**CSS**

Layout

Style

Selection Methods:

Given a paragraph in HTML that is as: *<p id=’abc’ class=’alphabets’>abcdefghij</p>*

* By the tag name.
* By the class name.
* By the Id.

All the following statements in CSS will change the color of this paragraph above to red:

*p {*

*color: red;*

*}*

*.alphabets {*

*color: red;*

*}*

*#abc {*

*color: red;*

*}*

Combinators

It is possible to target specific classes of elements in the DOM hierarchy as follows:

* Descendants (all the elements below a certain node, including the grandchildren)
* Children (all the elements that are directly under a certain node)
* Siblings (all the elements in the same level as a certain node)

From the above classes we can specify, even further, the specific elements we want:

Descendants: parentNode element(s) {…} A space separates the two. The first element is called the pre-selector and the second the second one is the specific selector.

Children: parentNode > element(s) {…} A greater than sign separates the two.

(N/B Fragile) Siblings: element1 ~ element2 {…} Targets all the siblings of element1

element1 + element2 {…} Targets the adjacent element (the one following immediately) of element1

Given more than one element having the same class name say, “regular text” but you want to target elements of one type, say, paragraphs, you can specify these as:

*p[class=”regular-text”] {…}*

To specify more than one target, given they share some common properties, you can separate the targets using a comma.

E.g, target1, target2 {…}

You can also give these targets more than one class name each (the same name), separated by a space in the HTML.

E.g. *<div class=”box-1 box”></div>*

*<div class=”box-2 box”></div>* In both these tags’ class names, there is “box”. Therefore in CSS, .box {…} will refer to both of them.

Pseudo Classes

These are used to target certain elements, but when they are in a certain state. For example, changing the opacity (to noticeably indicate that the button is clickable) and the cursor type of a button when a mouse hovers over it can be done as:

*button:hover {*

*cursor: pointer; The little pointed finger*

*opacity: 0.8;*

*}*

Factors affecting CSS specificity

* Location of the CSS rule in the file. The rules at the bottom of the file take precedence.
* Specificity of the CSS rule in the file. The rule is evaluated as where it targets a generic tag, a class name, or an Id. The highest precedence is for the Id, followed by the class, then for the tag. However, when combinators are involved, it becomes tricky. Here, calculations are involved:
* Count the number of Id selectors in the rules competing rules.
* Count the number of class selectors in the rules competing rules.
* Count the number of tag selectors in the rules competing rules.
* Concatenate the results into a single digit as IdsClassTag.
* The rule with the highest value, despite its position in the file, takes precedence

Inline Styles

You can specify the style of a HTML element within the HTML itself using the “style” keyword, and add CSS code, as:

<p style=”color: red; padding:20px;”>Some Text</p>

These inline rules take precedence over all other rules except one:

p{

color: blue !important;

padding: 10px !important;; This “!important” keyword takes the highest precedence but should be used sparingly.

}

Properties

padding: 20px; //10-20px

background-color: red;

color: white; // other ways of specifying color include: hex-codes, rgb, rgba. Check: adobe color &or search rgb to hex on google

width: 20px;

height: 20px;

line-height: 20px; // Set the element to occupy the height specified.

text-align: center;

box-shadow: 2px 3px 7px 0px rgba(0,0,0,0.35)

border-radius: 5px;

border: 5px solid green; //4-7px

overflow: hidden; // Rounded Corners

font-size: 16px; // default

font-family: “…”,”…” // The first is the one it will take, the second is the fallback font. Similar to “alt” in images.

Font-weight: 900; //bold

background-image: url(…);

width: 100vw;

height: 100vh;

background-size: cover;

background-repeat: no-repeat;

background-position: bottom-right;

margin: 20px // shorthand for all sides.

Margin: 10px 5px 10px 10ox // top,right,bottom,left respectively

*Box Model (Height, Width, Padding, Border, Margin)*

display: block // Elements occupy their own space. Adhere to their width and height properties. By default, occupy the whole line (the whole line).

inline // Elements are willing to share the available space on a line. Do not adhere to width and height properties

inline-block // Hybrid in that elements are willing to share lines but they also adhere to their width and height properties, if set.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

box-sizing: content-box; // The width and height specified apply to the actual content of the element.

border-box // The width and height specified apply to the actual content of the element as well as to the padding and border of the element.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

This concept is used in what is called the CSS Reset, where the entire CSS file is given a box-sizing of border-border and all other elements (that fall within it) inherit this property from this root element. This is done as:

html {

box-sizing: border-box;

}

\*, \*:before, \*:after {

box-sizing: inherit;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Position: static // Default on the element under focus.

Fixed // Doesn’t move even on scrolling. It is therefore used in navbars. However, it occupies no space in the webpage. Consequently, other elements are brought under it. To fix this, we set a “margin-top” property on the other elements (e.g. as classes) and assign it some value to ensure that these element are always below where this fixed element(s) are.

margin-top: 80px // For example if the height of the navbar is 60px. This forces all other elements to stay 20px away from the navbar. Set on the other elements, not the navbar itself.

Another issue is that the navbar should often take up the whole width. To guarantee this, on the navbar:

Width: 100% or 100vw // 100vh for height

To ensure that this navbar starts at the margin, clear any padding on the parent node that may be set by default, then “0 index” it on the body tag/view port. I.e:

Padding: 0px; // on the parent node

top: 0px // on the actual navbar

left: 0px // on the actual navbar

Relative // Activates the z-axis on the element. By default, the element assigned this property is elevated above the others, so it goes on top of other elements. To correct this for some specific element that is apart, say the navbar, set this other elements z-index higher, e.g. *z-index:2;* It affects other properties. E.g. when the same element with the relative position property is given a property of top:10px, it elongates downwards by 10px.

Absolute // Often used when some higher element, say a parent element, has a position property of ”relative”. If there is none, then the body element is used. With this property, other properties such as top, left, right as implement relative to the element above with the “relative” position property.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Measurement Units

Pixels

Em // Fonts - A fraction of the parent element

Rem //Fonts - A fraction of the root <html> element.

Percentages // Responsive programming

Media Queries and Breakpoints

@media <screen and> (min-width: someWidth px) {

Element {

someProperty: …;

}

} - Apply these CSS rules if the screen size >= someWidth

@media

only screen

and (min-width: someWidthA px)

and (max-width: someWidthB px)

{

Element {

someProperty: …;

}

} - Apply these CSS rules if the screen size falls between someWidthA and someWidthB

Responsive Design Checklist

Fluid Layout

No Overflow

Decent aspect ratio on images

No columns of text on mobile

700px max width of text on the screen (16px font-size)

**Flexbox**

**N/B**

“Flex” display property is declared as: “*display: flex;”*

It is usually used on divs.

A element can be both a flex container and a flex item.

A flex container is a parent div whose direct children inherit the “flex” property.

A flex item is the child of the parent div which inherits the “flex property”. This flex item can have children of its own. These children are by default having a display property of “block”. Explicitly declaring a display property of “flex” on this flex item makes its children flex items, but in relation to it, not to the grandparent container.

Flex Container Properties

*flex-direction: row;* *//or row-reverse, column, column-reverse* – Defines how the children flex items will flow. A flex direction of row makes the children flow from left to right. A flex direction of column does the opposite. The flex direction also defines what the main and cross axes will be. A flex direction of row makes the x-axis the main axis, and the y-axis as the cross axis. A direction of column does the opposite.

*justify-content: start; //or end, or center –* Similar to the align text feature on ms-word where start is left-align, end is right-align and center is center-align. However, the justify-content alignment depends on the flex-direction so that if it is set to “column” , then start corresponds to top, end to the bottom and center to the middle of the vertical axis.

Other values for “justify-content” include “space-between”, “space-around” and “space-evenly”.

space-between – Arranges the flex items (children) with even white spaces between them and with the first and last items being on the border.

space-around – Arranges the flex items (children) with even white spaces between them but with the first and last items being padded a little bit inside the parent container.

space-evenly – Arranges the flex items (children) with even white spaces between them and with the first and last items also being spaced evenly from the border of the container.

*Align-items: normal //*  – Similar to justify-content but for alignment on the cross-axis. So if the “flex-direction” is row, the “align-content” works vertically because the x-axis is the main axis. Its values work as:

normal: Stretches the width/height of the flex-items to fit along the cross-axis, unless explicit values for the same are given.

flex-start: Stops stretching the flex-items and pulls them to the top of the cross-axis. However, this pulling goes up to where the content can fit.

flex-end: Similar to “flex-start” except that the flex-items are pulled to the bottom of the cross-axis.

center: Similar to “flex-start” and “flex-end” except that the flex-items are pulled to the center of the cross-axis.

baseline: Similar to “flex-start”, “flex-end” and “center” except that the flex-items are pulled to where the bottoms of their individual contents align.

*flex-wrap: nowrap; // or wrap* – Defines what happens when the flex-items exceed what can be shown along the main-axis of the flex container. By default (nowrap), the flex items overflow if too many (despite being shrank to a reasonable degree). Ordinary, this is handled by setting an overflow property on the container and assigning it a value of auto as “overflow: auto”. This hides the overflowing content but adds a scroll bar to navigate and see the hidden items. However, this may not be what you want , e.g. when displaying many images. You would rather have them drop to the next line. This can be done via setting a “flex-wrap” property and assigning a value of “wrap” on the same.

*align-content: flex-start; // or other value equivalent to the “align-items” values e.g. “flex-end”, “flex-center” etc.* Working with a flex-wrap of “wrap” gives rise to problems in the alignment of flex-items, e.g. after wrapping, the overflowing items are pushed way down the cross axis, instead of next to the first items. Align-content remedies this so for example in the case above, “align-content: flex-start” removes the spaces between the first and second line, making the items adjacent to each other.

Flex Item Properties

*Align-self: flex-start; // or other value equivalent to the “align-items” values e.g. “flex-end”, “flex-center” etc.* Aligns an individual flex-item relative to the cross axis. It however, stills abides by the “justify-content” property of the flex-container

*flex-grow: 0 // or any other positive integer.* When the flex-items are smaller than the total available space in the flex container some space is left allocated. flex-grow is used to divide this unallocated space amongst the flex items, which then stretch to fill this now-divided space. By default, each flex item has a “flex-grow” value of 0 (no part of the unallocated space is allocated to the flex-item). It works as:

The sum of the flex-grow values of all the flex item is added up. Each flex-item is then allocated some space as per the ratio of its individual flex-grow value against the sum of all the flex-items’ flex-grow values.

*Flex-shrink: 1 // or any other positive integer*. Similar to “flex-grow” but in reverse. It is used to determine how the flex-items will shrink in the event of an overflow, only until the minimum width is reached after which the flex-items will be wrapped. This is because every flex-item has content, as well as padding. This content has a minimum size, lest it disappears. To avoid this, wrapping is done.

Here, with flex-shrink, the “space” that is divided is the amount of overflow. So, for example, if a flex-container has a width of 400px (margins ignored), and each flex-item has an explicit width of 50px, having 10 flex-items is obviously more than the space available. Therefore the overflow width to be divided is that of 2 flex items (2\*50 = 100px). Now, by default each flex-item has a “flex-shrink” value of 1. Therefore each flex-item will shrink by (100/10 = 10px).They will then be displayed having a width of (50-10 = 40px) If this is greater than the minimum width allowable, being made to fit in the flex-container. If it is less there will be a literal overflow needing to be wrapped.

*flex-basis: auto; // or other percentage or explicit pixel amount* - Similar to declaring the width/height of a normal element (depending on the main axis) but with higher precedence. It is more reliable than *flex-grow*.

Auto: A value of auto requires that the flex-item take up space according to its content, unless this space is explicitly declared.

…%: A percentage value assigns the flex-item’s width/height as a percentage of the total flex-container’s size. The sum of percentages needs not to add up to 100%.

*order: <No>;* // some position in the list of flex-items (not zero-indexed) - It specifies the sequence in which the flex-items will be displayed along the main axis. Here, an flex-item declared before another in the HTML can be swapped with another by specifying the number at which you want them to appear. This is useful when designing for responsive design when you may want some items to appear before others when the screen is resized.

flex: 0 1 auto; This is equivalent to:

flex-grow: 0;

flex-shrink: 1;

flex-basis: auto;

Keyboard Shortcuts

elementTag, period, className <Tab > - Creates an element of type ”elementTag”, with class name “className”