Front-end <Web /> Development

Lesson 5: CSS (Part 3)

University of Tehran

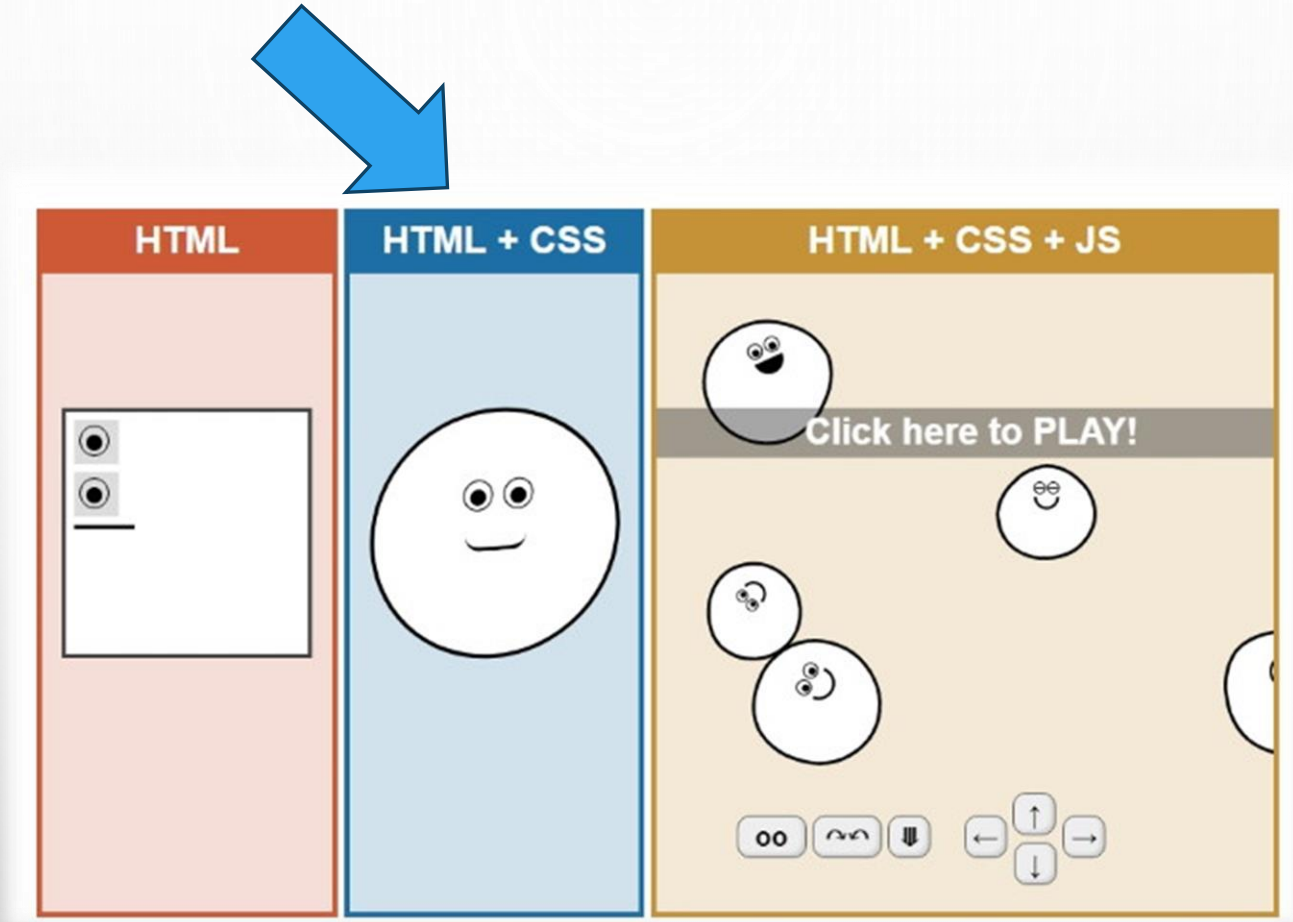
ACM
Summer of Code 2024

Lecturer:
Misagh Mohaghegh

A Short Review…

- Font types (sans-serif, serif, mono, cursive, fantasy)
- Font properties (font-family, font-size, font-style, font-weight, @font-face)
- rem and em units
- ch, vw, vh, and % units
- Pseudo-classes (:hover, inputs(:disabled, :invalid), links(:link, :visited))
- DOM pseudo-classes (:root, :first-child, :nth-child(n))
- Pseudo-elements (::first-letter, ::first-line, ::selection, ::placeholder, ::marker, ::before)
- Specificity (1-0-0, !important)
- Reset and Normalize
- Image object fit and filters

Continue…



Cascading Style Sheets

Position

Positioning Methods

There are 5 different **position** property values:

- static
- relative
- absolute
- fixed
- sticky

After the position property is set,
the four **top**, **right**, **bottom**, and **left** properties set the final location of the element.

The default position for elements is static:

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

Inset

inset is a shorthand property that sets the top, right, bottom, and left properties.
The syntax is similar to margin and padding.

```
#element {
    inset: 10px 0 0 20px; /* top: 10px, left: 20px */
    inset: 10px 0; /* top and bottom: 10px */
}
```

Static Position

This is the default positioning of all elements.
The top, right, bottom, and left properties have no effect in this mode.
The elements are positioned according to the normal flow of the page.

Relative Position

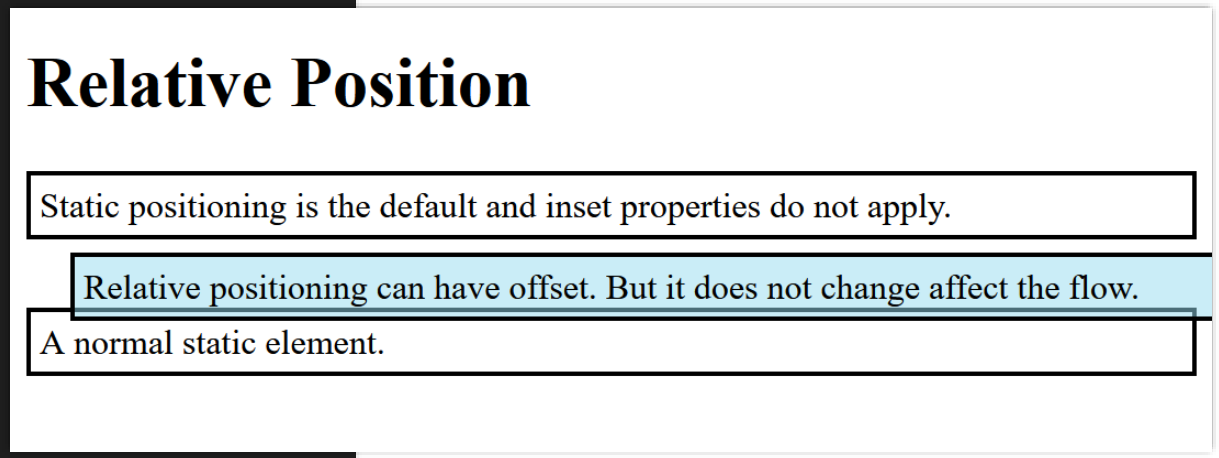
A *position: relative* element is positioned relative to its normal position.
This means that the element is first positioned normally,
then is adjusted by the top, right, bottom, and left offsets.

This moves the element relative to its normal static flow position, but it does not affect how other elements interact with it (gaps created by relatively moving are not filled by other elements).

```
.elem {
    position: relative;
    left: 10px;
}
```

Relative Position

```html
<head>
    <title>Relative Position</title>
    <style>
        div {
            border: 2px solid black;
            padding: 4px;
        }

        .relative {
            position: relative;
            top: 6px;
            left: 20px;
            background-color: rgba(150 220 240 / .5);
        }
    </style>
</head>

<body>
    <h1>Relative Position</h1>
    <div>Static positioning is the default and inset properties do not apply.</div>
    <div class="relative">
        Relative positioning can have offset.
        But it does not change affect the flow.
    </div>
    <div>A normal static element.</div>
</body>
```



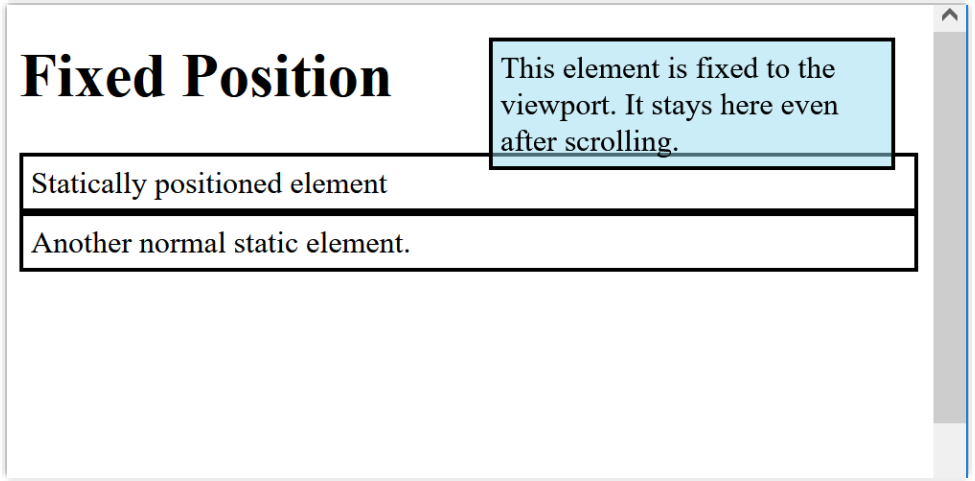
Fixed Position

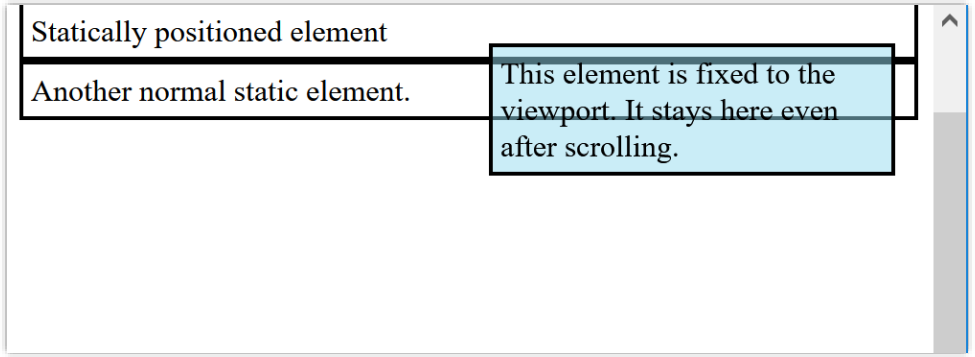
A *position: fixed* element is positioned relative to the viewport.
The element is removed from the normal flow of the page (no space is reserved for it).
The element is at the front of the viewport and is adjusted by the top, right, bottom, and left offsets.

```
.elem {
    position: fixed;
    top: 20px;
    left: 10px;
}
```

Fixed Position

```html
<head>
    <title>Fixed Position</title>
    <style>
        div {
            border: 2px solid black;
            padding: 4px;
        }

        .fixed {
            position: fixed;
            top: 20px;
            right: 20px;
            width: 200px;
            background-color: rgba(150 220 240 / .5);
        }

        body {
            height: 200vh;
        }
    </style>
</head>

<body>
    <h1>Fixed Position</h1>
    <div>Statically positioned element</div>
    <div class="fixed">
        This element is fixed to the viewport.
        It stays here even after scrolling.
    </div>
    <div>Another normal static element.</div>
</body>
```





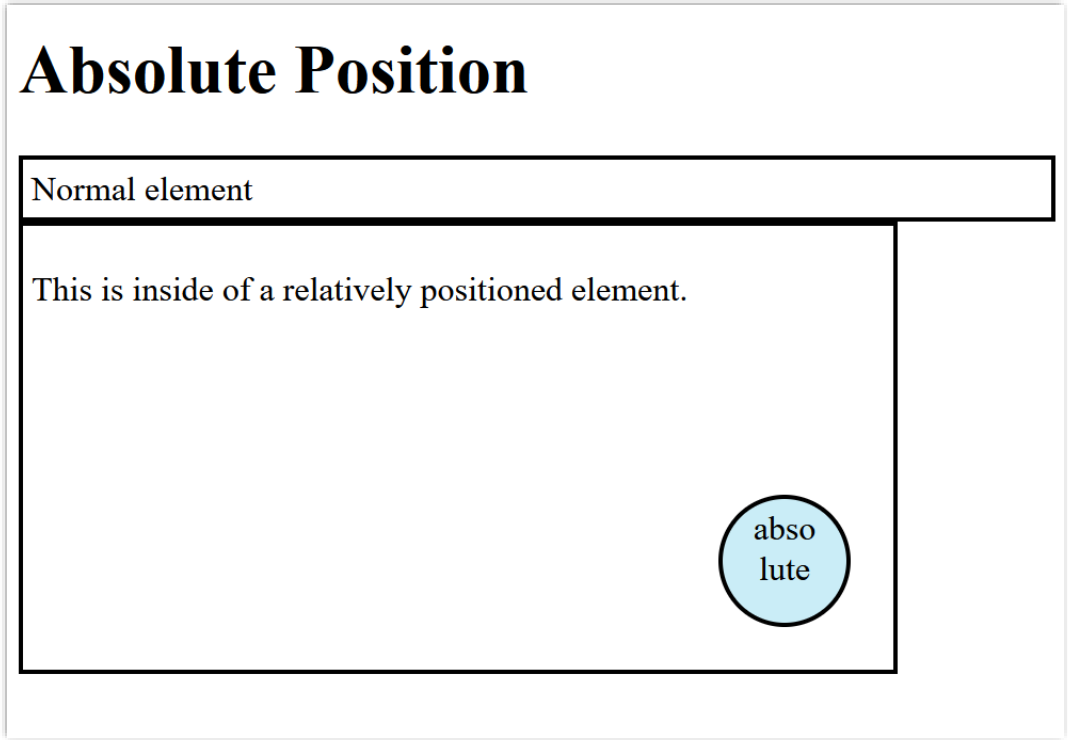
Absolute Position

A *position: absolute* element is positioned relative to the nearest positioned ancestor.
The element is removed from the normal flow of the page (no space is reserved for it).
If there is no positioned ancestor, it is like fixed positioning and relative to the viewport.

```
.parent {
    position: relative;
}

.elem {
    position: absolute;
    top: 20px;
    left: 10px;
}
```

Absolute Position

```html
<head>
    <title>Absolute Position</title>
    <style>
        div {
            border: 2px solid black;
            padding: 4px;
        }

        .relative {
            position: relative;
            width: 400px;
            height: 200px;
        }

        .absolute {
            position: absolute;
            bottom: 20px;
            right: 20px;
            width: 50px;
            height: 50px;
            border-radius: 50%;
            text-align: center;
            background-color: rgba(150 220 240 / .5);
        }
    </style>
</head>

<body>
    <h1>Absolute Position</h1>
    <div>Normal element</div>
    <div class="relative">
        <p>This is inside of a relatively positioned element.</p>
        <div class="absolute">abso<br>lute</div>
    </div>
</body>
```



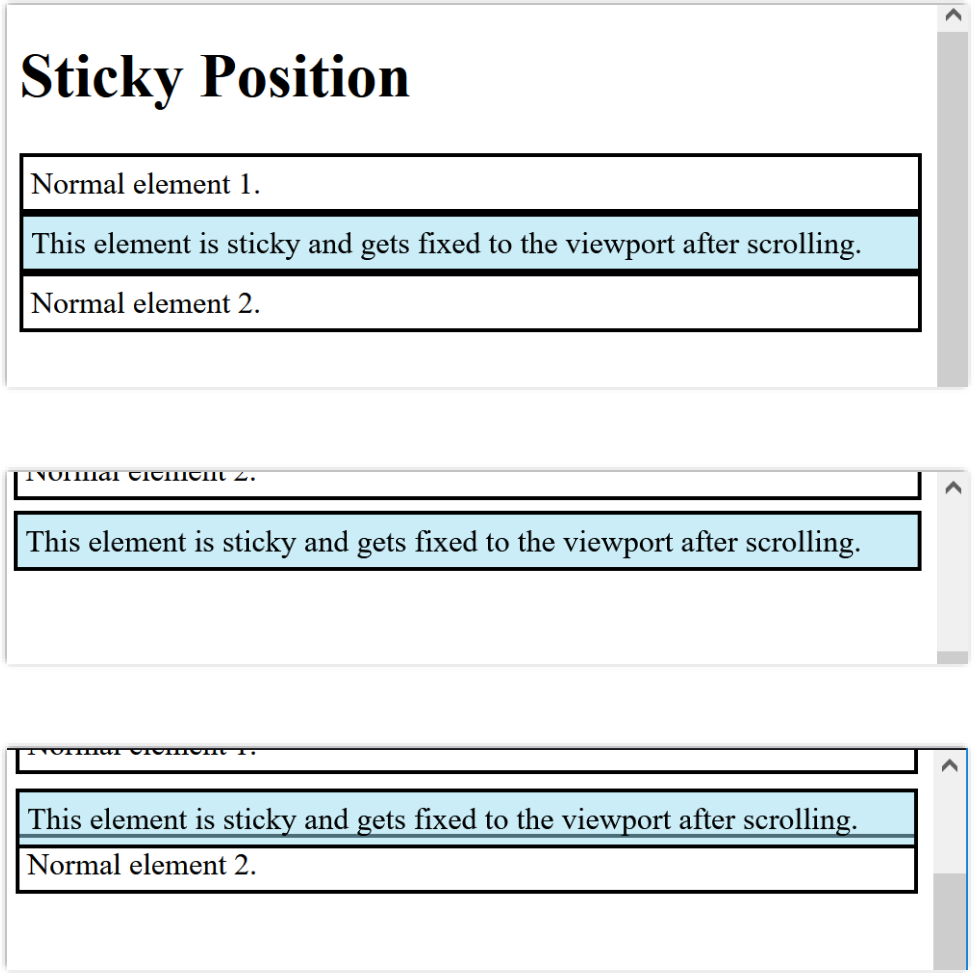
Sticky Position

A *position: sticky* element is positioned based on the scroll position.
The element is only removed from the normal flow if we scroll past it.
It is as if the element is *position: relative*, and becomes *position: fixed* after we scroll.

```
.elem {
    position: sticky;
    top: 20px;
    left: 10px;
}
```

Sticky Position

```html
<head>
    <title>Sticky Position</title>
    <style>
        div {
            border: 2px solid black;
            padding: 4px;
        }

        .sticky {
            position: sticky;
            top: 20px;
            background-color: rgba(150 220 240 / .5);
        }

        body {
            height: 200vh;
        }
    </style>
</head>

<body>
    <h1>Sticky Position</h1>
    <div>Normal element 1.</div>
    <div class="sticky">
        This element is sticky and gets fixed to the viewport after scrolling.
    </div>
    <div>Normal element 2.</div>
</body>
```



Flexbox

Layouts

Flexbox is a layout module.
Before flexbox, we had the choices of block and inline elements, tables for making 2D data, positions for explicit placement of elements, and float for placing block elements next to each other.

Flexbox makes creating flexible and responsive layout structures easy.

Flex Container

To use the flexbox model, we have to first define a flex container.

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

Now any **direct** children of an element which has *display: flex*, is a **flex item**.
By default, the flex items are aligned in a row next to each other.

Flex Container

```html
<head>
    <title>Flex</title>
    <style>
        .flex-container {
            display: flex;
            background-color: dodgerblue;
        }

        .flex-container>div {
            padding: 20px;
            margin: 10px;
            background-color: lightgrey;
        }
    </style>
</head>

<body>
    <div class="flex-container">
        <div>#1</div>
        <div>#2</div>
        <div>#3</div>
    </div>
</body>
```

Container Properties

The flexbox container (the parent div with *display: flex*) can have the following properties:

- flex-direction
- flex-wrap
- flex-flow
- justify-content
- align-items
- align-content
- gap

Flex Direction

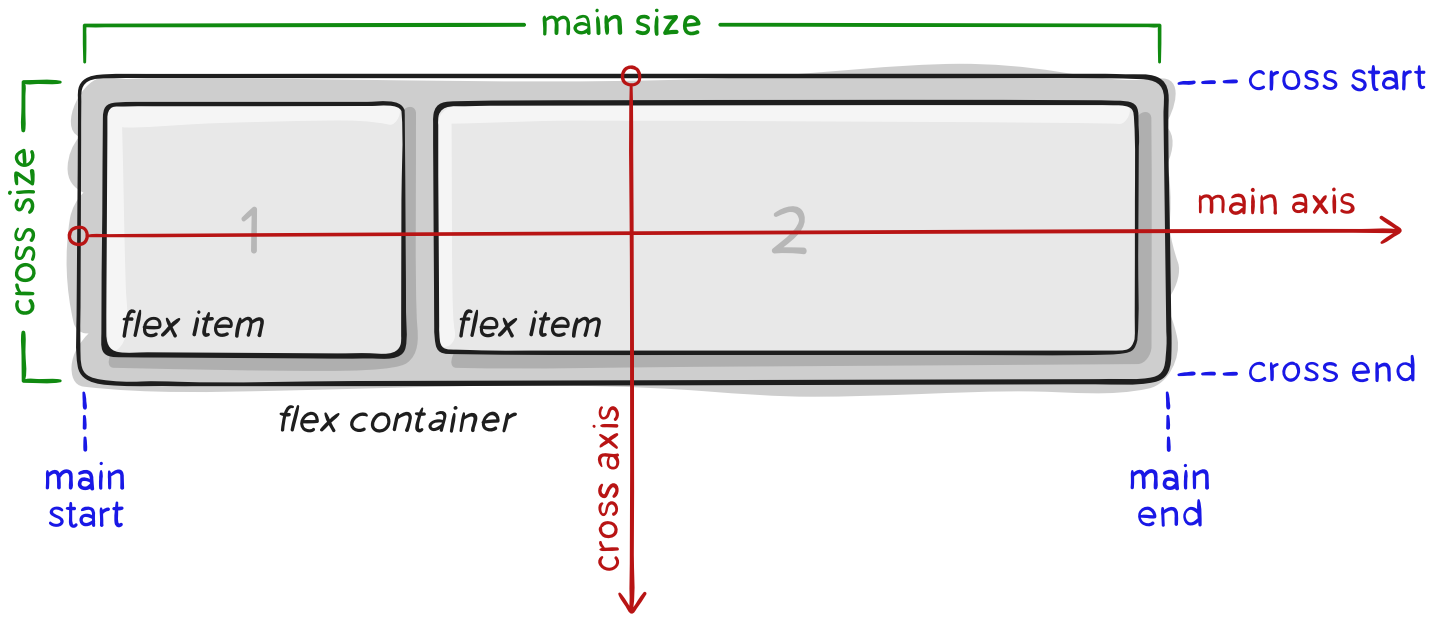
The flexbox direction is the direction that flex items are aligned.
This is row by default and can be changed to column using the **flex-direction** property:

```
.flex-container {
    display: flex;
    flex-direction: column;
}
```

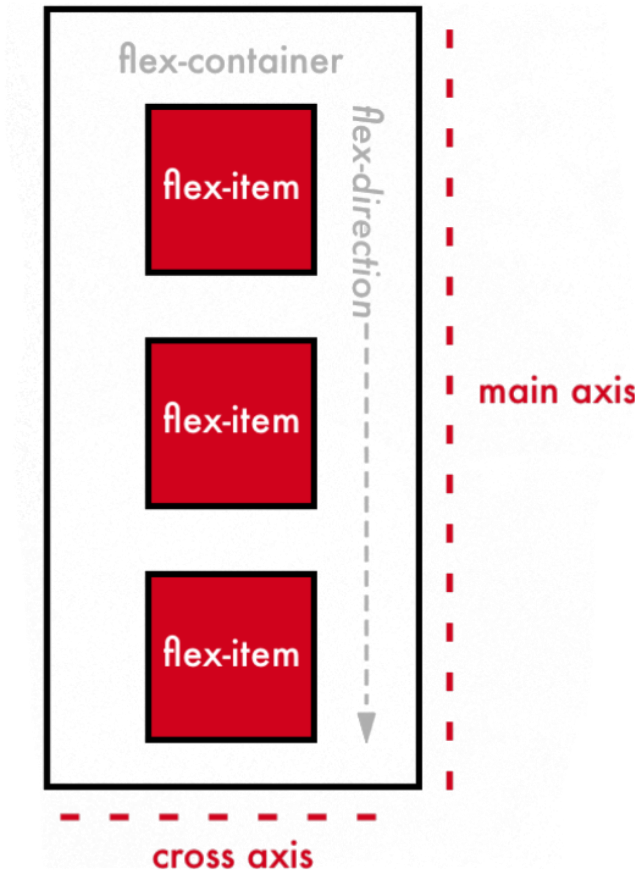
There is also row-reverse and column-reverse which stack the items in the opposite direction.
Row is left to right and column is top to bottom.

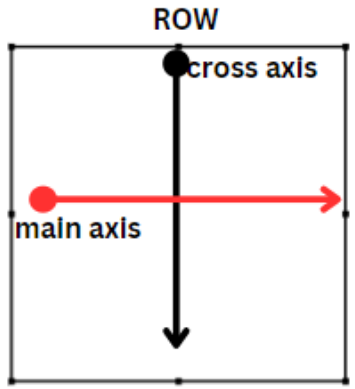
Flex Direction

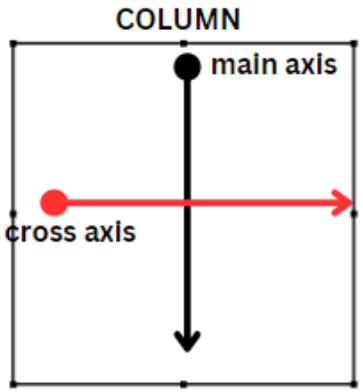
To better understand flexbox, we need to understand the flex flow directions (main and cross axis):

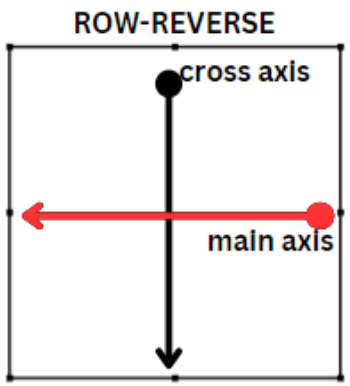


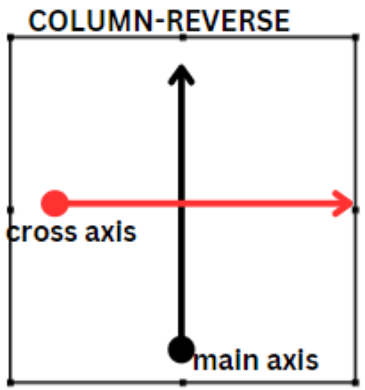
Flex Direction





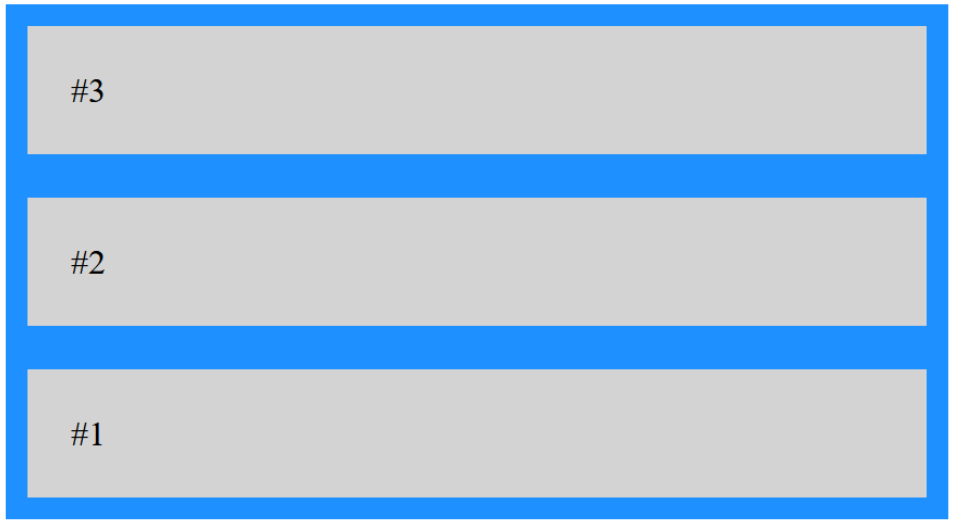






Flex Direction

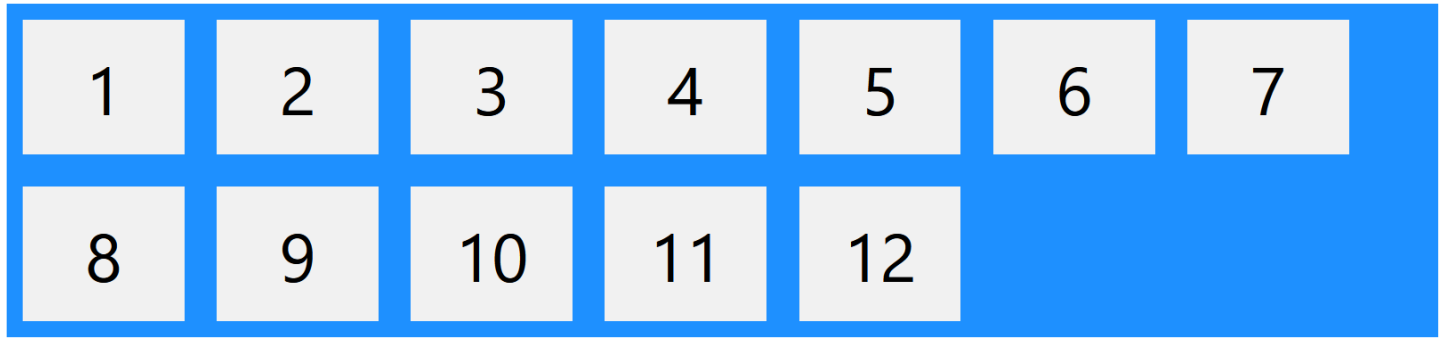
```
<head>
    <title>Flex</title>
    <style>
        .flex-container {
            display: flex;
            flex-direction: column-reverse;
            background-color: dodgerblue;
        }

        .flex-container>div {
            padding: 20px;
            margin: 10px;
            background-color: lightgrey;
        }
    </style>
</head>

<body>
    <div class="flex-container">
        <div>#1</div>
        <div>#2</div>
        <div>#3</div>
    </div>
</body>
```



Flex Wrap

flex-wrap allows wrapping of flex items into a new line if there are too many of them.
The default value is nowrap, and can become wrap or wrap-reverse.

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



Flex Flow

flex-flow is a shorthand property for flex-direction and flex-wrap:
`flex-flow: flex-direction flex-wrap`

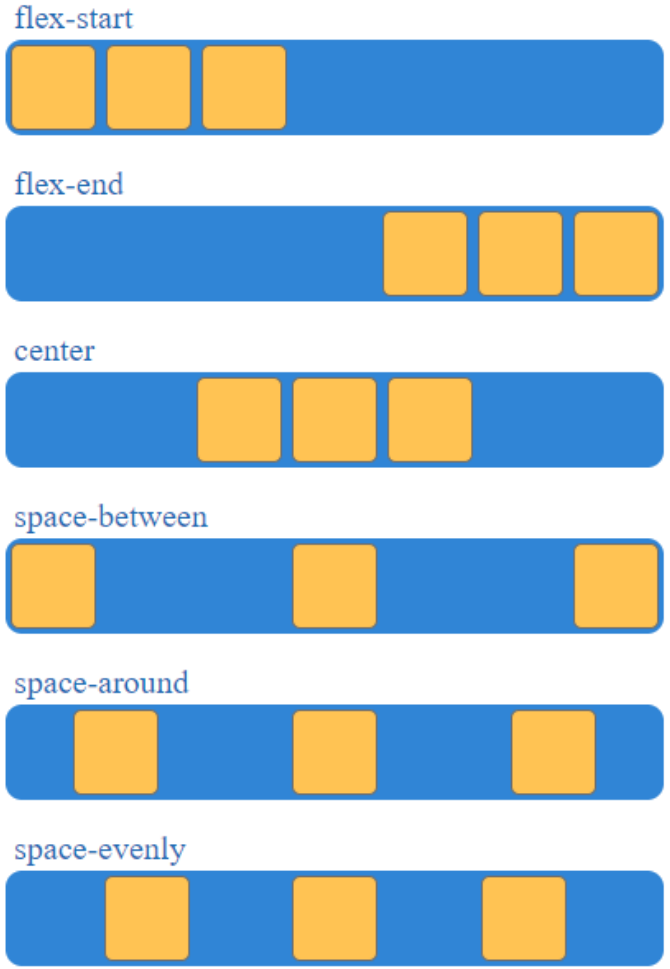
```
.flex-container {
    display: flex;
    flex-flow: row wrap;
}
```

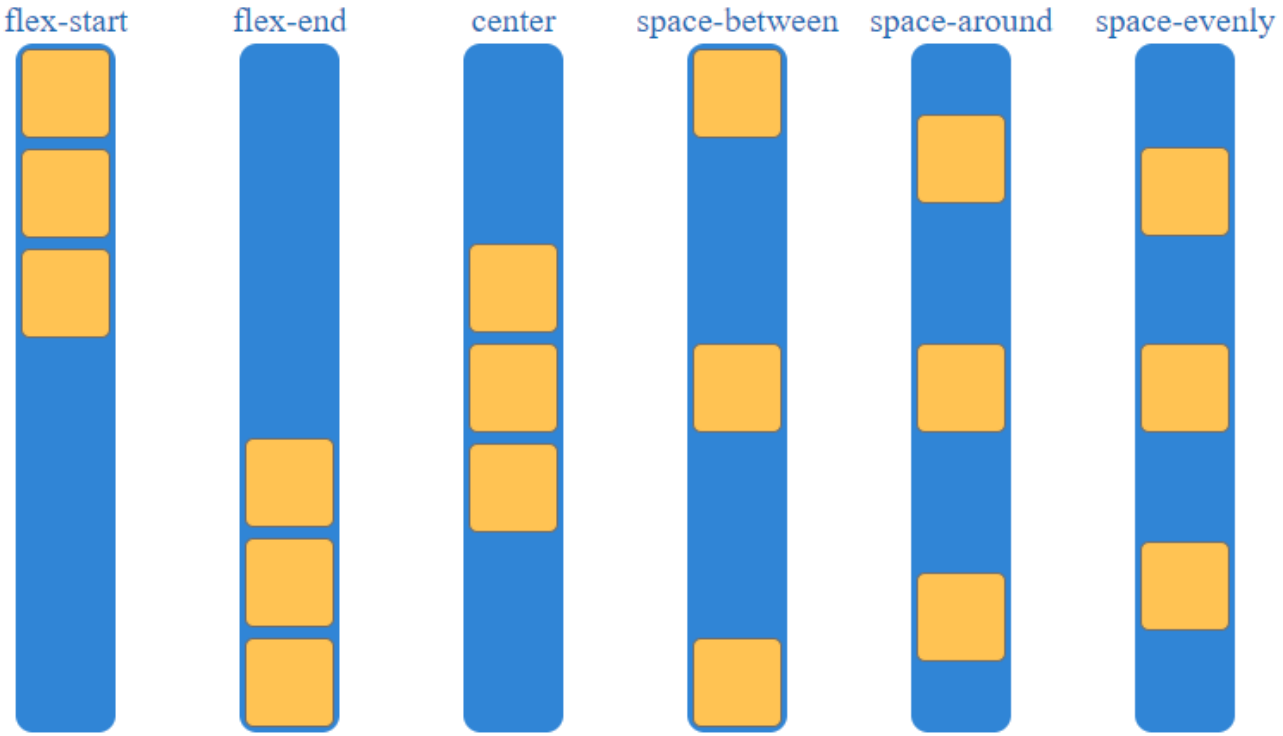
Justify Content

justify-content aligns the flex items on the main axis.
Accepted values are: center, flex-start (default), flex-end, space-around, space-between, space-evenly.

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

Justify Content



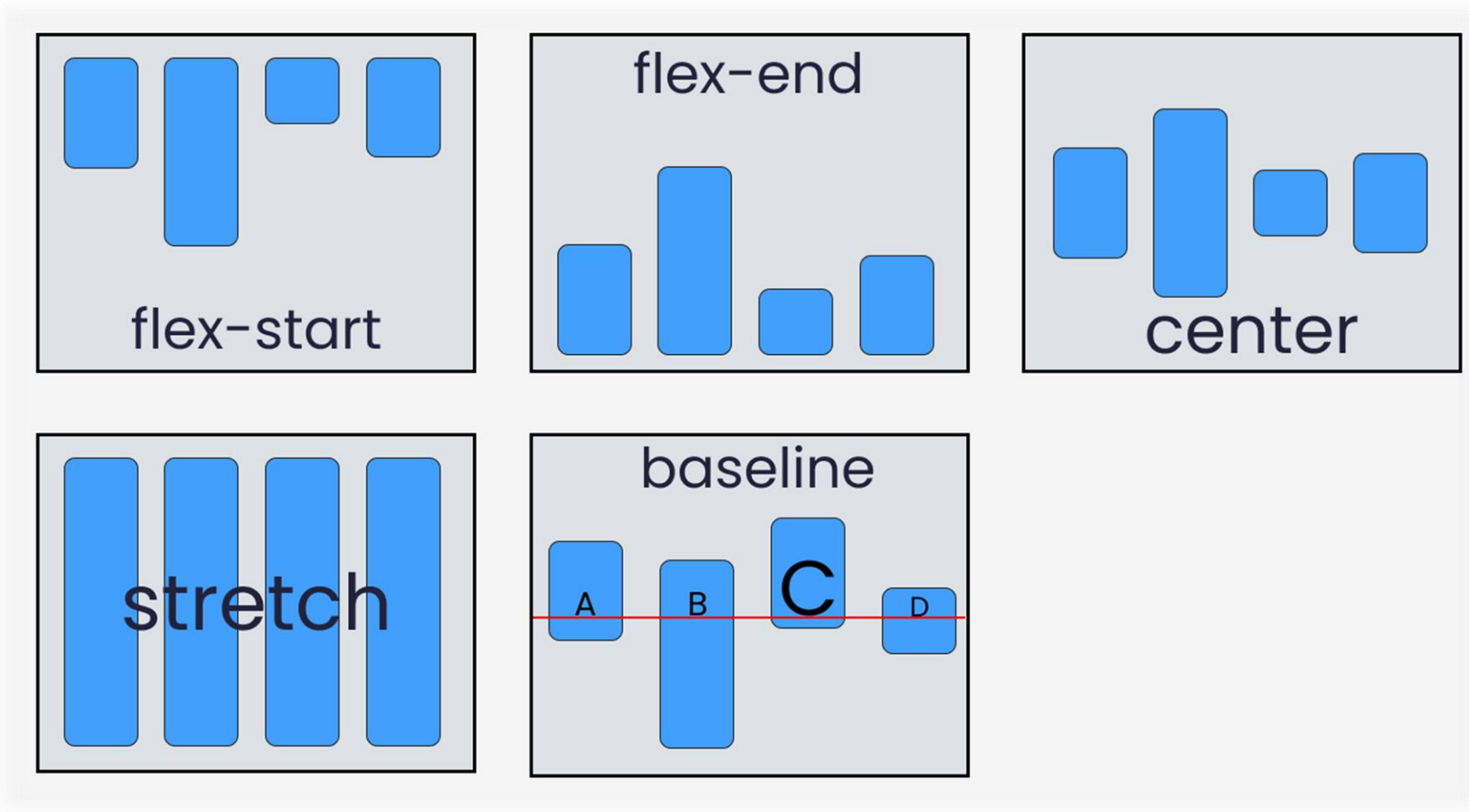


Align Items

align-items aligns the flex items on the cross axis.
Accepted values are: center, flex-start, flex-end, stretch (default), baseline.

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

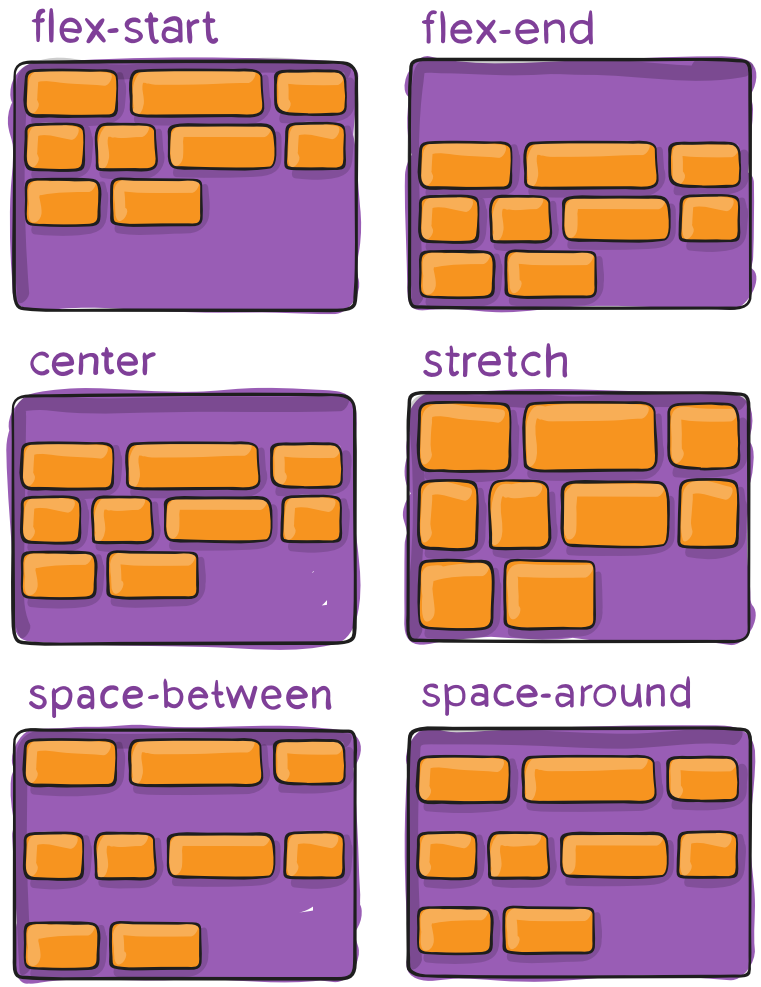
Align Items



Align Content

align-content aligns the flex lines when we have flex-wrap enabled.

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



Gap

gap is a shorthand property which sets the spacing between flex item columns and rows (if wrapped).
`gap: row-gap column-gap`

Entering only one value sets the gap for both the columns and rows.

```
.flex-container {
    display: flex;
    gap: 10px;
}
```

Flex Item Properties

The flexbox container had the following properties:

- flex-direction
- flex-wrap
- flex-flow
- justify-content
- align-items
- align-content
- gap

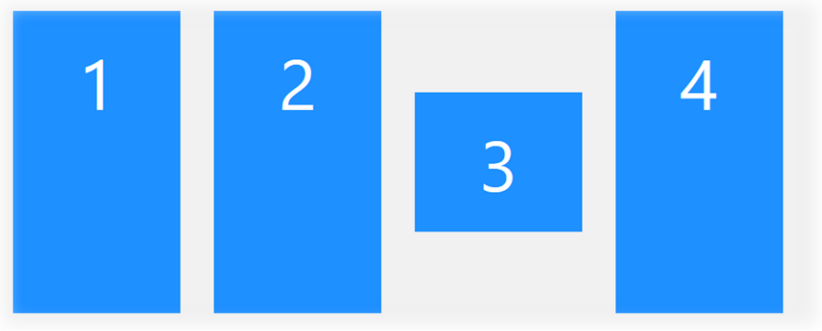
Flex items can also have some properties:

- align-self
- order
- flex-grow
- flex-shrink
- flex-basis
- flex

Align Self

The **align-self** property overrides the container **align-items** property for only one flex item.

```
.flex-container>div:nth-child(3) {
    align-self: center;
}
```



Order

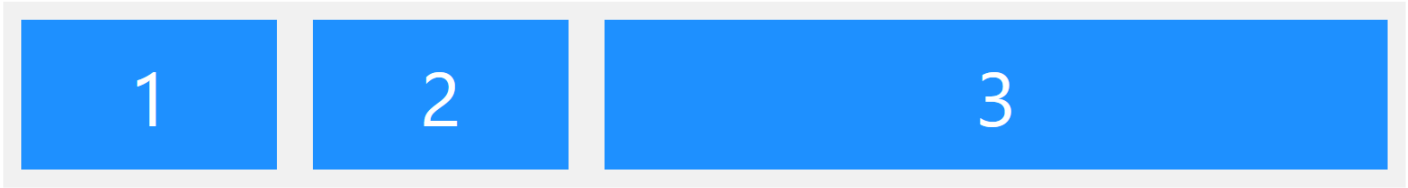
The **order** property changes the visual order of flex item appearances and ignores the HTML order.

```
div:nth-child(1) { order: 3; }
div:nth-child(2) { order: 2; }
div:nth-child(3) { order: 4; }
div:nth-child(4) { order: 1; }
```

Flex Grow

The **flex-grow** property specifies how much a flex item grows relative to the rest of the items.
The default is 0.

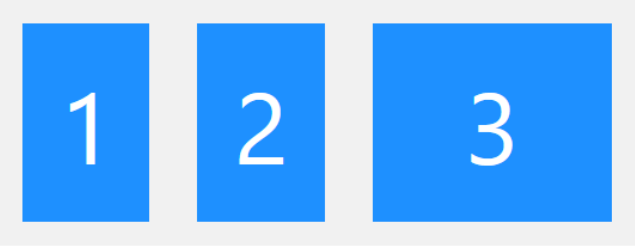
```
.flex-container>div:nth-child(3) {
    flex-grow: 1;
}
```



Flex Shrink

The **flex-shrink** property specifies how much a flex item will shrink relative to the rest of the items. The default is 1.

```
.flex-container>div:nth-child(3) {
    flex-shrink: 0;
}
```



Flex Basis

The **flex-basis** property specifies the initial length of a flex item.
The default is auto (uses the item width).
A value of 0 uses the minimum content width.

```
.flex-container>div:nth-child(3) {
    flex-basis: 200px;
}
```

Flex

flex is a shorthand property that sets flex-grow, flex-shrink, and flex-basis at the same time.
It is recommended to use the shorthand instead of the longhand properties.

```
.item {
    flex: 1; /* flex-grow: 1, flex-shrink: 1 */
    flex: 1 0; /* flex-grow: 1, flex-shrink: 0 */
    flex: 1 1 20px; /* flex-grow: 1, flex-shrink: 1, flex-basis: 20px */
}
```

Tips

To center an element inside of a container, we can use flex with centered justify-content and align-items:

```
.flex-center {
    display: flex;
    justify-content: center;
    align-items: center;
}
```

Tips

To make flex items of equal size, we can make all elements have flex-grow:

```
.flex-container>* {
    flex: 1;
}
```

Tips

To create a grid layout (fixed column count), we can use flex-basis.
First we would have flex-wrap so the items can go to the next line.
Now we use flex-basis to set how many items should be in each line.
For example, if we want 3 columns per line:

```
.flex-container>* {
    flex: 0 0 33.3333%; /* 100 divided by 3 */
}
```

Variables

Variables

Variables make code cleaner and easier to maintain.
For example, instead of copying the same color everywhere, we can define it in a variable.

Variables are defined inside selectors and can be global or local.
Global variables are defined inside the **:root** selector which is accessible everywhere.
Local variables are only accessible to the element's children.

Defining Variables

To define a variables, we make a property that starts with two dashes (--):

```
:root {
    --theme-blue: blue;
}
```

--theme-blue is a global variable.

Variable names are case sensitive and can be changed easily with JS and media queries.
Local scope variables are defined in the same manner.

Using Variables

To use a variable, the **var()** function is used:

```
:root {
    --theme-blue: blue;
}
header {
    background-color: var(--theme-blue);
}
```

The var function takes a variable name and returns the value set for that variable.
var can optionally take a second parameter which is the fallback value. It is used if the variable is not found.

```
header {
    background-color: var(--theme-blue, #0000ff);
}
```

Overriding Variables

Local variables can override global variables if they have the same name.

```
:root {
    --theme-blue: blue;
}
aside {
    --theme-blue: cyan;
}
aside article {
    background-color: var(--theme-blue);
}
```

Here, articles inside the aside element will use the cyan color instead of blue.

Transform & Transition

Transform

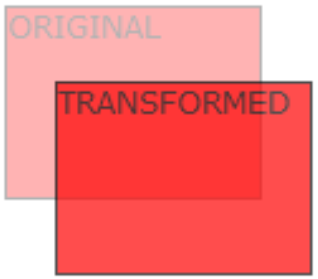
Transforms allow us to move, rotate, scale, and skew elements.
The **transform** property takes one or multiple transformation methods as the value.
Transform does not work on *display: inline* elements.

```
div {
    transform: method1(...) method2(...);
}
```

Transform Methods

Translate(X, Y): moves and offsets the element from its current position.
Rotate(degrees): rotates the element clockwise (positive degrees) or counter-clockwise.
Scale(X, Y): scales the element and makes it bigger or smaller. (There is also scaleX and scaleY)
Skew(X, Y): skews the element.

```
div {
    transform: rotate(20deg) scaleX(1.2);
}
```



translate

rotate

scale

skew

Transition

Transitions allow us to change a property value smoothly over time.
For example, we can change the color of an element smoothly by only stating the starting and ending color.

transition is a shorthand property for:
`transition: transition-property transition-duration transition-timing-function transition-delay`

We first specify the **transition-property**, which states which property change are we looking for.
For example, it can be **background-color** or **width** or **all** to set transition for everything.

The **transition-duration** sets how long it should take to smoothly move to the new value of the property.

```
button {
    transition: all 0.2s;
}
```

Transition

```html
<head>
    <title>Button Transition</title>
    <style>
        button {
            color: white;
            background-color: hsl(240 80% 60%);
            border: none;
            border-radius: 8px;
            padding: 20px;
            cursor: pointer;
            width: 120px;
            transition: all 0.2s;
        }

        button:hover {
            background-color: hsl(120 80% 40%);
            transform-origin: top left;
            transform: rotate(2deg);
            width: 200px;
        }
    </style>
</head>

<body>
    <button>Click Here!</button>
</body>
```

Thank you for your attention.