
CSS Class Notes
CSS Class notes for the 1st Semester
Are you ready to well designed UI with your HTML skills? Press space on your keyboard →

Table of contents

1. Getting Started with CSS?
2. Previous Class Recording
3. Selectors
4. Box Model
5. Using Block and Inline Axes in CSS
6. CSS Reset and Normalize
7. Inheritance
8. Colors/Units/Gradients
9. Debugging in browser
10. Inline, Internal and External CSS
11. FlexBox
12. Grid Layout
13. SubGrid
14. Positioned Layout
15. Stacking Context/Z-index
16. Overflow
17. Assignments
18. Important Links

Getting Started with CSS?

CSS which stands in for Cascading Style Sheets is a stylesheet language used to describe the presentation of a document written in HTML or XML.

Just as HTML serves as the skeletal part of the web, CSS describes how the element should be rendered on the web.
We use CSS to style our HTML elements and this is what you're going to learn throughout this module.

Previous Class Recording

- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22

Selectors

Before we move deeply into Selectors, let's dive into CSS rule which is a block of code, that has one or more selectors and one or more declarations.

Definition of selectors

Looking at the image in the previous slide we'll notice that CSS selector is the first part of a CSS rule. In order to choose or select HTML elements that's going to carry the CSS property values inside the rule we have to use CSS Selector. In summary, for us to add a style for a particular HTML element we need a selector.

Types of selectors

- Universal selector: This is also know as a wildcard, selects every single element in the document. It is represented by the asterisk character `*`

Code Example:

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
```

This rule is saying that remove any default margin and padding from all the elements in this document and also change the box-sizing value to border-box.

- Type selector: The CSS type selector matches elements by node/HTML name.

Code Example:

```
<p>I am taking color red and increasing my font size.</p>
```

```
p {
  color: red;
  font-size: 36px;
}
```

This CSS rule is saying that apply color of red to every `p` element and also increase its font size to 36px.

- Class selector: There is a class attribute associated to all HTML elements, this allows us to target a specific HTML element for its class name. To style an element using the class name we make use of the dot notation `.` before the class name when writing our selector in the CSS rule `.paragraph`

Code Example:

```
<p class="paragraph">You can style me using my class name.</p>
```

```
.paragraph {
  color: red;
  font-size: 36px;
}
```

This CSS rule is saying that apply color of red to the `p` element that has the class name of paragraph and also increase its font size to 36px.

- ID selector: The id selector uses the id attribute of an HTML element to select a specific element. Id value of an element must be unique which means you can only have a specific id value to an HTML element, unlike class where you can give 10 HTML elements same class name.

  To style an element using the id value we make use of the hash notation `#` before the id value when writing our selector in the CSS rule `#container-wrapper`

Code Example:

```
<span id="container-wrapper">
  You can style me using my id value which is container-wrapper.
</span>
```

```
#container-wrapper {
  color: red;
  font-size: 36px;
}
```

- Attribute selector: This gives you the power to select elements based on the presence of a certain HTML attribute or the value of an HTML attribute. To write the CSS rule for this you have to wrap the selector with square brackets.

Code Example:

```
<a href="https://altschoolafrica.com">
  You can style me using my attribute which is href.
</a>
```

```
[href] {
  color: red;
}
```

Code Example 2:

```
<a href="https://altschoolafrica.com">
  You can style me using my attribute and its value which is
  href="https://altschoolafrica.com".
</a>
```

```
[href="https://altschoolafrica.com"]{
  color: red;
  font-size: 36px;
}
```

Note: This method give you the access to style any element that has an attribute of data-type but with a specific value of href.

- `Pseudo-classes` : Pseudo-classes are keywords added to selectors using a single colon sign `:` just to specify a special state of the selected elements. They allow you to style elements based on their state, position, or user interactions, which cannot be targeted by regular CSS selectors alone. Here are some common pseudo-classes:

```
:link
:visited
:hover
:active
:focus
:nth-child()
```

Code Example

```
button:hover {
  background-color: orange;
}

li:nth-child(even) {
  text-transform: uppercase;
}

input:focus {
```

- `Pseudo-element` : To style specific parts of an element we attached double colon to our selector `::` followed by keywords to select the portion we want to apply styling to. Unlike the pseudo-classes, which target the entire element, pseudo-elements target specific parts of an element using a conventional keywords.

Here are some common pseudo-elements:

```
::before - Inserts content before the content of an element.
::after - Inserts content after the content of an element.
::first-letter - Styles the first letter of an element.
::first-line - Styles the first line of an element.
::selection - Styles the portion of an element that is selected by the user.
```

Note: Pseudo-elements are particularly useful for enhancing the design and readability of web content without the need for additional HTML elements.

Complex selectors

To have more power in accessing elements in the DOM we have some selectors which we will brief through but let's quickly look at parents and child elements using this code below:

```
<p>
  AltSchool Africa is a tech school that offers varieties of tech courses like
  <span>Frontend engineering</span>, <span>Backend engineering</span> and newly
  added <span>Cybersecurity</span> online.
</p>
```

In the code above, the parent element is the `p`, inside which we have 3 span elements, since all these 3 span elements are inside the `p` we call them the child elements of `p`.

- Descendant Selector: This selects all elements that are descendants and we achieve this by giving space `( )` to instruct the browser to look for child elements.

Code Example:

```
p span {
  color: red;
}
```

- Child selector (parent > child): This selects all elements that are direct children of a specified element.

Code Example:

```
ul > li {
  list-style: none;
}
```

- Adjacent Sibling Selector (prev + next): This selects an element that is immediately preceded by a specified element.

Code Example:

```
h1 + p {
  margin-top: 0;
}
```

- General Sibling Selector (prev ~ siblings): This selects all elements that are siblings of a specified element.

Code Example:

```
h1 ~ p {
  color: blue;
}
```

- Grouping Selector: Applies the same styles to multiple selectors.

Code Example:

```
h1,
h2,
h3 {
  margin-bottom: 10px;
}
```

- `Nesting Selectors &` : This is a way of writing CSS rules that are more specific and easier to read.They explicitly states the relationship between parent and child rules when using CSS nesting. It makes the nested child rule selectors relative to the parent element. Without the & nesting selector, the child rule selector selects child elements. The child rule selectors have the same specificity weight as if they were within `:is()` . Can be use with the Child Combinators.

Code Example:

```
<div class="container">
  <h1 class="title">Hello, CSS</h1>
</div>
```

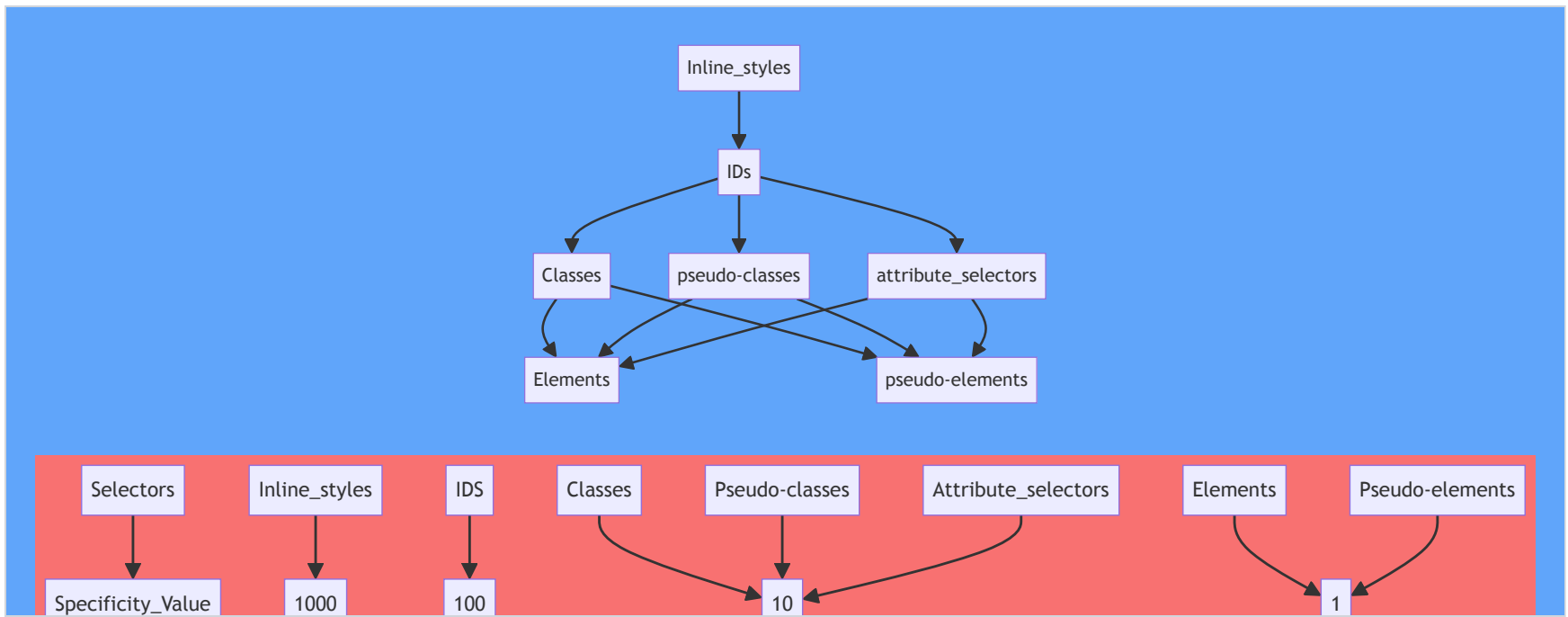
```
.container {
  padding: 20px;
  .title {
    color: red;
  }
  &:hover {
    background-color: lightblue;
  }
}
```

Read more about CSS nesting

Specificity Hierarchy

CSS selectors are of different forms and each of them has its place in the specificity hierarchy.

CSS Selectors decrease in specificity from top to bottom, meaning the selector at the top of the hierarchy has the highest specificity.



!impotant rule

In CSS, there is one rule that has the highest specificity score of 10,000. This rule is used to give a property-value pair the highest priority, allowing it to override any other declarations.

The only way to override inline styles which has specificity value of 1000 is by using this rule called !important, though this is considered as a bad practice and should be avoided. Read more

Code Example

```
selector {
  property: value !important;
}

.h1 {
  color: red !important;
}
```

Box Model

The CSS Box Model is a core concept in web design and layout. It describes how every element on a web page is rendered as a rectangular box. It's basically a box that wraps around every HTML element. Understanding this model is crucial for creating precise layouts and solving common design challenges.

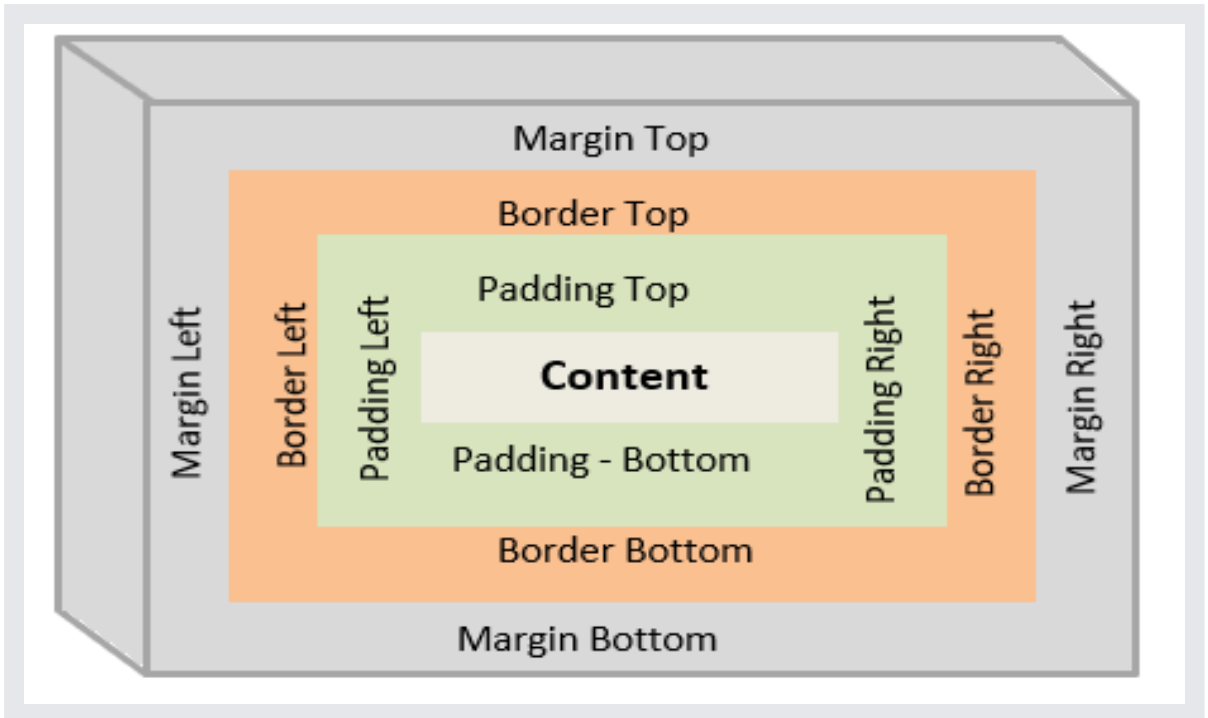
The four main areas of the box model: content box, padding box, border box and margin box.

Components Of Box Model

- a) Content:
  - This is the innermost layer.
  - It contains the actual content of the element (text, images, etc.).
  - Dimensions are set using 'width' and 'height' properties.
- b) Padding:
  - Surrounds the content area.
  - Creates space between the content and the border.
  - Can be set using 'padding' property (or padding-top, padding-right, etc.).
  - Is transparent, allowing the background of the element to show through.

- c) Border:
  - Encircles the padding (or content if no padding is set).
  - Can have different styles, colors, and widths.
  - Set using the 'border' property or individual properties like 'border-width'.
- d) Margin:
  - The outermost layer.
  - Creates space between the element and adjacent elements.
  - Is always transparent.
  - Set using the 'margin' property or individual properties (margin-top, etc.).

  Popular margin concepts are: Hungry margin(auto margin which only works for horizontal margins with explicit width), Collapsed margin, Negative margin.


Margin Top
Border Top
Padding Top
Margin Left
Border Left
Padding Left
Content
Padding - Bottom
Padding Right
Border Right
Margin Right
Border Bottom
Margin Bottom

- 1. Calculating Total Element Size

One of the most important aspects of the Box Model is understanding how the total size of an element is calculated:

- Total Width = width + left padding + right padding + left border + right border
- Total Height = height + top padding + bottom padding + top border + bottom border

Note: Margins are not included in these calculations as they affect spacing between elements, not the element's size itself.

- 2. Box-Sizing Property

The default box model can sometimes lead to unexpected results. CSS3 introduced the 'box-sizing' property to address this:

'content-box' (default): Width and height apply to content area only. 'border-box': Width and height include content, padding, and border.

```
* {
  box-sizing: border-box;
}
```

Example

```
div {
  box-sizing: border-box;
  width: 300px;
  padding: 20px;
  border: 10px solid black;
  margin: 25px;
}
```

Understanding the Box Model is crucial for:

- Centering elements
- Creating consistent spacing
- Implementing responsive designs
- Debugging layout issues

Using Block and Inline Axes in CSS

In CSS, the block and inline axes are used to determine how elements are laid out on a page. Understanding these axes is crucial for creating responsive layouts and designing web pages.

- Block Axis: The block axis is the vertical axis that runs from top to bottom. Block-level elements stack on top of each other in the block axis.
- Inline Axis: The inline axis is the horizontal axis that runs from left to right. Inline-level elements flow in the inline axis.

Possible css properties with block and inline axes are: `padding-block`, `margin-block`, `border-block`, `padding-inline`, `margin-inline`, `border-inline`, `block-size`, `inline-size`, `min-block-size`, `max-block-size`, `min-inline-size`, `max-inline-size`. Padding and margin can have the start, end variant like `padding-inline-start`, `padding-inline-end`, `margin-block-start`, `margin-block-end`.

The `block-size` and `inline-size` properties are used to set the width and height of an element, respectively. The `min-block-size` and `max-block-size` properties set the minimum and maximum width of an element, while the `min-inline-size` and `max-inline-size` properties set the minimum and maximum height of an element.

Calculating Width and Height

Width Calculation

- Width is calculated along the inline axis(left to right) and default is auto(using the content inside the element) but can be set using the `width` property. The `max-width` and `min-width` properties set the maximum and minimum width of an element. `max-width` is used to prevent an element from exceeding a certain width, while `min-width` ensures that an element is at least a certain width. `max-width` is useful for creating responsive designs that adapt to different screen sizes and should be used for containers/wrappers that need to be flexible. `min-width` is useful for ensuring that an element is at least a certain width, which can be helpful for maintaining the layout of a page and preventing elements from becoming too narrow. Width considers the parent element's width and the content inside the element. Avoid using fixed width and percentage width if you do not have a parent with an explicit width.

Height Calculation

- Height is calculated along the block axis(top to bottom) and default is auto(using the content inside the element) but can be set using the `height` property. The `max-height` and `min-height` properties set the maximum and minimum height of an element. `max-height` is used to prevent an element from exceeding a certain height, while `min-height` ensures that an element is at least a certain height. `min-height` is useful for creating responsive designs that adapt to different screen sizes and adjustable height. `max-height` is useful for ensuring that an element is at least a certain height, which can be helpful for maintaining the layout of a page and preventing elements from becoming too short. Height considers the content inside the element first before the parent element's height.

The `min-content` keyword will make a track as small as it can be without the track content overflowing. Changing the example grid layout to have three column tracks all at `min-content` size will mean they become as narrow as the longest word in the track.

The `max-content` keyword has the opposite effect. The track will become as wide enough for all of the content to display in one long unbroken string. This might cause overflows as the string will not wrap.

The `fit-content()` function acts like `max-content` at first. However, once the track reaches the size that you pass into the function, the content starts to wrap. So `fit-content(10em)` will create a track that is less than 10em, if the `max-content` size is less than 10em, but never larger than 10em.

CSS Reset and Normalize

CSS Reset

A CSS reset is a set of CSS rules that reset the styling of all HTML elements to a consistent baseline. This ensures that all browsers start with the same default styles, making it easier to create a consistent design across different browsers.

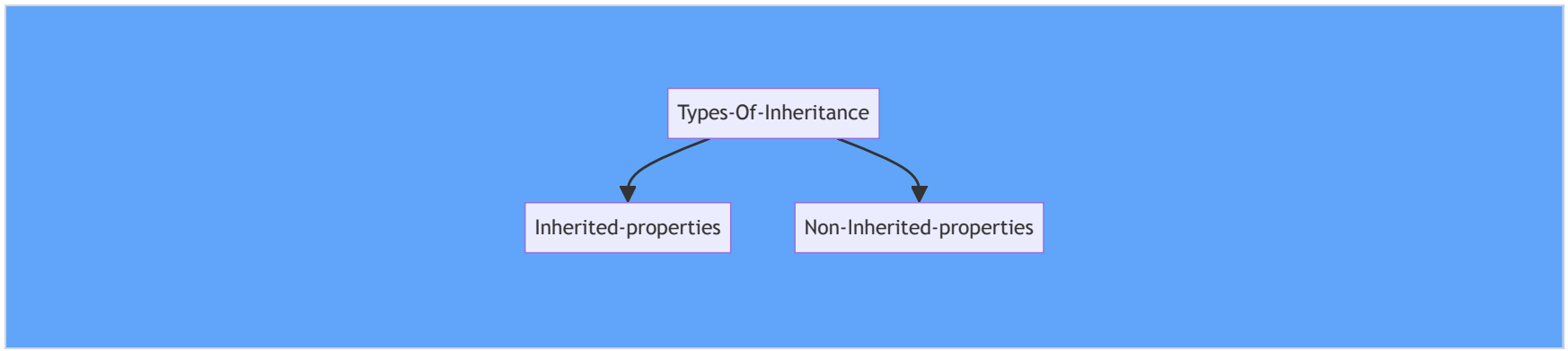
Link to Josh Comeau CSS Reset

Normalize.css

Normalize.css is a modern, HTML5-ready alternative to CSS resets. It makes browsers render all elements more consistently and in line with modern standards. It precisely targets only the styles that need normalizing.

Inheritance

Inheritance, this is when a child element get a computed value which represents its parent's value. Inheritance cascade downwards and every property has a default value in CSS.



- Inherited-properties: These are properties that by default passed down from a parent element to its children.
- Non-Inherited-properties: These are properties that by default can’t be passed down from a parent element to its children.

Some inherited and non-inherited CSS properties:

Inherited Properties	Non-Inherited Properties
list-style	border
color	margin
cursor	padding
font-family	width
font-size	height
font-style	position
font-weight	box-shadow

Inherited property

Code Example: The color property falls under the inherited properties, so the `em` element will inherit the color value from the parent element which is `p`

```
<p>This paragraph has <em>emphasized text</em>in it.</p>
```

```
p {
  color: green;
  font-weight-500;
}
```

This paragraph has *emphasized text* in it.

Non-inherited property

Code Example: The border property falls under the non-inherited properties so, the `em` element will not inherit the border value from the parent element which is `p`.

```
<p>This paragraph has <em>emphasized text</em>in it.</p>
```

```
p {
  border: 3px solid red;
}
```

This paragraph has *emphasized text* in it.

Setting inheritance explicitly in CSS

Using the `inherit` keyword

To keep everything under the developer's control, we have the inherit keyword that can make any property inherit its parent's computed value.

Code Example:

```
<p>This paragraph has <em>emphasized text</em>in it.</p>
```

```
p {
  border: 3px solid red;
}
em {
  border: inherit; //using the inherit keyword to make the em tag inherit the border style from its parent.
}
```

Controlling Inheritance

Note: Inheritance is always from the parent element in the document tree, even when the parent element is not the containing block.

There are 5 major keywords in inheritance:

inherit:
The inherit keyword causes element to take the computed value of the property from its parent element.

initial:
This keyword sets a property back to that initial, default value.

unset:
This keyword resets a property to its inherited value if the property naturally inherits from its parent, and to its initial value if not. This is like shuffling between the inherit and the initial keyword because in its first case it behaves like the inherit keyword when the property is an inherited property and like the initial keyword in the second case when the property is a non-inherited property.


HTML
CSS
JS
Result
EDIT ON
CODEPEN
Skip to
main
<!-- fragement identifier -->
<nav aria-label="Breadcrumbs">
<!-- role="list" was added back in because some
CSS display property values remove the semantics
from some elements. -->
<ol id="top" role="list">
Home
Living Room
<a aria-current="page"
href="/living/couch">Couches
<a
href="/living/couch/sectional">Sectional

</nav>
<main id="main">
Resources
1×
0.5×
0.25×
Rerun

The `all` CSS property

This shorthand resets all properties (except unicode-bidi and direction) of an element to their initial, inherited, or unset state. This property can be particularly useful when you want to ensure that an element does not inherit any styles from its parents or previous rules and instead starts with a clean slate.

```
<div class="parent">
  Parent Text
  <div class="child-inherit">Child Text with all: inherit</div>
</div>
```

```
.parent {
  color: red;
  font-size: 10px;
  background-color: lightgray;
}

.child-inherit {
  all: inherit;
}
```

Parent Text
Child Text with all: inherit

Colors/Units/Gradients

CSS Color

Colors in CSS can be defined in various ways, such as using color names, hexadecimal values, RGB, RGBA, HSL, HSLA, LCH, OKLCH, LAB, OKLAB, light-dark, color(), color-mix() and display-p3.

keywords: currentColor and transparent are also used in CSS to define colors.

Color Names

Definition: These are predefined color names in CSS, such as red, blue, green, black, white, etc. There are 140 named colors in CSS.

Named colors are convenient for quick, common colors but lack precision for more specific color needs.

```
p {
  color: red;
  background-color: lightblue;
}
```

Hexadecimal Colors

Hexadecimal colors are defined using a six-digit code consisting of letters and numbers, preceded by a "#".The first two digits represent the red component, the next two represent the green, and the last two represent the blue

You can also use a three-digit shorthand (e.g., #f00 for #ff0000), which is equivalent to doubling each digit.

```
p {
  color: #ff5733; /* Bright orange */
  color: #f53; /* Equivalent shorthand for #ff5533 */
  color: #ff0000; /* Red */
  color: #f00; /* Shorthand for Red*/
  background-color: #c0c0c0; /* Silver */
}
```

RGB and RGBA Colors

RGB stands for Red, Green, Blue, with values ranging from 0 to 255. RGBA adds an alpha channel for transparency, with a value between 0 (completely transparent) and 1 (completely opaque).

RGBA is particularly useful for overlay effects and blending colors.

```
color: rgb(255, 87, 51); /* Bright orange */
color: rgba(255, 87, 51, 0.5); /* 50% transparent */
```

This text is bright orange.

This text is 50% transparent orange.

HSL & HSLA Colors

HSL stands for Hue (0-360), Saturation (0%-100%), and Lightness (0%-100%). HSLA adds an alpha channel for transparency.

HSL is intuitive for adjusting colors based on human perception, making it easier to create shades and tints.

```
color: hsl(9, 100%, 60%); /* Bright orange */
color: hsla(9, 100%, 60%, 0.5); /* 50% transparent */
```

Opacity and Transparency

Transparency: Besides RGBA and HSLA, you can control an element's transparency using the opacity property, which affects the entire element, including its content.

```
opacity: 0.5; /* Makes the element 50% transparent */
```

The Future of Colors

LCH, OKLCH, LAB, OKLAB, Light-Dark, Color(), Color-Mix(), Display-P3

- LCH: Lightness, Chroma, Hue
- OKLCH: Lightness, Chroma, Hue with an alpha channel
- LAB: Lightness, A (green-red), B (blue-yellow)
- OKLAB: Lightness, A (green-red), B (blue-yellow) with an alpha channel
- Light-Dark: Adjusts the lightness of a color
- Color(): Creates a color from a string
- Color-Mix(): Mixes two colors
- Display-P3: Wide-gamut color space for digital displays

```
color: lch(60% 50 90); /* Lightness 60%, Chroma 50, Hue 90 */
color: lab(60% 50 90); /* Lightness 60%, A 50, B 90 */
color: light-dark(50%); /* Adjusts lightness to 50% */
color: color(display-p3 0.7 0.5 0); /* Display-P3 color */
color: color-mix(red blue 50%); /* Mixes red and blue 50% */
```

hideInToc: true layout: two-cols

```
<p style="color: var(--red)">This text is red.</p>
<p style="color: var(--dark-red)">This text is darker.</p>
<p style="color: var(--transparent-red)">This text is transparent.</p>
<p style="color: var(--soft-red)">This text is softer.</p>
```

CSS Units

CSS units are vital for defining the size, spacing, and layout of elements. Here's a more in-depth look at the types of units:

1. Absolute Units

- Fixed Units: These do not scale based on the viewport or parent elements.
- Pixels (px): Most common; ideal for precise control.

```
font-size: 14px; /* Fixed size */
```

2. Relative Units

- Flexible Units: These scale based on the parent element or viewport, making designs more responsive.
- em: Relative to the font size of the parent element. Useful for scalable spacing and typography.

```
padding: 1em; /* Equal to the current font size */
```

Unit	Name	Equivalent to
cm	Centimeters	1cm = 96px/2.54
mm	Millimeters	1mm = 1/10th of 1cm
Q	Quarter-millimeters	1Q = 1/40th of 1cm
in	Inches	1in = 2.54cm = 96px
pc	Picas	1pc = 1/6th of 1in
pt	Points	1pt = 1/72th of 1in
px	Pixels	1px = 1/96th of 1in

Contd (Relative Units)

- `rem`: Relative to the root element's font size (`html`), offering consistency across the page.

```
font-size: 1.2rem; /* 1.2 times the root font size */
```

- `%`: Relative to the parent element's size, commonly used in responsive design.

```
width: 80%; /* 80% of the parent element's width */
```

- `vw`, `vh`: Relative to the viewport's width or height. Ideal for full-screen layouts and responsive elements.

```
width: 100vw; /* Full width of the viewport */
height: 100vh; /* Full height of the viewport */
```

unit	relative to:
em	Relative to the font size, i.e. 1.5em will be 50% larger than the base computed font size of its parent. (Historically, the height of the capital letter "M").
ex	Heuristic to determine whether to use the x-height, a letter "x", or `.5em` in the current computed font size of the element.
cap	Height of the capital letters in the current computed font size of the element.
ch	Average character advance of a narrow glyph in the element's font (represented by the "0" glyph).
ic	Average character advance of a full width glyph in the element's font, as represented by the "水" (CJK water ideograph, U+6C34) glyph.
rem	Font size of the root element (default is 16px).
lh	Line height of the element.
rlh	Line height of the root element.

3. Viewport Units

- Viewport-based units: Perfect for responsive design.
- vw: 1% of the viewport width.
- vh: 1% of the viewport height.
- vmin and vmax: Relative to the smaller or larger of vw and vh.
- lvh and lvw
- ch: Relative to the width of the "0" (zero) character.
- svh and svw
- dvh and dvw

```
font-size: 5vw; /* Font size based on viewport width */
```

unit	relative to
vw	1% of viewport's width. People use this unit to do cool font tricks, like resizing a header font based on the width of the page so as the user resizes, the font will also resize.
vh	1% of viewport's height. You can use this to arrange items in a UI, if you have a footer toolbar for example.
vi	1% of viewport's size in the root element's inline axis. Axis refers to writing modes. In horizontal writing modes like English, the inline axis is horizontal. In vertical writing modes like some Japanese typefaces, the inline axis runs top to bottom.
vb	1% of viewport's size in the root element's block axis. For the block axis, this would be the directionality of the language. LTR languages like English would have a vertical block axis, since English language readers parse the page from top to bottom. A vertical writing mode has a horizontal block axis.
vmin	1% of the viewport's smaller dimension.
vmax	1% of the viewport's larger dimension.

CSS Gradients

Gradients are used to create smooth transitions between colors, adding depth and visual interest to designs. Here's a deeper look:

1. Linear Gradients A gradient that transitions along a straight line. You can control the direction and color stops.

`linear-gradient(direction, color-stop1, color-stop2, ...)`.

Direction: Can be specified with angles (e.g., 45deg) or keywords (to right, to bottom).

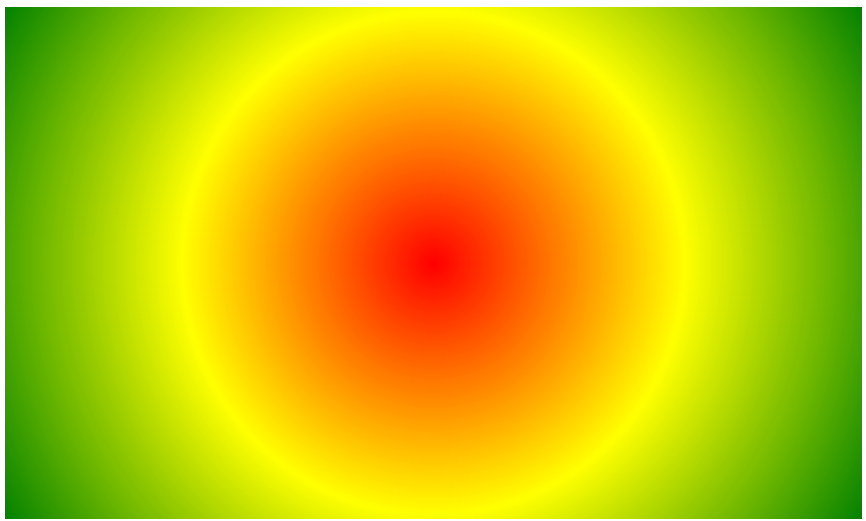
```
background: linear-gradient(45deg, red, yellow);
```

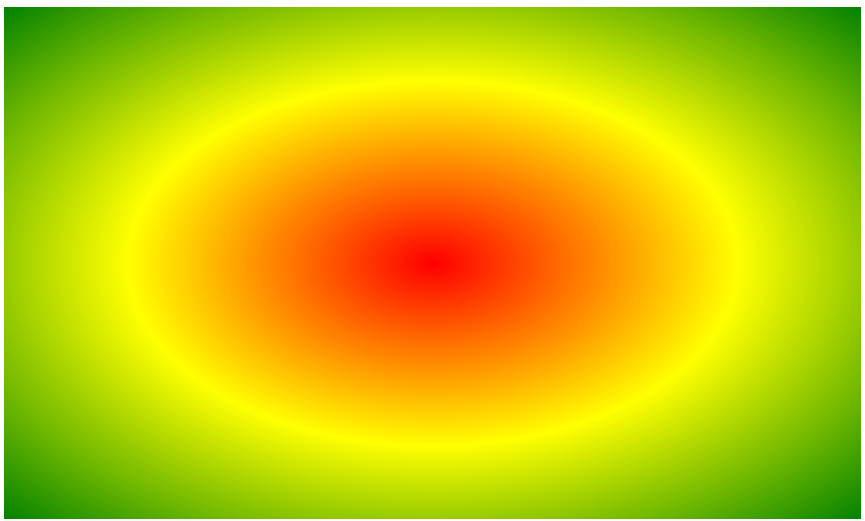
2. Radial Gradients Radiates from a central point outward, either circular or elliptical.

`radial-gradient(shape size at position, start-color, ..., end-color)`.

Shapes and Sizes: You can control the shape (circle or ellipse) and size (closest-side, farthest-corner, etc.).

```
background: radial-gradient(circle, red, yellow, green);
```



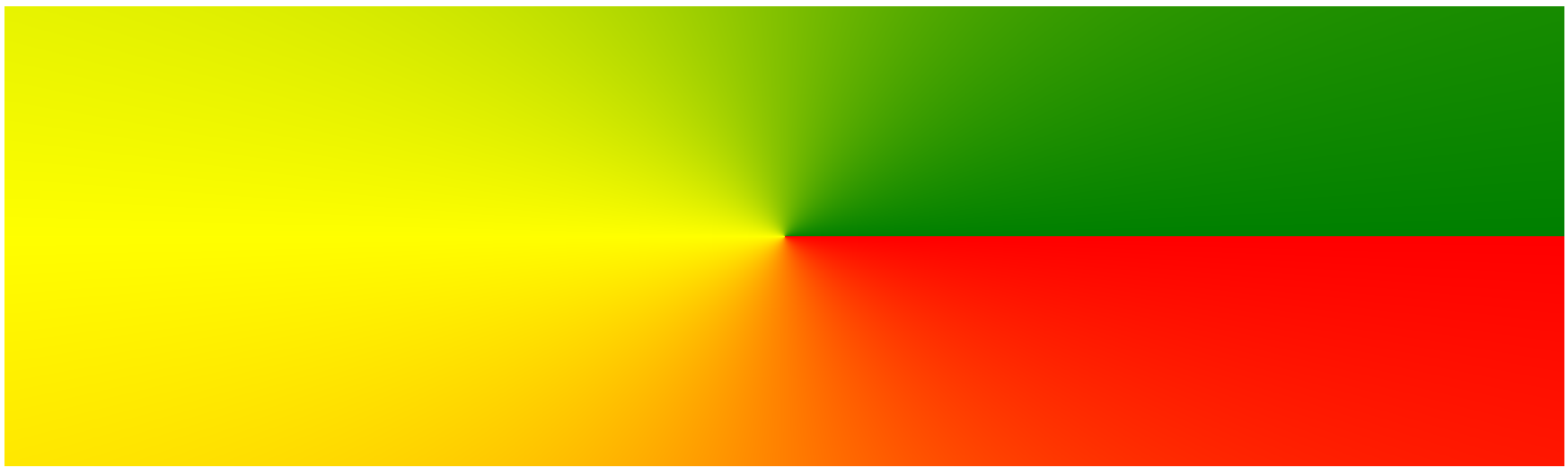


3. Conic Gradients

- A gradient that rotates around a central point, similar to slices of a pie.
- Often used for visualizations like pie charts.

`conic-gradient(from direction, color-stop1, color-stop2, ...)`

```
background: conic-gradient(from 90deg, red, yellow, green);
```



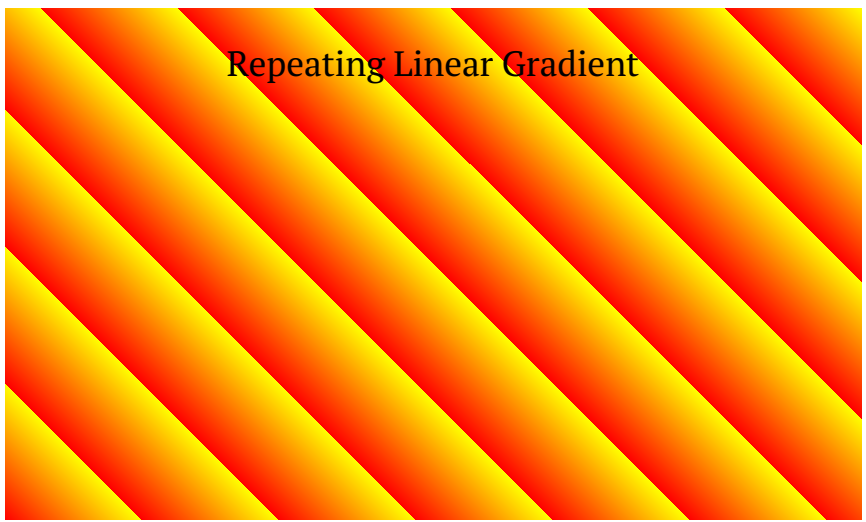
4. Repeating Gradients

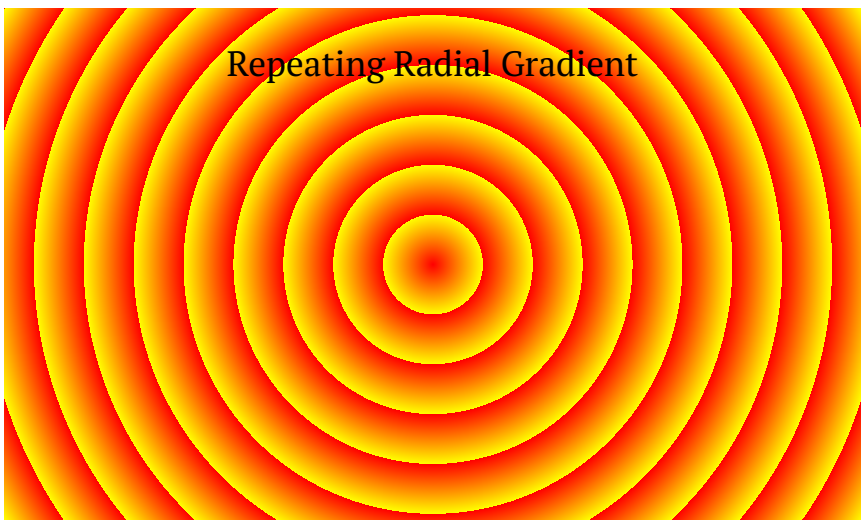
- Repeats the linear gradient pattern indefinitely.

```
background: repeating-linear-gradient(45deg, red, yellow 10%);
```

- Repeating Radial Gradients: Repeats the radial gradient pattern.

```
background: repeating-radial-gradient(circle, red, yellow 10%);
```





Practical Tips

- Combine Units: Use relative units (em, rem) for typography to maintain scalability and consistent spacing.
- Gradients with Transparency: Combine gradients with RGBA or HSLA colors for layered effects with transparency.
- Viewport Units for Responsiveness: Use vw and vh for elements that need to adapt to screen size changes, such as full-screen sections or responsive text sizes.

CSS Functions

Functions in CSS are used to manipulate values, perform calculations, and apply effects. Here are some common functions:

- calc(): Performs calculations on property values.
- var(): Defines custom properties (variables).
- rgb(), rgba(), hsl(), hsla(): Define colors using RGB, RGBA, HSL, and HSLA values.
- url(): Specifies the location of an external resource.
- linear-gradient(), radial-gradient(), conic-gradient(): Create gradients with smooth color transitions.
- clamp(): Restricts a value to a specified range.

```
p {
  font-size: calc(1rem + 1vw);
  color: var(--primary-color);
  background: linear-gradient(to right, red, blue);
}
```

CSS @Rules

@Rules are used to define special rules in CSS that control how styles are applied. Here are some common @Rules:

- @media: Defines media queries for responsive design.
- @keyframes: Creates animations with multiple keyframes.
- @font-face: Embeds custom fonts in a web page.
- @import: Imports external CSS files.
- @supports: Checks if a browser supports a particular CSS feature.
- @page: Defines the layout of printed pages.
- @layer: Specifies the layering order of elements.

CSS Variables or Custom Properties

CSS variables (also known as custom properties) are used to store reusable values in CSS. They are defined using the `--` prefix and can be used throughout the stylesheet.

```
:root {
  --primary-color: #ff5733;
  --secondary-color: #f0f0f0;
}

p {
  color: var(--primary-color);
  background-color: var(--secondary-color);
}
```

Can be used to store colors, font sizes, spacing, and other values that are reused across the stylesheet. They are particularly useful for maintaining consistency and making global changes easier. The new `@property` rule in CSS allows you to define custom properties with specific types and values.

```
@property --primary-color {
  syntax: "<color>";
  inherits: false;
  initial-value: black;
}
```

JS-Free hamburger menu demo

With the Popover API

Click open the hamburger menu to see a demo of JavaScript-free interaction handling! By using the `popover` attribute, you can allow the browser to handle the keyboard management (including navigation via `esc`, `spacebar`, and `enter`), optional light-dismiss (clicking outside the boundaries of the popover), and click handlers such as on the open and close buttons.

Typography in CSS

Typography is a crucial aspect of web design, as it affects readability, accessibility, and overall user experience. Here are some key CSS properties for typography: `font-style`, `font-weight`, `font-size`, `line-height`, `font-family`, `text-align`, `text-transform`, `text-decoration`, `letter-spacing`, `word-spacing`, `text-shadow`, `white-space`, `overflow-wrap`, `word-break`, `hyphens`, `text-overflow`, `vertical-align`, `text-orientation`, `font-variant`.

```css
p {
  font-family: "Arial", sans-serif;
  font-size: 16px;
  line-height: 1.5;
  font-weight: 400;
  text-align: center;
  text-transform: uppercase;
  text-decoration: underline;
  letter-spacing: 1px;
  word-spacing: 2px;
```

We can use Google Fonts(any other font hosting service) or custom fonts in CSS to enhance the typography of a web page. Google Fonts offers a wide range of free, open-source fonts that can be easily integrated into a website.

```css
@import url("https://fonts.googleapis.com/css2?family=Roboto
body {
  font-family: "Roboto", sans-serif;
}
```

The @font-face rule can be used embed custom fonts in a web page, allowing the use of font file format (e.g., .woff, .woff2, .ttf) and define font properties like font-weight and font-style.

```css
@font-face {
  font-family: "CustomFont";
  src: url("custom-font.woff2") format("woff2");
  font-weight: 400;
  font-style: normal;
}
body {font-family: "CustomFont", sans-serif;}
```

Debugging in browser

Debugging

Debugging is the process of finding and fixing errors or bug in the source code of any software. When writing code, everything may appear normal during development, but errors can arise during runtime. These errors typically fall into two categories:

- Syntax Error: Occurs when the code does not adhere to the language's rules or grammar, preventing it from being compiled or interpreted correctly.
- Logic Error: Occurs when the code is syntactically correct but produces incorrect or unintended results due to flawed reasoning or incorrect algorithm implementation.

You might wonder when you'd need to debug CSS. Let me explain: Sometimes, when writing CSS rules for an element, you may encounter situations where your styles aren't being applied as expected, or the element isn't behaving the way you intended.

Note: When in doubt in CSS put a border on the element.

Debugging in the browser

One of the fastest way to get your CSS debugged is to use the browser. Browser like Chrome, Firefox etc offer powerful developer tools for debugging and this is what we are referring to as debugging in the browser.

How to access browser DevTools

This devtools live inside the browser and you can access it by:

1. Press-and-hold/right-click an item on a webpage and choose inspect from the context menu that appears. This will show all the code that made up the UI but highlighted the code of the element you right-clicked. Click on Elements to see how the HTML looks like on runtime and their respective CSS applied.
2. Keyboard: On WIndows `Ctrl` + `Shift` + `I` On macOS: `Command` + `Shift` + `I`

DevTools

In the developer tools, you can immediately modify the HTML and CSS, with the changes reflected live in the browser. This feature is valuable for previewing your intended modifications before implementing them locally.

Also, you can toggle CSS rules by unchecking the corresponding checkboxes in the devTools, allowing you to experiment with different styles on the fly.

Additionally, we have talked about the Box Model in the previous lesson, the devTools layout view shows you the box model on a selected elements and gives you proper insight on the element box property like border, margin, padding, height and width.

Inspecting the applied CSS

To examine the CSS that an element inherits or has applied to it, right-click on the element and choose "Inspect" to open the devTools. In the devTools, one section displays the HTML, while another shows the CSS inherited by the element as well as the styles directly applied to it. This is particularly helpful for identifying any unexpected CSS affecting the element.In the image below the developer is trying to check the CSS on the body element.

Inline, Internal and External CSS

Inline CSS is used to apply a unique style to a single HTML element. It is done using the style attribute directly within the HTML tag

```
<p style="color: blue; font-size: 20px;">
  This is a paragraph with inline CSS.
</p>
```

Advantages:

- Quick and easy for small, specific changes.
- Good for overriding styles in a pinch.

Disadvantages:

- Makes the HTML code harder to read and maintain.
- Not suitable for styling multiple elements.

Internal CSS

Internal CSS is used to define styles for an entire HTML document. It is placed within the `style` tag in the `head` section of the HTML file.

```
<head>
  <style>
    p {
      color: red;
      font-size: 18px;
    }
  </style>
</head>
<body>
  <p>This is a paragraph with internal CSS.</p>
</body>
```

Advantages

- Keeps styles in one place within the document.
- Useful for applying styles to a single page.
- Easier to manage and maintain than inline CSS.

Disadvantages

- Not efficient for styling across multiple pages.

External CSS

External CSS involves linking an external .css file to your HTML document. This file contains all the styles, which can be applied to multiple HTML documents.

Syntax

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

```
/* In styles.css */
p {
  color: green;
  font-size: 16px;
}
```

```
<head>
  <link rel="stylesheet" href="styles.css" />
</head>
<body>
  <p>This is a paragraph with external CSS.</p>
</body>
```

External CSS

Advantages

- Keeps HTML files clean and separates content from design..
- Efficient for applying the same styles across multiple pages.
- Easier to maintain and update, as changes in the external CSS file are reflected across all linked pages.

Disadvantages

- Requires an additional HTTP request to load the CSS file.
- No styles will be visible if the CSS file fails to load.

Summary

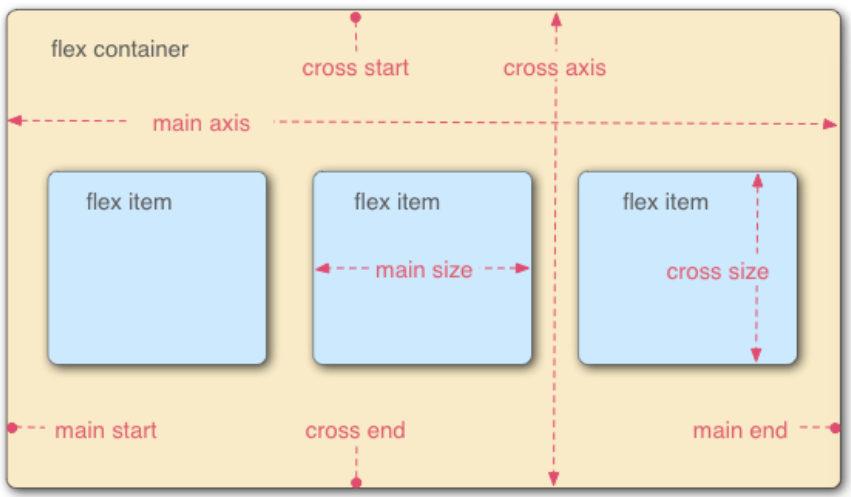
- Inline CSS: Best for quick, single-use styles but not ideal for maintainability.
- Internal CSS: Good for single-page styling, better than inline but still not ideal for multiple pages.
- External CSS: Preferred method for styling, offering maintainability and scalability across multiple documents.

FlexBox

Flexbox is a one-dimensional layout method for arranging items vertically(columns) or horizontally(rows).

To implement a flexbox layout in CSS, you need to set `display: flex;` in your CSS rules.

When elements are laid out as flex items, they are laid out along two axis:



- The main axis is the direction in which flex items are laid out, such as across the page in a row or down the page in a column. The start and end points of this axis are referred to as the main start and main end. The distance between the main start and main end is known as the main size.
- The cross axis runs perpendicular to the direction in which flex items are laid out. The start and end points of this axis are called the cross start and

Why Flexbox?

- It allows you to display item(s) as a row, or a column
- Vertically center a block of content inside its parent
- They respect the writing mode of the document
- Items in the layout can be visually reordered, away from their order in the DOM
- Make all columns in a multiple-column layout adopt the same height even if they contain a different amount of content.
- Space can be distributed inside the items, so they become bigger and smaller according to the space available in their parent.
- Make all the children of a container take up an equal amount of the available width/height, regardless of how much width/height is available.

Direction and Alignment

To determine how flex items are arranged within a flex contaniner, direction and alignment are the key aspects.

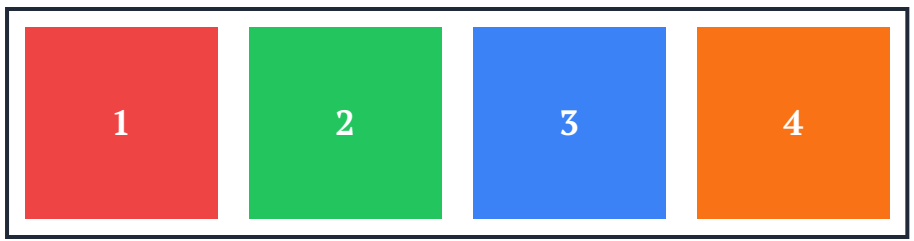
Flex Direction: The `flex-direction` property defines the direction in which the flex items are placed within the flex container. The direction can be either block (column) or inline (row).

The following values can be assigned to it:

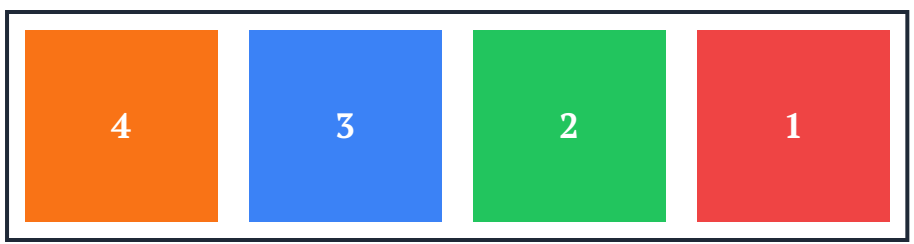
```
.container {
  display: flex;
  flex-direction: row;
}
.container {
  display: flex;
  flex-direction: row-reverse; //row-reverse arranges items order from right to left
}
.container {
 display: flex;
 flex-direction: column;
}
.container {
 display: flex;
```

Flex Direction Code Example:

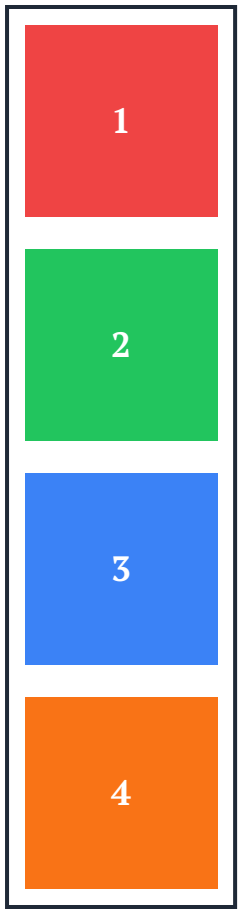
Flex Direction: Row (Default)



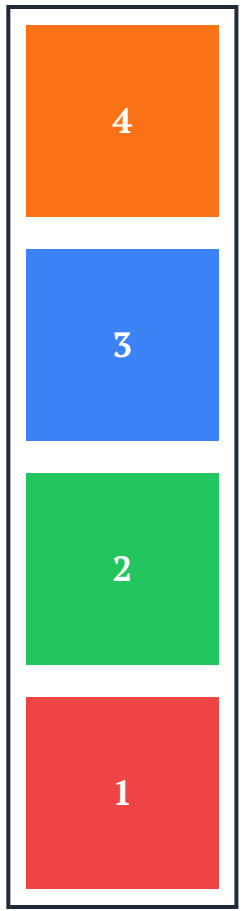
Flex Direction: Row-Reverse



Flex Direction: Column



Flex Direction: Column-Reverse



Alignment

Absolutely! Flexbox is indeed powerful for aligning elements with precision. It excels in both horizontal and vertical alignment, making it easier to create responsive layouts that adapt to different screen sizes.

Let's take a look at the flexbox properties that controls alignment and spacing

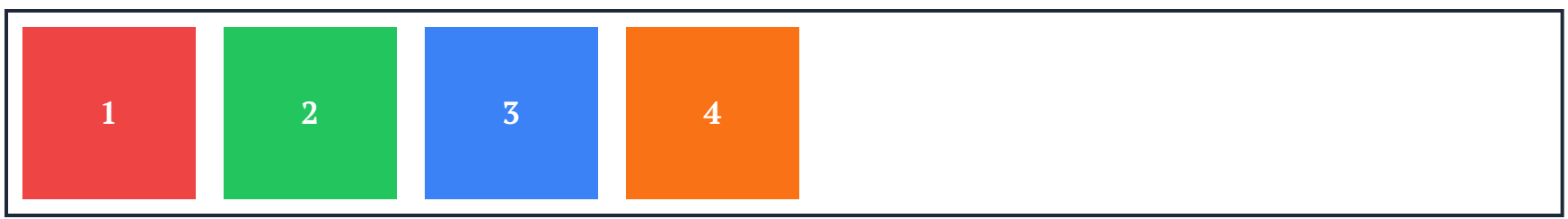
1. Justify Content(Main Axis Alignment)
2. Align Items(Cross Axis Alignment)
3. Align Content (Multi-line Cross Axis Alignment)
4. Align Self

Justify Content(Main Axis Alignment)

The main axis is the natural way the flex items are laid out across the page in a row. With the `justify-content` flex property you can control how you want your items to be laid out. Code Example:

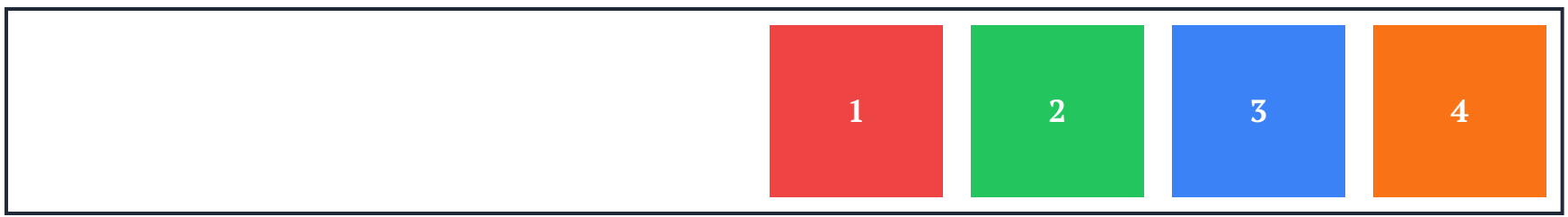
`flex-start` : Items are aligned to the start of the container.

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



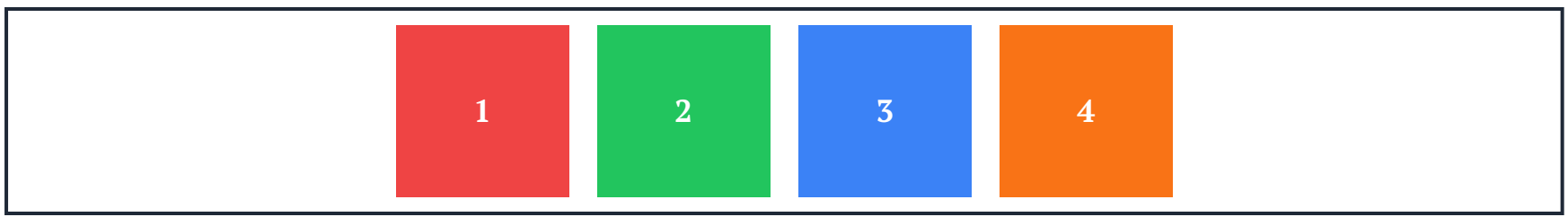
`flex-end:` Items are aligned to the end of the container.

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



`center` : Items are centered along the main axis.

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



`space-between` : Items are evenly distributed in the line; the first item is on the start line and the last item is on the end line.

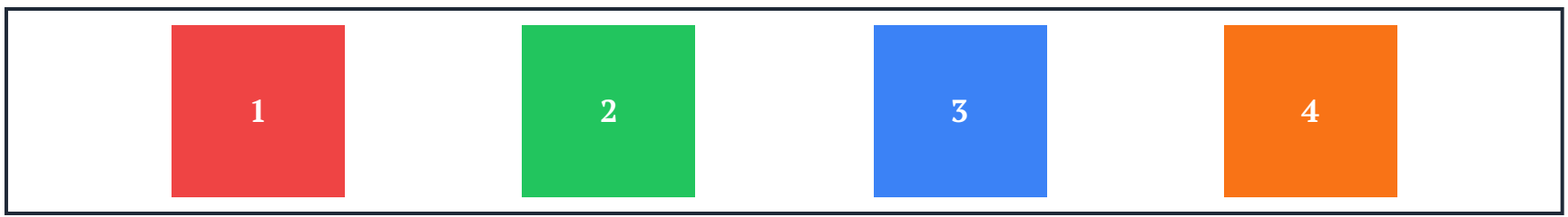
```
.container {
  display: flex;
  justify-content: space-between;
}
```

`space- around` : Items are evenly distributed in the line with equal space around them.

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

`space-evenly` : Items are evenly distributed with equal space between them.

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

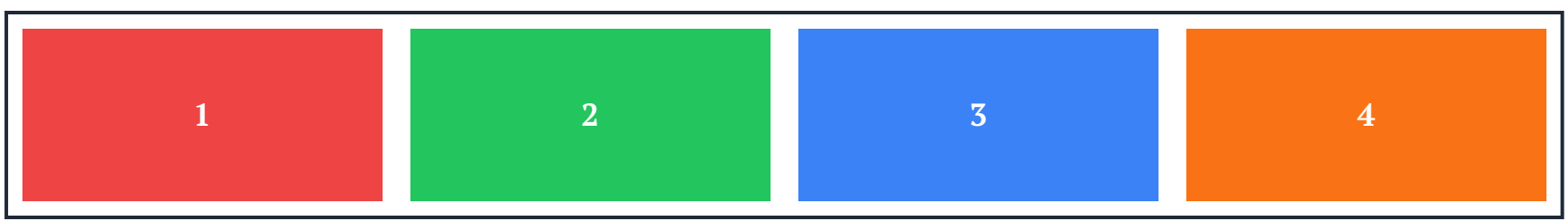


Align Items (Cross Axis Alignment)

The cross axis runs perpendicular to the direction in which flex items are laid out. The `align-items` property aligns the flex items along the cross axis (perpendicular to the main axis).

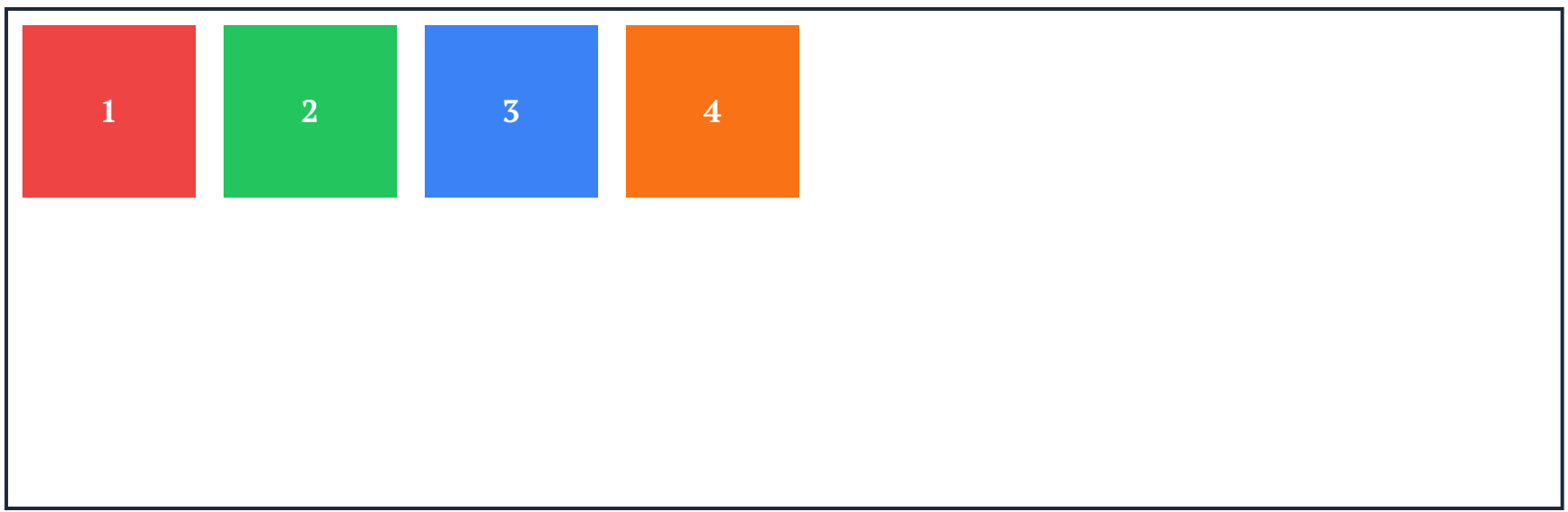
`stretch` : Items stretch to fill the container (default).

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



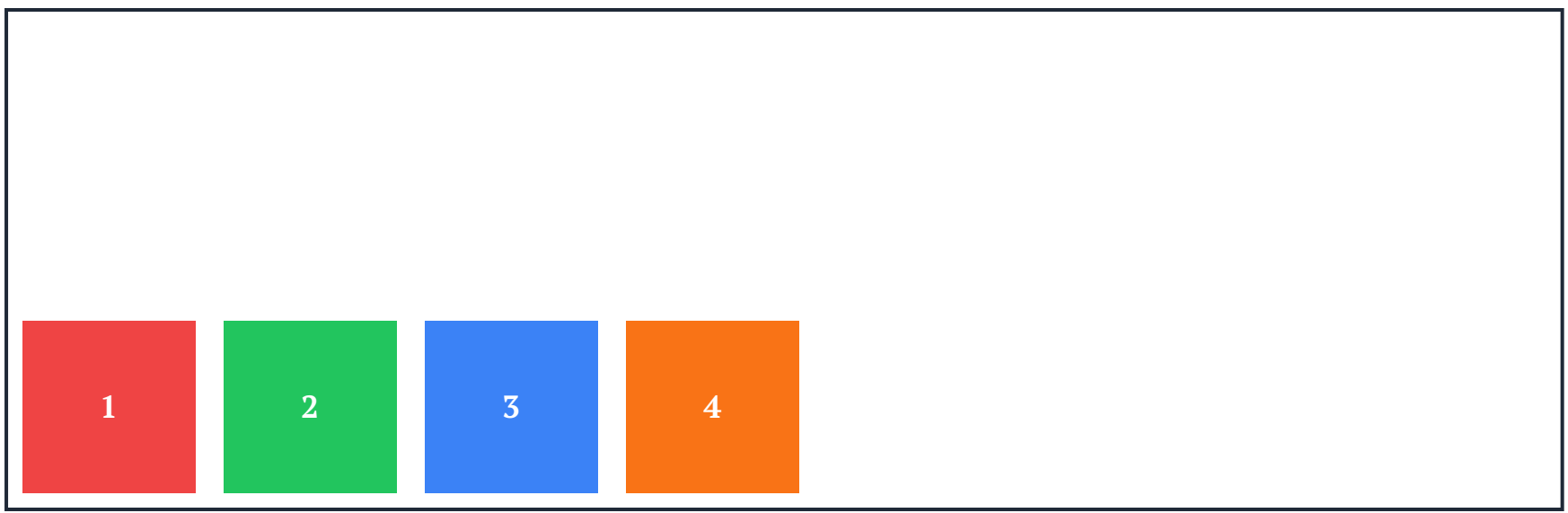
`flex-start` : Items are aligned to the start of the cross axis.

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



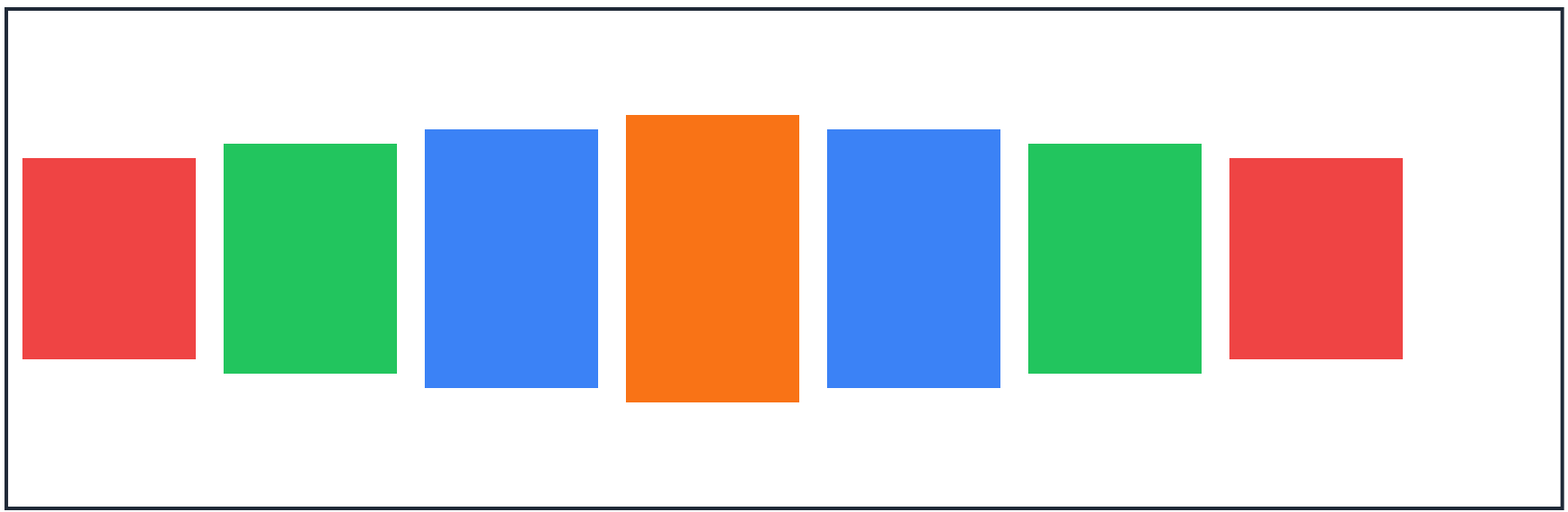
`flex-end` : Items are aligned to the end of the cross axis.

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



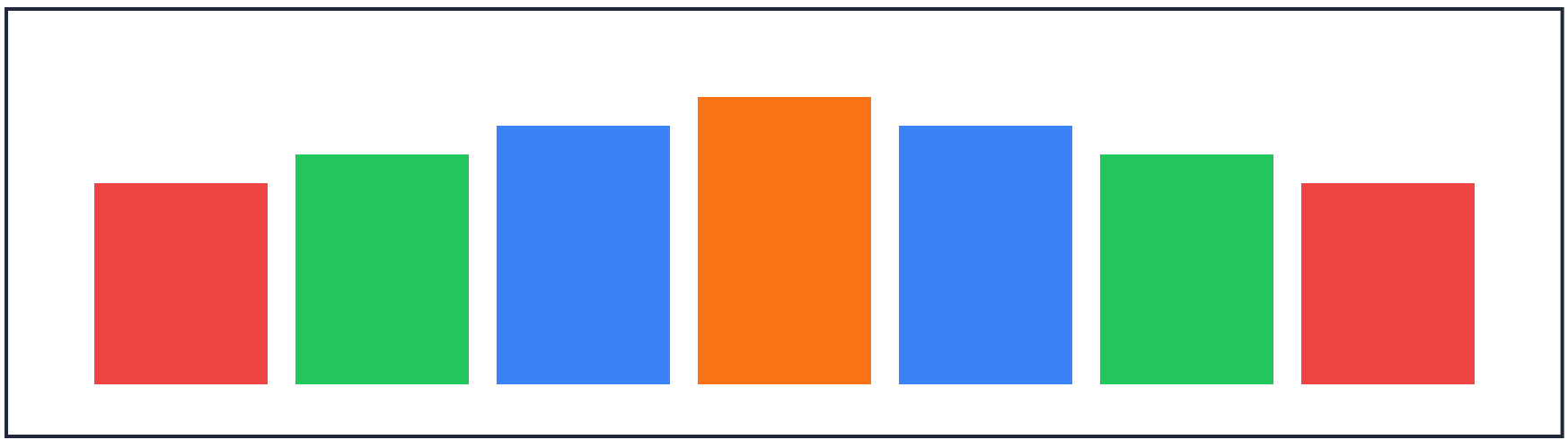
`center`: Items are centered along the cross axis.

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



`baseline`: Items are aligned along their baseline. If you want to make sure the bottoms of each character are aligned, as they would be if they were written on a page then `align-items: baseline;` is used instead of `align-items: center;`.

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

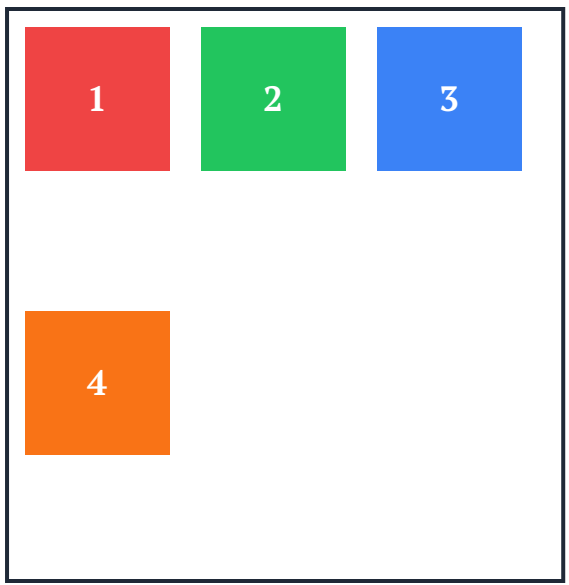


Align Content (Multi-line Cross Axis Alignment)

The `align-content` property is used to control the alignment of multiple lines of items along the cross axis (which is perpendicular to the main axis). This property is only relevant when the flex container has more than one line of items, typically when flex-wrap is set to wrap or wrap-reverse.

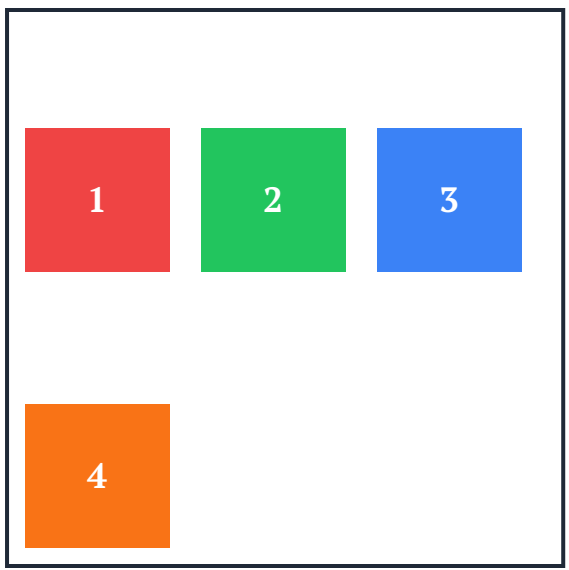
`flex-start` : Rows are packed to the start of the container.

```
.container {
  display: flex;
  flex-wrap: wrap;
  align-content: flex-start;
}
```



`flex-end` : Rows are packed to the end of the container.

```
.container {
  display: flex;
  flex-wrap: wrap;
  align-content: flex-end;
}
```

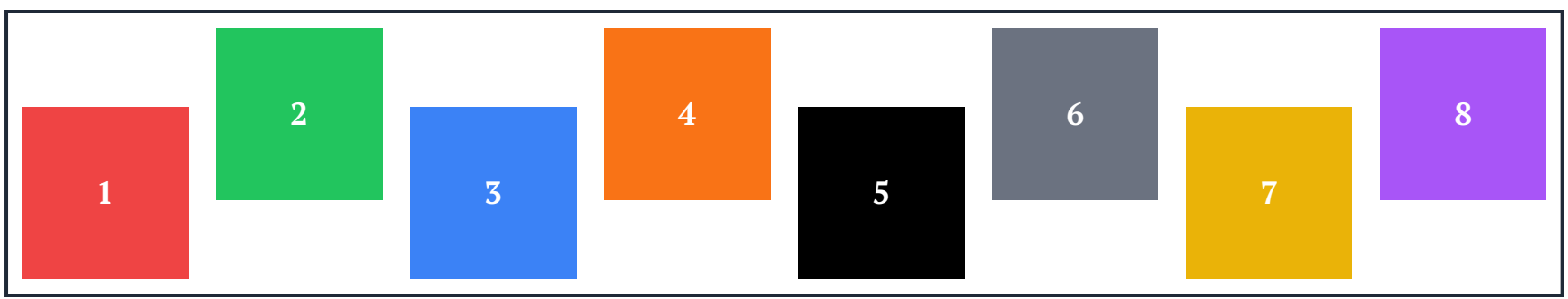


Align Self

In a case where you want a specific child(ren) to have specific alignments instead of aligning all the children, flexbox gives you the `align-self` property to achieve this.

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

.container:nth-child(odd) {
  align-self: flex-end;
}
```



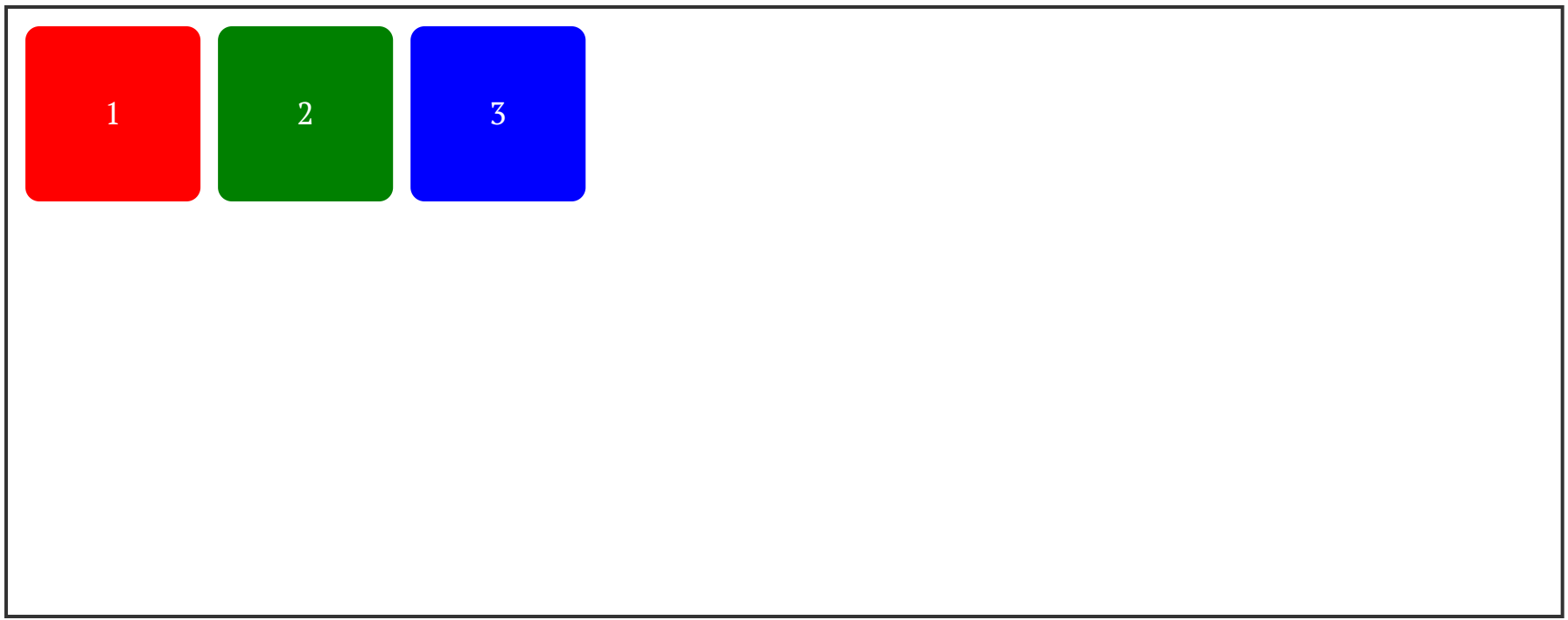
@Oluwasetemi

Practice FlexBox: Justify Content and Align Items

flex-direction: row

justify-content: flex-start

align-items: stretch



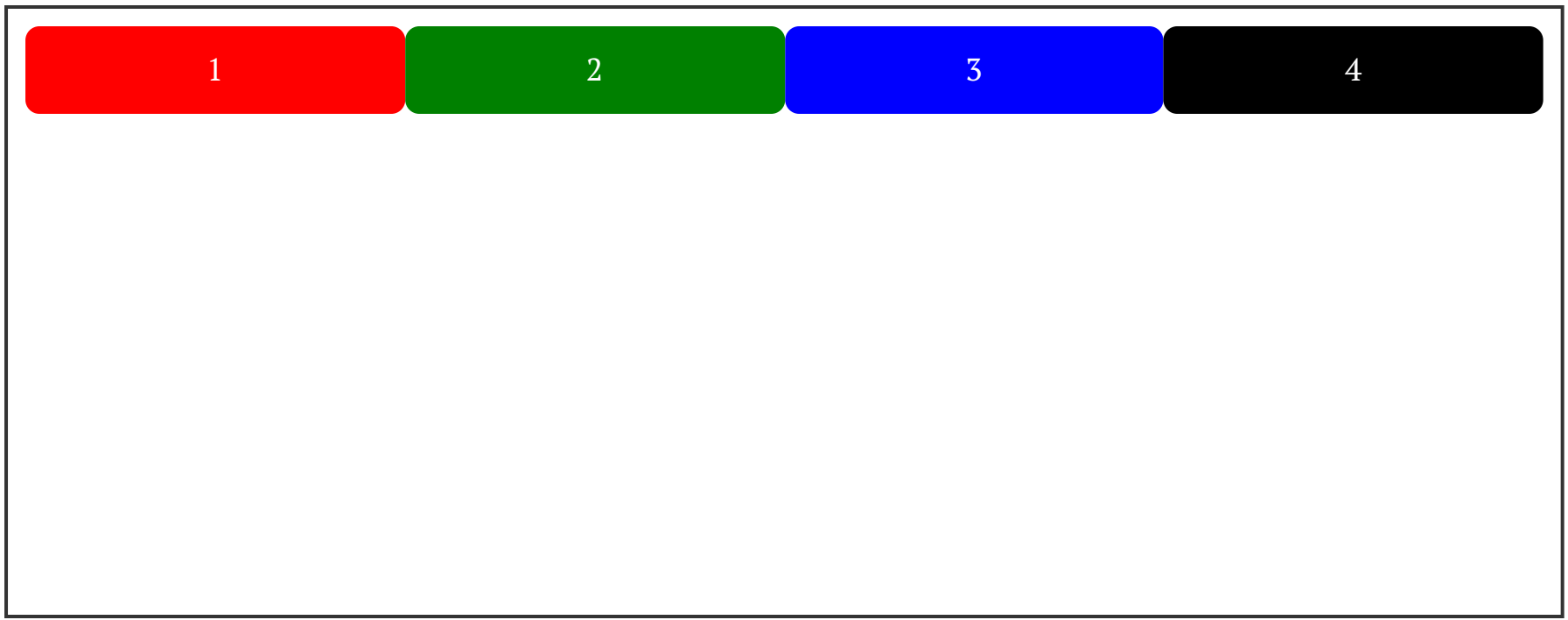
@Oluwasetemi

Practice FlexBox: Align-Content

flex-direction: row

flex-wrap: nowrap

align-content: stretch



Growing and Shrinking

There are two important sizes when dealing with Flexbox: the minimum content size, and the hypothetical size.

- The minimum content size is the smallest an item can get without its contents overflowing.
- The hypothetical size refers to the size a flex item would take up if it were not subjected to the flex-grow, flex-shrink, or any other flex properties that might cause it to stretch or shrink. It’s the size that the item "wants" to be, based on its content and its initial settings like width, height, padding, and margin, before any flex-related adjustments are applied.

Flex-Grow

The `flex-grow` CSS property specifies how much a flex item will grow relative to the other flex items inside the same container when there is positive free space available.

The value of flex-grow is a unitless number that serves as a proportion, determining how much of the available space inside the flex container the item should take up compared to other items.

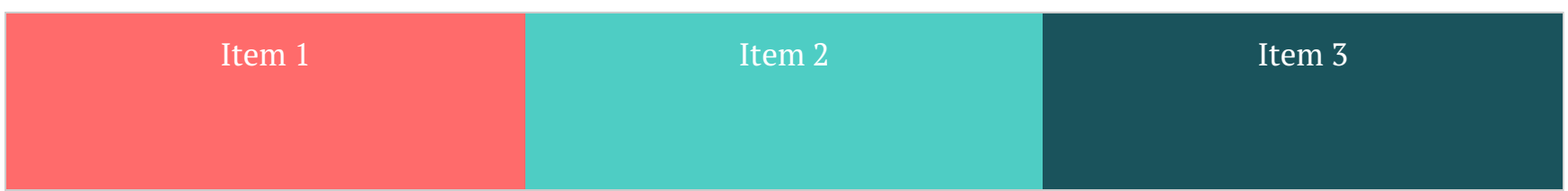
Flex grow is about consuming additional space and it only does something when items are above their hypothetical size

Adjust Flex-Grow Values

Item 1 (flex-grow: 1)

Item 2 (flex-grow: 1)

Item 3 (flex-grow: 1)



Flex-Shrink

The `flex-shrink` CSS property determines how much flex items will shrink relative to each other when the flex container is too small to accommodate their full size.

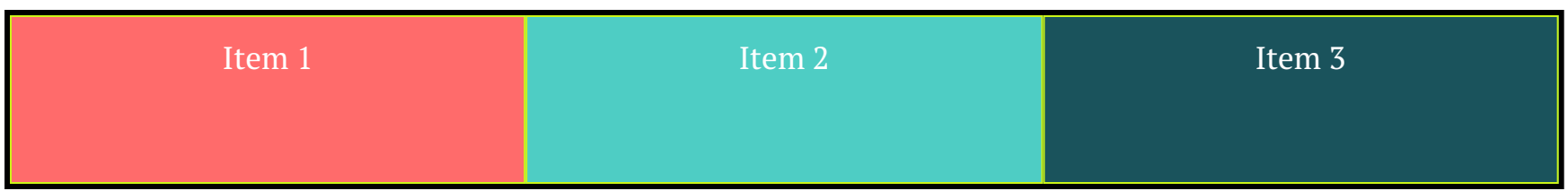
Flex shrink only does something when the items are between their minimum size and hypothetical size and you can disable the ability of an item to shrink by setting `flex-shrink: 0;`.

Adjust Flex-Shrink Values

Item 1 (flex-shrink: 1)

Item 2 (flex-shrink: 1)

Item 3 (flex-shrink: 1)



Flex-Basis

The `flex-basis` CSS property has the same effect as `width` in a flex row (height in a column). You can use them interchangeably, but flex-basis will win if there's a conflict. `flex-basis` can't scale an element below its minimum content size, but width can.

The "flex" Shorthand

The `flex` CSS property takes 3 individual values:

1. `flex-grow`
2. `flex-shrink`
3. `flex-basis`

`flex` sets how a flex item will grow or shrink to fit the space available in its flex container. It does the basic management automatically.

It is recommended to use the `flex` shorthand instead of separate `flex-grow` `flex-shrink` `flex-basis` declarations.

```
/*instead of this */
.container {
  flex-grow: 1;
  flex-shrink: 1;
  flex-basis: 0px;
}
```

```
/*try this */
.container {
  flex: 1;
}
```

Grid Layout

Grid Layout is a two-dimensional layout system that allows you to create complex web designs with minimal code. It enables you to align elements into rows and columns, making it easier to design web pages that are responsive and adaptable to different screen sizes.

Grid Container

The grid container is the parent element that contains the grid items (child elements). To create a grid container, you set the display property of the parent element to grid or inline-grid.

```
<div class="grid-container">
  <div class="grid-item">Item 1</div>
  <div class="grid-item">Item 2</div>
  <div class="grid-item">Item 3</div>
</div>
```

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

Item 1

Item 2

Item 3

Defining Rows and Columns

You can define the structure of the grid using the **grid-template-rows** and **grid-template-columns** properties. These properties determine the number of rows and columns in the grid and their respective sizes.

```
.grid-container {
  display: grid;
  grid-template-columns: 200px 1fr 100px;
  grid-template-rows: 100px 200px;
  grid-gap: 4;
}
```

Example

```
.grid-container {
  display: grid;
  grid-template-columns: 200px 1fr 100px;
  grid-template-rows: 100px 200px;
  grid-gap: 4;
}
```

In this example:

- **grid-template-columns: 200px 1fr 100px;** creates three columns. The first column is 200px wide, the second column takes up the remaining space (**1fr**), and the third column is 100px wide.
- **grid-template-rows: 100px 200px;** creates two rows, the first row being 100px tall, and the second row being 200px tall.

Placing Grid Items

By default, grid items are placed in the grid based on the order they appear in the HTML. However, you can control their placement using the grid-column and grid-row properties.

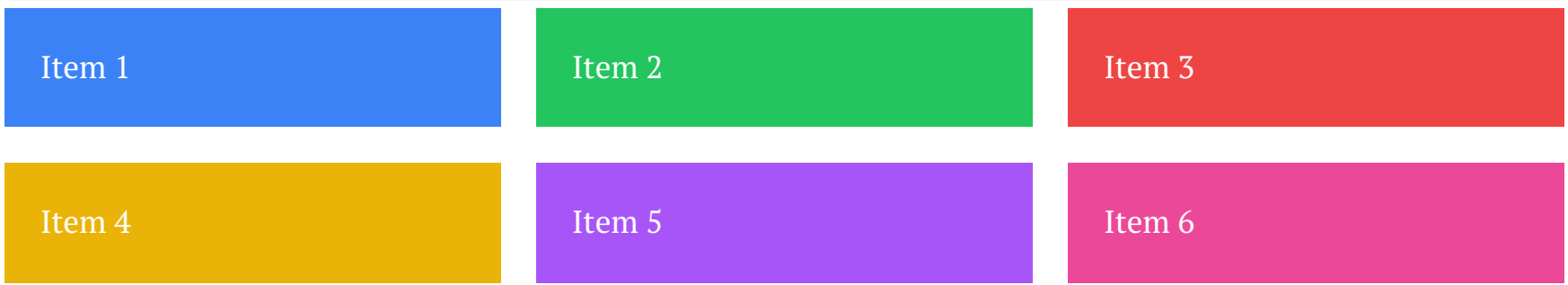
```
.grid-item:nth-child(1) {
  grid-column: 1 / 3; /* Spans across the first and second columns */
  grid-row: 1; /* Placed in the first row */
}

.grid-item:nth-child(2) {
  grid-column: 3; /* Placed in the third column */
  grid-row: 1 / 3; /* Spans across the first and second rows */
}
```

Grid Gaps

To create space between grid items, you can use the **grid-gap**, **row-gap**, and **column-gap** properties.

```
.grid-container {
  display: grid;
  grid-template-columns: repeat(3, 1fr);
  grid-gap: 20px; /* 20px space between all grid items */
}
```



Grid Areas

Grid areas allow you to name specific sections of the grid, making it easier to define complex layouts. You can use grid-template-areas to define areas and grid-area to place grid items within those areas.

```
.grid-container {
  display: grid;
  grid-template-areas:
    "header header header"
    "sidebar main main"
    "footer footer footer";
  grid-template-rows: auto 1fr auto;
  grid-template-columns: 150px 1fr 1fr;
}

.header { grid-area: header; }

.sidebar { grid-area: sidebar; }

.main { grid-area: main; }

.footer {grid-area: footer;}
```

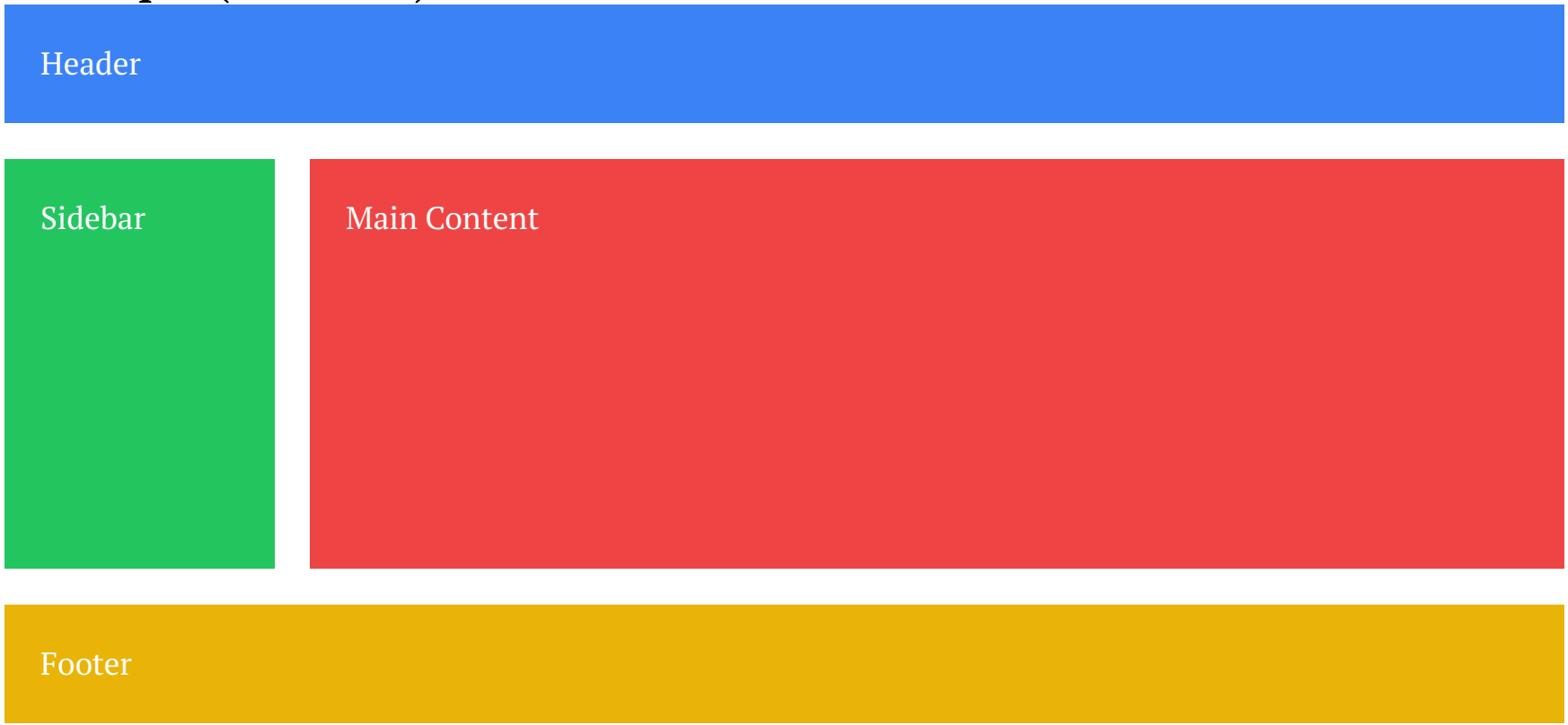
Responsive Design with Grid

CSS Grid makes it easy to create responsive designs. You can use functions like repeat() and minmax() to create grids that adjust based on the available space.

```css
.grid-container {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(100px, 1fr));
  grid-gap: 10px;
}
```

- repeat(auto-fit, minmax(100px, 1fr)); automatically creates as many columns as will fit into the container, with each column being at least 100px wide and taking up a fraction of the remaining space.
- This ensures that the grid adjusts dynamically as the viewport size changes.

Example(CONTD)



SubGrid

Pug | SCSS | Result | EDIT ON CODEPEN

```
mixin card(title, content)
  .box
    img(
      src="https://picsum.photos/1600/900",
      alt="",
      width="1600",
      height="900"
    )
    h1.title= title
    .content
      p= content
    .actions
      button.button Details
      button.button Buy

.grid
  +card("Short title, short text", "Lorem ipsum
```

Resources | View Compiled | 1× | 0.5× | 0.25× | Rerun

Positioned Layout

Positioned Layout is another layout mode we'll explore in this section. Unlike the flow layout algorithm, which ensures that multiple elements never occupy the same pixels, positioned layout allows items to overlap and break out of the box.

To style your layout, use the `position` property with one of the following values: `relative`, `absolute`, `fixed`, or `sticky`. Each of these positioning values works uniquely to place the element. Combine it with the `top`, `right`, `bottom`, and `left` properties to specify the exact location of the element within its containing block.

Relative Positioning

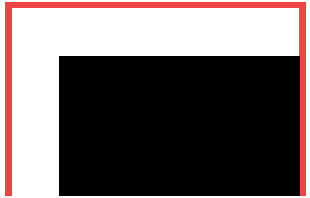
The element is positioned `position: relative;` based on the normal document flow and then adjusted relative to its original position using the top, right, bottom, and left values. This adjustment does not impact the layout or positioning of surrounding elements, so the space allocated for the element remains the same as if it were using static positioning.

Absolute Positioning

Every element is contained by a block which is referred to containing block. When you absolutety positioned an element, it ignore their parents block to cause an overflow unless the parent use positioned layout.

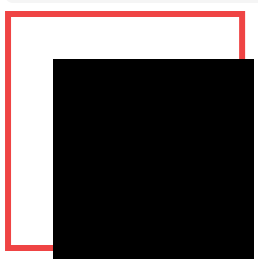
Absolutely-positioned elements act just like static-positioned elements when it comes to overflow. If the parent sets `overflow: auto;`, as long as that parent is the containing block, it will allow that child to be scrolled into view:

```
.wrapper {
  overflow: auto;
  position: relative;
  /* other styles here */
}
.box {
  position: absolute;
  /* other styles here */
}
```



```css
.wrapper {
  overflow: hidden;
  width: 100px;
  height: 100px;
  border: 3px solid red;
}

.box {
  position: absolute;
  top: 24px;
  left: 24px;
  background: black;
  width: 150px;
  height: 200px;
}
```

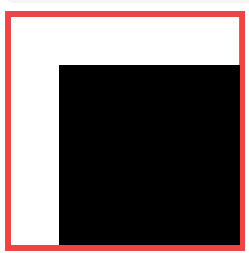


`.box` is not been contained by wrapper even with the `overflow: hidden;` passed into the `wrapper` CSS rule because the parent which is `wrapper` is not using positioned layout.

Error fixed by adding `position: relative;` to the parent.

```
.wrapper {
  overflow: hidden;
  position: relative;
  width: 100px;
  height: 100px;
  border: 3px solid red;
}

.box {
  position: absolute;
  top: 24px;
  left: 24px;
  background: black;
  width: 150px;
  height: 200px;
}
```



Fixed Positioning

To create a "floating" element that stays in the same position regardless of scrolling, you should use `position: fixed;` This is similar to absolute positioning, but there are key differences:

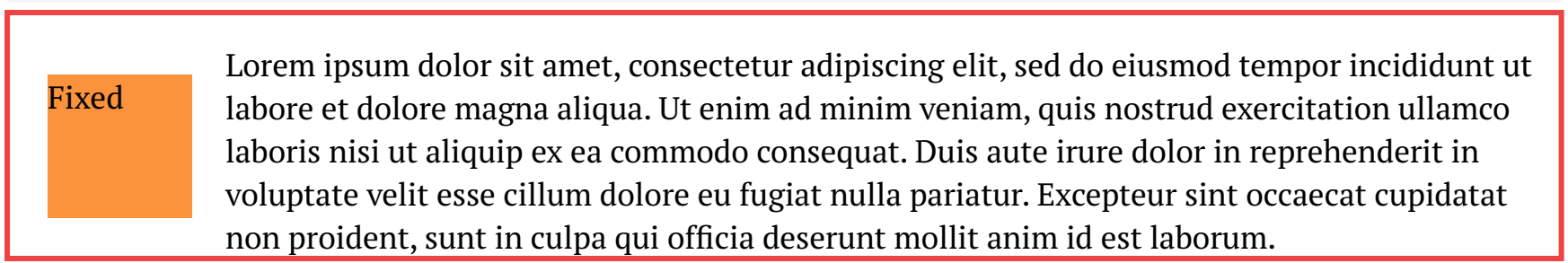
Fixed Positioning: A fixed element is positioned relative to the viewport, meaning it stays in the same place on the screen even when you scroll. The element is contained by the "initial containing block," which is essentially the entire browser window or viewport. With position: fixed, the element will not move when the user scrolls the page.

Absolute Positioning: An absolutely positioned element is positioned relative to its nearest positioned ancestor (an ancestor with position set to relative, absolute, or fixed). If no such ancestor exists, it will be positioned relative to the initial containing block, just like a fixed element. With `position: absolute;`, the element will move with its parent element if the parent is scrolled.

```css
.scroll-container {
  width: 100%;
  height: 35px;
  overflow: scroll;
  border: 3px solid red;
}

.fixed-box {
  position: fixed;
  bottom: 30px;
  left: 80px;
  width: 80px;
  height: 80px;
  background: orange;
}

.scroll-content-box {
  padding-left: 120px;
}
```



Sticky Positioning

In this form of positioning `position: sticky;`, an element transitions from being relatively-positioned to being fixed-positioned and this happens when you scroll, the element get stuck to the edge. To pass `position: sticky;` to an element and work effectively, you must specify a threshold with at least one of to top, right, bottom, or left.

While using `position: sticky;` note that the element will never follow the scroll outside of its parent container. The sticky elements only stick while their container is in view.

Code Example

```
dt {
  position: sticky;
  top: -1px;
  /* other styles */
}

dd {
  margin: 0;
  /* other styles */
}
```

A

Andrew W.K.

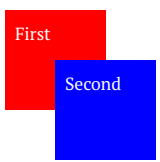
Apparat

Arcade Fire

Stacking Context/Z-index

In CSS, the stacking order of elements is a crucial aspect of layout and design. Two key concepts that control how elements stack on top of each other are stacking contexts and the z-index property. Alright, imagine you're stacking a bunch of transparent sheets on top of each other. That's basically what's happening when you're building a webpage with CSS. But sometimes, you want to control which sheet goes on top, right? That's where z-index and stacking contexts come in. Let's break it down!

Natural Stacking Order First things first. When you're writing your HTML, the browser stacks elements in the order they appear. It's like if you're laying down those transparent sheets one by one. The last one you put down ends up on top. Check this out:



Natural Stacking Order

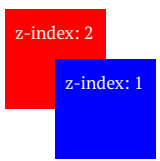
```
<div style="background: red; width: 100px; height: 100px;"></div>
<div style="background: blue; width: 100px; height: 100px; margin-top: -50px; margin-left: 50px;"></div>
```

See how the blue box is on top of the red one? That's because in our HTML, it came after the red box. Simple, right?

Z-index

The z-index property only works on positioned elements. If applied to a non-positioned element, it has no effect. However, there's an exception: flex children can use z-index even if they are non-positioned.

Now, what if you want to flip that order? That’s where z-index comes in. It’s like giving each element a number, and the higher the number, the closer it gets to you (and the further it gets from the screen). Here’s what it looks like:



Z-Index Stacking

```
<div style="position: relative; z-index: 2; background: red; width: 100px; height: 100px;"></div>
<div style="position: relative; z-index: 1; background: blue; width: 100px; height: 100px; margin-top: -50px; margin-le
```

Look at that! Now the red box is on top, even though it came first in our HTML. That’s the power of z-index. But here’s the catch: z-index only works on positioned elements. That means you need to set position to something other than static (like relative, absolute, or fixed) for z-index to do its thing.

Stacking Context

Okay, now here's where it gets a bit tricky. Sometimes, elements form what we call a "stacking context". It's like creating a new stack of transparent sheets that all move together.

A stacking context is a three-dimensional conceptualization of HTML elements along an imaginary z-axis relative to the user. Within a stacking context, child elements are stacked according to the same rules, but the context as a whole is considered a unit in the parent stacking context.

Creating Stacking Contexts

So, how do you create these stacking contexts? There are a bunch of ways, but here are the most common:

- Give an element a z-index and any position value except static.
- Set opacity to less than 1.
- Use transforms, filters, or clip-path.
- Use isolation: isolate (this one's handy if you don't want to mess with the element's position or appearance).

```
<div id="parent1" style="position: relative; z-index: 1;">
  Parent 1
  <div id="child1" style="position: absolute; z-index: 999999;">Child 1</div>
</div>
<div id="parent2" style="position: relative; z-index: 2;">
  Parent 2
  <div id="child2" style="position: absolute; z-index: 1;">Child 2</div>
</div>
```

In this example:

Both parent divs create their own stacking contexts due to having position: relative and a z-index. Child1 has a much higher z-index than Child2. However, Child1 will appear behind Parent2 and Child2, because its parent (Parent1) has a lower z-index than Parent2.

Flex and Grid Exception

An interesting exception to the positioning rule for z-index is that children of flex and grid containers can use z-index without needing to be positioned:

```
<div style="display: flex;">
  <div style="background: red; z-index: 1;">First</div>
  <div style="background: blue; z-index: 2; margin-left: -20px;">Second</div>
</div>
```

In this example, the blue div will appear on top of the red div due to its higher z-index, even though neither has a position set.

Isolation

The isolation property provides a way to create a new stacking context without changing the element's position or z-index

```
.new-context {
  isolation: isolate;
}
This is particularly useful for creating self-contained components that don't
interfere with the stacking order of other elements on the page.
```

Debugging Stacking Contexts

Debugging stacking context issues can be challenging. Here are some tools and techniques:

- Browser Dev Tools: Some browsers (like Microsoft Edge) offer 3D views of the stacking contexts.
- offsetParent: This JavaScript property can sometimes help identify the nearest positioned ancestor.
- VSCode extensions: Some extensions highlight when stacking contexts are created in CSS files.
- Browser extensions: There are extensions available for Chrome and Firefox that add information about z-index and stacking contexts to the developer tools.

Key Takeways

Understanding stacking contexts and z-index is crucial for creating complex layouts and resolving layout issues in CSS. Remember these key points:

- By default, elements stack in the order they appear in your HTML.
- z-index lets you control the stacking order, but only for positioned elements.
- Stacking contexts group elements together in the stacking order.
- z-index values only compete within the same stacking context.

Overflow

The `overflow` CSS property allows you to control how content is handled when it exceeds the boundaries of an element. It has a default value of `visible`.

This property is a shorthand for:

- `overflow-x`
- `overflow-y`

overflow keyword values

- `overflow: auto;`
- `overflow: hidden;`
- `overflow: scroll;`
- `overflow: visible;`
- `overflow: clip;`

overflow: auto;

`overflow: auto;` property makes an element scrollable when its content exceeds its bounds. Although the overflow content is clipped at the element's padding box, it can still be scrolled into view.

```
<div class="content">
  <strong><kbd>overflow: auto;</kbd></strong> property makes an element
  scrollable when its content exceeds its bounds. Although the overflow content
  is clipped at the element's padding box, it can still be scrolled into view.
</div>
```

```
.content {
  overflow: auto;
  border: 3px solid black;
  max-height: 100px;
  width: 100px;
}
```

overflow: auto; property makes an element scrollable when its content exceeds its bounds. Although the overflow content is clipped at the

overflow: hidden;

The `overflow: hidden;` property makes an element truncate its content when it overflows its boundaries. It behaves similarly to `overflow: scroll;`, but without displaying scrollbars. When `overflow: hidden;` is applied to an element, a scroll container is created without visible scrollbars.

```
.content {
  overflow: hidden;
  /* other styles */
}
```

overflow: hidden;
property causes an element truncate its content when it exceeds its boundaries, but the scroll container is still active so use the tab key to confirm.

- Track 1

overflow: scroll;

`overflow: scroll;` property causes an element overflow content to be scrolled into view using scroll bars. The scroll bars shows whether the content is going to overflow or not.

```
<div class="content">
  <strong><kbd>overflow: scroll;</kbd></strong> property causes an element
  overflow content to be scrolled into view using scroll bars.
</div>
```

```
.content {
  overflow: scroll;
  border: 3px solid black;
  max-height: 30px;
  width: 50px;
}
```

overflow: scroll; property causes an element overflow content to be scrolled

overflow: visible;

`overflow: visible;` property is the default setting for the `overflow` property. When overflow occurs outside the element's padding box, it will be displayed.

```
<div class="content">
  <strong><kbd>overflow: visible;</kbd></strong> property is the default setting
  for the <strong><kbd>overflow</kbd></strong> property. When overflow occurs
  outside the element's padding box, it will be displayed.
</div>
```

```
.content {
  overflow: visible;
  border: 3px solid black;
  max-height: 23px;
  width: 50px;
}
```

overflow: visible; property causes an element overflow

Before we move into the last value which is `overflow: clip;` let's learn about:

Scroll Containers

Whenever we set `overflow` property to `scroll`, `hidden`, or `auto` we automatically create what we referred to as a scroll container which manages overflow in both directions(`overflow-x` `overflow-y`).

A scroll container acts like a portal to a confined space. Any element within a scroll container is effectively trapped inside, ensuring it won't overflow beyond the boundaries of the container's four corners.

You can think of a scroll container as a "magical big box" that is confined within a specific height. While the "box" itself has defined boundaries, the content inside it can move around (scroll) without ever spilling out beyond those boundaries.

This metaphor helps illustrate how the scroll container behaves — it allows you to see different parts of its content by scrolling, but it keeps everything neatly contained within its fixed dimensions.

overflow: clip;

`overflow: clip;` property causes element's content to clipped at the element's overflow clip edge. The content outside the clipped region is not visible, and also no addition of scroll container. This work exactly the way most developers think `overflow: hidden;` should work.

```
.content {
  overflow: clip;
  /* other styles */
}
```

The content outside the clipped region is not visible, and also no addition of scroll container.

- Track 1
- Track 2
- Track 3

Horizontal Overflow

When you have inline elements that automatically wrap to the next line when they can't all fit within the container's width, and you want them to scroll horizontally instead, simply using the `overflow: auto;` property won't be sufficient. This is where the `white-space: nowrap;` property becomes useful.

`white-space` is a CSS property that allows developers to control how words and other inline or inline-block elements wrap.

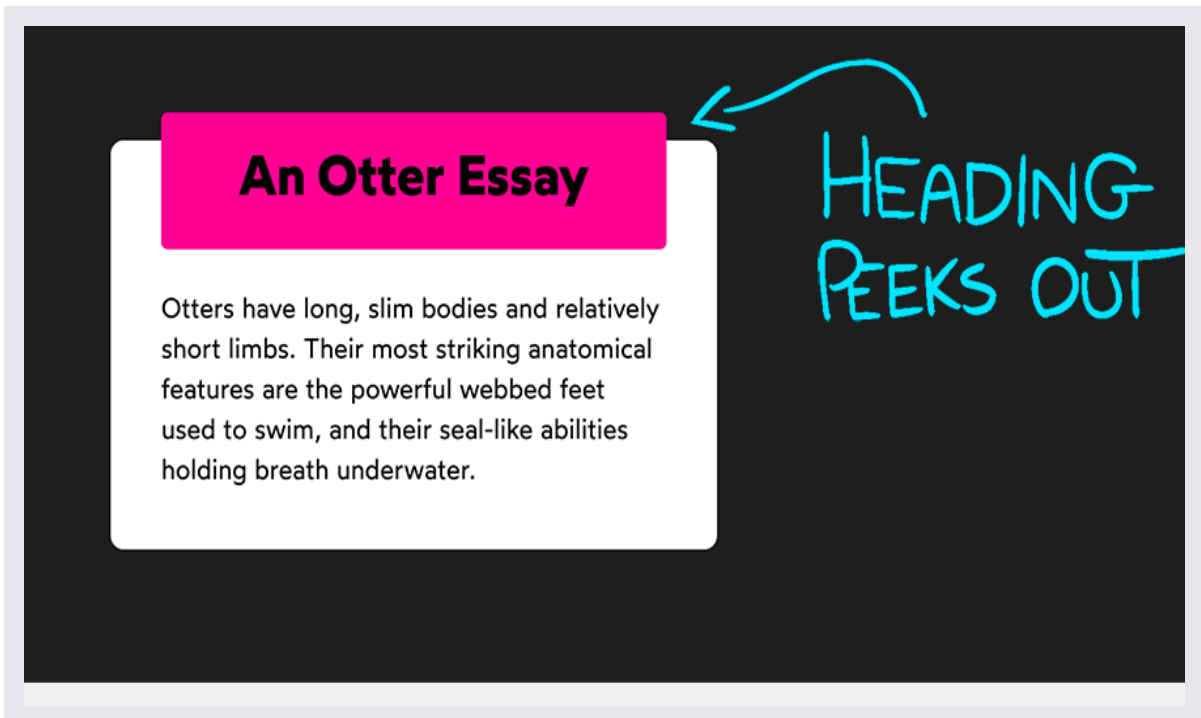
```
.img-wrapper {
  overflow: auto;
  white-space: nowrap;
  /* other styles */
}
```

Assignments

- Assignment 1
- Assignment 2
- Assignment 3
- Assignment 4
- Assignment 5
- Assignment 6

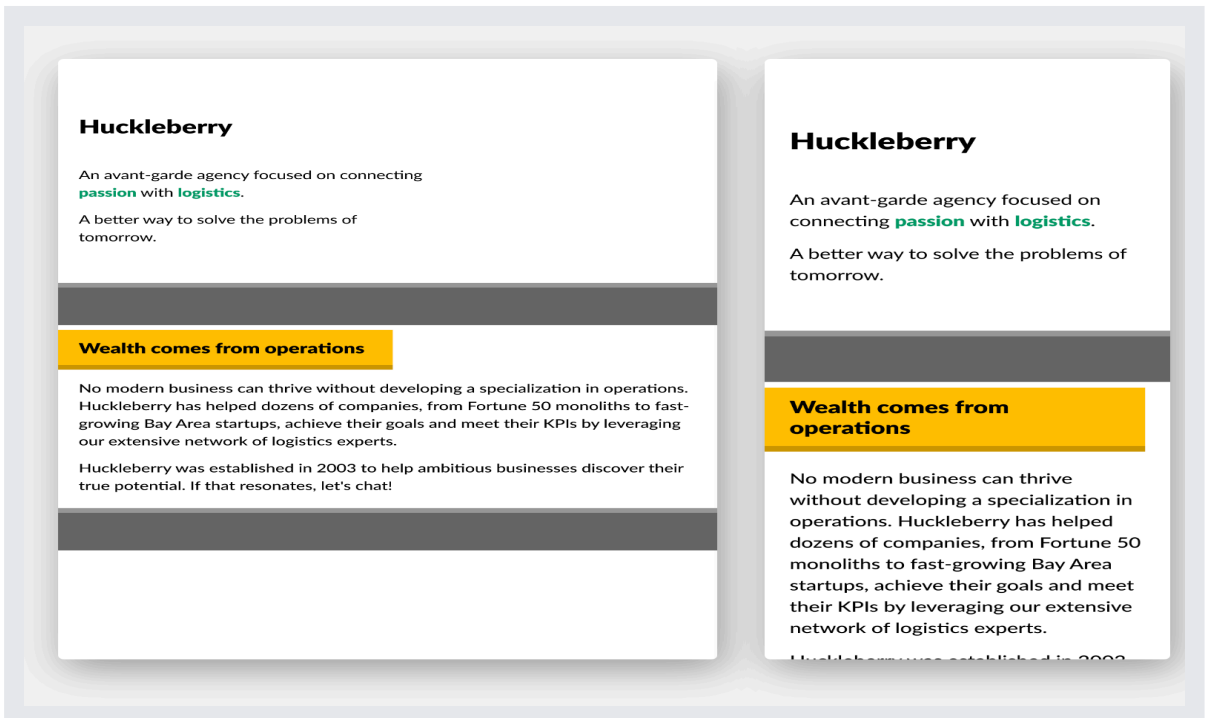
Assignment 1

Convert the design in the image (without the arrow and heading peeks out) to HTML and CSS.



Assignment 2 (Huckleberry)

Convert the design in the image to HTML and CSS. Check this figma file for pixel perfect design.

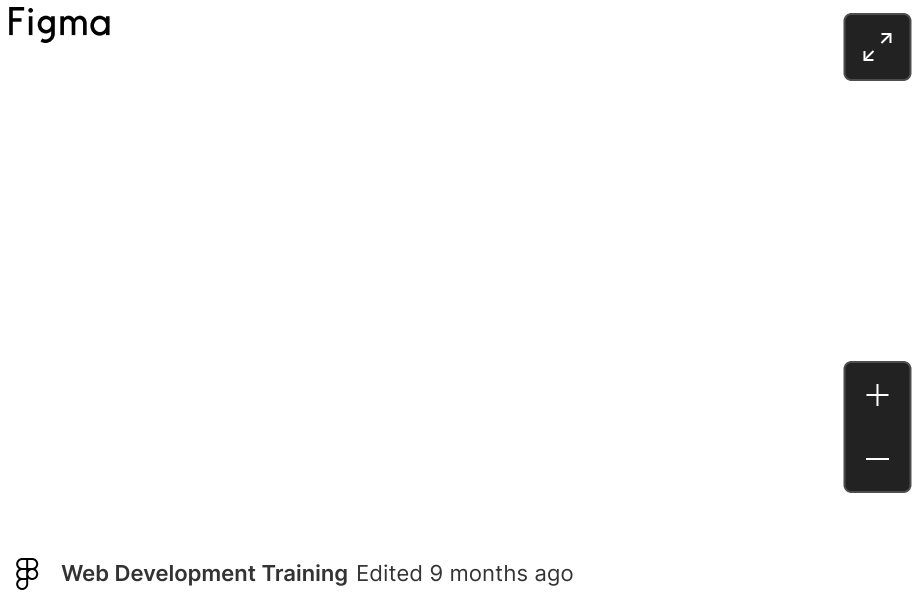


Assignment 3 (Novus Watch)

Convert the design in the image to HTML and CSS.

Assignment 4 (Web Development training)

Convert the design in the figma to HTML and CSS. Check this figma file for pixel perfect design.



Assignment 5 (WP Pusher)

Convert the design in the figma to HTML and CSS. Check this figma file for pixel perfect design.

Assignment 6 (Scissors)

Convert the design in the figma to HTML and CSS. Check this figma file for pixel perfect design.

Figma

AltSchoolV2-Exam Edited 1 year ago

Important Links

- CSS Tricks
- MDN Web Docs
- Selector Game
- Selectors Explained
- Variable Fonts
- CSS Cascade
- Understanding % unit
- interactive guide to CSS Grid

Contributors

- Ridwan Adebosin
- Oluwibe Faith