17-06-25 Tuesday

Top 10 Questions on CSS (expert level)

Q1: What is the specificity hierarchy in CSS?
Inline styles > IDs > Classes/Attributes/Pseudo-classes >Elements/Pseudo-elements>Universal.
Calculated using a tuple (a,b,c)**.**

```
pracrice.html > html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>

<style>
    #demo {       /* ID selector */
      color: blue;
    }
    .test {      /* Class selector */
      color: green;
    }
    p {        /* Element selector */
      color: red;
    }
  </style>
</head>
<body>
      <p id="demo" class="test" style="color: red;">Hello World!</p>
</body>
</html>
```

Result : Hello world !

Q2: Difference between `:nth-child()` vs. `:nth-of-type()`?
`:nth-child()` Selects all **the nth child** of a **parent**, regardless of its type.
`:nth-of-type()` filters by Specific type (e.g., only `<p>` elements) , select only elements of the same tag.

pracrice.html > html > body > div > p

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Important questions</title>

<style>
    p:nth-child(2n) {
     color: red;
    }

  </style>
</head>
<body>
    <div>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
    <p>Paragraph 3</p>
    <span>Span 1</span>
    <p>Paragraph 4</p>
    <p>Paragraph 6</p>
    </div>

</body>
</html>
```

Result: For nth -child(2n)

Paragraph 1

Paragraph 2

Paragraph 3

Span 1

Paragraph 4

Paragraph 6

pracrice.html > html > head > title

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Important</title>
<style>
    p:nth-of-type(2n) {
    color: blue;
    }
  </style>
</head>
<body>
    <div>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
    <p>Paragraph 3</p>
    <span>Span 1</span>
    <p>Paragraph 4</p>
    <p>Paragraph 6</p>
    </div>
</body>
</html>
```

Result: For nth -of-type(2n)

Paragraph 1

Paragraph 2

Paragraph 3

Span 1

Paragraph 4

Paragraph 6

Q3: What does the `:not()` pseudo-class do?

The :not() CSS pseudo-class represents elements that do not match a list of selectors. Since it prevents specific items from being selected, it is known as the negation pseudo-class or select every product except the elements that match the selector passed to `:not()`.

pracrice.html > html > head > style > p:not(.special)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Important</title>
<style>
    /* Apply blue color to all <p> elements except the one with class 'special' */
    p:not(.special) {
      color: blue;
      font-weight: bold;
    }
    /* Style the excluded one differently */
    .special {
      color: red;
      font-style: italic;
    }
  </style>
</head>
<body>
  <h2>Example of :not() in CSS</h2>
  <p>This paragraph is blue and bold.</p>
  <p class="special">This paragraph is red and italic (excluded from :not).</p>
  <p>This paragraph is also blue and bold.</p>
</body>
</html>
```

Result:

Example of :not() in CSS

This paragraph is blue and bold.

This paragraph is red and italic (excluded from :not).

This paragraph is also blue and bold.

Q4: How do `:is()` and `:where()` differ?

`:is()` inherits the highest specificity of its arguments of the selectors in its list.

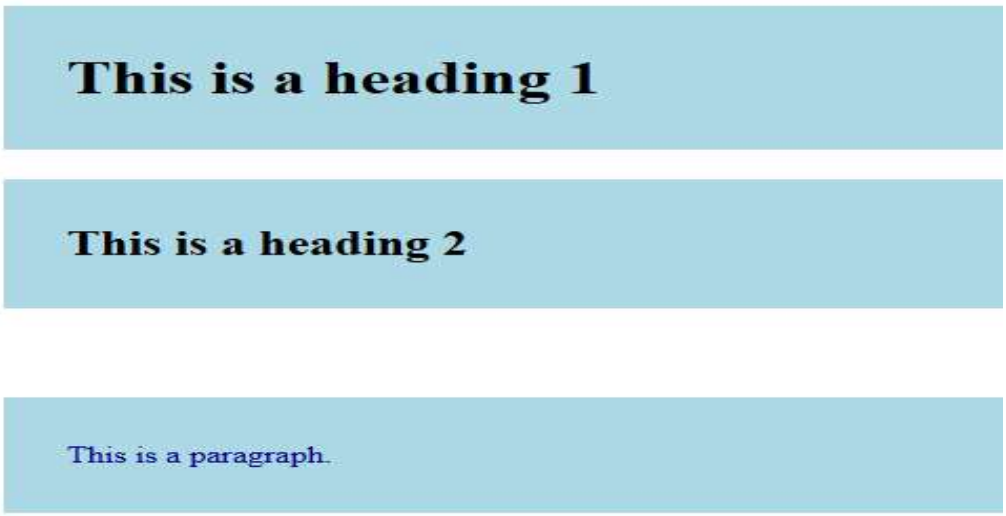
`:where()` contributes zero specificity, making it easier to override.

- Use :is() for concise selectors where specificity matters.
- Use :where() when you want to group selectors but keep specificity low for flexibility.

pracrice.html > html > body > p

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Important</title>
<style>
    /* This will style ALL h1, h2, and p with a blue background */
    :is(h1, h2, p) {
      background-color: lightblue;
      padding: 30px;
    }
    /* This gives all h1, h2, p 10px margin at the bottom */
    /* But this is easy to override later because specificity = 0 */
    :where(h1, h2, p) {
      margin-bottom: 10px;
    }
    /* Overriding only the paragraph margin using regular CSS */
    /* Since :where() has 0 specificity, this wins easily */
    p {
      margin-top: 60px;
      color: darkblue;
    }
  </style>
</head>
<body>
  <h1>This is a heading 1</h1>
  <h2>This is a heading 2</h2>
  <p>This is a paragraph.</p>
</body>
</html>
```

Result:



Q5: How does `!important` affect specificity?

The !important rule in CSS is used to add more importance to a property/value than normal.In fact, if you use the !important rule, it will override ALL previous styling rules for that specific property on that element or `!important` is used to **force a style to apply**, no matter what the normal rules of **specificity** or **order** say.

pracrice.html > html > body > p

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Important</title>
 <style>
    /* Lower specificity: element selector */
    p {
      color: green;
    }
    /* Higher specificity: class selector */
    .highlight {
      color: blue;
    }
    /* Forceful override using !important */
    #force {
      color: red !important;
    }
  </style>
</head>
<body>
  <p>This paragraph is green (from normal tag selector).</p>
  <p class="highlight">This paragraph is blue (from class selector).</p>
  <p class="highlight" id="force">This paragraph is red (because of !important).</p>
</body>
</html>
```

Result:

This paragraph is green (from normal tag selector).

This paragraph is blue (from class selector).

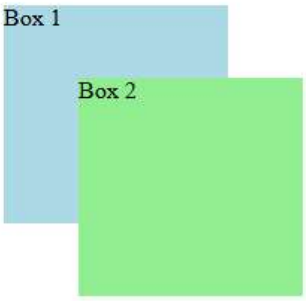
This paragraph is red (because of !important).

Q6: What is `z-index` in CSS and how does it affect overlapping elements on a webpage?

`z-index` in CSS defines the **stacking order** (or *stack level*) of elements along the z-axis (depth). It determines which elements appear **in front of** or **behind** others when they overlap. By default, element boxes are rendered on Layer 0. The z-index property allows you to position elements on different layers along the z-axis, in addition to the default rendering layer. Each element's position along the imaginary z-axis (z-index value) is expressed as an integer (positive, negative, or zero) and controls the stacking order during rendering. Greater numbers mean elements are closer to the observer. It only works on elements with a **position** set to `relative`, `absolute`, `fixed`, or `sticky` (not `static`). Default value is `auto` (treated as `0`).

```
pracrice.html > html > body
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Important</title>
 <style>
    .box1 {
      width: 150px;
      height: 150px;
      background-color: lightblue;
      position: absolute;
      top: 50px;
      left: 50px;
      z-index: 1; /* Lower value, appears behind */
    }
    .box2 {
      width: 150px;
      height: 150px;
      background-color: lightgreen;
      position: absolute;
      top: 100px;
      left: 100px;
      z-index: 2; /* Higher value, appears on top */
    }
  </style>
</head>
<body>
  <div class="box1">Box 1</div>
  <div class="box2">Box 2</div>
</body>
</html>
```

Result:



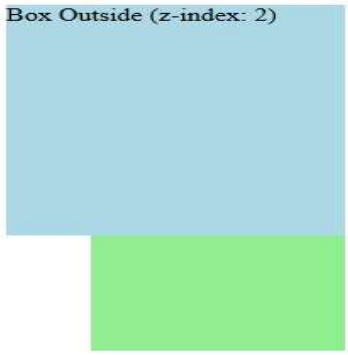
Q7: What is the stacking context in CSS?

A stacking context is a group of elements that have a common parent and move up and down the z axis together. The z-index of elements inside of a stacking context are always relative to the parent's current order in its own stacking context.

Within a stacking context, child elements are stacked according to the z-index values of all the siblings.

```
pracrice.html > html > head > style > .box-inside
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Important</title>
<style>
    .box-outside {   /* Box outside stacking context */
      width: 200px;
      height: 200px;
      background-color: lightblue;
      position: relative;
      z-index: 2;
    }
    .container {  /* This container creates a new stacking context */
      position: relative;
      z-index: 1;
    }
    .box-inside { /* Even with high z-index, it's inside the container context */
      width: 150px;
      height: 150px;
      background-color: lightgreen;
      position: absolute;
      top: -50px;
      left: 50px;
      z-index: 999;
    }
  </style>
</head>
<body>
  <div class="box-outside">Box Outside (z-index: 2)</div>
  <div class="container">
    <div class="box-inside">Box Inside (z-index: 999)</div>
  </div>
</body>
```

Result:



Q8 : How does flex-grow, flex-shrink, and flex-basis work?

Flex grow - Controls how much an item **can grow** relative to other items.

Value: `0` (default) = don't grow, `1` = grow if space is available.

`Flex-shrink-` Controls how much an item **can shrink** when there's not enough space.

Value: `1` (default) = can shrink, `0` = won't shrink.

Flex-basis- Sets the **initial size** of the item **before** `grow` or `shrink` is applied. Can be in `px`, `%`, or `auto`.

pracrice.html > html > head > style > .flex-container

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Position Example</title>
<style>
    .flex-container {
      display: flex;
      border: 2px solid #333;
      padding: 10px;
    }
    .box {
      color: white;
      text-align: center;
      padding: 10px;
      font-weight: bold;
    }
    .box1 {
      background-color: steelblue;
      flex-grow: 1;
      flex-shrink: 1;
      flex-basis: 100px;
    }
    .box2 {
      background-color: darkorange;
      flex-grow: 2;
      flex-shrink: 1;
      flex-basis: 100px;
    }
    .box3 {
      background-color: forestgreen;
      flex-grow: 1;
      flex-shrink: 1;
      flex-basis: 100px;
    }
  </style>
</head>
<body>
    <div class="flex-container">
    <div class="box box1">Box 1</div>
    <div class="box box2">Box 2</div>
    <div class="box box3">Box 3</div>
  </div>
</body>
</html>
```

Result:

Q9: How is CSS Grid different from Flexbox?

Flexbox- One dimensional layout system , Can layout items **horizontally (row)** or **vertically (column)** but **not both at the same time.** Controls layout along **main axis** and **cross axis. Content-first** layout. we place content and Flexbox arranges it. Use Flexbox for a navigation bar

CSS Grid - Two-dimensional layout system. Can layout items in **both rows and columns.** Controls **row and column tracks** separately. **Layout-first.** we define the structure (rows/columns) and place content into it. Use Grid for a full web page layout with headers,sidebars, and content areas.

pracrice.html > html > body > h2

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Position Example</title>
<style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
    }
    .flex-container {
      display: flex;
      background-color: #e0f7fa;
      padding: 10px;
      margin-bottom: 20px;
    }
    .grid-container {
      display: grid;
      grid-template-columns: repeat(3, 1fr);
      background-color: #fff3e0;
      gap: 10px;
      padding: 10px;
    }
    .item {
      background-color: #4db6ac;
      color: white;
      padding: 20px;
      text-align: center;
      font-weight: bold;
    }
</style>
</head>
<body>
  <h2>Flexbox Layout (One-dimensional)</h2>
  <div class="flex-container">
    <div class="item ">Item 1</div>
    <div class="item ">Item 2</div>
    <div class="item ">Item 3</div>
  </div>
  <h2>Grid Layout (Two-dimensional)</h2>
  <div class="grid-container">
    <div class="item ">Item A</div>
    <div class="item ">Item B</div>
    <div class="item ">Item C</div>
    <div class="item ">Item D</div>
    <div class="item ">Item E</div>
    <div class="item ">Item F</div>
  </div>
</body>
```

Result :

Flexbox Layout (One-dimensional)

Item 1 Item 2 Item 3

Grid Layout (Two-dimensional)

Item A	Item B	Item C
Item D	Item E	Item F

Q10: What's the difference between `auto-fill` vs `auto-fit` in grid?

`auto-fill` = "Reserve space for possible items", **Keeps empty tracks** if there's space, even without content.

`auto-fit` = "Stretch existing items to fill space", **Collapses** empty tracks — makes other items expand to fill space.

```
pracrice.html > html > head > style > .container
<head>
  <meta charset="UTF-8">
  <title>Position Example</title>
  <style>
    body {
      font-family: sans-serif;
      padding: 20px;
    }
    .container {
      display: grid;
      gap: 10px;
      margin-bottom: 40px;
    }
    .box {
      background: steelblue;
      color: white;
      padding: 20px;
      text-align: center;
      font-weight: bold;
    }
    .fill-grid {  /* auto-fill grid */
      grid-template-columns: repeat(auto-fill, minmax(150px, 1fr));
    }
    .fit-grid {  /* auto-fit grid */
      grid-template-columns: repeat(auto-fit, minmax(150px, 1fr));
    }
  </style>
</head>
<body>
  <h2>Grid with auto-fill</h2>
  <div class="container fill-grid">
    <div class="box">1</div>
    <div class="box">2</div>
    <div class="box">3</div>
  </div>
  <h2>Grid with auto-fit</h2>
  <div class="container fit-grid">
    <div class="box">1</div>
    <div class="box">2</div>
    <div class="box">3</div>
  </div>
</body>
</html>
```

Result:

Grid with auto-fill

Grid with auto-fit