Module 2 – WD – CSS and CSS3

Theory Assignment

CSS Selectors & Styling

Question 1:

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

Ans:

A CSS selector is a pattern used to select and target HTML elements on a web page to apply specific styles defined in CSS rules. Selectors allow for precise control over which elements receive certain styling properties.

Here are examples of common basic selectors :

❖ Element Selector:

- This selector targets all instances of a specific HTML element based on its tag name.

Code:

```
p {
  color: blue;
  font-size: 16px;
}
```

- In this example, all `<p>` (paragraph) elements on the page will have blue text and a font size of 16 pixels.

❖ Class Selector:

- This selector targets elements that have a specific `class` attribute. Multiple elements can share the same class, allowing for consistent styling across various elements.

Code:

```
.highlight {
  background-color: yellow;
  font-weight: bold;
}
```

- Any HTML element with `class="highlight"` will have a yellow background and bold text. For example: `<span class="highlight">Important text</span>`

❖ ID Selector:

- This selector targets a single, unique HTML element based on its `id` attribute. An `id` should be unique within a given HTML document.

Code:

```
#main-header {
  text-align: center;
  padding: 20px;
}
```

- Only the HTML element with `id="main-header"` will have its text centered and a padding of 20 pixels. For example: `<h1 id="main-header">Welcome</h1>`.

Question 2:

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

Ans:

CSS specificity is a set of rules used by browsers to determine which CSS declarations apply to an HTML element when multiple, conflicting styles target the same element. It essentially assigns a "weight" or "score" to each CSS selector, and the selector with the highest specificity takes precedence.

How Specificity is Calculated:

Specificity is calculated based on four categories, which are compared in descending order of importance:

Inline styles:

Styles applied directly to an HTML element using the `style` attribute have the highest specificity.

IDs:

Selectors targeting an element by its `id` attribute (e.g., `#myElement`) have the next highest specificity.

Classes, attributes, and pseudo-classes:

Selectors using classes (e.g., `.myClass`), attribute selectors (e.g., `[type="text"]`), and pseudo-classes (e.g., `:hover`, `:nth-child`) have a lower specificity than IDs.

Elements and pseudo-elements:

Selectors targeting an element by its tag name (e.g., `p`, `div`) or pseudo-elements (e.g., `::before`, `::after`) have the lowest specificity.

> When comparing two selectors, the browser first compares the number of inline styles, then the number of IDs, then classes/attributes/pseudo-classes, and finally elements/pseudo-elements. The selector with the higher count in the first differing category wins.

How Conflicts Between Multiple Styles Are Resolved:

> Conflicts between multiple styles are resolved by the browser following a hierarchical process:

Importance (`!important`):

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

Origin and Cascade Layers:

Styles from different origins (e.g., user agent stylesheets, user stylesheets, author stylesheets) and within CSS Cascade Layers have a defined order of precedence.

Specificity:

If declarations have the same importance and origin/layer precedence, the declaration with the higher specificity wins.

Order of Appearance:

If two selectors have the exact same specificity and target the same property, the declaration that appears later in the stylesheet (or in the order of linked stylesheets) takes precedence. This is also known as the "last rule wins" principle.

Inheritance:

If a property is not explicitly defined for an element but is inherited from a parent element, the inherited value is used unless a more specific rule is applied directly to the element.

Question 3:

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

Ans:

CSS (Cascading Style Sheets) is essential for defining the presentation of web pages, including layout, colors, and fonts. There are three primary methods to apply CSS:

- **Inline CSS**: Applied directly within an HTML element's style attribute, affecting only that specific element.

- **Internal CSS**: Defined within a <style> tag in the <head> section of an HTML document, influencing the entire page.

- **External CSS**: Contained in a separate .css file linked to the HTML document, allowing for consistent styling across multiple pages.

Differences between Inline, Internal, and External CSS

Feature	Inline CSS	Internal CSS	External CSS
Location	It is used within HTML tag using the style attribute.	It is used within <head> section of HTML document.	It is used in a separate .css file.
Selector Scope	Affects a single element or a group of elements.	Affects multiple elements within the same HTML element.	Affects multiple HTML documents or an entire website.
Reusability	Not reusable. Styles need to be repeated for each element.	Can be reused on multiple elements within the same HTML document.	Can be reused on multiple HTML documents or an entire website.
Priority	Highest priority. Overrides internal and external styles.	Medium priority. Overrides external styles but can be overridden by inline styles.	Lowest priority. Can be overridden by both inline and internal styles.
File Size	Inline styles increase the HTML file size, which can affect the page load time.	Internal styles are part of the HTML file, which increases the file size.	External styles are in a separate file, which reduces the HTML file size and can be cached for faster page loads.
Maintainability	Not easy to maintain. Changes need to be made	Relatively easy to maintain. Changes need to be made in one place in the <head> section.	Easiest to maintain. Changes need to be made in one place in the external .css file.

	manually to each element.		

The advantages and disadvantages:

1. Inline CSS

Advantages:

- Quick and easy for specific, one-off styling or testing.
- High specificity, overriding other styles for that element.

Disadvantages:

- Poor maintainability and scalability, as styles are scattered throughout the HTML.
- Increases HTML file size.
- Not reusable across multiple elements or pages.

2. Internal CSS

Advantages:

- Centralized styling for a single HTML page.
- Avoids separate CSS files, simplifying management for small, single-page projects.

Disadvantages:

- Styles are not reusable across different HTML pages, leading to repetition.
- Increases the size of the HTML file.
- Can become cumbersome for larger projects with many styles.

3. External CSS

Advantages:

- Promotes reusability and consistency across multiple HTML pages.
- Separates content (HTML) from presentation (CSS), improving code organization and readability.
- Faster loading times after initial caching of the CSS file.
- Easier maintenance, as changes in one CSS file reflect across all linked pages.

Disadvantages:

- Requires an additional HTTP request to fetch the CSS file, potentially slightly impacting initial page load.
- Can be more complex to set up initially for very small projects.

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?

Ans:

The CSS Box Model describes how HTML elements are rendered as rectangular boxes, encompassing their content and surrounding space. This model comprises four distinct components, each affecting the element's overall size and layout:

- **Content:**

This is the innermost area where the actual content of the element, such as text, images, or other media, resides. The `width` and `height` CSS properties directly control the dimensions of this content area.

- **Padding:**

This is the transparent space between the content area and the element's border. Padding adds space inside the element, pushing the content away from the border. It is controlled by properties like `padding-top`, `padding-right`, `padding-bottom`, and `padding-left`, or the shorthand `padding`. Padding adds to the element's total visible size.

- **Border:**

This is a line that surrounds the padding and content. It defines the visual edge of the element. Border properties like `border-width`, `border-style`, and `border-color` control its appearance. The border's width adds to the element's total visible size.

- **Margin:**

This is the transparent space outside the element's border, creating separation between the element and other adjacent elements. Margins do not have a background and are used for spacing elements apart. They are controlled by properties like `margin-top`, `margin-right`, `margin-bottom`, and `margin-left`, or the shorthand `margin`. Margins affect the element's position relative to other elements but are not considered part of the element's visible size in the same way padding and border are when calculating `width` and `height` based on `box-sizing: content-box`.

How each affects the size of an element:

- **Content:**

The `width` and `height` properties explicitly set the size of the content area.

- **Padding:**

Adds to the element's total calculated width and height. For example, a `width: 100px` element with `padding: 10px` on all sides will occupy `120px` of horizontal space (10px left padding + 100px content + 10px right padding).

- **Border:**

Adds to the element's total calculated width and height, similar to padding. A `width: 100px` element with `border: 5px solid black` and `padding: 10px` will

occupy `130px` of horizontal space (5px left border + 10px left padding + 100px content + 10px right padding + 5px right border).

- **Margin:**

Does not directly add to the element's intrinsic width or height but creates external space around the element, influencing its overall footprint in the document flow and its distance from other elements.

Question 2:

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

Ans:

The `box-sizing` CSS property determines how an element's total width and height are calculated, specifically in relation to its padding and border. There are two primary values: `content-box` and `border-box`.

1. `content-box` (Default):

- When `box-sizing` is set to `content-box`, the `width` and `height` properties you define for an element apply only to its content area.
- Any `padding` and `border` you add to the element are then added outside of this specified width and height, increasing the element's total occupied space on the page.
- **Example:** If you set `width: 200px; padding: 10px; border: 2px;`, the element's total width will be `200px (content) + 10px (left padding) + 10px (right padding) + 2px (left border) + 2px (right border) = 224px`.

2. `border-box`:

- When `box-sizing` is set to `border-box`, the `width` and `height` properties you define for an element include the content, padding, and border.
- This means that any padding or border you add will be included within the specified width and height, reducing the available space for the content. The element's total occupied space on the page will remain exactly as specified by the `width` and `height`.
- **Example:** If you set `width: 200px; padding: 10px; border: 2px;`, the element's total width will remain `200px`. The content area will shrink to accommodate the padding and border within that 200px.

❖ **Which is the default?**

The default value for the `box-sizing` property in CSS is `content-box`. This means that unless you explicitly declare `box-sizing: border-box;` for an element or use a CSS reset that applies it globally, elements will behave according to the `content-box` model.

CSS Flexbox

Question 1:

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

Ans:

CSS Flexbox, formally known as the Flexible Box Layout module, is a one-dimensional CSS layout model designed to provide an efficient way to lay out, align, and distribute space among items within a container, even when their size is unknown or dynamic. It simplifies the creation of flexible and responsive web designs, particularly for component layouts like navigation menus, forms, and card grids.

Flexbox is highly useful for layout design because it offers powerful capabilities for:

- **Alignment and Distribution:**

It provides properties to align items along both the main axis (the direction of the flex container) and the cross-axis (perpendicular to the main axis), and to distribute extra space among items or around them.

- **Responsiveness:**

Flex items can grow or shrink to fill available space or prevent overflow, making layouts adaptable to different screen sizes and viewports.

- **Ordering:**

The visual order of flex items can be easily changed, independent of their source order in the HTML, which is beneficial for responsive design.

Flex-container and Flex-item:

- **Flex-container:** This is the parent element that holds the flex items. To enable Flexbox, the `display` property of this parent element is set to `flex` or `inline-flex`. The flex-container defines the main and cross axes and controls the overall layout of its direct children (flex-items) using properties like `flex-direction`, `justify-content`, `align-items`, and `align-content`.

Code :

```
.flex-container {
  display: flex; /* Makes this element a flex container */
  flex-direction: row; /* Arranges items in a row */
  justify-content: center; /* Centers items horizontally */
}
```

- **Flex-item:** These are the direct child elements within a flex-container. They are the individual elements whose layout is managed by the flex-container. Flex-items can be controlled individually using properties like `flex-grow`, `flex-shrink`, `flex-basis`, and `align-self`.

  Code :

```
.flex-item {
  flex-grow: 1; /* Allows the item to grow and fill available space */
  align-self: center; /* Aligns this specific item to the center on the cross-axis */
}
```

Question 2:

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

Ans:

Flexbox utilizes several properties to control the layout and alignment of items within a container. Among these, `justify-content`, `align-items`, and `flex-direction` are fundamental for defining the primary layout behavior.

- **`flex-direction`:**

This property defines the main axis of the flex container, determining the direction in which flex items are laid out. It can be set to `row` (default, items arranged horizontally), `row-reverse`, `column` (items arranged vertically), or `column-reverse`. The main axis dictates the direction for `justify-content`.

❖ justify-content:

This property controls the alignment of flex items along the main axis of the flex container. It manages the distribution of free space between and around content items. Common values include:

- **`flex-start`**: Items are packed towards the start of the main axis.
- **`flex-end`:** Items are packed towards the end of the main axis.
- **`center`**: Items are centered along the main axis.
- **`space-between`**: Items are evenly distributed, with the first item at the start and the last item at the end.
- **`space-around`**: Items are evenly distributed with equal space around them.
- **`space-evenly`**: Items are distributed so that the spacing between any two adjacent items, and the space at the start and end, are all equal.

❖ align-items:

This property controls the alignment of flex items along the cross axis of the flex container. The cross axis is perpendicular to the main axis. Common values include:

- **`flex-start`**: Items are aligned to the start of the cross axis.
- **`flex-end`**: Items are aligned to the end of the cross axis.
- **`center`**: Items are centered along the cross axis.
- **`stretch`**: Items are stretched to fill the container along the cross axis (default if no explicit height/width is set).
- **`baseline`**: Items are aligned based on their baselines.

CSS Grid

Question 1:

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

Ans:

CSS Grid and Flexbox are both powerful CSS layout modules, but they are designed for different purposes.

CSS Grid is a two-dimensional layout system that allows you to define rows and columns simultaneously. It provides precise control over the placement and alignment of elements within a grid structure, making it ideal for creating complex page layouts and complete user interfaces. You define the grid structure on a container element using properties like `grid-template-rows`, `grid-template-columns`, and `grid-gap`. Child elements are then placed within this grid using properties like `grid-row`, `grid-column`, `grid-area`, etc.

CSS Flexbox (Flexible Box Layout) is a one-dimensional layout system designed for distributing and aligning items within a single row or column. It excels at arranging items dynamically based on

available space, making it suitable for components like navigation menus, card layouts, or distributing items within a section. Flexbox operates on a main axis (the direction of the flow, either row or column) and a cross axis (perpendicular to the main axis). Properties like `justify-content`, `align-items`, `flex-grow`, and `flex-shrink` control the behavior of flex items.

❖ Key Differences:

- **Dimensionality:**

Grid is for two-dimensional layouts (rows AND columns), while Flexbox is for one-dimensional layouts (rows OR columns).

- **Control:**

Grid offers explicit control over item placement within a defined grid, while Flexbox focuses on distributing and aligning items along an axis.

- **Use Cases:**

Grid is generally preferred for overall page layouts and complex structures, while Flexbox is better for arranging components and content within sections.

When to Use Grid Over Flexbox:

- **Full Page Layouts:**

When designing the entire structure of a webpage with defined rows and columns for header, sidebar, main content, and footer.

- **Complex Component Layouts:**

For components requiring precise alignment and spacing across both rows and columns, such as dashboards or image galleries with specific column and row spans.

- **Content-Independent Layout:**

When the layout structure is more important than the content's intrinsic size, as Grid allows you to define the layout first and then place content within it.

- **Overlapping Elements:**

Grid provides features like `grid-area` that simplify the positioning of elements that might overlap or span multiple grid cells.

Question 2:

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

Ans:

The CSS Grid Layout module provides powerful tools for creating two-dimensional layouts. Key properties for defining the grid structure and spacing include `grid-template-columns`, `grid-template-rows`, and `gap` (or its longhands `row-gap` and `column-gap`).

1. `grid-template-columns`:

This property defines the number, size, and order of columns in a grid layout. Values can be lengths (e.g., `px`, `em`, `rem`), percentages, the `fr` unit (fractional unit, distributing available space), `auto` (auto-sizing based on content), or `repeat()` function for repeating patterns.

Code:

```
.container {
display: grid;
grid-template-columns: 100px 1fr 2fr; /* Three columns: 100px fixed, then 1/3 and 2/3 of remaining space */ }
```

2. `grid-template-rows`:

Similar to `grid-template-columns`, this property defines the number, size, and order of rows in a grid layout. Values can be lengths, percentages, `fr` units, `auto`, or the `repeat()` function.

Code:

```
.container {
display: grid;
grid-template-rows: auto 50px 100px; /* Three rows: auto-sized, then 50px fixed, then 100px fixed */
}
```

3. `gap` (or `row-gap` and `column-gap`):

These properties control the spacing between grid tracks (rows and columns).

- **`row-gap`**: Specifies the gap between grid rows.
- **`column-gap`:** Specifies the gap between grid columns.
- **`gap`**: A shorthand property for both `row-gap` and `column-gap`. If one value is provided, it applies to both. If two values are provided, the first is for `row-gap` and the second for `column-gap`.

```
.container { display: grid;
grid-template-columns: 1fr 1fr 1fr;
 grid-template-rows: auto auto; gap: 20px;
 /* 20px gap between both rows and columns */ }
/* Equivalent using longhands: /
 .container-alt {
display: grid;
 grid-template-columns: 1fr 1fr 1fr;
grid-template-rows: auto auto; row-gap: 15px;
 column-gap: 30px;
/ 15px between rows, 30px between columns */ }
```

Responsive Web Design with Media Queries

Question 1:

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

Ans:

Media queries in CSS are a powerful feature that allow web developers to apply different styles to a webpage based on the characteristics of the device displaying the content. This is achieved using the `@media` at-rule, which conditionally

applies a block of CSS properties if a specified media condition is true.

Why are they important for responsive design?

Media queries are fundamental to responsive web design because they enable websites to adapt their layout and appearance to various screen sizes and device types, ensuring an optimal viewing experience for all users. Their importance stems from the following:

Device Compatibility:
They allow content to be presented effectively on a wide range of devices, from small mobile phones and tablets to large desktop monitors, without requiring separate versions of the website.

Flexible Layouts:
Media queries facilitate the creation of fluid and flexible layouts that can adjust dynamically. For instance, a multi-column layout on a desktop might transform into a single-column layout on a mobile device, improving readability and usability.

Enhanced User Experience:
By tailoring the design to the user's device, media queries contribute to a better user experience. This includes optimizing font sizes, image scaling, navigation menus, and the overall visual presentation to suit the specific viewing context.

Targeted Styling:
They enable targeted styling based on various factors beyond just screen width, such as device orientation (portrait or landscape), resolution, media type (screen, print), and even user preferences like dark mode.

Future-Proofing:
As new devices and screen sizes emerge, media queries provide a flexible framework for adapting designs without a complete overhaul, making websites more future-proof.

Question 2:

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

Ans:

A basic media query to adjust the font size of a webpage for screens smaller than 600px can be implemented in CSS as follows:

Code:

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

/* Styles for screens smaller than 600px */
@media screen and (max-width: 599px) {
 body {
  font-size: 14px; /* Adjust font size for smaller screens */
 }
}
```

In this example:

- The `body` selector defines a default font size for the entire webpage, which applies to screens wider than 599px.
- The `@media screen and (max-width: 599px)` rule targets devices with a screen width of 599 pixels or less.

- Inside this media query, the `body` font size is adjusted to `14px`, overriding the default for smaller screens.

This approach ensures that the content remains readable and visually appealing across various screen sizes, providing a better user experience on mobile devices and smaller viewports.

Typography and Web Fonts

Question 1:

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

Ans:

Web-safe fonts are a limited set of typefaces that are pre-installed on the vast majority of computer operating systems and devices. This ensures that when a user accesses a webpage using a web-safe font, the font is rendered consistently across different systems without needing to be downloaded. Examples include Arial, Verdana, Times New Roman, and Georgia.

Custom web fonts, on the other hand, are typefaces that are not universally installed on users' devices. These fonts are typically hosted on a server (either external services like Google Fonts or a self-hosted server) and are downloaded by the user's browser when a webpage is accessed. This allows for greater creative freedom and unique branding opportunities.

Reasons to use a web-safe font over a custom font:

- **Performance and Load Time:**

Web-safe fonts load instantly because they are already present on the user's system. Custom fonts require a download, which can increase page load times, especially for users with slower internet connections.

- **Reliability and Consistency:**

Web-safe fonts guarantee that the text will appear as intended on virtually any device or browser, eliminating potential display issues or font fallback inconsistencies that can occur with custom fonts if they fail to load.

- **Simplicity and Ease of Implementation:**

Using web-safe fonts simplifies web development as there is no need to manage font files, external services, or potential licensing issues associated with custom fonts.

- **Accessibility:**

While custom fonts can be optimized for accessibility, web-safe fonts are generally well-established and highly legible, ensuring a good reading experience for a broad audience.

Question 2:

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

Ans:

The `font-family` property in CSS specifies the font for an element. It can hold several font names as a "fallback" system, ordered by priority and separated by commas. If the browser does not support the first font, it tries the next font in the list. This property can use specific font names like "Arial" or generic font families like "serif" or "sans-serif".

To apply a custom Google Font to a webpage, follow these steps:

- **Choose a Font:** Navigate to Google Fonts and select the desired font(s). Click "Select this style" for each style you want to use.
- **Embed the Font:** In the "Selected families" drawer that appears, copy the `<link>` tag provided for embedding the font into your HTML.
- **Paste into HTML:** Paste this `<link>` tag into the `<head>` section of your HTML file.

Code :

```
<head>
  <link rel="stylesheet"
href="https://fonts.googleapis.com/css2?family=Your+Font+Name&display=swap">
</head>
```

Replace `Your+Font+Name` with the actual name of your chosen font, replacing spaces with + signs.

- **Apply in CSS:** In your CSS file or within a `<style>` block in your HTML, use the `font-family` property to apply the custom font to the desired elements. Always include a generic font family as a fallback.

Code:

```
body {
    font-family: 'Your Font Name', sans-serif;
```

```
/* Replace with your chosen font name */
  }



  h1 {
     font-family: 'Another Font', serif;
/* Example with a different font */
  }
```