CSS Positioning, Display Properties Flexbox and Grid

1. Positioning in CSS

CSS positioning determines how elements are placed on the page. Here are the main positioning values:

1.1 static (Default Positioning)

- The default behavior of elements.
- Elements flow naturally in the document.

```
{ position: static; }
```

1.2 relative

- The element is positioned relative to its normal position.
- Use top, right, bottom, and left to offset.

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

> Use relative positioning as a reference for positioning child elements.

1.3 absolute

- The element is positioned relative to the nearest positioned (non-static) ancestor.
- Removed from the normal document flow.

```
.parent {
  position: relative;
}
.child {
  position: absolute;
  top: 10px;
  left: 15px;
}
```

> If no ancestor is positioned, the element positions itself relative to the html (root element).

1.4 fixed

- The element is positioned relative to the viewport.
- Stays in place even during scrolling.

```
.fixed-header {
  position: fixed;
  top: 0;
  width: 100%;
  background-color: #333;
  color: white;
}
```

> Use fixed positioning for fixed headers, footers.

1.5 sticky

- A hybrid of relative and fixed.

- Behaves as relative until a specific scroll point is reached, then acts like fixed.

```
.sticky-nav {
  position: sticky;
  top: 0;
  background-color: yellow;
}
```

> Use sticky for slides like table headers or navigation bars.

1.6 z-index

- Controls the stacking order of elements.
- Higher values appear on top of lower values.

```
.header {
  position: fixed;
  top: 0;
  z-index: 1000;
}
.modal {
  position: fixed;
  top: 50%;
  left: 50%;
  z-index: 2000;
}
```

> Use z-index to control the stacking order of elements like modals, dropdowns.

2. Display Properties

The `display` property controls the box type of an element and its behavior in the document flow.

2.1 block

- The element takes up the full width of its parent container.
- Starts on a new line.

```
div { display: block; }
```

2.2 inline

- The element takes up only as much width as its content.
- Doesn’t start on a new line.

```
span { display: inline; }
```

2.3 inline-block

- Behaves like inline, but allows setting width and height.

```
.menu-item {
  display: inline-block;
  padding: 10px;
  background-color: #ddd;
}
```

2.4 none

- Hides the element entirely.

```
.hidden {
```

```
  display: none;
}
```

> - Combine inline-block for horizontal layouts without float.
> - Use none for toggling visibility dynamically via JavaScript.

3. Flexbox (Flexible Box Layout)

Flexbox is a powerful CSS layout model for distributing space and aligning items.

3.1 Setting Up Flexbox

```
.container {
  display: flex;
}
```

3.2 Key Flexbox Properties

```
.container {
  flex-direction: row;
  justify-content: center;
  align-items: center;
  flex-wrap: wrap;
  gap: 10px;
}
```

flex-direction

Controls the direction of the items in the container.

- row (default): Items are placed horizontally.
- column: Items are placed vertically.

```
.container {
  flex-direction: row; /* Horizontal */
}
```

justify-content

Aligns items horizontally in the container.

- flex-start (default), center, space-between, space-around, flex-end

```
.container {
  justify-content: center; /* Center items */
}
```

align-items

Aligns items vertically in the container.

- stretch (default), center, flex-start, flex-end

```
.container {
  align-items: center; /* Center items vertically */
}
```

flex-wrap

Allows items to wrap onto multiple lines.

```
.container {
  flex-wrap: wrap;
}
```

gap

Adds spacing between flex items.

```
.container {
  gap: 10px;
}
```

3.3 Child Flex Properties

```
.item {
  flex: 1;
  align-self: flex-end;
}
```

flex

Specifies the flex-grow, flex-shrink, and flex-basis values.

```
.item {
```

```
  flex: 1 1 150px;
}
```

Card 1 Card 2 Card 3

align-self

Overrides align-items for individual items.

```
.item {
  align-self: flex-end;
}
```

Practical Example

Item 1 Item 2 Item 3

> Flex Box Best Practices
> Fallbacks: Provide fallbacks for older browsers that don’t support Flexbox.
> Avoid Overuse: Use Flexbox for layout structure, not for small alignments where simpler CSS would suffice.
> Combine Flexbox and Grid: Use Flexbox for one-dimensional layouts and CSS Grid for two-dimensional layouts.
> Debugging Tools: Use browser developer tools to visualize flex containers and properties.

4. Grid Layout : (Two Dimensional Box Layout)

CSS Grid Layout is a two-dimensional layout system in CSS that allows you to design web layouts in rows and columns. It offers control over alignment, spacing, and positioning of items.

Grid Basics

1. Grid Container

- To use Grid, set the display property to grid or inline-grid on a container.

```
.container {
  display: grid;
}
```

2. Grid Properties

- Define columns and rows using the grid-template-columns and grid-template-rows properties.

```
.container {
  display: grid;
  grid-template-columns: 100px 200px 1fr;
  grid-template-rows: 50px auto 100px;
}
```

```
- 100px, 200px: Fixed sizes.
- 1fr: Fixed unit that divides remaining space.
- auto: Adjusts size based on content.

Item 1                  Item 2
```

```
Item 3
```

3. Grid Gap

- Control spacing between rows and columns.

```
.container {
  gap: 10px; /* Equal gap between rows and columns */
  row-gap: 15px; /* Row-specific spacing */
  column-gap: 20px; /* Column-specific spacing */
}
```

Spanning Items: grid-column and grid-row

- Position and stretch items across specific rows or columns.

```
.item {
  grid-column: 1 / 3; /* Span columns 1 to 3 */
  grid-row: 1 / 2; /* Span rows 1 to 2 */
}
```

```
Spanning 2 Columns          Item 2
Item 3
```

Grid Advanced

1. repeat() Function

- Repeat a pattern of columns or rows.

```
.container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-template-rows: repeat(2, 100px);
}
```

2. Auto Sizing Tracks

Automatically fit items into available space.

- auto-fit: Adjusts items to fill available space.
- auto-fill: Creates empty tracks if items don’t fill the space.

```
.container {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(100px, 1fr));
}
```

```
Item 1                    Item 2                    Item 3
```

3. Grid Areas

Define named grid areas for layout.

- Use grid-template-areas to define areas.
- Assign areas to items using grid-area.

```
.container {
  grid-template-areas:
    "header header"
    "sidebar content";
}
.item1 { grid-area: header; }
.item2 { grid-area: sidebar; }
.item3 { grid-area: content; }
```

```
Header

Sidebar            Content
```

4. Grid Auto Placement

Automatically place items in the grid.

- grid-auto-flow: Controls the direction of auto-placement.

```
.container {
  display: grid;
  grid-auto-flow: column;
}
```

```
Item 1        Item 2        Item 3
```

Aligning Items

Control alignment of items in the grid.

- justify-items: Aligns items horizontally.
- align-items: Aligns items vertically.

```
.container {
  display: grid;
  justify-items: center;
  align-items: center;
}
```

```
   Item 1        Item 2        Item 3
```

Practical Example

Header

Navigation Content

Footer

> Grid Best Practices
> Use minmax() for Responsive Design: Combines minimum and maximum sizes to create flexible grids. Example: grid-template-columns: repeat(3, minmax(100px, 1fr));
> Avoid Overlapping Items Accidentally: Use explicit grid definitions like grid-area to manage layout.
> Named Grid Areas for Readability: Simplify layout debugging by using grid-template-areas.
> Leverage auto-fit for Responsive Layouts: Automatically adjust item placement without fixed column counts.

> Experiment with DevTools: Use browser DevTools to visualize grid structure for debugging.