, we will just call them buttons, but since that kind of button is unique to carossels, we write it that way.

They do not represent the structure. Meaning we may want to design a button in a navbar and this what we should avoid: header\_\_nav\_\_btn . Just do header\_\_btn. The normal chronology will go thus : header then navbar then btn right? But we don’t have to list of them, the goal or the goal Is not to describe the structure! ut to identify the main block , then pick the element of that block we want to style.

Finally **Modifiers**..

help express different **types , genre** or **states** of the same base component. They are named like this : block**--modifier** If for instance we were to style cards differently for classic , premium and gold subscription, This is how we will go about it.

Card**—classic{}**Card**—premium{}**

Card**—gold{}**

Notice that as desribed previously, classic , premium and gold are just different **types or genres** of Card. Perfect. Another example could be –highlighted , --compact, --secondary, --disabled etc. Note that this examples will describe a type of block. Ok.

An advantage of BEM is that watch this :

.header btn{} will have more specificity than .header\_\_btn. This helps to better hand priority and specificity in you styling. Perfect!

Comprehensive Documentation help: [CSS Entries | Codrops (tympanus.net)](https://tympanus.net/codrops/css_reference/)

**Refresher**

Link styling should have the following order:

a{}

a:visited{}

a:hover, focus{}  
a:active{}

Just keep **Avha** for short.

- Notice you should always associate hover to focus.

- You can use the following  
 a:any-link{ } *// It will select both visited and none visited links.*

- :focus-within *//matches an element if it or any of it’s descendents are in focus state. (suitable for navbars)*

Typography

line-height : 1.5;

letter-spacing : 1em;

text-indent : 2em; ( Indents the first line of text . Suitable for paragraphs)

word-spacing : 1em;

font-style: **oblique**; (which is a stronger italic)

**font used in web design**

* Roboto
* Libre Franklin
* Raleway
* Inter
* Source Sans Pro

Text-tranform: uppercase != Text-tranform: capitalize;

List-style

List-style-image: ule(“ “); //This replaces the list buttets by the image pointed by the url.

:nth-child(**even**) will select even children  
:nth-child(**odd**) will select odd children.

For ordered lists, you can use the property

List-style-type: lower-alpha; // sets points to alpha in lowercase.

Columns

Short hand to grid column.

Column-count : 4; *// defines the number of columns*

Column-width : 250px; *// gives a minimum width of 250px to columns.*

Columns : 4 250px; *// is a shorthand to the above lines.*

Column-rule : 1px solid white; *//defines a rule between columns. Note that the rule(s) disapear when there’s only one column*

Column-gap : 2rem; *//defines the gap between columns.*

Column-span : all; *// makes an element span throught out all the columns*

Animations & Transitions

btn{

transition: transform 1s ease-in-out 0s;

}

You could also do this :

Transition : all 1s ease-in-out 0s forwards;

*// notice we selected all the transitions instead of just one. Perfect!*

*// notice the* ***forward***  *keyword which basically says that the object or element should remain in the state it has at the end of the animation. The default is forward*

Shorthand

Do this :is(header, footer) .btn { }

Instead of header.btn , footer.btn *//it’s a shorthand*

Clipping image to text

<p class=”clip”>Name</p>

.clip{

Background-size: 100%;

-webkit-background-clip: text;

Color: hsla(0, 0%, 0%, 0.196);

}

*//This will clip the image to the text*

**Kevin Powell**

Scroll-based animations ( check the responsive web dev repo)

We add images in html! CSS more for when the image is not static

The *inherit*  property is a monster property! Use it well and it will do wonderful stuff!

**Images secrets revelled!!**

**Html** : object-fit: cover

**CSS**: background-size: cover

This is a stunning discovery 😅😁. So henceforth depending on what you wann do.. those are the options! 🤔😄

More amazingly, you can even use the css and inject it in the html like this:

                <div class="project\_\_tile" style="background-image: url(' https://images.pexels.com/photos/230544/pexels-photo-230544.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=1 ' ); ">

Notice the style attribute . Yes! We are styling the background of the element directly in the html 😅😍.

***Pexels*** *is a nice option of online images!* 🤔