# CSS Specificity

**Specificity** is the algorithm used by browsers to determine the CSS declaration that is the most relevant to an element, which in turn, determines the property value to apply to the element. The specificity algorithm calculates the weight of a CSS selector to determine which rule from competing CSS declarations gets applied to an element.

There are four categories which define the specificity level of a selector:

1. **Inline styles**
2. **IDs**
3. **Classes, pseudo-classes, attribute selectors**
4. **Elements and pseudo-elements**

**Note:**

- The universal selector (*) has no specificity value (0,0,0,0)
- The pseudo-class :not() adds no specificity by itself, only what's inside it's parentheses.
- The universal selector and the pseudo-class `:where()` and its parameters aren't counted when calculating the weight so their value is 0-0-0, but they do match elements. These selectors do not impact the specificity weight value.

# CSS units

We can divide CSS units into 2 categories:

1. Absolute Lengths
2. Relative Lengths

### Absolute Lengths

Absolute length units are based on an actual physical unit, and are generally considered to be the same size across devices. However, depending on your screen size and quality, or settings in your browser or OS, there may be some exceptions.

- cm - centimeters
- mm - millimeters
- in - inches (1in = 96px = 2.54cm)
- px - pixels (1px = 1/96th of 1in)
- pt - points (1pt = 1/72 of 1in)
- pc - picas (1pc = 12 pt)

`cm`

Centimeters.

In CSS, `1cm` is roughly 37.8 pixels, or about 25.2/64 of an inch.

`mm`

Millimeters.

In CSS, `1mm` is roughly 3.78 pixels, or 1/10th of a centimeter.

`in`

Inches.

In CSS, `1in` is roughly 96 pixels, or about 2.54cm.

`px`

Pixels, or `px`, are one of the most common length units in CSS.

1 pixel is formally defined as 1/96 of an inch. All other absolute length units are based on this definition of a pixel.

But when that standard was originally formulated, most monitors had a resolution of 1024 x 768, and a DPI (dots per inch) of 96.

Screens on modern devices have much higher resolutions and DPIs, so a line that's 96 pixels long may not measure exactly 1 inch, depending on the device.

Even though sizing in pixels can vary across devices, it's generally considered better to use pixels for screens.

If you know that your page will be printed on a high quality printer, then you may consider using another unit like `cm` or `mm`.

`pt`

Points.

In CSS, `1pt` is roughly 1.3333 pixels, or 1/72th of an inch.

`pc`

Picas.

In CSS, `1pc` is roughly 16 pixels, or 1/6 of an inch.

## Relative Length Units

Relative length units are relative to something else, perhaps the size of the parent element's font, or the size of the viewport. The benefit of using relative units is that with some careful planning you can make it so the size of text or other elements scales relative to everything else on the page. Some of the most useful units for web development are listed in the table below.

`em`

Font size of the parent, in the case of typographical properties like `font-size`, and font size of the element itself, in the case of other properties like `width`.

```
.container {
  font-size: 16px;
}
.container p {
  font-size: 1em;
}
.container h2 {
  font-size: 3em;
}
.container h3 {
  font-size: 2em;
}
```

`rem`

Root `em`. This relative unit is not affected by the size or setting of a parent element, and is instead based on the root of the document. For websites, the root of the document is the `html` element.

```
p {  font-size: 1.25rem;}
```

In most browsers, the default font size is 16, so the `font-size` of `html` elements is `16px`. So in this case, `p` is `20px` (16 * 1.25).

`%`

Percentages, or the percent size relative to the parent's size:

```
div {  width: 400px;}
div p {  width: 75%;}
```

Since the parent's width is `400px`, the width of the inner paragraph is be `300px` (400 * .75).

`vw`

View width. `1vw` is 1% of the width of the viewport.

```
body {  width: 100vw;}
```

Since the `body` element is set to `100vw`, or 100% of the viewport's width, it will take up the full width available to it. So if you resize your browser  to 690 pixels wide, then the `body` will take up all 690 pixels in width.

`vh`

View height. `1vh` is 1% of the height of the viewport.

For example:

```
div {  height: 50vh;}
```

The `div` will fill 50% of the viewport's height. So if the browser window is 900 pixels high, the `height` of the `div` will be 450 pixels.

`vmin and vmax`

Viewport minimum (`vmin`) and viewport maximum (`vmax`) units are based on the values of `vw` and `vh`.

`1vmin` is 1% of the viewport's smallest dimension, and `1vmax` is 1% of the viewports largest dimension.

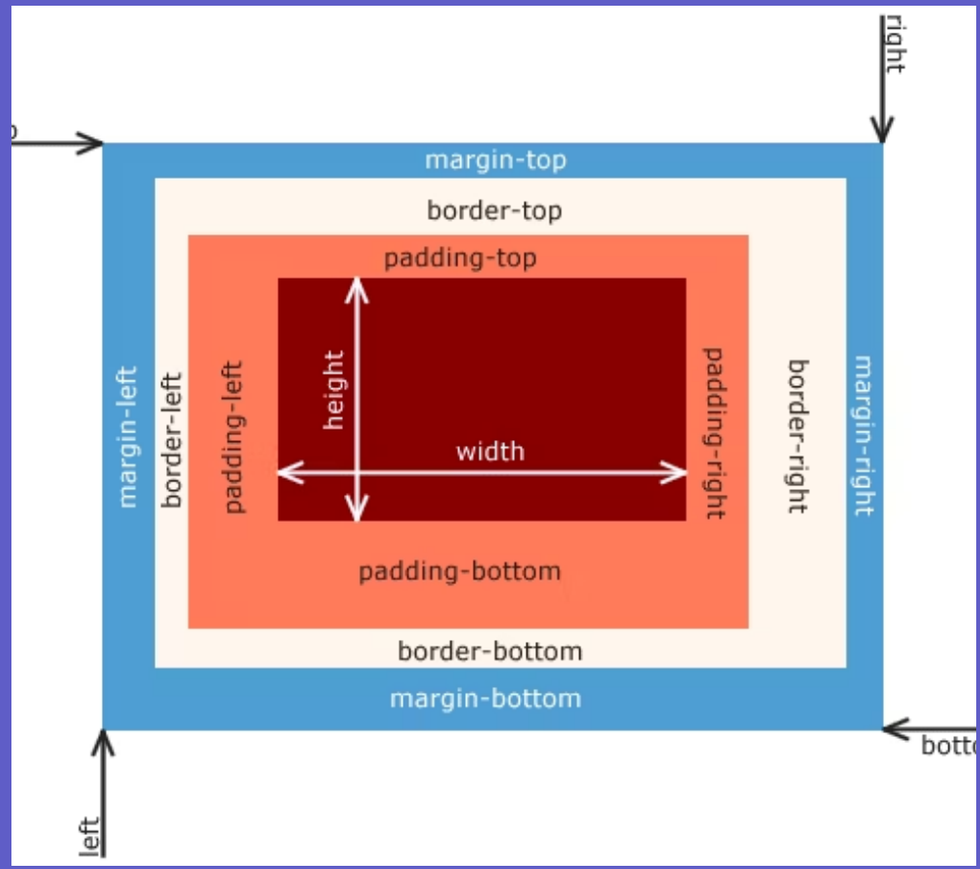
For example, imagine a browser window that is 1200 pixels wide and 600 pixels high. In this case, `1vmin` is `6px` (1% of `vh`, which is smaller at 600 pixels). Meanwhile, `1vmax` is `12px` (1% of `vh`, which is the larger value at 1200 pixels).

# Css Box Model

Everything in CSS has a box around it. It consists of:

- Content - The content of the box, where text and images appear
- Padding - Clears an area around the content. The padding is transparent
- Border - A border that goes around the padding and content
- Margin - Clears an area outside the border. The margin is transparent

```
div {
  width: 350px;
  padding: 10px;
  border: 5px solid gray;
  margin: 0;
}
```



# Box Sizing

The `box-sizing` CSS property sets how the total width and height of an element is calculated.

By default in the CSS box model, the `width` and `height` you assign to an element is applied only to the element's content box.

`content-box`

This is the initial and default value as specified by the CSS standard. The `width` and `height` properties include the content, but does not include the padding, border, or margin. For example,

`.box {width: 350px; border: 10px solid black;}` renders a box that is 370px wide.

Here, the dimensions of the element are calculated as: *width = width of the content*, and *height = height of the content*. (Borders and padding are not included in the calculation.)

`border-box`

The `width` and `height` properties include the content, padding, and border, but do not include the margin. Note that padding and border will be inside of the box. For example,

`.box {width: 350px; border: 10px solid black;}` renders a box that is 350px wide, with the area for content being 330px wide.

Here the dimensions of the element are calculated as: *width = border + padding + width of the content*, and *height = border + padding + height of the content*.

# Overflow

The overflow property controls what happens to content that breaks outside of its bounds: imagine a `div` in which you've explicitly set to be 200px wide, but contains an image that is 300px wide. That image will stick out of the div and be visible by default. Whereas if you set the overflow value to hidden,the image will cut off at 200px.

- - visible: content is not clipped when it proceeds outside its box. This is the default value of theproperty
- - hidden: overflowing content will be hidden.
- - scroll: similar to hidden except users will be able to scroll through the hidden content
- should be visible for users to read the rest of the content.
- - initial: uses the default value which is visible
- - inherit: sets the overflow to the value of its parent element.

### Overflow-x and Overflow-y

It's also possible to manipulate the overflow of content horizontally or vertically with the `overflow-x` and `overflow-y` properties.

# Css Position

Positioning allows us to produce interesting results by overriding normal document flow.

There are five different position values:

- `static`
- `relative`
- `fixed`
- `absolute`
- `sticky`

## Static positioning

Static positioning is the default that every element gets. It just means "put the element into its normal position in the document flow — nothing special to see here."

**Static positioned elements are not affected by the top, bottom, left, and right properties.**

## Relative positioning

This is very similar to static positioning, except that once the positioned element has taken its place in the normal flow, you can then modify its final position, including making it overlap other elements on the page

**Setting the top, right, bottom, and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.**

## Fixed positioning

An element with `position: fixed;` is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled. The top, right, bottom, and left properties are used to position the element.

**A fixed element does not leave a gap in the page where it would normally have been located.**

## Absolute positioning

An element with `position: absolute;` is positioned relative to the nearest positioned ancestor (instead of positioned relative to the viewport, like fixed).

However; if an absolute positioned element has no positioned ancestors, it uses the document **body**, and moves along with page scrolling.

**Absolute positioned elements are removed from the normal flow, and can overlap elements.**

## Sticky positioning

This is basically a hybrid between relative and fixed position. It allows a positioned element to act like it's relatively positioned until it's scrolled to a certain threshold (e.g., 10px from the top of the viewport), after which it becomes fixed.

**Note:** Internet Explorer does not support sticky positioning. Safari requires a -webkit- prefix (see example below). You must also specify at least one of `top`, `right`, `bottom` or `left` for sticky positioning to work.

# z-index

All this absolute positioning is good fun, but there's another feature we haven't considered yet. When elements start to overlap, what determines which elements appear over others and which elements appear under others? In the example we've seen so far, we only have one positioned element in the positioning context, and it appears on the top since positioned elements win over non-positioned elements. What about when we have more than one?

**Note:** If two positioned elements overlap each other without a `z-index` specified, the element defined **last in the HTML code** will be shown on top.

**Note:** `z-index` only works on positioned elements (position: absolute, position: relative, position: fixed, or position: sticky) and flex items

# Opacity

The `opacity` property specifies the opacity/transparency of an element.

the value of opacity property is in the range `0.0` to `1.0`, or a `<percentage>` in the range `0%` to `100%`

The `opacity` property is often used together with the `:hover` selector to change the opacity on mouse-over:

# float

The CSS `float` property specifies how an element should float.

The `float` property can have one of the following values:

- `left` - The element floats to the left of its container
- `right` - The element floats to the right of its container
- `none` - The element does not float (will be displayed just where it occurs in the text). This is default
- `inherit` - The element inherits the float value of its parent

## Clearing floats

The CSS `clear` property specifies what elements can float beside the cleared element and on which side.

The `clear` property accepts the following values:

- `left`: Clear items floated to the left.
- `right`: Clear items floated to the right.
- `both`: Clear any floated items, left or right.