**THEORY EXERCISE:**

**Module 3 – Frontend-CSS & CSS3**

**CSS Selectors and Styling:**

**Question 1:** What is a CSS selector? Provide examples of element, class, and ID selectors.

* A CSS selector is a pattern used to select and style specific HTML elements.
* It tells the browser which element in the HTML should receive the CSS rules.

1. **Element Selector**

* Selects HTML elements directly by their tag name.
* Used for select all matching tags.
* Ex.CSS

P {

Color: blue;

}

1. **Class Selector**

* Selects elements with a specific class attribute.
* Used for can be reused on multiple elements.
* Syntax: .classname
* Ex.CSS

.highlight {

background-color: yellow;

}

1. **ID Selector**

* Selects an element with a specific id attribute.
* That’s unique, used for one element only.
* Syntax: #idname
* Ex.CSS

#header {

font-size: 24px;

}

**Question 2:** Explain the concept of CSS specificity. How do conflicts between multiple styles get resolved?

* CSS specificity is a set of rules that determines which CSS styles are applied to an HTML element when multiple, conflicting styles target the same element. It acts as a ranking system for selectors, assigning a "weight" to each based on its type. The selector with the highest specificity "wins" and its styles are applied.
* **Conflict Resolution:**
* When multiple styles target the same element and conflict, CSS resolves these conflicts using the following cascade order:
* **Importance (!important):**

Declarations marked with !important override all other styles, regardless of specificity. However, overuse of !important is generally discouraged as it can lead to difficult-to-manage CSS.

* **Cascade Layers:**

With CSS Cascade Layers, styles within a higher-priority layer will override styles in a lower-priority layer, even if the lower-priority layer's selectors have higher specificity.

* **Specificity:**

If styles are from the same cascade layer (or no layers are used), the declaration with the highest specificity is applied.

* **Order of Appearance:**

If multiple selectors have the exact same specificity and are within the same cascade layer, the last declaration encountered in the stylesheet (or the one appearing later in the source order) will take precedence.

**Question 3:** What is the difference between internal, external, and inline CSS? Discuss the advantages and disadvantages of each approach.

**Types of CSS**

**1. Inline CSS**

* CSS written directly inside the **HTML element** using the style attribute.

<p style="color: red; font-size: 18px;">This is inline CSS</p>

**Advantages:**

* Quick and easy to apply for a single element.
* No need for a separate CSS file.
* Useful for testing or overriding other styles.

**Disadvantages:**

* Not reusable (applies only to one element).
* Hard to maintain for large projects.
* Mixes content with style (bad practice).

**2. Internal CSS**

* CSS written inside a <style> tag in the **HTML document’s <head> section**.

<head>

<style>

p { color: blue; font-size: 18px; }

</style>

</head>

**Advantages:**

* Styles are centralized for one page.
* Better than inline (can style multiple elements).
* Easy to test and debug for single-page projects.

**Disadvantages:**

* CSS applies only to that **specific page**, not reusable across multiple pages.
* Increases page size if repeated in many HTML files.

**3. External CSS**

* CSS written in a **separate .css file**, linked with <link> tag in <head>.

<head>

<link rel="stylesheet" href="style.css">

</head>

**Advantages:**

* Best for **reusability** (same CSS file can style multiple pages).
* Cleaner HTML (separates content and design).
* Reduces code duplication.
* Faster loading once the CSS file is cached.

**Disadvantages:**

* Requires an extra HTTP request (slightly slower on first load).
* If the CSS file is missing or link is broken → page appears unstyled.

**CSS Box Model**

**Question 1:** Explain the CSS box model and its components (content, padding, border, margin). How does each affect the size of an element?

* Every HTML element in CSS is treated as a **rectangular box**.  
  The **box model** describes how the size of this box is calculated and how spacing works around it.

It consists of **4 main components (from inside → outside):**

**1. Content**

* The actual text, image, or other content inside the element.
* Controlled by properties like width, height, line-height.
* Changing content size directly changes the element’s total size.

**2. Padding**

* Space **between the content and the border**.
* Transparent (doesn’t have color unless background is applied).
* Increases the clickable/visible area of an element.
* Increases element size.

**3. Border**

* The edge around the padding and content.
* Controlled by border-width, border-style, border-color.
* Increases element size.

**4. Margin**

* Space **outside the border**, separating the element from other elements.
* Transparent (doesn’t take background color).
* Creates distance between elements but does not affect the content size.

**Effect on Total Size**

By default (using box-sizing: content-box):

**Total Element Width** = content width + padding-left + padding-right + border-left + border-right + margin-left + margin-right

**Total Element Height** = content height + padding-top + padding-bottom + border-top + border-bottom + margin-top + margin-bottom

* If box-sizing: border-box; is used → **padding and border are included inside the declared width/height**, making layout easier.

**Question 2:** What is the difference between border-box and content-box box-sizing in CSS? Which is the default?

* **box-sizing in CSS**

The box-sizing property defines **how the browser calculates the width and height of an element**.

There are two common values:

**1. content-box (Default)**

* **Width/height includes only the content.**
* Padding and border are added **outside** the declared width/height.

**Example: CSS**

.box1 {

box-sizing: content-box; /\* default \*/

width: 200px;

padding: 20px;

border: 10px solid black;

}

* Declared width = 200px
* **Actual width =** 200 (content) + 40 (padding left+right) + 20 (border left+right) = 260px

**2. border-box**

* **Width/height includes content + padding + border.**
* Declared width is the **final size of the element**.

**Example: CSS**

.box2 {

box-sizing: border-box;

width: 200px;

padding: 20px;

border: 10px solid black;

}

* Declared width = 200px
* **Actual width = 200px total** (browser automatically shrinks content box to fit padding + border).

**CSS Flexbox**

**Question 1:** What is CSS Flexbox, and how is it useful for layout design? Explain the terms flex-container and flex-item.

* **Flexbox (Flexible Box Layout Module)** is a CSS layout model that allows you to design **one-dimensional layouts** (row or column) easily.
* It provides better alignment, spacing, and distribution of items inside a container compared to older methods like floats and tables.
* Makes layouts **responsive** and adjusts automatically to screen size.
* Easily aligns elements **horizontally and vertically**.
* Provides control over **spacing, ordering, and alignment** of items.
* Eliminates the need for complex float/position hacks.

**Key Terms**

**1. Flex Container**

* The parent element where **Flexbox is applied**.
* Defined by setting: **CSS**

.container {

display: flex; /\* or display: inline-flex \*/

}

* The container controls how its children (flex items) are arranged.
* Properties like flex-direction, justify-content, align-items are applied **to the container**.

**2. Flex Items**

* The **child elements** inside a flex container.
* Automatically become flexible and adjust based on container rules.
* Each flex item can be controlled individually using properties like flex-grow, flex-shrink, and flex-basis.

**Example. HTML**

<div class="container">

<div class="item">One</div>

<div class="item">Two</div>

<div class="item">Three</div>

</div>

**CSS**

.container {

display: flex; /\* Flex container \*/

justify-content: space-around; /\* Align items horizontally \*/

align-items: center; /\* Align items vertically \*/

}

.item {

background: lightblue;

padding: 20px;

}

**Question 2:** Describe the properties justify-content, align-items, and flex-direction used inFlexbox.

**1. flex-direction**

* Defines the **main axis** along which flex items are placed inside the flex container.
* Determines whether items are arranged **horizontally (row)** or **vertically (column)**.

**Values:**

* row → Default. Items placed left → right.
* row-reverse → Items placed right → left.
* column → Items placed top → bottom.
* column-reverse → Items placed bottom → top.

**CSS.**

.container {

display: flex;

flex-direction: row; /\* or column, row-reverse, column-reverse \*/

}

**2. justify-content**

* Aligns flex items **along the main axis** (horizontal if row, vertical if column).
* Controls spacing between items.

**Values:**

* flex-start → Items packed at start of main axis (default).
* flex-end → Items packed at end.
* center → Items centered.
* space-between → Equal space **between** items, edges flush with container.
* space-around → Equal space **around** each item.
* space-evenly → Equal space between items **and** at edges.

**CSS.**

.container {

display: flex;

justify-content: space-between;

}

**3. align-items**

* Aligns flex items **along the cross axis** (perpendicular to main axis).
* Controls vertical alignment if flex-direction: row.
* Controls horizontal alignment if flex-direction: column.

**Values:**

* flex-start → Items aligned to start of cross axis.
* flex-end → Items aligned to end.
* center → Items centered along cross axis.
* stretch → Items stretch to fill container (default).
* baseline → Items aligned according to their text baseline.

**CSS**

.container {

display: flex;

align-items: center;

}

**CSS Grid**

**Question 1:** Explain CSS Grid and how it differs from Flexbox. When would you use Grid over Flexbox?

* **CSS Grid Layout** is a **two-dimensional layout system** in CSS.
* It allows you to design layouts in **rows and columns** at the same time.
* You work with a **grid container** and its **grid items**.

**CSS.**

.container {

display: grid;

grid-template-columns: 200px 200px 200px;

grid-template-rows: auto auto;

gap: 20px;

}

**HTML.**

<div class="container">

<div>Item 1</div>

<div>Item 2</div>

<div>Item 3</div>

<div>Item 4</div>

</div>

**How Grid Differs from Flexbox**

| **Feature** | **Flexbox (1D)** | **Grid (2D)** |
| --- | --- | --- |
| **Layout type** | One-dimensional → works in **row OR column** | Two-dimensional → works in **rows AND columns** |
| **Main use** | Aligning items in a line (navbars, buttons, small layouts) | Creating full page or section layouts (dashboards, galleries) |
| **Content vs. Layout** | Content-based → items flow and adjust automatically | Layout-based → you define rows/columns, then place items |
| **Axis** | Works along **main axis & cross axis** | Works with **rows & columns** simultaneously |
| **Complex layouts** | Difficult (requires nesting flex containers) | Easier (define rows, columns, and place items anywhere) |

**When to Use Grid vs. Flexbox**

* **Use Grid when:**
* You need a **full page layout** or a section with rows + columns (e.g., dashboard, image gallery, product listing).
* Layout is **grid-based** and requires precise row/column alignment.
* **Use Flexbox when:**
* You need a **1D alignment** (row **or** column).
* For smaller UI components → navbars, buttons, menus, forms.
* When items should **flow dynamically** (auto-adjust).
* **Simple Example**

**Grid (2D layout): CSS**

.container {

display: grid;

grid-template-columns: 1fr 1fr 1fr; /\* 3 equal columns \*/

grid-template-rows: auto auto; /\* 2 rows \*/

gap: 10px;

}

**Flexbox (1D layout): CSS**

.container {

display: flex;

justify-content: space-between; /\* align items in a row \*/

}

**Question 2:** Describe the grid-template-columns, grid-template-rows, and grid-gap properties. Provide examples of how to use them.

**1. grid-template-columns**

* Defines the **number and width of columns** in a grid.
* Values can be in px, %, fr (fractional units), or auto.

**Example: CSS**

.container {

display: grid;

grid-template-columns: 200px 1fr 2fr;

}

👉 Creates **3 columns**:

* 1st = fixed 200px,
* 2nd = flexible 1fr (1 fraction of available space),
* 3rd = flexible 2fr (twice the size of 2nd).

**2. grid-template-rows**

* Defines the **number and height of rows** in a grid.
* Same units as columns.

**Example: CSS**

.container {

display: grid;

grid-template-rows: 100px auto 50px;

}

👉 Creates **3 rows**:

* 1st row = 100px,
* 2nd row = auto (adjusts to content),
* 3rd row = 50px.

**3. grid-gap (shorthand for row-gap + column-gap)**

* Defines the **space between grid items**.
* Does not add space at the container edges, only between items.

**Example: CSS.**

.container {

display: grid;

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

grid-template-rows: auto auto;

grid-gap: 20px; /\* same gap for rows and columns \*/

}

👉 Creates a grid of **3 columns × 2 rows** with 20px space between items.

**Complete Example: HTML.**

<div class="container">

<div class="item">1</div>

<div class="item">2</div>

<div class="item">3</div>

<div class="item">4</div>

<div class="item">5</div>

<div class="item">6</div>

</div>

**CSS.**

.container {

display: grid;

grid-template-columns: 150px 1fr 2fr; /\* 3 columns \*/

grid-template-rows: 100px auto; /\* 2 rows \*/

grid-gap: 15px; /\* spacing \*/

}

.item {

background: lightblue;

padding: 20px;

text-align: center;

}

**Responsive Web Design with Media Queries**

**Question 1:** What are media queries in CSS, and why are they important for responsive design?

* **Media queries** are CSS rules that apply styles based on **conditions** such as screen size, device type, or orientation.
* They allow you to create **responsive designs** that adapt to different devices (desktop, tablet, mobile).

**Syntax: CSS.**

@media (condition) {

/\* CSS rules here \*/

}

* Websites are viewed on devices with **different screen sizes** (phones, tablets, laptops, desktops).
* Media queries help ensure content is **readable and usable** on all devices.
* They improve **user experience** by making designs responsive.
* They reduce the need for separate mobile websites.

**Examples:**

**1. Change background color on small screens**

@media (max-width: 600px) {

body {

background-color: lightblue;

}

}

👉 When screen width is **600px or less** (like on mobile), background turns light blue.

**2. Responsive navigation menu**

@media (max-width: 768px) {

nav ul {

flex-direction: column; /\* Stack menu items vertically \*/

}

}

👉 On tablets/mobiles, the horizontal menu becomes vertical.

**3. Orientation-based query**

@media (orientation: landscape) {

body {

font-size: 18px;

}

}

👉 Applies only when the device is in **landscape mode**.

**Common Media Features Used**

* max-width / min-width → Screen width conditions
* orientation → Portrait or landscape
* resolution → Screen resolution (e.g., retina displays)
* prefers-color-scheme → Light or dark mode

**Question 2:** Write a basic media query that adjusts the font size of a webpage for screens smaller than 600px.

/\* Default font size for larger screens \*/

body {

font-size: 18px;

}

/\* Media query for small screens \*/

@media (max-width: 600px) {

body {

font-size: 14px;

}

}

**Typography and Web Fonts**

**Question 1:** Explain the difference between web-safe fonts and custom web fonts. Why might you use a web-safe font over a custom font?

**1. Web-Safe Fonts**

* **Definition:** Fonts that are pre-installed on most operating systems (Windows, macOS, Linux, etc.) and browsers.
* **Examples:** Arial, Times New Roman, Verdana, Georgia, Courier New, Trebuchet MS.
* **How They Work:** Since these fonts are already available on users’ devices, the browser just applies them directly without needing to download anything.

**2. Custom Web Fonts**

* **Definition:** Fonts that are not pre-installed on devices but are loaded via the web, usually through services like Google Fonts, Adobe Fonts, or self-hosted font files (e.g., .woff, .ttf).
* **Examples:** Open Sans, Roboto, Lato, Poppins, Montserrat.
* **How They Work:** The browser downloads the font file from a server and then renders the text with it.

**You might prefer web-safe fonts because:**

* **Performance:** No extra download time—faster page load speed.
* **Compatibility:** Works consistently across different devices and browsers.
* **Fallback Reliability:** If custom fonts fail to load due to poor network, users still see the intended font.
* **Accessibility:** Lighter for low-bandwidth users and ensures readability.
* **Simplicity:** Easy to implement without needing external services.

**Question 2:** What is the font-family property in CSS? How do you apply a custom Google Font to a webpage?

* The font-family property in CSS specifies the font to be used for an HTML element. It can accept multiple font names as a "fallback" system, meaning if the browser cannot render the first specified font, it will attempt to use the next one in the list. This ensures that the text will always be displayed with a suitable font.

**Applying a Custom Google Font to a Webpage**

**Option A: Using <link> in HTML <head>**

<link href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;700&display=swap" rel="stylesheet">

**Option B: Using @import in CSS**

@import url('https://fonts.googleapis.com/css2?family=Roboto:wght@400;700&display=swa