Module 3 – Frontend – CSS and CSS3

CSS Selectors & Styling

Theory Assignment

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

A CSS selector is a pattern used to select (choose) HTML elements so that we can apply styles to them. In simple words, selectors tell the browser “Which HTML part to style.” There are different types of selectors, like element, class, and ID.

1. Element Selector:

a. It selects HTML tags directly.

b. Example:

```
p {
color: blue;
}
```

This makes all (paragraphs) text blue.

2. Class Selector:

a. It selects elements with a specific class (using a dot . before the name).

b. Example:

```
. highlight {
background-color: yellow;
}
```

This applies yellow background to all elements having class="highlight".

3. ID Selector:

a. It selects an element with a unique ID (using a hash # before the name).

b. Example:

#title {

font-size: 24px;

}

This makes the element with id="title" have bigger text.

Conclusion:

CSS selectors are very important because they decide which part of the webpage will get styled.

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

CSS specificity means which style is stronger when two or more CSS rules try to style the same element. The browser follows this order:

1. Inline style → strongest

2. ID selector (#id) → strong

3. Class selector (.class) → medium

4. Element selector (p, h1, div) → weakest

If two rules have the same power, the last written rule will win.

Example:

p { color: blue; }

#title { color: red; }

hello

Here text will be red because ID is stronger than element selector.

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

1. Inline CSS

Definition: CSS is written directly inside an element's `style` attribute.
Example:

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

✅ **Advantages**

- Quick and easy for small changes.
- Useful for testing or overriding styles.
- Styles apply immediately to that specific element.

❌ **Disadvantages**

- Hard to maintain (styles are scattered).
- Breaks the separation of content and design.
- Increases code repetition (no reusability).
- Not suitable for large project

2. Internal CSS

Definition: CSS is placed inside a `<style>` tag within the `<head>` section of an HTML document.
Example:

<head>

```
<style>
 p {
  color: blue;
  font-size: 18px;
 }
</style>
</head>
```

✅ **Advantages**

- Better organization than inline CSS.
- Styles are reusable within the same page.
- Easier debugging since CSS is in one place.

❌ **Disadvantages**

- Styles apply only to that single HTML page.
- Not efficient for large websites with multiple pages (code duplication).
- Larger page size since CSS is embedded in HTML.

3. External CSS

Definition: CSS is written in a separate `.css` file and linked using `<link>` inside the `<head>`.

Example:

```
<head>
 <link rel="stylesheet" href="style.css">
</head>
```

Style.css

```
p {
 color: green;
 font-size: 18px;
}
```

✅ Advantages

- Best for large projects (styles in one file, used across multiple pages).
- Keeps HTML clean (separates structure from style).
- Reduces redundancy (one change updates all linked pages).
- Faster load times (CSS file is cached by browsers).

❌ Disadvantages

- Requires additional HTTP request (can slightly slow first load).
- If the CSS file is missing/not linked properly, pages lose styling

Comparison Table

Approach	Scope	Best Use Case	Maintainability	Performance
Inline	Single element	Small fixes, quick testing	Poor	Fast but messy
Internal	One HTML page	Small websites, unique pages	Medium	Moderate
External	Multiple pages	Large websites, reusable CSS	Excellent	Best (with caching)

CSS Box Model

Theory Assignment

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

CSS Box Model

In CSS, every element on a webpage is treated as a **box**.
The **box model** describes how the size of an element is calculated and how spacing is applied around it.

An element consists of **four main components** (inside → out):

1. Content

- The innermost part of the box (text, images, or other elements).
- Controlled by properties like `width`, `height`, `font-size`, etc.

👉 **Effect on size:** Increasing `width` or `height` increases only the content area.

2. Padding

- Space **between content and border**.
- Transparent area (background color extends into padding).
- Defined using `padding`, `padding-top`, `padding-right`, etc.

👉 **Effect on size:** Adds extra space inside the box, pushing the border outward.

3. Border

- A line surrounding the padding and content.
- Can be styled with `border-width`, `border-style`, `border-color`.

👉 **Effect on size:** Border thickness adds to the total size of the element.

4. Margin

- The outermost space **outside the border**, creating distance between elements.
- Transparent, doesn't have background color.
- Defined using `margin`, `margin-top`, `margin-right`, etc.

👉 **Effect on size:** Doesn't increase the element's size but increases the **space around it**.

Total Element Size Calculation

By default (with `box-sizing: content-box`):

$$\textit{Total Width=Content Width+Padding (left + right)+Border (left + right)+Margin (left + right)}$$ $$\text{Total Width} = \text{Content Width} + \text{Padding (left + right)} + \text{Border (left + right)} + \text{Margin (left + right)}$$

Total Width=Content Width+Padding (left + right)+Border (left + right)+Margin (left + right) $$\textit{Total Height=Content Height+Padding (top + bottom)+Border (top + bottom)+Margin (top + bottom)}$$ $$\text{Total Height} = \text{Content Height} + \text{Padding (top + bottom)} + \text{Border (top + bottom)} + \text{Margin (top + bottom)}$$

Total Height=Content Height+Padding (top + bottom)+Border (top + bottom)+Margin (top + bottom)

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

Box-Sizing in CSS

The `box-sizing` property defines **how the total width and height of an element are calculated**.
There are two main values: **`content-box`** and **`border-box`**.

1. content-box (Default)

- **Default behavior in CSS.**
- The `width` and `height` apply **only to the content area**.
- Padding and border are added **on top of** the width and height.

✅ Example:

```
.box1 {
  box-sizing: content-box; /* default */
  width: 200px;
  padding: 20px;
  border: 10px solid black;
}
```

👉 Total width = `200 (content) + 40 (padding) + 20 (border)` = **260px**
👉 Total height = `height + padding + border`

2. border-box

- The `width` and `height` include **content + padding + border**.
- Makes it easier to manage layouts (size stays consistent).

✅ Example:

```
.box2 {
  box-sizing: border-box;
  width: 200px;
  padding: 20px;
  border: 10px solid black;
}
```

👉 Total width = **200px fixed** (content shrinks to fit padding + border inside).

👉 Total height = **200px fixed** (same logic).

Comparison Table

Feature	`content-box` (default)	`border-box`
Width/Height apply to	Only content	Content + padding + border
Padding & border effect	Increase total size	Included inside size
Layout control	Harder, requires calculations	Easier, consistent sizing
Default in CSS	✅ Yes	❌ No (must be set manually)

CSS Flexbox

Theory Assignment

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

CSS Flexbox

Definition:
CSS **Flexbox (Flexible Box Layout)** is a layout model that makes it easier to arrange elements in a **row or column**, align them, and distribute space dynamically—even when screen sizes or content change.

Unlike older methods (like floats, tables, or inline-block), Flexbox is **responsive by design** and simplifies alignment.

Key Concepts

1. Flex Container

- The **parent element** where `display: flex;` (or `display: inline-flex;`) is applied.
- It defines a **flex context** for its child elements.
- Properties that apply to the container include:
  - `flex-direction` → row, column, row-reverse, column-reverse
  - `justify-content` → alignment on the main axis
  - `align-items` → alignment on the cross axis
  - `flex-wrap` → whether items wrap or not

✅ Example:

```
.container {
  display: flex;
  flex-direction: row;  /* horizontal layout */
  justify-content: space-between;
  align-items: center;
}
```

2. Flex Items

- The **children** of a flex container.
- They follow the container's flex rules but can also have their own flex properties.
- Properties that apply to flex items:
  - `flex-grow` → how much an item can expand
  - `flex-shrink` → how much an item can shrink
  - `flex-basis` → initial size before growing/shrinking
  - `align-self` → override container's `align-items`

✅ Example:

```
.item {
  flex-grow: 1;   /* expand to fill available space */
  flex-basis: 100px; /* initial size */
}
```

Visual Example

```
<!DOCTYPE html>
<html>
<head>
<style>
.container {
  display: flex;
  justify-content: space-around; /* distribute space */
  align-items: center;           /* vertical alignment */
  height: 200px;
  background: lightgray;
}
```

```
.item {
  background: steelblue;
  color: white;
  padding: 20px;
  font-size: 18px;
}
</style>
</head>
<body>
  <div class="container">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
  </div>
</body>
</html>
```

👉 Here:

- `.container` = **flex-container**
- `.item` = **flex-items**
- Items are aligned **horizontally** with space between them.

Why Flexbox is Useful

✅ Simplifies responsive design (elements adapt to screen sizes).
✅ Easier alignment (center, stretch, space-between).
✅ Eliminates need for floats, clears, and complex hacks.
✅ Great for **1D layouts** (rows OR columns).

Question 2: Describe the properties justify-content, align-items, and flexdirection used in Flexbox.

1. flex-direction

- Defines the **main axis** of the flex container → determines whether items are arranged **horizontally (row)** or **vertically (column)**.
- Values:
  - `row` → items placed left to right (**default**)
  - `row-reverse` → items placed right to left
  - `column` → items placed top to bottom
  - `column-reverse` → items placed bottom to top

✅ Example:

```
.container {
  display: flex;
  flex-direction: column; /* stack items vertically */
}
```

2. justify-content

- Aligns items **along the main axis** (defined by `flex-direction`).
- Useful for controlling **horizontal spacing in row** or **vertical spacing in column**.
- Values:
  - `flex-start` → items start at the beginning (default)
  - `flex-end` → items align at the end
  - `center` → items centered
  - `space-between` → equal space *between* items
  - `space-around` → equal space *around* items

- `space-evenly` → equal space *between and outside* items

✅ Example:

```
.container {
  display: flex;
  justify-content: space-between;
}
```

3. align-items

- Aligns items **along the cross axis** (the perpendicular axis).
- If `flex-direction: row`, cross axis = vertical.
- If `flex-direction: column`, cross axis = horizontal.
- Values:
  - `stretch` → items stretch to fill container (default)
  - `flex-start` → align to start of cross axis (top for row, left for column)
  - `flex-end` → align to end of cross axis
  - `center` → align at center of cross axis
  - `baseline` → align based on text baseline

✅ Example:

```
.container {
  display: flex;
  align-items: center; /* vertically center items if row */
}
```

Quick Visualization

If `flex-direction: row;` (default):

- `justify-content` → controls **horizontal alignment**
- `align-items` → controls **vertical alignment**

If `flex-direction: column;`:

- `justify-content` → controls **vertical alignment**
- `align-items` → controls **horizontal alignment**

CSS Grid

Theory Assignment

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

What is CSS Grid?

- **CSS Grid Layout** is a **two-dimensional** layout system in CSS.
- It allows you to arrange elements into **rows and columns** simultaneously.
- Works like a spreadsheet (with rows, columns, and cells).

✅ Example:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr; /* 3 equal columns */
  grid-template-rows: auto 200px;     /* first row auto,
second row fixed */
  gap: 20px; /* spacing between cells */
```

```
}

<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 direction	One-dimensional (row **or** column)	Two-dimensional (rows **and** columns)
Alignment	Works best for aligning items in a line	Works best for page/section layouts
Content vs Structure	Content-driven (items push layout)	Layout-driven (explicit rows/columns)
Control	Controls spacing and alignment of items	Controls full grid layout and placement
Example use case	Navbars, buttons, small UI pieces	Full-page layouts, galleries, dashboards

When to Use Grid vs Flexbox

☑ Use Grid when:

- You need a **two-dimensional layout** (rows + columns).
- You're building a **page structure** (header, sidebar, main content, footer).
- You want precise placement of elements in a grid-like fashion (e.g., image galleries, dashboards).

☑ Use Flexbox when:

- You need a **one-dimensional layout** (row **or** column).
- You're aligning items inside a single container (like navbar, buttons, or form fields).
- You want flexible spacing and alignment without defining strict rows/columns.

Quick Example: Grid vs Flexbox

Flexbox (Row Layout)

```
.container {
  display: flex;
  justify-content: space-between;
}
```

👉 Items arranged in **one row**.

Grid (Row + Column Layout)

```
.container {
  display: grid;
  grid-template-columns: 1fr 2fr 1fr;
  grid-template-rows: auto auto;
  gap: 10px;
}
```

👉 Items arranged in **rows and columns**, like a table.

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 the grid.
- You can use:
  - Fixed units (`px`, `em`)
  - Relative units (`%`)
  - Flexible units (`fr`) → fraction of available space
  - Keywords like `auto`

✅ Example:

```
.container {
  display: grid;
  grid-template-columns: 200px 1fr 2fr;
}
```

👉 This creates **3 columns**:

- First column = 200px
- Second column = 1 part of remaining space
- Third column = 2 parts of remaining space

2. grid-template-rows

- Defines the **number and height of rows** in the grid.
- Works the same way as columns.

✅ Example:

```
.container {
  display: grid;
  grid-template-rows: 100px auto 50px;
}
```

👉 This creates **3 rows**:

- Row 1 = 100px
- Row 2 = auto (adjusts to content)
- Row 3 = 50px

3. grid-gap (or gap in modern CSS)

- Defines the **space between rows and columns**.
- You can set:
  - One value → applies to both rows & columns
  - Two values → first = row gap, second = column gap

✅ Example:

```
.container {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  grid-template-rows: auto auto;
  gap: 20px;          /* shorthand for row + column */
  row-gap: 10px;      /* only row spacing */
  column-gap: 30px;   /* only column spacing */
}
```

Full Example

```
<!DOCTYPE html>
<html>
<head>
<style>
.container {
  display: grid;
  grid-template-columns: 1fr 2fr 1fr;  /* 3 columns */
  grid-template-rows: 100px auto 50px; /* 3 rows */
  gap: 15px; /* space between rows & columns */
  background: lightgray;
  padding: 10px;
}
.item {
  background: steelblue;
  color: white;
  padding: 20px;
  text-align: center;
  font-size: 18px;
}
</style>
</head>
<body>
  <div class="container">
    <div class="item">Item 1</div>
    <div class="item">Item 2</div>
    <div class="item">Item 3</div>
    <div class="item">Item 4</div>
    <div class="item">Item 5</div>
    <div class="item">Item 6</div>
  </div>
</body>
```

```
</html>
```

👉 This creates a **3x3 grid-like structure** with gaps between items.

Responsive Web Design with Media Queries

Theory Assignment

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

What are Media Queries in CSS?

- **Media queries** are CSS techniques used to apply styles **based on device characteristics**, such as:
  - Screen size (width, height)
  - Screen resolution (dpi)
  - Device orientation (portrait, landscape)
  - Media type (screen, print, speech)
- They let you create **responsive designs** that adapt to different devices (desktop, tablet, mobile).

✅ Basic Syntax:

```
@media (condition) {
  /* CSS rules applied only if condition is true */
}
```

Why Are Media Queries Important for Responsive Design?

1. **Device Adaptability** → Websites look good on desktops, tablets, and phones.
2. **Improved User Experience** → Content is readable and accessible without zooming or scrolling.
3. **Performance** → Serve optimized layouts/styles for different devices.
4. **Modern Web Standard** → Essential for mobile-first design (Google ranks mobile-friendly sites higher).

Common Examples

1. Responsive Layout by Screen Width

```
/* For desktops */
@media (min-width: 1024px) {
  body {
    background: lightblue;
    font-size: 18px;
  }
}

/* For tablets */
@media (min-width: 768px) and (max-width: 1023px) {
  body {
    background: lightgreen;
    font-size: 16px;
  }
}

/* For mobile */
```

```
@media (max-width: 767px) {
  body {
    background: lightpink;
    font-size: 14px;
  }
}
```

2. Orientation (portrait vs landscape)

```
@media (orientation: portrait) {
  .container {
    flex-direction: column;
  }
}

@media (orientation: landscape) {
  .container {
    flex-direction: row;
  }
}
```

3. Print Styles

```
@media print {
  body {
    color: black;
    background: white;
  }
  nav, footer {
    display: none; /* hide navigation & footer when printing
*/
```

```
  }
}
```

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

Here's a simple example 👇

```
/* Default font size for larger screens */
body {
  font-size: 18px;
}

/* Media query for screens smaller than 600px */
@media (max-width: 600px) {
  body {
    font-size: 14px; /* smaller font for small screens */
  }
}
```

Typography and Web Fonts

Theory Assignment

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

1. Web-Safe Fonts

- **Definition:** Pre-installed fonts that are commonly available across most operating systems (Windows, macOS, Linux).
- Since they are already installed on users' devices, the browser can display them **without downloading extra files**.

- Examples:
    - Arial
    - Times New Roman
    - Georgia
    - Verdana
    - Courier New

✅ **Advantages**

- Fast loading (no need to download font files).
- Reliable (works across devices).
- Good for performance-sensitive websites.

❌ **Disadvantages**

- Limited variety (not much design freedom).
- Can look plain or generic.

2. Custom Web Fonts

- **Definition:** Fonts that are **not pre-installed** on devices but loaded from the web using `@font-face` or font providers (like **Google Fonts**, Adobe Fonts).
- Example using Google Fonts:

```
@import
url('https://fonts.googleapis.com/css2?family=Roboto&display
=swap');

body {
  font-family: 'Roboto', sans-serif;
}
```

✅ **Advantages**

- Huge design flexibility (thousands of fonts available).
- Better branding and unique look.

❌ **Disadvantages**

- Requires downloading font files (slightly slower load time).
- If the font fails to load, fallback fonts may change design.
- Too many font weights/styles can hurt performance.

Why Use a Web-Safe Font Over a Custom Font?

You might prefer a **web-safe font** when:

1. **Performance is critical** → no extra font files to load.
2. **Email templates** → many email clients block external fonts, so web-safe fonts are more reliable.
3. **Fallback safety** → ensures the text always displays correctly even if custom fonts fail.
4. **Minimalist designs** → sometimes a clean, classic look is enough.

✅ **In short:**

- **Web-safe fonts** = reliable, fast, but limited.
- **Custom web fonts** = more creative freedom, but require extra loading.
- Use **web-safe** when performance and reliability matter most (emails, small sites, critical apps).

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

1. What is the `font-family` Property?

- The **`font-family` property** in CSS specifies the typeface (font) for text.
- You can list **multiple fonts** in order of preference:
  - The browser will try the first font;
  - If not available, it will try the next;
  - Finally, it falls back to a **generic family** (`serif`, `sans-serif`, `monospace`, `cursive`, `fantasy`).

✅ Example:

```
p {
  font-family: "Times New Roman", Georgia, serif;
}
```

👉 Here, the browser will try:

1. `"Times New Roman"`
2. If not available → `Georgia`
3. If still not → generic `serif`

2. Applying a Custom Google Font

Step 1: Import the Font

You can add Google Fonts in two ways:

a) Using `<link>` in HTML `<head>`

```
<link
href="https://fonts.googleapis.com/css2?family=Roboto&displa
```

```
y=swap" rel="stylesheet">
```

b) Using `@import` in CSS

```
@import
url('https://fonts.googleapis.com/css2?family=Roboto&display
=swap');
```

Step 2: Apply the Font with `font-family`

```
body {
  font-family: 'Roboto', sans-serif;
}
```

Full Example

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Google Fonts Example</title>
  <!-- Step 1: Import Google Font -->
  <link
href="https://fonts.googleapis.com/css2?family=Roboto&displa
y=swap" rel="stylesheet">
  <style>
    /* Step 2: Apply Google Font */
    body {
      font-family: 'Roboto', sans-serif;
      font-size: 18px;
```

```
    }
  </style>
</head>
<body>
  <h1>Hello, World!</h1>
  <p>This text uses the Roboto Google Font.</p>
</body>
</html>
```

✅ **In short:**

- `font-family` → sets the font of text, with fallbacks.
- To use a Google Font → import it (`<link>` or `@import`) + apply with `font-family`.