## Q.1 What’s Box Model in CSS ?

**Ans:**
The box model is a fundamental concept in CSS (Cascading Style Sheets) that describes how elements are structured and displayed on a web page. It defines the properties and dimensions of an element's rectangular box, including its content, padding, borders, and margins.

The box model consists of the following components:

1. Content: This is the actual content of the element, such as text, images, or other HTML elements.

2. Padding: The padding is a transparent area surrounding the content, inside the element's border. It provides spacing between the content and the border.

3. Border: The border is a line that surrounds the padding and content of an element. It separates the element from its neighboring elements.

4. Margin: The margin is the transparent area outside the element's border. It provides spacing between the element and other elements on the page.

## Q.2 What are the Different Types of Selectors in CSS & what are the advantages of them?

**Ans:**

Types of CSS Selectors:

1. Element Selectors:
- Target elements based on their HTML tag names.
- Simple and widely applicable.

2. Class Selectors:
- Target elements based on the value of their `class` attribute.
- Reusable across multiple elements for consistent styling.

3. ID Selectors:
- Target elements based on the value of their `id` attribute.
- Provide high specificity and uniqueness.
- Useful for JavaScript interactions and specific styles.

4. Attribute Selectors:
- Target elements based on the presence or value of their attributes.
- Provide flexibility in targeting elements with specific attribute conditions.

5. Descendant Selectors:
- Target elements that are descendants of another element.
- Style elements based on their position within the HTML structure.

6. Pseudo-classes and Pseudo-elements:
- Target elements based on a specific state or position.
- Examples: `:hover`, `:first-child`, `::before`, `::after`.
- Provide additional styling options and effects.

Advantages of CSS Selectors:

- Flexibility: CSS selectors offer a wide range of options for targeting elements.

- Reusability: Class selectors promote code reusability and consistency.

- Specificity: ID selectors allow for targeting specific elements with high specificity.

- Control: Different selectors provide control over styling based on various conditions.

- Targeting: Attribute selectors enable precise targeting based on attribute values.

- Hierarchy: Descendant selectors allow styling based on element hierarchy.

- State and Position: Pseudo-classes and pseudo-elements provide additional styling options.

## Q.3 What is VW/VH?

Ans

In CSS, `vw` and `vh` are units of measurement that represent viewport width and viewport height, respectively. They are relative units that allow you to specify dimensions based on a percentage of the viewport size.

Here's what `vw` and `vh` mean:

1. VW (Viewport Width):

- 1vw represents 1% of the viewport width.
- For example, if the viewport width is 1000 pixels, 1vw would be equal to 10 pixels (1% of 1000 pixels).

2. VH (Viewport Height):
   - 1vh represents 1% of the viewport height.
   - For example, if the viewport height is 800 pixels, 1vh would be equal to 8 pixels (1% of 800 pixels).

The `vw` and `vh` units are useful for creating responsive designs that adapt to different screen sizes. By using these units, you can ensure that elements on your web page scale proportionally based on the viewport dimensions.

## Q.4 What difference between Inline, Inline Block and block ?

Ans:

In CSS, there are three display properties that affect how elements are rendered on a web page: inline, inline-block, and block. These display properties determine how elements flow and interact with other elements. Here's the difference between them:

1. Inline:

- Inline elements do not start on a new line and flow alongside adjacent elements.
- They do not have width and height properties and take up only as much space as their content requires.
- Examples of inline elements include `<span>`, `<a>`, `<strong>`, `<em>`, etc.
- You can apply properties like padding and margin to an inline element, but the top and bottom margins do not push other elements away.
- Inline elements do not respect the `width` and `height` properties, and they cannot have line breaks.

2. Inline-block:

- Inline-block elements share characteristics of both inline and block elements.
- They flow inline with other elements but can have width and height properties.
- They respect the `width` and `height` properties and can have line breaks.
- Unlike inline elements, inline-block elements allow vertical alignment and can have padding and margins applied to them.
- Examples of inline-block elements include `<img>`, `<button>`, and elements with `display: inline-block` explicitly set.

3. Block:

- Block elements start on a new line and occupy the full available width of their parent container by default.
- They have width and height properties, and their width expands to fill the available horizontal space.
- Block elements respect the `width` and `height` properties and can have line breaks.
- You can apply padding, margin, and other properties to block elements.
- Examples of block elements include `<div>`, `<p>`, `<h1>` to `<h6>`, `<ul>`, `<li>`, etc.

## Q.5 How is Border-box different from Content Box?

**Ans**:
The `box-sizing` property in CSS controls how the total width and height of an element are calculated, taking into account its content, padding, border, and margin. There are two values for the `box-sizing` property: `content-box` (the default) and `border-box`. Here's how they differ:

1. Content Box (Default):

- With the `box-sizing: content-box` value, the total width and height of an element are calculated by considering only the content area.
- This means that the width and height specified for an element do not include the padding, border, or margin. They apply only to the content area.
- If you add padding or border to an element with `box-sizing: content-box`, it will increase the overall size of the element beyond the specified width and height.

Example:

```
.element {

  box-sizing: content-box;
  width: 200px;
  padding: 20px;
  border: 2px solid black;
}
```

In the example above, the total width of the element will be calculated as `200px` (specified width) + `40px` (padding: 20px on each side) + `4px` (border: 2px on each side) = `244px`.

2. Border Box:

- With the `box-sizing: border-box` value, the total width and height of an element are calculated by including the content, padding, and border within the specified width and height.

- This means that the padding and border are drawn inward from the specified width and height, without increasing the overall size of the element.

- When you use `box-sizing: border-box`, the element's specified width and height encompass the content area, padding, and border.

Example:

```
.element {

  box-sizing: border-box;

  width: 200px;

  padding: 20px;

  border: 2px solid black;

}
```

In the example above, the total width of the element will be calculated as `200px` (specified width) without considering padding and border. The padding and border will be drawn inward from the specified width.

## Q.6 What’s z-index and How Does it Function?

Ans:

In CSS, the `z-index` property controls the stacking order of positioned elements along the z-axis, which determines the visibility and overlapping of elements on a web page. It specifies the depth or layering of an element relative to other elements in the stacking context.

Here's how `z-index` functions:

1. Stacking Context:

- Each positioned element with a `z-index` value other than `auto` creates a stacking context.
- The stacking context represents a specific layer in the z-axis and determines the order in which elements are rendered.

2. Integer Values:
- The `z-index` property accepts integer values, positive or negative, or the value `auto`.
- Higher `z-index` values indicate a higher stacking order within the stacking context.
- Elements with a higher `z-index` value are stacked on top of elements with lower `z-index` values.

3. Sibling Elements:
- The `z-index` property affects the stacking order of sibling elements within the same stacking context.
- Elements with a higher `z-index` value will appear in front of elements with lower `z-index` values.

4. Stacking Context Hierarchy:
- Stacking contexts can be nested, creating a hierarchy of stacking orders.
- Elements within a higher-level stacking context can have a higher stacking order than elements within a lower-level stacking context.

5. Default `z-index`:
- By default, positioned elements have a `z-index` value of `auto`.
- In this case, the stacking order depends on the order of elements in the HTML markup and their position in the document flow.

6. Non-positioned Elements:
- Non-positioned elements (elements with `position: static`) are not affected by the `z-index` property.
- The `z-index` property only applies to positioned elements (elements with `position: relative`, `position: absolute`, or `position: fixed`).

Here's an example of how `z-index` can be used:

```
.element1 {
  position: relative;
  z-index: 2;
}

.element2 {
  position: relative;
  z-index: 1;
}
```

In the example above, `element1` has a higher `z-index` value than `element2`. Therefore, `element1` will be stacked on top of `element2`, and it will appear visually above `element2`.

## Remaining question below

## Q.7 What's Grid & Flex and difference between them?

**Ans:**

CSS Grid and Flexbox are two powerful layout systems in CSS that provide different approaches for creating flexible and responsive web layouts.

Flexbox:

- Flexbox, short for Flexible Box Layout, is a one-dimensional layout system primarily designed for arranging elements in a single row or column.

- It works along the main axis (horizontal or vertical) and allows flexible resizing and alignment of elements.

- Flexbox is ideal for creating dynamic and vertically or horizontally centered layouts, such as navigation menus, card layouts, or vertically stacked elements.

- It provides properties like `flex-direction`, `justify-content`, and `align-items` for controlling the alignment, spacing, and order of flex items.

CSS Grid:

- CSS Grid Layout is a two-dimensional layout system designed for creating complex and grid-based layouts.

- It allows you to define rows and columns and place elements anywhere in the grid, both vertically and horizontally.

- CSS Grid is suitable for creating grid-based designs, like multi-column layouts, responsive grids, and asymmetrical layouts.

- It provides properties like `grid-template-rows`, `grid-template-columns`, and `grid-area` for defining the grid structure and placing items within the grid.

Key Differences:

1. Layout Model:

- Flexbox is a one-dimensional layout system, working along either a horizontal or vertical axis.

- CSS Grid is a two-dimensional layout system, creating rows and columns to form a grid.

2. Direction:

- Flexbox is useful for arranging elements in a linear format, either in a row or column.

- CSS Grid allows placement and control of elements in both horizontal and vertical dimensions, forming a grid.

3. Complexity:

- Flexbox is simpler to understand and implement for one-dimensional layouts.

- CSS Grid is more powerful and suited for complex and multi-dimensional layouts.

4. Alignment:

- Flexbox provides excellent support for alignment and distribution of elements within a container, both vertically and horizontally.

- CSS Grid offers more fine-grained control over the placement and alignment of elements in both rows and columns.

5. Browser Support:

- Flexbox has good browser support, including modern browsers and some older versions.

- CSS Grid has relatively newer browser support but is increasingly being adopted.

## Q.8 Difference between absolute and relative and sticky and fixed position explain with example.

Ans:

The differences between absolute, relative, sticky, and fixed positioning in CSS, along with examples:

1. Absolute Positioning:

- Elements with `position: absolute` are positioned relative to their closest positioned ancestor or the initial containing block if there is no positioned ancestor.
- The element is taken out of the normal document flow, and other elements on the page are unaffected by its position.
- The position of an absolutely positioned element can be defined using `top`, `right`, `bottom`, and `left` properties.

- Example:

```
.parent {
  position: relative;
}

.child {
  position: absolute;
  top: 20px;
  left: 30px;
}
```

In the example, the `.child` element will be positioned 20 pixels down from the top and 30 pixels from the left of its closest positioned ancestor (`.parent`).

2. Relative Positioning:

- Elements with `position: relative` are positioned relative to their normal position in the document flow.
- The element still occupies its original space in the flow, and other elements are not affected by its position.
- The position of a relatively positioned element can be adjusted using `top`, `right`, `bottom`, and `left` properties.

- Example:

```
.child {
  position: relative;
  top: 10px;
  left: 20px;
}
```

In the example, the `.child` element will be moved 10 pixels down from its normal position and 20 pixels from the left.

3. Sticky Positioning:

- Elements with `position: sticky` are positioned based on the user's scroll position.
- The element behaves like `position: relative` until the user scrolls to a specific threshold, then it becomes `position: fixed` until scrolling reaches another threshold.
- The sticky element remains in the document flow and affects the layout of other elements.

- Example:

```
.sticky-element {
  position: sticky;
  top: 0;
}
```

In the example, the `.sticky-element` will stick to the top of its parent container as the user scrolls, creating a sticky effect.

4. Fixed Positioning:

- Elements with `position: fixed` are positioned relative to the viewport, meaning they remain fixed in their position even when the page is scrolled.
- Fixed elements are taken out of the normal document flow and do not affect the layout of other elements.
- The position of a fixed element can be defined using `top`, `right`, `bottom`, and `left` properties.
- Example:

```
.fixed-element {
  position: fixed;
  top: 20px;
  right: 30px;
}
```

In the example, the `.fixed-element` will be positioned 20 pixels down from the top and 30 pixels from the right of the viewport.