What is CSS selector specificity and how does it work?

The browser determines what styles to show on an element depending on the specificity of CSS rules. We assume that the browser has already determined the rules that match a particular element. Among the matching rules, the specificity, four comma-separate values, `a, b, c, d` are calculated for each rule based on the following:

1. `a` is whether inline styles are being used. If the property declaration is an inline style on the element, `a` is 1, else 0.
2. `b` is the number of ID selectors.
3. `c` is the number of classes, attributes and pseudo-classes selectors.
4. `d` is the number of tags and pseudo-elements selectors.

The resulting specificity is not a score, but a matrix of values that can be compared column by column. When comparing selectors to determine which has the highest specificity, look from left to right, and compare the highest value in each column. So a value in column `b` will override values in columns `c` and `d`, no matter what they might be. As such, specificity of `0,1,0,0` would be greater than one of `0,0,10,10`.

In the cases of equal specificity: the latest rule is the one that counts. If you have written the same rule into your stylesheet (regardless of internal or external) twice, then the lower rule in your style sheet is closer to the element to be styled, it is deemed to be more specific and therefore will be applied.

I would write CSS rules with low specificity so that they can be easily overridden if necessary. When writing CSS UI component library code, it is important that they have low specificities so that users of the library can override them without using too complicated CSS rules just for the sake of increasing specificity or resorting to `!important`.

References

- https://www.smashingmagazine.com/2007/07/css-specificity-things-you-should-know/
- https://www.sitepoint.com/web-foundations/specificity/

[↑] Back to top

What's the difference between "resetting" and "normalizing" CSS? Which would you choose, and why?

- **Resetting** - Resetting is meant to strip all default browser styling on elements. For e.g. `margin`s, `padding`s, `font-size`s of all elements are reset to be the same. You will have to redeclare styling for common typographic elements.
- **Normalizing** - Normalizing preserves useful default styles rather than "unstyling" everything. It also corrects bugs for common browser dependencies.

I would choose resetting when I have a very customized or unconventional site design such that I need to do a lot of my own styling and do not need any default styling to be preserved.

References

- https://stackoverflow.com/questions/6887336/what-is-the-difference-between-normalize-css-and-reset-css

[↑] Back to top

Describe `floats` and how they work.

Float is a CSS positioning property. Floated elements remain a part of the flow of the page, and will affect the positioning of other elements (e.g. text will flow around floated elements), unlike `position: absolute` elements, which are removed from the flow of the page.

The CSS `clear` property can be used to be positioned below `left`/`right`/`both` floated elements.

If a parent element contains nothing but floated elements, its height will be collapsed to nothing. It can be fixed by clearing the float after the floated elements in the container but before the close of the container.

The `.clearfix` hack uses a clever CSS pseudo selector (`:after`) to clear floats. Rather than setting the overflow on the parent, you apply an additional class `clearfix` to it. Then apply this CSS:

```
.clearfix:after {
  content: ' ';
  visibility: hidden;
  display: block;
  height: 0;
  clear: both;
}
```

Alternatively, give `overflow: auto` or `overflow: hidden` property to the parent element which will establish a new block formatting context inside the children and it will expand to contain its children.

References

- https://css-tricks.com/all-about-floats/

[↑] Back to top

Describe `z-index` and how stacking context is formed.

The `z-index` property in CSS controls the vertical stacking order of elements that overlap. `z-index` only affects elements that have a `position` value which is not `static`.

Without any `z-index` value, elements stack in the order that they appear in the DOM (the lowest one down at the same hierarchy level appears on top). Elements with non-static positioning (and their children) will always appear on top of elements with default static positioning, regardless of HTML hierarchy.

A stacking context is an element that contains a set of layers. Within a local stacking context, the `z-index` values of its children are set relative to that element rather than to the document root. Layers outside of that context — i.e. sibling elements of a local stacking context — can't sit between layers within it. If an element B sits on top of element A, a child element of element A, element C, can never be higher than element B even if element C has a higher `z-index` than element B.

Each stacking context is self-contained - after the element's contents are stacked, the whole element is considered in the stacking order of the parent stacking context. A handful of CSS properties trigger a new stacking context, such as `opacity` less than 1, `filter` that is not `none`, and `transform` that is not `none`.

Note: What exactly qualifies an element to create a stacking context is listed in this long set of rules.

References

- https://css-tricks.com/almanac/properties/z/z-index/
- https://philipwalton.com/articles/what-no-one-told-you-about-z-index/
- https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Positioning/Understanding_z_index/The_stacking_context

[↑] Back to top

Describe Block Formatting Context (BFC) and how it works.

A Block Formatting Context (BFC) is part of the visual CSS rendering of a web page in which block boxes are laid out. Floats, absolutely positioned elements, `inline-blocks`, `table-cells`, `table-caption`s, and elements with `overflow` other than `visible` (except when that value has been propagated to the viewport) establish new block formatting contexts.

Knowing how to establish a block formatting context is important, because without doing so, the containing box will not contain floated children. This is similar to collapsing margins, but more insidious as you will find entire boxes collapsing in odd ways.

A BFC is an HTML box that satisfies at least one of the following conditions:

- The value of `float` is not `none`.
- The value of `position` is neither `static` nor `relative`.
- The value of `display` is `table-cell`, `table-caption`, `inline-block`, `flex`, or `inline-flex`.
- The value of `overflow` is not `visible`.

In a BFC, each box's left outer edge touches the left edge of the containing block (for right-to-left formatting, right edges touch).

Vertical margins between adjacent block-level boxes in a BFC collapse. Read more on collapsing margins.

References

- https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Block_formatting_context

- https://www.sitepoint.com/understanding-block-formatting-contexts-in-css/

[↑] Back to top

What are the various clearing techniques and which is appropriate for what context?

- Empty `div` method - `<div style="clear:both;"></div>`.
- Clearfix method - Refer to the `.clearfix` class above.
- `overflow: auto` or `overflow: hidden` method - Parent will establish a new block formatting context and expand to contains its floated children.

In large projects, I would write a utility `.clearfix` class and use them in places where I need it. `overflow: hidden` might clip children if the children is taller than the parent and is not very ideal.

[↑] Back to top

Explain CSS sprites, and how you would implement them on a page or site.

CSS sprites combine multiple images into one single larger image. It is a commonly-used technique for icons (Gmail uses it). How to implement it:

1. Use a sprite generator that packs multiple images into one and generate the appropriate CSS for it.
2. Each image would have a corresponding CSS class with `background-image`, `background-position` and `background-size` properties defined.
3. To use that image, add the corresponding class to your element.

Advantages:

- Reduce the number of HTTP requests for multiple images (only one single request is required per spritesheet). But with HTTP2, loading multiple images is no longer much of an issue.
- Advance downloading of assets that won't be downloaded until needed, such as images that only appear upon `:hover` pseudo-states. Blinking wouldn't be seen.

References

- https://css-tricks.com/css-sprites/

[↑] Back to top

How would you approach fixing browser-specific styling issues?

- After identifying the issue and the offending browser, use a separate style sheet that only loads when that specific browser is being used. This technique requires server-side rendering though.

- Use libraries like Bootstrap that already handles these styling issues for you.
- Use `autoprefixer` to automatically add vendor prefixes to your code.
- Use Reset CSS or Normalize.css.
- If you're using Postcss (or a similar transpiling library), there may be plugins which allow you to opt in for using modern CSS syntax (and even W3C proposals) that will transform those sections of your code into corresponding safe code that will work in the targets you've used.

[↑] Back to top

How do you serve your pages for feature-constrained browsers? What techniques/processes do you use?

- Graceful degradation - The practice of building an application for modern browsers while ensuring it remains functional in older browsers.
- Progressive enhancement - The practice of building an application for a base level of user experience, but adding functional enhancements when a browser supports it.
- Use caniuse.com to check for feature support.
- Autoprefixer for automatic vendor prefix insertion.
- Feature detection using Modernizr.
- Use CSS Feature queries @support

[↑] Back to top

What are the different ways to visually hide content (and make it available only for screen readers)?

These techniques are related to accessibility (a11y).

- `visibility: hidden`. However, the element is still in the flow of the page, and still takes up space.
- `width: 0; height: 0`. Make the element not take up any space on the screen at all, resulting in not showing it.
- `position: absolute; left: -99999px`. Position it outside of the screen.
- `text-indent: -9999px`. This only works on text within the `block` elements.
- Metadata. For example by using Schema.org, RDF, and JSON-LD.
- WAI-ARIA. A W3C technical specification that specifies how to increase the accessibility of web pages.

Even if WAI-ARIA is the ideal solution, I would go with the `absolute` positioning approach, as it has the least caveats, works for most elements and it's an easy technique.

References

- https://www.w3.org/TR/wai-aria-1.1/
- https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA
- http://a11yproject.com/

[↑] Back to top

Have you ever used a grid system, and if so, what do you prefer?

I like the `float`-based grid system because it still has the most browser support among the alternative existing systems (flex, grid). It has been used in Bootstrap for years and has been proven to work.

[↑] Back to top

Have you used or implemented media queries or mobile-specific layouts/CSS?

Yes. An example would be transforming a stacked pill navigation into a fixed-bottom tab navigation beyond a certain breakpoint.

[↑] Back to top

Are you familiar with styling SVG?

Yes, there are several ways to color shapes (including specifying attributes on the object) using inline CSS, an embedded CSS section, or an external CSS file. Most SVG you'll find around the web use inline CSS, but there are advantages and disadvantages associated with each type.

Basic coloring can be done by setting two attributes on the node: `fill` and `stroke`. `fill` sets the color inside the object and `stroke` sets the color of the line drawn around the object. You can use the same CSS color naming schemes that you use in HTML, whether that's color names (that is `red`), RGB values (that is `rgb(255,0,0)`), Hex values, RGBA values, etc.

```
<rect x="10" y="10" width="100" height="100" stroke="blue"
  fill="purple" fill-opacity="0.5" stroke-opacity="0.8"/>
```

The above `fill="purple"` is an example of a *presentational attribute*. Interestingly, and unlike inline styles like `style="fill: purple"` which also happens to be an attribute, presentational attributes can be overriden by CSS styles defined in a stylesheet. So, if you did something like `svg { fill: blue; }` it would override the purple fill we've defined.

References

- https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/Fills_and_Strokes

[↑] Back to top

Can you give an example of an @media property other than screen?

Yes, there are four types of @media properties (including *screen*):

- `all` - for all media type devices

- `print` - for printers
- `speech` - for screenreaders that "reads" the page out loud
- `screen` - for computer screens, tablets, smart-phones etc.

Here is an example of `print` media type's usage:

```
@media print {
  body {
    color: black;
  }
}
```

References

- https://developer.mozilla.org/en-US/docs/Web/CSS/@media#Syntax

[↑] Back to top

What are some of the "gotchas" for writing efficient CSS?

Firstly, understand that browsers match selectors from rightmost (key selector) to left. Browsers filter out elements in the DOM according to the key selector and traverse up its parent elements to determine matches. The shorter the length of the selector chain, the faster the browser can determine if that element matches the selector. Hence avoid key selectors that are tag and universal selectors. They match a large number of elements and browsers will have to do more work in determining if the parents do match.

BEM (Block Element Modifier) methodology recommends that everything has a single class, and, where you need hierarchy, that gets baked into the name of the class as well, this naturally makes the selector efficient and easy to override.

Be aware of which CSS properties trigger reflow, repaint, and compositing. Avoid writing styles that change the layout (trigger reflow) where possible.

References

- https://developers.google.com/web/fundamentals/performance/rendering/
- https://csstriggers.com/

[↑] Back to top

What are the advantages/disadvantages of using CSS preprocessors?

Advantages:

- CSS is made more maintainable.
- Easy to write nested selectors.
- Variables for consistent theming. Can share theme files across different projects.
- Mixins to generate repeated CSS.
- Sass features like loops, lists, and maps can make configuration easier and less verbose.

- Splitting your code into multiple files. CSS files can be split up too but doing so will require an HTTP request to download each CSS file.

Disadvantages:

- Requires tools for preprocessing. Re-compilation time can be slow.
- Not writing currently and potentially usable CSS. For example, by using something like postcss-loader with webpack, you can write potentially future-compatible CSS, allowing you to use things like CSS variables instead of Sass variables. Thus, you're learning new skills that could pay off if/when they become standardized.

[↑] Back to top

Describe what you like and dislike about the CSS preprocessors you have used.

Likes:

- Mostly the advantages mentioned above.
- Less is written in JavaScript, which plays well with Node.

Dislikes:

- I use Sass via `node-sass`, which is a binding for LibSass written in C++. I have to frequently recompile it when switching between node versions.
- In Less, variable names are prefixed with `@`, which can be confused with native CSS keywords like `@media`, `@import` and `@font-face` rule.

[↑] Back to top

How would you implement a web design comp that uses non-standard fonts?

Use `@font-face` and define `font-family` for different `font-weight`s.

[↑] Back to top

Explain how a browser determines what elements match a CSS selector.

This part is related to the above about writing efficient CSS. Browsers match selectors from rightmost (key selector) to left. Browsers filter out elements in the DOM according to the key selector and traverse up its parent elements to determine matches. The shorter the length of the selector chain, the faster the browser can determine if that element matches the selector.

For example with this selector `p span`, browsers firstly find all the `<span>` elements and traverse up its parent all the way up to the root to find the `<p>` element. For a particular `<span>`, as soon as it finds a `<p>`, it knows that the `<span>`matches and can stop its matching.

References

- https://stackoverflow.com/questions/5797014/why-do-browsers-match-css-selectors-from-right-to-left

[↑] Back to top

Describe pseudo-elements and discuss what they are used for.

A CSS pseudo-element is a keyword added to a selector that lets you style a specific part of the selected element(s). They can be used for decoration (`:first-line`, `:first-letter`) or adding elements to the markup (combined with `content: ...`) without having to modify the markup (`:before`, `:after`).

- `:first-line` and `:first-letter` can be used to decorate text.
- Used in the `.clearfix` hack as shown above to add a zero-space element with `clear: both`.
- Triangular arrows in tooltips use `:before` and `:after`. Encourages separation of concerns because the triangle is considered part of styling and not really the DOM. It's not really possible to draw a triangle with just CSS styles without using an additional HTML element.

References

- https://css-tricks.com/almanac/selectors/a/after-and-before/

[↑] Back to top

Explain your understanding of the box model and how you would tell the browser in CSS to render your layout in different box models.

The CSS box model describes the rectangular boxes that are generated for elements in the document tree and laid out according to the visual formatting model. Each box has a content area (e.g. text, an image, etc.) and optional surrounding `padding`, `border`, and `margin` areas.

The CSS box model is responsible for calculating:

- How much space a block element takes up.
- Whether or not borders and/or margins overlap, or collapse.
- A box's dimensions.

The box model has the following rules:

- The dimensions of a block element are calculated by `width`, `height`, `padding`, `border`s, and `margin`s.
- If no `height` is specified, a block element will be as high as the content it contains, plus `padding` (unless there are floats, for which see below).
- If no `width` is specified, a non-floated block element will expand to fit the width of its parent minus `padding`.
- The `height` of an element is calculated by the content's `height`.

- The `width` of an element is calculated by the content's `width`.
- By default, `padding`s and `border`s are not part of the `width` and `height` of an element.

References

- https://www.smashingmagazine.com/2010/06/the-principles-of-cross-browser-css-coding/#understand-the-css-box-model

[↑] Back to top

What does `* { box-sizing: border-box; }` do? What are its advantages?

- By default, elements have `box-sizing: content-box` applied, and only the content size is being accounted for.
- `box-sizing: border-box` changes how the `width` and `height` of elements are being calculated, `border` and `padding` are also being included in the calculation.
- The `height` of an element is now calculated by the content's `height` + vertical `padding` + vertical `border` width.
- The `width` of an element is now calculated by the content's `width` + horizontal `padding` + horizontal `border` width.
- Taking into account `padding`s and `border`s as part of our box model resonates better with how designers actually imagine content in grids.

References

- https://www.paulirish.com/2012/box-sizing-border-box-ftw/

[↑] Back to top

What is the CSS `display` property and can you give a few examples of its use?

- `none`, `block`, `inline`, `inline-block`, `table`, `table-row`, `table-cell`, `list-item`.

TODO

[↑] Back to top

What's the difference between `inline` and `inline-block`?

I shall throw in a comparison with `block` for good measure.

	`block`	`inline-block`	`inline`
Size	Fills up the width of its parent container.	Depends on content.	Depends on content.
Positioning	Start on a new line and tolerates no HTML elements next to it (except when you add `float`)	Flows along with other content and allows other elements beside it.	Flows along with other content and allows other elements beside it.
Can specify `width` and `height`	Yes	Yes	No. Will ignore if being set.
Can be aligned with `vertical-align`	No	Yes	Yes
Margins and paddings	All sides respected.	All sides respected.	Only horizontal sides respected. Vertical sides, if specified, do not affect layout. Vertical space it takes up depends on `line-height`, even though the `border` and `padding` appear visually around the content.
Float	-	-	Becomes like a `block` element where you can set vertical margins and paddings.

[↑] Back to top

What's the difference between a `relative`, `fixed`, `absolute` and `static`ally positioned element?

A positioned element is an element whose computed `position` property is either `relative`, `absolute`, `fixed` or `sticky`.

- `static` - The default position; the element will flow into the page as it normally would. The `top`, `right`, `bottom`, `left` and `z-index` properties do not apply.
- `relative` - The element's position is adjusted relative to itself, without changing layout (and thus leaving a gap for the element where it would have been had it not been positioned).
- `absolute` - The element is removed from the flow of the page and positioned at a specified position relative to its closest positioned ancestor if any, or otherwise relative to the initial containing block. Absolutely positioned boxes can have margins, and they do not collapse with any other margins. These elements do not affect the position of other elements.
- `fixed` - The element is removed from the flow of the page and positioned at a specified position relative to the viewport and doesn't move when scrolled.
- `sticky` - Sticky positioning is a hybrid of relative and fixed positioning. The element is treated as `relative` positioned until it crosses a specified threshold, at which point it is treated as `fixed` positioned.

References

- https://developer.mozilla.org/en/docs/Web/CSS/position

[↑] Back to top

What existing CSS frameworks have you used locally, or in production? How would you change/improve them?

- **Bootstrap** - Slow release cycle. Bootstrap 4 has been in alpha for almost 2 years. Add a spinner button component, as it is widely used.
- **Semantic UI** - Source code structure makes theme customization extremely hard to understand. Its unconventional theming system is a pain to customize. Hardcoded config path within the vendor library. Not well-designed for overriding variables unlike in Bootstrap.
- **Bulma** - A lot of non-semantic and superfluous classes and markup required. Not backward compatible. Upgrading versions breaks the app in subtle manners.

[↑] Back to top

Have you played around with the new CSS Flexbox or Grid specs?

Yes. Flexbox is mainly meant for 1-dimensional layouts while Grid is meant for 2-dimensional layouts.

Flexbox solves many common problems in CSS, such as vertical centering of elements within a container, sticky footer, etc. Bootstrap and Bulma are based on Flexbox, and it is probably the recommended way to create layouts these days. Have tried Flexbox before but ran into some browser incompatibility issues (Safari) in using `flex-grow`, and I had to rewrite my code using `inline-blocks` and math to calculate the widths in percentages, it wasn't a nice experience.

Grid is by far the most intuitive approach for creating grid-based layouts (it better be!) but browser support is not wide at the moment.

References

- https://philipwalton.github.io/solved-by-flexbox/

[↑] Back to top

Can you explain the difference between coding a website to be responsive versus using a mobile-first strategy?

Note that these two 2 approaches are not exclusive.

Making a website responsive means the some elements will respond by adapting its size or other functionality according to the device's screen size, typically the viewport width, through CSS media queries, for example, making the font size smaller on smaller devices.

```
@media (min-width: 601px) {
  .my-class {
    font-size: 24px;
  }
}
@media (max-width: 600px) {
  .my-class {
    font-size: 12px;
  }
}
```

A mobile-first strategy is also responsive, however it agrees we should default and define all the styles for mobile devices, and only add specific responsive rules to other devices later. Following the previous example:

```
.my-class {
  font-size: 12px;
}

@media (min-width: 600px) {
  .my-class {
    font-size: 24px;
  }
}
```

A mobile-first strategy has 2 main advantages:

- It's more performant on mobile devices, since all the rules applied for them don't have to be validated against any media queries.
- It forces to write cleaner code in respect to responsive CSS rules.

[↑] Back to top

How is responsive design different from adaptive design?

Both responsive and adaptive design attempt to optimize the user experience across different devices, adjusting for different viewport sizes, resolutions, usage contexts, control mechanisms, and so on.

Responsive design works on the principle of flexibility - a single fluid website that can look good on any device. Responsive websites use media queries, flexible grids, and responsive images to create a user experience that flexes and changes based on a multitude of factors. Like a single ball growing or shrinking to fit through several different hoops.

Adaptive design is more like the modern definition of progressive enhancement. Instead of one flexible design, adaptive design detects the device and other features and then provides the appropriate feature and layout based on a predefined set of viewport sizes and other characteristics. The site detects the type of device used and delivers the pre-set layout for that device. Instead of a single ball going through several different-sized hoops, you'd have several different balls to use depending on the hoop size.

Both have these methods have some issues that need to be weighed:

- Responsive design can be quite challenging, as you're essentially using a single albeit responsive layout to fit all situations. How to set the media query breakpoints is one such challenge. Do you use standardized breakpoint values? Or, do you use breakpoints that make sense to your particular layout? What if that layout changes?
- Adaptive design generally requires user agent sniffing, or DPI detection, etc., all of which can prove unreliable.

References

- https://developer.mozilla.org/en-US/docs/Archive/Apps/Design/UI_layout_basics/Responsive_design_versus_adaptive_design
- http://mediumwell.com/responsive-adaptive-mobile/
- https://css-tricks.com/the-difference-between-responsive-and-adaptive-design/

[↑] Back to top

Have you ever worked with retina graphics? If so, when and what techniques did you use?

Retina is just a marketing term to refer to high resolution screens with a pixel ratio bigger than 1. The key thing to know is that using a pixel ratio means these displays are emulating a lower resolution screen in order to show elements with the same size. Nowadays we consider all mobile devices *retina* defacto displays.

Browsers by default render DOM elements according to the device resolution, except for images.

In order to have crisp, good-looking graphics that make the best of retina displays we need to use high resolution images whenever possible. However using always the highest resolution images will have an impact on performance as more bytes will need to be sent over the wire.

To overcome this problem, we can use responsive images, as specified in HTML5. It requires making available different resolution files of the same image to the browser and let it decide which image is best, using the html attribute `srcset` and optionally `sizes`, for instance:

```
<div responsive-background-image>
  <img src="/images/test-1600.jpg"
    sizes="
      (min-width: 768px) 50vw,
      (min-width: 1024px) 66vw,
      100vw"
    srcset="
      /images/test-400.jpg 400w,
      /images/test-800.jpg 800w,
      /images/test-1200.jpg 1200w">
</div>
```

It is important to note that browsers which don't support HTML5's `srcset` (i.e. IE11) will ignore it and use `src` instead. If we really need to support IE11 and we want to provide this feature for performance reasons, we can use a JavaScript polyfill, e.g. Picturefill (link in the references).

For icons, I would also opt to use SVGs and icon fonts where possible, as they render very crisply regardless of resolution.

References

- https://css-tricks.com/responsive-images-youre-just-changing-resolutions-use-srcset/
- http://scottjehl.github.io/picturefill/
- https://aclaes.com/responsive-background-images-with-srcset-and-sizes/

[↑] Back to top

Is there any reason you'd want to use `translate()` instead of `absolute` positioning, or vice-versa? And why?

`translate()` is a value of CSS `transform`. Changing `transform` or `opacity` does not trigger browser reflow or repaint but does trigger compositions; whereas changing the absolute positioning triggers `reflow`. `transform` causes the browser to create a GPU layer for the element but changing absolute positioning properties uses the CPU. Hence `translate()` is more efficient and will result in shorter paint times for smoother animations.

When using `translate()`, the element still occupies its original space (sort of like `position: relative`), unlike in changing the absolute positioning.

References

- https://www.paulirish.com/2012/why-moving-elements-with-translate-is-better-than-posabs-topleft/