**fontsizes**

**CSS NOTES :-**

TEXT(font weight, font size, font family, Text align, )

**Font weight:-**

* **font-weight**: This property sets the thickness of the characters in a text. Common values are **normal**, **bold**, **bolder**, and **lighter**.

**Example:**

P {

font-weight : bold;:

}

**Font size :-**

* **font-size**: T: This property sets the size of the text. You can specify the size using various units like pixels (**px**) , ems (em), rems (**rem**), percentages (**%**), etc.
* if we have not specified a specific font size, it takes the default size of 16px;

**Relative font sizee**

**rem (root em):**

rem units are relative to the font size of the root element (usually <html> or :root in your CSS).

If the root font size is set to 16px, 1rem is equal to 16px.

:root {

font-size: 16px;

}

body {

font-size: 1.5rem; /\* 1.5 times the root font size, so 24px \*/

}

body{  
font-size:50%; /means 50% of the root fontsize- 16/2=8px;

}

**em:**

em units are relative to the font size of the parent element.

If an element has a font size of 16px, 1em is equal to 16px. If a parent element has a font size of 20px, 1em within that element is equal to 20px.

body {

font-size: 16px;

}

p {

font-size: 1.5em; /\* 1.5 times the font size of the parent element, so 24px \*/

}

In summary, while both rem and em are relative units, rem is relative to the root font size, providing a more global reference point, whereas em is relative to the font size of the immediate parent element, making it more context-specific. rem is often preferred for overall layout and sizing consistency, while em is useful for creating designs that scale based on the size of their immediate parent elements.

**Absolute font size:-**

giving absolue fixed size to the elements in px,. fontsze:10px;

**Example:**

P {

font-size: 10px;

}

**Font family :-**

To include custom fonts from google navigate to google.fonts.com, and select the font u needed and copy the font code in text and include under the <style> </style> tag or include in the head part

this property sets the font for an element. It can take a specific font name, a generic font family (like **sans-serif**, **serif**, **monospace**), or a list of font names separated by commas.

H1{  
font-family: “Times new roman”, serif;

}

Linking external style sheets-

<link rel=”stylesheets” href=”filename.css”;p

ee {ed;  
}

**Selectors** - By classname, by id , selectors

**Text transform**-captilazie, lowercase, uppercase

Specificity and inheritance :-



combining css selectors ;-

**group selector**

h1, h2{  
color :red;

}

**child combiner**-

child nested in a parent tag then we can apply css to particular child node only

parent tag > child {

color:red;

}

**Descendent combiner**-

child can be nested anwhere inside parent.. we can apply css to child one,

parent child{  
color: ;

}

**Chaining** -

applying when all the selectors are same, for ex having same id and same class names. but needed to apply for specific item then we use this

<p class="done">Do these things today!</p>

<ul class="list">

<li>Wash Clothes</li>

**<li class="done">Read</li>**

**<li class="done">Maths Questions</li>**

</ul>

ex;- elementname.classname or element#id { -no space between  
color: ;

}

**Positioning in css:-**

**1)Static Positioning:-**

If you have inserted any html element then this positioning is inserted. It is a Html default flow.

ex:- background color: red;

width :200px; height:200px;

**2)Relative positioning**

Position relative to its default position. when we insert default html and we want to change its position then we apply this relative position in order apply changes from the default position.

ex:-

background color: red;

width :200px; height:200px;

position : relative;

left:20px;

top:10px;

**3)Absolute Positioning:-**

Position relative to the nearest positoned ancestor or top left corner of the webpage.

\*it checks for the nearest ancestor in order to set if its not having any ancestor then it will be positioned with top left corner of webpage.

**--> Z-Index :-**It is a property which sets the stacking order of the overlapping elements.every default element is having z-index as 0 by default.

In order to achieve z-index apply z-index syntax in absolute position elemnent and make it as -1.(lesser then default)

ex :- z-index : -1

**4) Fixed Positioning:-**

Position relative to top left corner of the browser window.It is fixed even when u scroll window it is fixed at same position

**DISPLAY PROPERTIES IN CSS:**

**Inline:-**

Definition: An inline element is a type of element that does not start on a new line and only takes up as much width as necessary. It flows within the content and allows other elements to sit beside it horizontally.

**Examples:** <span>, <a>, <strong>, <em>

span {

display: inline;

}

**Block:-**

**Definition**: A block-level element is an element that starts on a new line and stretches the full width of its container. It creates a "block" of content and pushes subsequent elements onto the next line.

Examples: <div>, <p>, <h1> to <h6>

div {

display: block;

}

**Inline-block:**

**Definition**: An inline-block element is a combination of both inline and block elements. It remains inline (it doesn't start on a new line) but can have its width and height set like a block-level element. It allows elements to sit side by side horizontally and still be styled like block-level elements. it takes the size of the content size.

Examples: <img>, <button>

img {

display: inline-block;

}

**None:-**

**Definition**: The none value for the display property is used to hide an element. The element is not rendered on the page, and its space in the layout is completely removed.

ex:- display:none, list-style:none (to remove dots for ul for li's)

**Example**:

.hidden-element {

display: none;

}

**Float Property in CSS:-**

he float property is used to push an element to one side of its containing element, allowing other elements to wrap around it. The values for the float property are usually "left" or "right.

**ex:-**

.float-example {

float: left;

width: 50%; /\* or any desired width \*/

}

**to create resposnsive website we can use 4 different methods as below:-**

1)Media queries

2)Css Grids

3)Css Flexbox

4)External Frameworks ex:- Bootstrap

**1)Media Queries in CSS:**

Media queries are used to apply different styles for different devices or screen sizes. They allow you to specify different styles based on characteristics like screen width, height, device orientation, and more. Media queries are commonly used for creating responsive designs that adapt to various devices.

***ex:-***

@media(min-width:100px) and (max-width:600px) {

.logo{

width: 100px;

}

h1{

font-size: 2rem;

text-align: center;

}

**3)FLEXBOX in CSS:**

[**https://css-tricks.com/snippets/css/a-guide-to-flexbox/**](https://css-tricks.com/snippets/css/a-guide-to-flexbox/)

**Flexbox Definition:**

Flexbox is a one-dimensional layout system that allows you to design a layout structure along a single axis (either horizontally or vertically). It simplifies the process of distributing space and aligning items within a container, making it particularly useful for creating responsive and dynamic layouts.

.It will create a block(container) and store them in it and its width will be based on the content value.

<style>

.flex-container {

***display: flex;***

justify-content: space-between;

align-items: center;

height: 200px;

background-color: #f0f0f0;

}

.flex-item {

width: 100px;

height: 100px;

background-color: #3498db;

color: #ffffff;

text-align: center;

line-height: 100px;

}

</style>

<body>

<div class="flex-container">

<div class="flex-item">Item 1</div>

<div class="flex-item">Item 2</div>

<div class="flex-item">Item 3</div>

</div>

</body>

**Properties :-**

**1)flex direction:-**

Defines the direction of the main axis (the primary axis along which flex items are laid out).

**Values**: row, row-reverse, column, column-reverse.

**NOTE:- the flex-direction is only applies at container class level not on child elements.**



.container{

flex-direction:row or coloumn;

}

Default value: row.

**2) flex basis:-**

it is the size of the block or element u wanted to extend. it is applied to the elements not on the container level.

.div{  
**flex-basis:20px;** //20px width increses if flex-direction is row and 20px height increases if flex direction is coloumn.

}

**3) flex wrap:-**

**nowrap (default):** all flex items will be on one line

**wrap**: flex items will wrap onto multiple lines, from top to bottom.

wrap-reverse: flex items will wrap onto multiple lines from bottom to top.

**4) Justify content:-**



**flex-start (default):** items are packed toward the start of the flex-direction.

flex-end: items are packed toward the end of the flex-direction.

**start**: items are packed toward the start of the writing-mode direction.

end: items are packed toward the end of the writing-mode direction.

**left**: items are packed toward left edge of the container, unless that doesn’t make sense with the flex-direction, then it behaves like start.

**right:** items are packed toward right edge of the container, unless that doesn’t make sense with the flex-direction, then it behaves like end.

center: items are centered along the line

**space-between:** items are evenly distributed in the line; first item is on the start line, last item on the end line

**space-around**: items are evenly distributed in the line with equal space around them. Note that visually the spaces aren’t equal, since all the items have equal space on both sides. The first item will have one unit of space against the container edge, but two units of space between the next item because that next item has its own spacing that applies.

**space-evenly:** items are distributed so that the spacing between any two items (and the space to the edges) is equal

**.container {**

**justify-content: flex-start | flex-end | center | space-between | space-around | space-evenly | start | end | left | right ... + safe | unsafe;**

**}**

**5)Align-Items:-**use flexwrap as no wrap and use this,



This defines the default behavior for how flex items are laid out along the cross axis on the current line. Think of it as the justify-content version for the cross-axis (perpendicular to the main-axis).

**.container {**

**align-items: stretch | flex-start | flex-end | center | baseline | first baseline | last baseline | start | end | self-start | self-end + ... safe | unsafe;**

**}**

**stretch (default):** stretch to fill the container (still respect min-width/max-width)

**flex-start / start / self-start:** items are placed at the start of the cross axis. The difference between these is subtle, and is about respecting the flex-direction rules or the writing-mode rules.

**flex-end / end / self-end:** items are placed at the end of the cross axis. The difference again is subtle and is about respecting flex-direction rules vs. writing-mode rules.

**center**: items are centered in the cross-axis

**baseline**: items are aligned such as their baselines align

**6)FLEX FLOW:-**

flex-flow is a shorthand property in CSS that combines the flex-direction and flex-wrap properties. It allows you to set both the direction of the main axis and the wrapping behavior of flex items in a single declaration.

**Syntax:**

flex-flow: <flex-direction> <flex-wrap>;

**FLEX-SIZING:-**

max-width and min-width

**Content width < Width < Flex Basis < Min width and Max Width**

\*by default if we give elements for the flex box container it takes the default behaviour ,when we shrink the page it is to check the possible min width of a word in a div or paragraph. once we exceeded the min width the words will shrink and cant be appeared. it is default behaviour of flexbox without setting any sizing properties.

\* max-width look at the longest possible line of Text and min-Width looks at the longest word to deternmine the min-width

\*if we give max or min width also flex-basis, if max or min width is > than flex basis then, it applies the max or min width. if it is smaller then it applies the flex basis.

\*if we give flex-grow and flex shrink both on



in this case flex-basis is ignored because grow and shrink both are on. so it shrinks upto the min width of size of word and also grows and fills the empty container until it filled by flex elements.

**case 2:**



in this case it respects the flex-basis which is 100px and also grows and fills the remaining left container. it doesnt keep the container empty it fills it with flex items when we increase our screen.

**case 3:**



in this case it applies flex basis by default, and it not going to grow to fill the space but when we shrink it is going to shrink beyond flex basis until it reaches its min width(if not set it goes for min content width)



here all are equal but different syntaxes in shortcuts.

1 1 0 - grow-1,shrink-1,flex-basis-0;

1 - grow-1,shrink-1,basis-0;

order

,justify-content:space-between,space-around,space-evenly,Flex start.,Flex end, center,

Align-items :- use flexwrap as no wrap and use this, center,start,end , baseline,stretch

Align self

**4) GRID IN CSS:-**

CSS Grid is a layout system in CSS that allows you to create two-dimensional layouts with both rows and columns. It provides a powerful and flexible way to design complex web layouts. Here are some key definitions and basic syntaxes for CSS Grid:

**display: grid;** or display: inline-grid;

Defines the element as a grid container.

**grid-template-rows and grid-template-columns**

Specifies the size of rows and columns in the grid.

You can use values like auto, 1fr (fractional unit), fixed sizes, etc.

**1fr,2fr -Fractional units. (size in grid)**

The fr unit in CSS Grid stands for "fractional unit." It represents a fraction of the available space in the grid container. The 1fr unit means one fraction of the available space.

When you set the grid-template-columns or grid-template-rows property to use fr units, you are essentially dividing the available space into equal fractions. For example:

**.grid-container {**

**display: grid;**

**grid-template-columns: 1fr 2fr 1fr;**

**}**

The first column takes up 1 part of the available space.

The second column takes up 2 parts of the available space.

The third column takes up 1 part of the available space.

So, if the total available width is, for example, 1200 pixels, the columns would take up 300px, 600px, and 300px respectively. The fr unit is a flexible and powerful way to create responsive and dynamic layouts in CSS Grid.

**syntax:-**

.container{

display : grid;

grid-template-rows: 100px 200px;

/\* Two rows, 100px and 200px \*/

grid-template-columns: 1fr 2fr; /\* Two columns, 1fr and 2fr \*/

**or**

grid-template :100px 200px / 100px 200px;

(rows) (coloumns)

}

**GRID SIZING :-**

1)fixed size

2)Auto size

3)fractional size

4) Min max size

5)Repeat

6)Test

**1) Fixed Size:**

.grid-container {

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

grid-template-rows:20px 100px;

(or)

grid-template:100px 200px/ 200px 100px; (100px row and 200px row and 200px and 100px coloumn will be set)

}

**Explanation**:

Columns are set to a fixed size of 100px each.

All columns will have the same width.

**2) Auto Size:**

**ex:-** grid-template-rows: 100px auto; (auto in row is applied as content height and in coloumn it applies 100%of the width)

grid-template-coloumn:50px auto;

auto means its occupies the total width which is available for that div, might be the row or might be the coloumn.

.grid-container {

grid-template-columns: auto auto auto;

}

Explanation:

Columns will automatically size based on their content.

The width of each column is determined by the content it contains.

**3) Fractional Size:**

.grid-container {

grid-template-columns: 1fr 2fr 1fr;

}

**Explanation:**

Columns are set to fractional units (fr).

The available space is divided into parts, where the second column (2fr) takes up twice as much space as the other columns.

**4) Min Max Size:**

.grid-container {

grid-template-columns: minmax(100px, 1fr) minmax(200px, 2fr) minmax(100px, 1fr);

}

***Explanation:***

Columns are set with a minimum and maximum size using minmax().

The second column (minmax(200px, 2fr)) can be at least 200px wide but can expand to take up any remaining space (2fr).

**5) Repeat:**

.grid-container {

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

}

**Explanation:**

The repeat() function is used to repeat the specified pattern.

In this example, it repeats 1fr three times, creating three equal-width columns.

**6) Test:**

.grid-container {

grid-template-columns: 100px minmax(200px, 1fr) auto repeat(2, 1fr);

}

**Explanation:**

A combination of different sizing techniques in one grid layout.

First column is 100px wide, second column has a minimum size of 200px and can expand, third column is automatic size based on content, and the last two columns are equal-width fractional columns repeated twice.