**Ways of adding CSS**

CSS can be added to HTML documents in 3 ways:

* **Inline** - by using the **style** attribute inside HTML elements
* **Internal** - by using a **<style>** element in the **<head>** section
* **External** - by using a **<link>** element to link to an external CSS file

**Inline CSS**

* **Inline CSS** requires the style attribute placed **inside an HTML element**.
* To add inline CSS, you use a style attribute and place it inside the opening tag of an HTML element. Here's the syntax:

<element style="CSS property: value">

* **It will override any other CSS targeting the same elements**. For this reason, inline CSS is effective for targeting a single element with unique style properties.
* Let’s say you want to change the color of a key term in a paragraph to a bright orange and leave the other text as is.

|  |  |
| --- | --- |
| **inline.html** | **Result** |
| <!DOCTYPE html>  <html lang="en">    <head>      <meta charset="UTF-8" />      <title>Inline</title>    </head>    <body>  **<h1 style="color: blue">Style Me in Blue!</h1>**    </body>  </html> |  |

**Internal CSS**

Instead of being placed inside a style attribute, it is placed inside brackets and defined by a CSS selector. This rule set is then wrapped in **<style></style>** tags and found in the **head section** **of the HTML file**.

**Internal CSS** allows you to **style groups of elements at once** — rather than having to add the same style attributes to elements over and over again.

If you have a multi-page site and would like to make changes across your site, you would still have to open up each HTML file representing those pages and add or change the internal CSS in each head section (or you can use external CSS, which is usually even better).

|  |  |
| --- | --- |
| **Internal.css** | **Result** |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <title>Internal</title>    <style>      h1 {        color: red;      }    </style>  </head>  <body>    <h1>Style Me in Red!</h1>  </body>  </html> |  |

**External CSS**

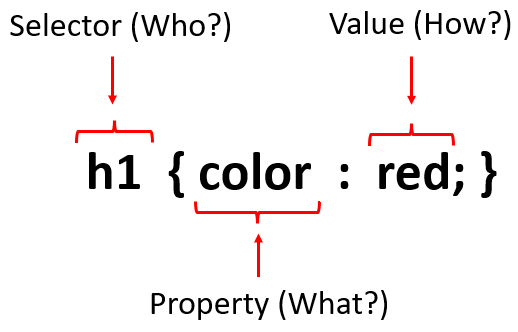
**External** **CSS** is formatted like internal CSS, but it isn’t wrapped in **<style>** tags or placed in the head section of your HTML file. Instead, it’s placed in an external file with the extension “.css”.

Using external CSS is considered the **best practice** for a few reasons. It’s also the fastest and most **SEO-friendly**. Storing CSS in another file makes your HTML file **easier** **to** **read** for search engines. It also enables a visitor’s browser to cache the CSS file to load your website faster for their next visit.

|  |  |
| --- | --- |
| **HTML File** | **CSS File** |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="UTF-8">    <title>External</title>    <link rel="stylesheet" href="./style.css" />  </head>  <body>    <h1>Style Me in Green</h1>  </body>  </html> | h1 {    color: green;  } |
| **Result** | |
|  | |

**Selector**

In CSS, **selectors** are patterns used to select the element(s) you want to style. See below the anatomy of CSS syntax used to style. In this example, we change the color of h1 to red.



There are different types of selectors:

* **Element selector:** selects a particular HTML tag. So, in the case below, it’s targeting all the paragraphs.

p {

  color: red;

}

* **Class selector:** selects all HTML elements with the specified class. The example below turns the background of the elements with the class “intro” into yellow.

|  |  |
| --- | --- |
| **index.html** | **Index.css** |
| <div class="intro">    <p>My name is Donald.</p>    <p>I live in Duckburg.</p>  </div>  <p>My best friend is Mickey.</p>  <p class="intro">My best friend is Mickey.</p> | .intro {    background-color: yellow;  } |
| **Result** | |
|  | |

* **ID selector:** ID selectors target elements based on the value of the id. The example below turns the background of the elements with the class “intro” into yellow.

|  |  |
| --- | --- |
| **index.html** | **Index.css** |
| <p id="para1">Hello World!</p>  <p>This paragraph is not affected by the style.</p> | #para1 {   text-align: center;   color: red; } |
| **Result** | |
|  | |

* **Attribute selector:** The attribute selector is used to select elements with a specified attribute. The following example selects all **<a>** elements with a target attribute:

|  |  |
| --- | --- |
| **index.html** | **Index.css** |
| <h2>CSS [attribute] Selector</h2>  <p>The links with a target attribute gets a yellow background:</p>  <a href="https://www.w3schools.com">w3schools.com</a>  <a href="http://www.disney.com" target="\_blank">disney.com</a>  <a href="http://www.wikipedia.org" target="\_top">wikipedia.org</a> | a[target] {   background-color: yellow; } |
| **Result** | |
|  | |

You can also select based on the value with the **[attribute="value"]** format, selecting elements with a specified attribute value. The following example selects all **<a>** elements with " attribute:

|  |  |
| --- | --- |
| **index.html** | **Index.css** |
| <h2>CSS [attribute="value"] Selector</h2>  <p>The link with target="\_blank" gets a yellow background:</p>  <a href="https://www.w3schools.com">w3schools.com</a>  <a href="http://www.disney.com" target="\_blank">disney.com</a>  <a href="http://www.wikipedia.org" target="\_top">wikipedia.org</a> | a[target="\_blank"] {   background-color: yellow; } |
| **Result** | |
|  | |

* **Universal selector**: The universal selector targets all elements. The example below selects all elements and set their background to yellow.

|  |  |
| --- | --- |
| **index.html** | **Index.css** |
| <h1>Demo of the \* selector</h1>  <div class="intro">    <p id="firstname">My name is Donald.</p>    <p id="hometown">I live in Duckburg.</p>  </div>  <p>My best friend is Mickey.</p> | \* {     background-color: yellow; } |
| **Result** | |
|  | |

The **\*** selector can also select all elements inside another element. The example below selects all elements inside **<div>** elements and set their background color to yellow:

|  |  |
| --- | --- |
| **index.html** | **Index.css** |
| <h1>Demo of the \* selector</h1>  <div class="intro">    <p id="firstname">My name is Donald.</p>    <p id="hometown">I live in Duckburg.</p>  </div>  <p>My best friend is Mickey.</p> | div \* {     background-color: yellow; } |
| **Result** | |
|  | |

Here is an example of a program using all 5 types of selectors:

|  |
| --- |
| **Index.html** |
| <body>    <h1>CSS Selectors</h1>    <h2>Applying CSS to Different Parts of HTML</h2>    <p class="note">      1. The element selector targets elements based on their HTML tag name.    </p>    <ol>      <li class="note" value="2">        Class selectors target elements based on the value of the class        attribute.      </li>      <li class="note" id="id-selector-demo" value="3">        ID selectors target elements based on the value of the id attribute.      </li>        <li class="note" value="4">        Attribute selectors target elements based on their attributes and        values.      </li>      <li class="note" value="5">        The universal selector targets all elements.      </li>    </ol>  </body> |
| **index.css** |
| ol {    margin-left: -40px;    margin-top: -20px;    list-style-position: inside;  }  p {    color: red;  }  .note {    font-size: 20px;  }  #id-selector-demo {    color: green;  }  li[value="4"] {    color: blue;  }  \* {    text-align: center;  } |
| **Result** |
|  |

|  |  |  |
| --- | --- | --- |
| **List of Selectors (and combinations)** | | |
| **Selector** | **Example** | **Example description** |
| [*.*class](https://www.w3schools.com/cssref/sel_class.asp) | *.intro* | *Selects all elements with* **class="intro"** |
| Selectorselector… | *.name1.name2* | *Selects all elements with both***name1***and***name2***set within its class attribute – Very Specific!* |
| *.class1 .class2* | *.name1 .name2* | *Selects all elements with***name2***that is a descendant of an element with***name1** |
| [*\**](https://www.w3schools.com/cssref/sel_all.asp) | *\** | *Selects* ***all*** *elements* |
| [element](https://www.w3schools.com/cssref/sel_element.asp) | *p* | *Selects all* **<p>** *elements* |
| [selector,selector](https://www.w3schools.com/cssref/sel_element_comma.asp) | *div, p* | *Selects all* **<div>** *elements and all <p> elements* |
| [selectorselector](https://www.w3schools.com/cssref/sel_element_element.asp) | *div p* | *Selects all* **<p>** *elements inside <***div>** *elements* |
| [*:hover*](https://www.w3schools.com/cssref/sel_hover.asp) | *a:hover* | *Selects links on mouse over* |
| [*:nth-of-type(*n*)*](https://www.w3schools.com/cssref/sel_nth-of-type.asp) | *p:nth-of-type(2)* | *Selects every* **<p>** *element that is the second* **<p>** *element of its parent* |
| selector > selector | *div > p* | *Selects the* **<p>** *element inside the* **<div>** *element (***<p>** *is direct child of* **<div>** *)* |

**Font Properties**

***Font Size***

The **font-size** property can have the following values:

* **px** (approx. 0.25mm)
* **pt** (approx. 0.35mm)
* **em** (equal to 100% of parent size) – which means that if you have a **h1** element inside a **div** (**div** is the **direct** **parent**), and you set the **div** to **20px** and **h1** to **1em**, **h1** will have **20px**. If you set **h1** to **2em**, **h1** will have **40px**.
* **rem** (equal to 100% of root). As this is relative to the root, this is an **independent** **way** **of** **setting** **the** **font** **size**, as the font size of the root is constant.

***Font-weight***

The **font-weight** property can have the following values:

* **normal/bold**
* **lighter / bolder (-100/+100)** – relative to the parent.
* **number (100-900)** – set the weight of the font within a range.

***Font Family***

The **font-family** property determines how the text will look like. This can include a specific font type that only exists in Mac computers (e.g. Helvetica) and a generic one in case the first one in case the first one doesn’t exist:

h1 {

  font-family: Helvetica, sans-serif

}

Also, if the font family has spaces, it needs to be inside double quotes:

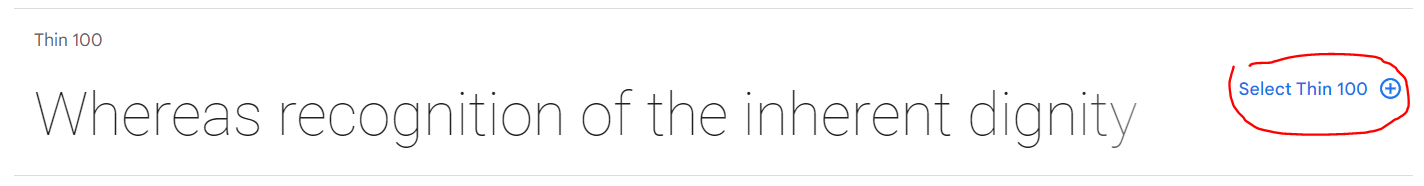
h1 {

  font-family: 'Times New Roman', serif

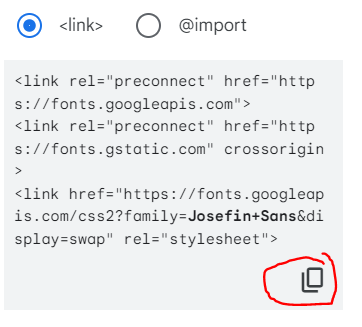
}

If you want to download a custom font:

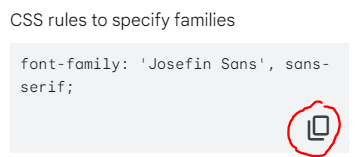
1. navigate to [www.fonts.google.com](http://www.fonts.google.com)
2. select the font you want
3. select the weight you want



1. Copy the Link.



1. Insert it inside your head element, just after your style tag (in case your have internal CSS).
2. Copy the CSS rule and paste it in your CSS, inside the element you want to style.



***Text Alignment***

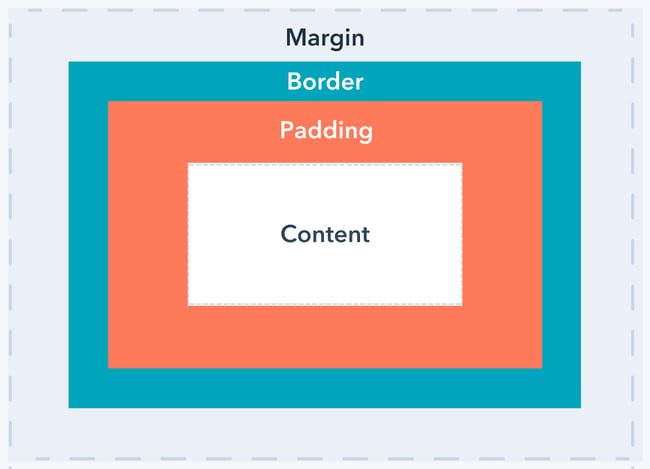
The **text-align** properties have the following properties:

* **Left** - Aligns the text to the left.
* **Right** - Aligns the text to the right.
* **Center** - Centers the text.
* **Justify** - Stretches the lines so that each line has equal width.
* **Initial** - Sets this property to its default value.
* **Inherit** - Inherits this property from its parent element.

**CSS Box Model**

**Introduction**

Every element in HTML is inside its own box with its own height and width, even though you can’t see without inspecting. CSS allows to change size and positioning of those boxes.



**Properties**

**Height**

The **height** property can have the following type of values:

* **px**
* **%** (0% - 100%) - percentage of the parent height

**Width**

The **width** property can have the following type of values:

* **px**
* **%** (0% - 100%) - percentage of the parent width

**Border**

The border property can take up to 3 separate values:

* **Thickness** of the border – expressed in px.
* **Style** – can be solid, dashed…
* **Color** – black, blue…

For example:

div {

  border: 30px solid black

}

Note that the border goes **outwards** from the dimensions of your existing box.

The 4 sides of the border can have different thicknesses:

* **border-bottom**
* **border-top**
* **border-right**
* **border-left**

You can also use **border-width** to specify the thickness of each side. In this case, it starts from the top and then goes clockwise.

div {

  border-width: 0px 10px 20px 30px

}

Or you could just set 2 values, which in this case the first would be top and bottom, and the second left and right.

div {

  border-width: 0px 30px

}

**Padding**

The **padding** property pushes the border out:

div {

  padding: 20px

}

The same rule applies that you can specify more than one value to affect each side of the box differently.

**Margin**

The **margin** property defines the space outside of the border

div {

  margin: 0px 30px

}

**Properties**

Boxes are created by element, and we can actually group different elements in a box. One common HTML element to wrap elements is **div**:

* The **<div>** tag defines a division or a section in an HTML document.
* The **<div>** tag is used as a container for HTML elements - which is then styled with CSS or manipulated with JavaScript.
* The **<div>** tag is easily styled by using the **class** or **id** attribute.
* Any sort of content can be put inside the **<div>** tag!

|  |  |
| --- | --- |
| **HTML file** | **Result** |
|  |  |

**The Cascade – Specificity and Inheritance**

**Position**

**Position** refers to whether if a rule is at a higher or a lower position. For example, in the code below, the text will be blue, because lower positions take priority over higher positions.

li {

  color:red;

  color:blue

}

Also, it also applies if the element is selected twice:

li {

  color:red;

}

li {

  color:blue;

}

**Specificity**

**Specificity** refers to how specific a selector is in terms of the elements that you're applying the CSS rule to. The most specific selector will have priority over the less specific ones.

The following list orders the specificity of selectors, from the least specific to the most specific.

|  |  |
| --- | --- |
| 1. Element 2. Class 3. Attribute 4. ID | li {    color: blue;  }  .first-class {    color: red;  }  li[draggable] {    color: purple;  }  #first-id {    color: orange;  } |

**Type**

The type refers to the way in which the CSS was applied. The list below shows from the least important to the most important:

1. External CSS
2. Internal CSS
3. Inline CSS

**Importance**

The importance keyword makes that specific style have priority over everything else.

li {

  color: blue;

}

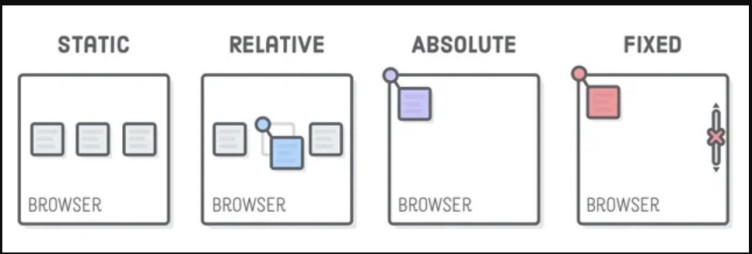
li {

  color: red !important;

}

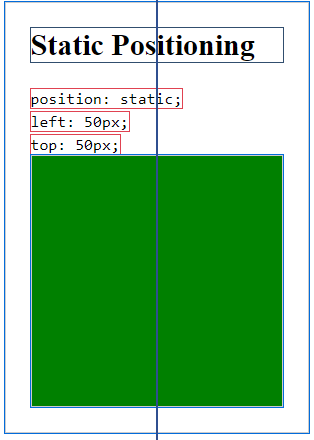
**Positioning**

There are 4 different values for the **position** property: **static**, **relative**, **absolute** and **fixed**.



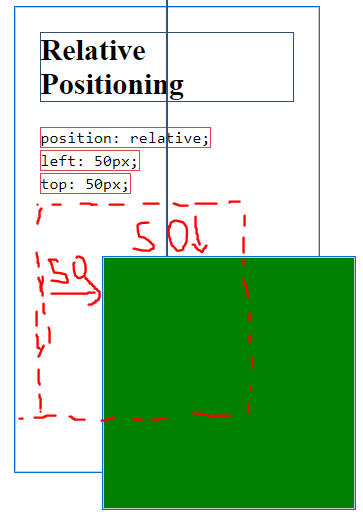
**CSS Positioning – Static**

* HTML elements are positioned static by **default**.
* An element with **static** **positioning** is **not positioned in any special way**; it is always positioned according to the **normal flow of the page**.
* Static positioned elements are not affected by the top, bottom, left, and right properties.
* If you add any properties like **left**, **right**, **bottom** and **top** it won’t have any effect on the positioning.



**CSS Positioning – Relative**

* An element with **relative position** is positioned **relative to its default position**.
* Setting the **top**, **right**, **bottom**, and **left** properties of a relatively-positioned element will cause it to be adjusted away from its normal position. **Other content will not be adjusted to fit into any gap left by the element**.

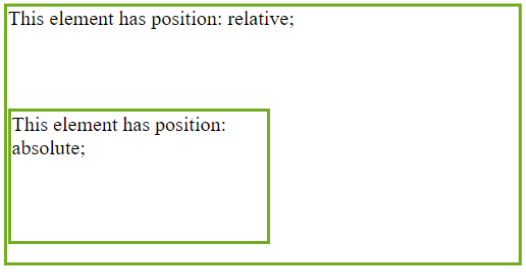


**CSS Positioning – Absolute**

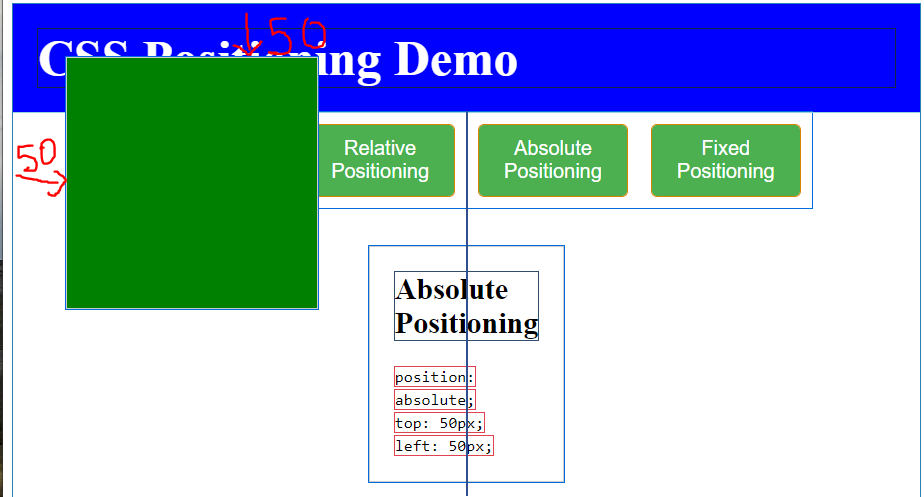
* An element with **absolute position** is positioned **relative to the nearest positioned ancestor**.
* If an absolute positioned element has no positioned ancestors, it uses the document body, and moves along with page scrolling.

**Note**: Absolute positioned elements are **removed from the normal flow**, and **can overlap elements**.

If you give relative position to a parent, and absolute position to a child, then you can then position the child inside the parent using top, left, right and bottom properties.

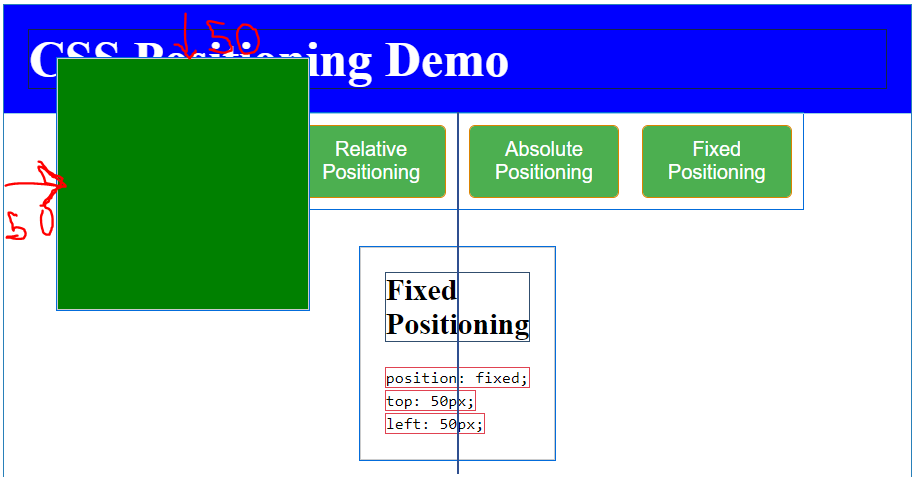


In the example below, the green block does not have an ancestor (or better, it’s ancestor is the body of the webpage).



**CSS Positioning – Fixed**

* An element with **fixed position** is positioned **relative to the top-left corner of the viewport**, which means it **always stays in the same place even if the page is scrolled**.
* The **top**, **right**, **bottom**, and **left** properties are used to position the element.
* A fixed element does not leave a gap in the page where it would normally have been located.



**Z-index**

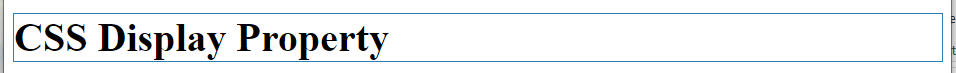
You can use the **z-index** property to define which elements go on top or behind.



**Display**

**Introduction**

Previously we've seen that when we create an HTML element without setting the width, it would take the full width of the screen.



CSS provides tools to not only change the size of the block, but also re-arrange them using display values: **inline, block-inline, block and none**.

**Display: inline**

* Displays an element as an inline element (like **<span>**).
* Any **height** and **width** properties will have **no effect**.

|  |  |
| --- | --- |
| <style>     p {       display: inline;     }  </style> | <body>      <h1>Hello</h1>      <h1>World</h1>    </body> |



**Display: block**

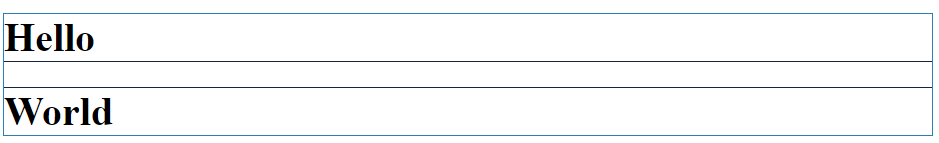
* Displays an element as a block element (like **<p>**).
* It starts on a new line, and takes up the whole width.
* Default value of display property.

  <body>

    <h1>Hello</h1>

    <h1>World</h1>

  </body>



**Display: inline-block**

* Displays an element as an inline-level container.
* The element itself is **formatted as an inline element**, but **you can apply height and width values**.
* If the width is small enough to fit on the screen (together with the previous containers sitting on the side), it will be displayed it will be displayed in-line with the other containers. If not, it goes to the following line.

|  |  |
| --- | --- |
| **HTML code** | **Style code** |
| <body>      <h1 class="first">Hello</h1>      <h1 class="second">World</h1>    </body> | <style>        h1 {          display: inline-block;        }      </style> |
| **Result – As there is space in the same line, both elements are inline** | |
|  | |

|  |  |
| --- | --- |
| **HTML code** | **Style code** |
| <body>      <h1 class="first">Hello</h1>      <h1 class="second">World</h1>    </body> | <style>        h1 {          display: inline-block;        }        .second {          width: 700px;        }      </style> |
| **Result – As there’s no more space, the 2nd heading goes to the 2nd line** | |
|  | |

**Display: none**

* Hiding an element can be done by setting the display property to **none**.
* The **element will be hidden**, and the page will be displayed as if the element is not there.

**Float**

The float property is used for positioning and formatting content, e.g. let an image float left to the text in a container.

The float property can have one of the following values:

* **left** - The element floats to the left of its container
* **right** - The element floats to the right of its container
* **none** - The element does not float (will be displayed just where it occurs in the text). This is default
* **inherit** - The element inherits the float value of its parent

In its simplest use, the float property can be used to wrap text around images.

|  |  |
| --- | --- |
| **HTML code** | **Style code** |
| <div></div>      <p>(some text)</p> | div {          width: 100px;          height: 100px;          background-color: blue;          float: left;        } |
| **Result – div floats to the left of the text** | |
|  | |

|  |  |
| --- | --- |
| **HTML code** | **Style code** |
| <div></div>      <p>(some text)</p> | div {          width: 100px;          height: 100px;          background-color: blue;          float: right;        } |
| **Result – div floats to the right of the text** | |
|  | |

We are not actually targeting the text with the CSS, but the div instead, because by default (normal CSS flow) the image block would take the whole width of the screen. So, other elements (like footers) might wrap around the image as well, which might not be what you want.

To solve that, there is a property called clear, which stops a specific element from wrapping around the float element.

|  |  |
| --- | --- |
| **HTML code** | **Style code** |
| <div></div>  <p>(some text)</p>  <footer>Copyright App Brewery</footer> | div {          width: 100px;          height: 100px;          background-color: blue;          float: right;        } |
| **Result – the footer is wrapping around the div** | |
|  | |

|  |  |
| --- | --- |
| **HTML code** | **Style code** |
| <div></div>  <p>(some text)</p>  <footer>Copyright App Brewery</footer> | div {    width: 100px;    height: 100px;    background-color: blue;    float: right;  }  footer {    clear: right;  } |
| **Result – the footer is wrapping around the div** | |
|  | |

**Responsive Websites**

The layout of your website should change depending on the size of the screen. It should fit desktops, laptops, IPads, mobile phones, etc… in order to do that, we need to make our websites responsive so it responds to the changes in the screen size.

|  |  |
| --- | --- |
|  |  |

To achieve this, there are 3 main ways:

* **Media Queries** – Makes your CSS change if certain conditions are met (for example, it will have a different appearance if the screen width is less than 700px)
* **CSS Grid** – To do 2D layouts that adapt to the size of the screen.
* **CSS Flexbox** - To do 1D layouts that adapt to the size of the screen.

**Media Queries**

**Max-Width**

Defines the CSS applied to **a screen smaller than a certain width**.

|  |  |
| --- | --- |
| div {     width: 200px;     height: 200px;     background-color: blue;  }  @media (max-width: 600px) {     div {      width: 100px;       height: 100px;     }  } |  |
|  |

**Min-Width**

Defines the CSS applied to **a screen smaller than a certain width**.

|  |  |
| --- | --- |
| div {     width: 200px;     height: 200px;     background-color: blue;  }  @media (min-width: 600px) {     div {      width: 100px;       height: 100px;     }  } |  |
|  |

**Operators**

It is also possible to **combine different media queries**.

Media queries support logical operators like many programming languages so that we can match media types based on certain conditions. The **@media** rule is itself a logical operator that is basically stating that “if” the following types and features are matches, then do some stuff.

* **and**: we can use the and operator if we want to target screens within a range of widths.

|  |  |
| --- | --- |
|  |  |

* **or**: We can also comma-separate features as a way of using an or operator to match different ones.

/\*

  Matches screens where either the user prefers dark mode or the screen is at least 1200px wide \*/

@media screen (prefers-color-scheme: dark), (min-width 1200px) {

  .element {

    /\* Styles! \*/

  }

}

* **not**: We can target devices by what they do not support or match. This declaration removes the body’s background color when the device is a printer and can only show one color.

@media print and ( not(color) ) {

  body {

    background-color: none;

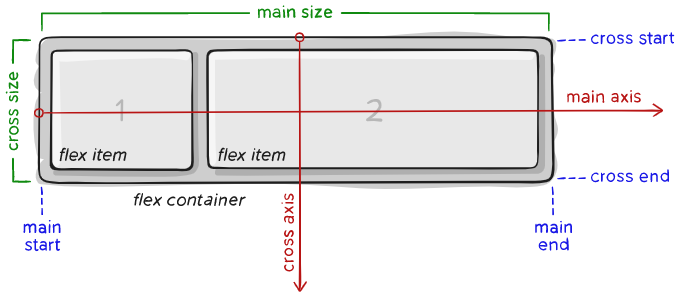
  }

}

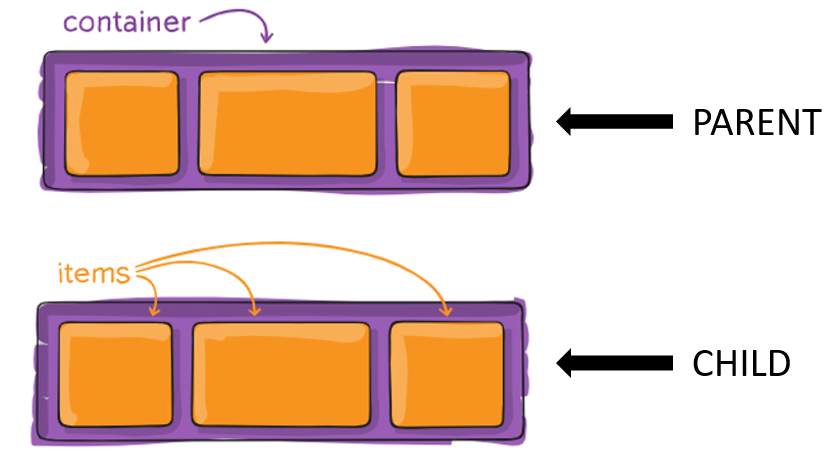
**Flexbox**

**Introduction**

* More **efficient** way to **lay out**, **align** and **distribute** space among items in a container, even when their size is unknown and/or dynamic.
* A flex container is **flexible**, **expanding items** and **changing their order** to fill available free space or **shrinks them to prevent overflow**.
* Please have a look at this figure from the specification, explaining the main idea behind the flex layout.



* When you make a parent a flex display, you have access to different flex properties to help to create complex and dynamic layouts, which some are only applicable to the **parent** (container), and some are only applicable to the **children** (items).



In order to show these properties, a simple CSS with 5 div’s was set up. By default, all elements (except span) have block display by default, which means they will occupy the whole screen width.

|  |  |
| --- | --- |
| **HTML Code** | **Result** |
| <div class="container">     <div class=" item item1">item1</div>     <div class=" item item2">item2</div>     <div class=" item item3">item3</div>     <div class=" item item4">item4</div>     <div class=" item item5">item5</div>  </div> |  |

**Flexbox – Properties**

The properties used for in CSS flexbox are:

|  |  |
| --- | --- |
| **Parent** | **Children** |
| * display * flex-direction * gap / row-gap / column gap * flex-wrap * justify-content * align-items * align content | * order * flex-grow * flex-shrink * flex-basis * align-self |

**Display: flex**

When you use Flexbox and you enclose all of the elements that you want to display inside a flexible container, **all of the previous display values will be ignored**.

Instead, we're going to have everything laid out by Flexbox and the **width of each of the elements inside the Flexbox will normally be based on the content size**. So, it's going to try and let you have all of your content displayed in one line.

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {    display: flex;  } |  |

**Display: inline-flex**

By default, if you set a parent container to be flex, then it’s going to be the width of the screen. Instead, you can use inline-flex and the parent container will be just wide enough to fit the items, so leaves space for other things can be on the side. Also, its **width is customizable**.

**Gap**

You can set a **gap** between each item. You can also use **row-gap** and **column-gap** to set them separately.

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {    display: flex;  gap: 30px;  } |  |

**Flex-direction**

By default, in Flexbox, as long as they fit in the row, if you keep adding items to the container they will keep stacking up in the right. This is because the **flex-direction** property is set to **row**. So, **row is the main axis by default**, and the **direction is from left to right**.

The perpendicular axis is called cross-axis.

If we set the flex-direction to column, we are changing the main axis to be vertical, and the cross axis to horizontal.

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {    display: flex;    flex-direction: column;  } |  |

**Flex-wrap**

By default, flex items will all try to fit onto one line. If you have too many items, it will keep pushing them until it runs out of space and, eventually, they won’t show up on the window.

You can change that and allow the items to wrap as needed with this property.

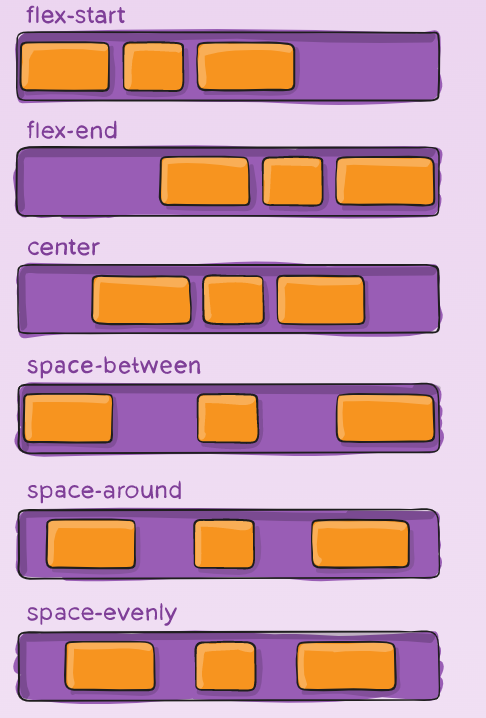
* **nowrap** (default): all flex items will be on one line.
* **wrap**: flex items will wrap onto multiple lines, from top to bottom.
* **wrap-reverse**: flex items will wrap onto multiple lines from bottom to top.

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {    display: flex;   gap: 30px;    flex-wrap: wrap;  }  .item {    width: 100px;  } |  |
| .container {    display: flex;    gap: 30px;    flex-wrap: wrap-reverse;  }  .item {   width: 100px;  } |  |

**Justify-content**

This defines the **alignment along the main axis**. It helps **distribute extra free space** leftover when either all the flex items on a line are **inflexible**, or are **flexible but have reached their maximum size**. It also exerts some control over the **alignment of items when they overflow the line**.

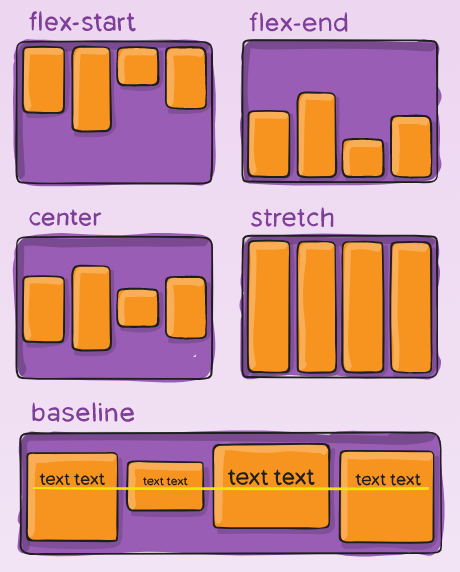
* **flex-start (default)**: items are packed toward the start of the flex-direction.
* **flex-end**: items are packed toward the end of the flex-direction.
* **center**: items are centered along the line.
* **space-between**: items are evenly distributed in the line; first item is on the start line, last item on the end line
* **space-around**: items are evenly distributed in the line with equal space around them. Note that visually the spaces aren’t equal, since all the items have equal space on both sides. The first item will have one unit of space against the container edge, but two units of space between the next item because that next item has its own spacing that applies.
* **space-evenly**: items are distributed so that the spacing between any two items (and the space to the edges) is equal.



**align-items**

This defines the default behavior for how flex items are **laid out along the cross axis** on the current line. Think of it as the justify-content version for the cross-axis (perpendicular to the main-axis).

* **stretch (default)**: stretch to fill the container (still respect min-width/max-width)
* **flex-start**: items are placed at the start of the cross axis.
* **flex-end**: items are placed at the end of the cross axis.
* **center**: items are centered in the cross-axis
* **baseline**: items are aligned such as their baselines align

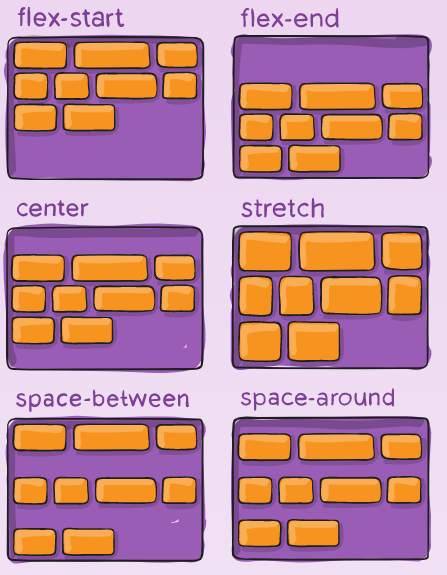


**align-content**

This aligns a flex container’s items when there is extra space in the **cross-axis**, similar to how **justify-content**aligns individual items within the main-axis.

This property **only takes effect on multi-line flexible containers**, where **flex-wrap** is set to either wrap or wrap-reverse). A single-line flexible container (i.e. where flex-wrap is set to its default value, no-wrap) will not reflect align-content.

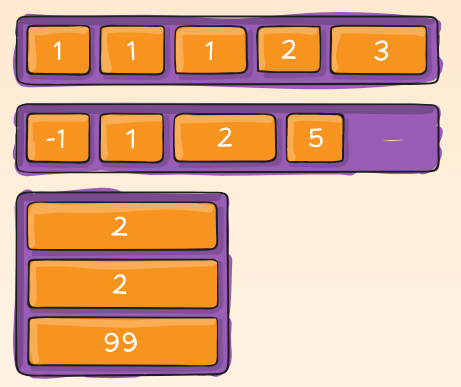
* **normal (default)**: items are packed in their default position as if no value was set.
* **flex-start**: items packed to the start of the container. The flex-start honors the flex-direction while start honors the writing-mode direction.
* **flex-end**: items packed to the end of the container. The flex-end honors the flex-direction while end honors the writing-mode direction.
* **center**: items centered in the container
* **space-between**: items evenly distributed; the first line is at the start of the container while the last one is at the end
* **space-around**: items evenly distributed with equal space around each line
* **space-evenly**: items are evenly distributed with equal space around them
* **stretch**: lines stretch to take up the remaining space



**Order**

By default, flex items are laid out in the source order (everything set to 0). However, the **order** property controls the order in which they appear in the flex container.

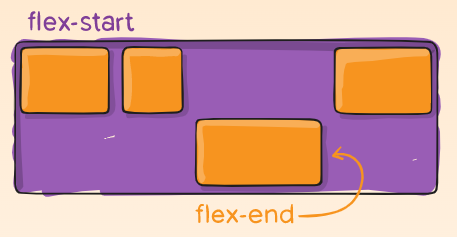
Items with the same order revert to source order.



**align-self**

This allows the default alignment (or the one specified by align-items) to be overridden for individual flex items.

* **stretch (default)**: stretch to fill the container (still respect min-width/max-width)
* **flex-start**: items are placed at the start of the cross axis.
* **flex-end**: items are placed at the end of the cross axis.
* **center**: items are centered in the cross-axis
* **baseline**: items are aligned such as their baselines align



**Flex-basis**

Sets the **width** of the items inside the container if **flex-direction: row**

Sets the **height** of the items inside the container if **flex-direction: column**

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {     display: flex;     justify-content: space-between;  }  .item {     flex-basis: 50px;  } |  |
| .container {     display: flex;     flex-direction: column;  }  .item {     flex-basis: 50px;  } |  |

The **default value** of **flex-basis** is **auto**, which means it's going to look at the amount of content in each of the items. So, it's going to **give more flex basis to the items with more content.**

If you don't want that and instead you want **everything to be equal width**, then you simply switch it off by making **flex-basis:0**.

**Flex-grow**

This defines the **ability for a flex item to grow** if necessary. It dictates what amount of the **available space inside the flex container** the item should take up.

It accepts a unitless value which is the **growth factor**.

If all items (or an item) have **flex-grow** set to **0**, then **they won’t grow**. This is the **default behavior**.

The items with this property (**flex-grow 0**) will **grow until they occupy the whole screen** unless there’s another property like **max-width** to stop them. If all items have flex-grow **set to 1**, the remaining space in the container will be distributed **equally to all children**.

If one of the children has a value of 2, that child would take up twice as much of the space either one of the others (or it will try, at least, if there is no other property limiting it).

In the example below, they all start with the same width because the window **width** is less than 500px. But as the window increases enough to let the items have 100px width, only **item2** and **item3** grow, and **item3** will grow at twice the rate of **item2**.

|  |  |
| --- | --- |
| **CSS Code** | |
| .container {    display: flex;    justify-content: space-between;  }  .item {  flex-basis: 100px;  } | .item2 {   flex-grow: 2;  }  .item3 {   flex-grow: 4;  } |
| **Result – small window** | |
|  | |
| **Result – big window** | |
|  | |

**Flex-shrink**

This defines the ability for a flex item to **shrink** if necessary.

The default value is 1.

In the example below, the items start shrinking as the window becomes too small to accommodate 5 blocks of 150px width. Then, **item2** and **item3** start shrinking at a faster rate than the other ones.

|  |  |
| --- | --- |
| **CSS Code** | |
| .container {    display: flex;    justify-content: space-between;  }  .item {  flex-basis: 150px;  } | .item2 {   flex-shrink: 2;  }  .item3 {   flex-shrink: 4;  } |
| **Result – big window** | |
|  | |
| **Result – small window** | |
|  | |

**Flex**

This is the shorthand for **flex-grow** (1st value), **flex-shrink** (2nd value) and **flex-basis** (3rd value) combined. The second and third parameters (flex-shrink and flex-basis) are optional. For example

* **flex: 1 1 200px**
* **flex:1** = **flex: 1 1 0**
* **flex:2** = **flex: 2 2 0**

The default value is **flex: 0 1 auto**

**Flexbox – Sizing**

**Priority List**

Flexbox uses a **priority list** to decide what happens to the items when the screen changes size. Below is that priority list, from the most important to the least important.

1. min-width / max-width
2. flex-basis
3. width
4. content-width

If you make a container flex, as you are shrinking the window **all the items will shrink by the same amount/rate by default**.

1. As it shrinks, all the items will shrink until they reach the **minimum-content width**. After that, the content will start disappearing. The minimum-content width is equal to the width of the longest word in the item (e.g. Programming in **item4**).

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {    display: flex;    gap: 10px;  } |  |
|  |

1. If, however, you have a **width** property set, they will going to look at that instead the **content-width**. So, for example, in the example below they will start shrinking as soon as the screen can’t fit 4 items of 200px each with 10px gap.

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {    display: flex;    gap: 10px;  }  .item {     width: 200px;  } |  |

If, we add **flex-basis**, Flexbox would completely **ignore the** **width** property, as **flex-basis is more important**. In the example below, the items would start shrinking as they hit 250px.

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {     display: flex;     gap: 10px;  }  .item {  ~~width: 150px;~~    flex-basis: 250px;  } |  |
|  |

As for the **max-width** and **min-width** properties, the default values are:

* **max-width**: the width which fits the content in one line.
* **min-width**: determined by the width of longest word of the item (in this case “Programming”.

|  |  |
| --- | --- |
|  |  |

**max-width** determines the **maximum width that the item is allowed to grow to**, even if you’ve got a flex-basis property (if **max-width < flex-basis**). In the example below, it will only allow the items to grow to 100px, and ignore flex-basis.

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {    display: flex;    gap: 10px;  }  .item {  ~~flex-basis: 200px;~~   max-width: 100px;  } |  |

**min-width** determines **how small the items can shrink to**. In the example below, if the items shrink up to 100px and the screen keeps shrinking, then they will start disappearing from the screen.

.container {

   display: flex;

   gap: 10px;

}

.item {

   max-width: 100px;

 }

If you use **min-width** together with flex-basis (width **flex-basis < min-width**) then it will ignore the **flex-basis** and respect the **min-width**.

**flex-grow and flex-shrink examples**

If **flex-grow = 0** and **flex-shrink=0**, we get items that don’t grow or shrink. They will just stay with the initial width of **100px** set by flex-basis.

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {    display: flex;    gap: 10px;  }  .item {    flex-basis: 100px;    flex-grow: 0;    flex-shrink: 0;  } |  |
|  |

If **flex-grow = 1** and **flex-shrink=0**, then the items will try to be 100px, but because there is space to grow, they will and **leftover space will be evenly distributed**.

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {    display: flex;    gap: 10px;  }  .item {    flex-basis: 100px;    flex-grow: 1;    flex-shrink: 0;  } |  |

If **flex-grow = 0** and **flex-shrink=1**, then we have the **default Flexbox behavior**. It **won’t grow if there is space available**, but it will **shrink after it reaches the flex-basis value**.

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {    display: flex;    gap: 10px;  }  .item {    flex-basis: 100px;    flex-grow: 0;    flex-shrink: 1;  } |  |
|  |

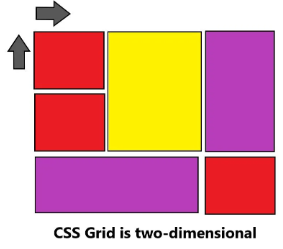
If **flex-grow = 1** and **flex-shrink=1**, then it will basically **ignore flex-basis**, because it will just **try to grow and shrink as much as possible**.

|  |  |
| --- | --- |
| **CSS Code** | **Result** |
| .container {    display: flex;  }  .item {    flex: 1 1 200px;  } |  |
|  |

If you set **max-width**, it will **grow until it reaches it**.

If you set **min-width**, it will **shrink until it reaches it**.

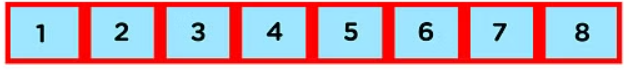
If these two are not set, then it's going to infer that from the content of those items.

**CSS Grid**

The CSS Grid is a powerful grid-based layout system in that makes it simple to design **complex** **responsive** web layouts across browsers using a two-dimensional grid.

**Flexbox vs Grid**

Flexbox helps me **align content along a one-dimensional line**, placing our items and defining how we want them aligned in a certain container in our page.



Grid is good to use to create nice and **organized general layouts**, and then you can use Flexbox to organize the content inside the grid containers. However, you can also have grid inside flexbox.

|  |  |
| --- | --- |
| Flexbox inside grid container | Grid inside a Flexbox |
|  |  |

Another difference is how the containers wrap when the screen changes size. While Grid wraps everything in a nice and organized way (keeping all the alignments), Flexbox doesn’t.

|  |  |
| --- | --- |
| **Flexbox** | **Grid** |
|  |  |

|  |  |
| --- | --- |
| **Flexbox** | **Grid** |
| 1-dimensional layouts | 2-dimensional layouts |
| Good to organize content within a container (center, spacing items, etc) | Good to do the general layout of a website |
| Wrapping doesn’t keep alignment | Wrapping keeps the alignment |

**CSS Grid – Special Units & Functions**

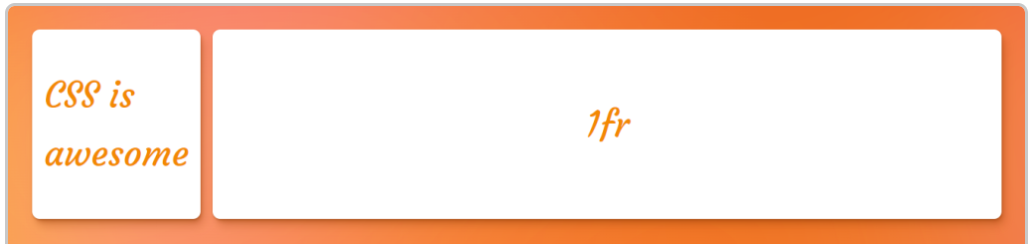
When sizing rows and columns, you can use all the lengths you are used to, like **px**, **rem**, **%**, etc, but you also have keywords:

* **fr units:** They essentially mean “portion of the remaining space”. So, a declaration like:

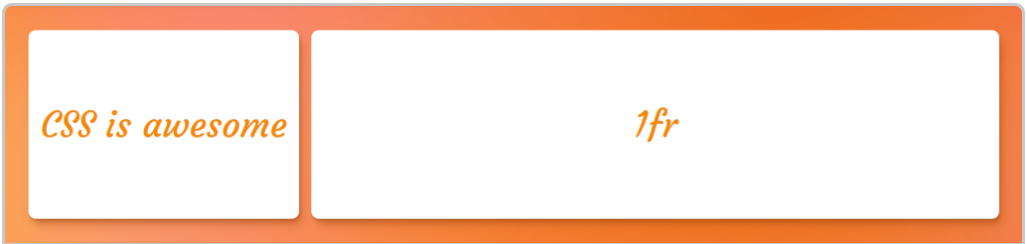
grid-template-columns: 1fr 3fr;

Means, loosely, 25% 75%. Except that those percentage values are much more firm than fractional units are.

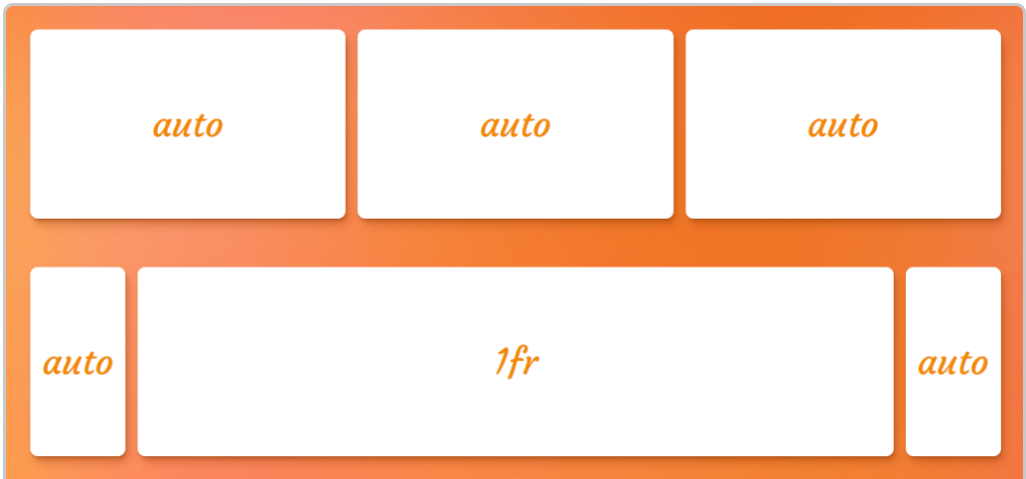
* **min-content**: the minimum size of the content. Imagine a line of text like “E pluribus unum”, the **min-content** is likely the width of the word “pluribus”, because it’s the longest word.



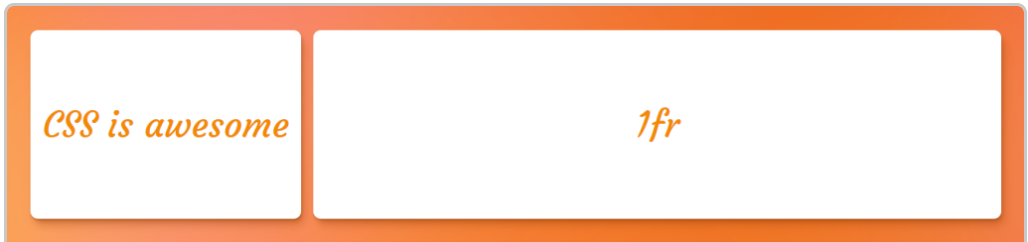
* **max-content**: the maximum size of the content. Imagine the sentence above, the max-content is the length of the whole sentence in one line.



* **auto**: makes the item occupy the space left (when using width) or the content height (when using height). It will lose against an 1fr though, leaving just enough width to fit the content.



* **fit-content**: use the space available, but never less than min-content and never more than max-content.



* **minmax:** sets a minimum and maximum value for what the size is able to be. This is useful for in combination with relative units. Like you may want a column to be only able to shrink so far.

grid-template-columns: minmax(100px, 1fr) 3fr;

**The repeat() Function and Keywords**

The **repeat()** function can save some typing:

grid-template-columns: 1fr 1fr 1fr 1fr 1fr 1fr 1fr 1fr;

/\* easier: \*/

grid-template-columns: repeat(8, 1fr);

But **repeat()** can get extra fancy when combined with keywords:

* **auto-fill**: Fit as many possible columns as possible on a row, even if they are empty.
* **auto-fit**: Fit whatever columns there are into the space. Prefer expanding

**CSS Grid – Sizing**

In order to show these properties, a simple CSS with 4 **div** was set up. By default, all elements (except **span**) have **block display** by **default**, which means they will **occupy the whole screen width**.

|  |  |
| --- | --- |
| **HTML Code** | **Result** |
| <div class="item item1">item1</div>  <div class="item item2">item1</div>  <div class="item item3">item1</div>  <div class="item item4">item4</div> |  |

**Grid-template-rows / grid-template-columns**

You can use **grid-template-rows** and **grid-template-columns** properties to define the layout of your container as shown below.

|  |  |
| --- | --- |
| **CSS code** | **Result** |
| .container {    display: grid;    grid-template-rows: 50px 100px;    grid-template-columns: 200px 300px;  } |  |

A shorthand for this would be grid-template: 50px 100px / 200px 300px;

The problem with using fixed sizes like pixels is that you layout is **not responsive**, they will always have that size.

A way to make it a bit more responsive is by using **auto**, which has 2 different effects depending on the property:

* grid-template-rows: make the cell stretch as much as it needs so the whole container takes 100 of the screen width.
* grid-template-columns: make the cell adapt to the content height.

|  |  |
| --- | --- |
| **CSS code** | **Result** |
| .container {    display: grid;    grid-template-rows: 50px auto;    grid-template-columns: 200px auto;  } |  |

Another very common way to make grid responsive is by using **fractions**. In the example below, no matter how much we change the screen size, the first column will have the half of the width of the first column, and the same happens with the row’s height:

|  |  |
| --- | --- |
| **CSS code** | **Result** |
| .container {    display: grid;    grid-template-columns: 1fr 2fr;    grid-template-rows: 1fr 2fr;  } |  |

Another sizing property is **minmax**, which lets us decide what is the minimum and maximum height we want our column to be.

|  |
| --- |
| **CSS code** |
| .container {     display: grid;     grid-template-rows: 100px 200px;     grid-template-columns: 100px minmax(200px, 400px);  } |
| **Result** |
|  |

You can use the **repeat** keyword to set column and row sizes that are all the same, by setting the size and the times you want to repeat. For example:

|  |  |
| --- | --- |
| .container {    display: grid;    grid-template-rows: 200px 200px;    grid-template-columns: 100px 100px;  } | .container {    display: grid;    grid-template-rows: repeat(2,200px);    grid-template-columns: repeat(2,100px);  } |

When the number of cells specified in the grid-template property is bigger than the number of items, the missing item spots are left empty.

|  |  |
| --- | --- |
| **CSS code** | **Result** |
| .container {    display: grid;    grid-template-rows: 50px auto;    grid-template-columns: 200px auto;  } |  |

When the number of cells specified in the grid-template property is smaller than the number of items, it pushes the leftover item to the next position with **auto height** and **width similar to the column where it is**.

|  |  |
| --- | --- |
| **CSS code** | **Result** |
| .container {    display: grid;    grid-template-rows: repeat(2, 100px);    grid-template-columns: repeat(2, 50px);  } |  |

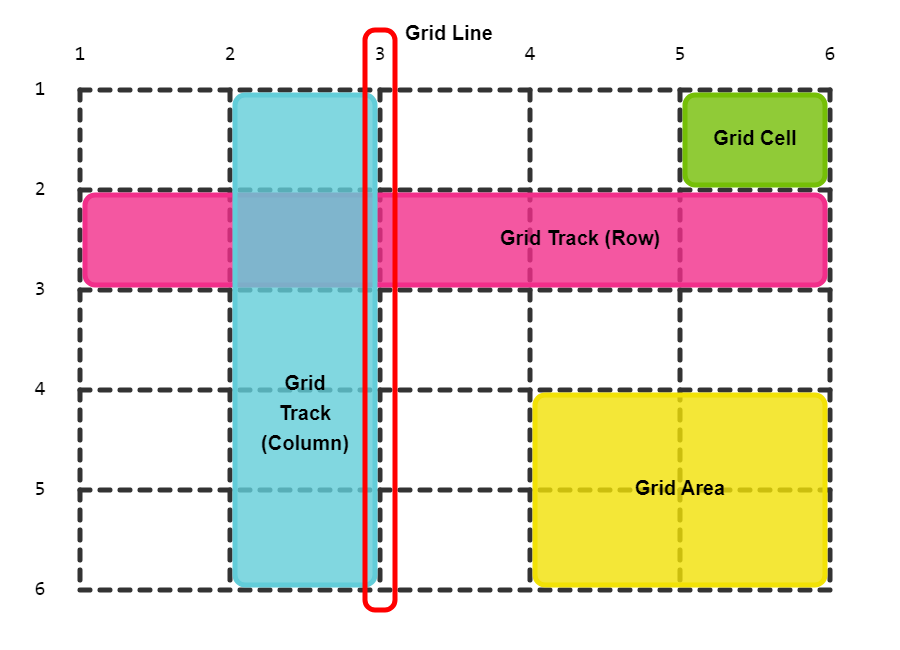
You can change the height of this extra row by using grid-auto-rows property. In the example below, item5 has 50px height.

|  |  |
| --- | --- |
| **CSS code** | **Result** |
| .container {    display: grid;    grid-template-rows: repeat(2, 100px);    grid-template-columns: repeat(2, 50px);    grid-auto-rows: 50px;  } |  |

**CSS Grid – Placement**

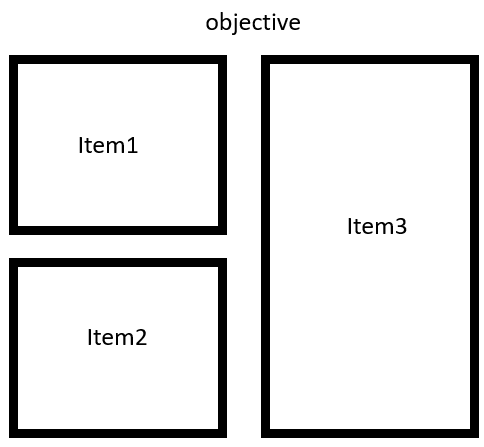
**Terminology**

* **Container**: contains all our items (html elements).
* **Items**: items inside the grid container.
* **Tracks**: rows and columns in our grid, sized using grid-template property.
* **Cell**: interceptions between a row and a column
* **Line**: line that separates the tracks. We can control the size of the grid lines using the gap property. We can’t change its color or add anything inside the lines.



**Example – initial code**

We will start with a simple grid display, with the aim of getting the layout below.



Notice that if we don’t add any positioning, the default is for the items to go from the top-left to the bottom-right.

|  |  |
| --- | --- |
| **CSS code** | **Result** |
| .container {     height: 100vh;     display: grid;     gap: 3rem;     grid-template-rows: 1fr 1fr;     grid-template-columns: 1fr 1fr 1.5fr;  } |  |

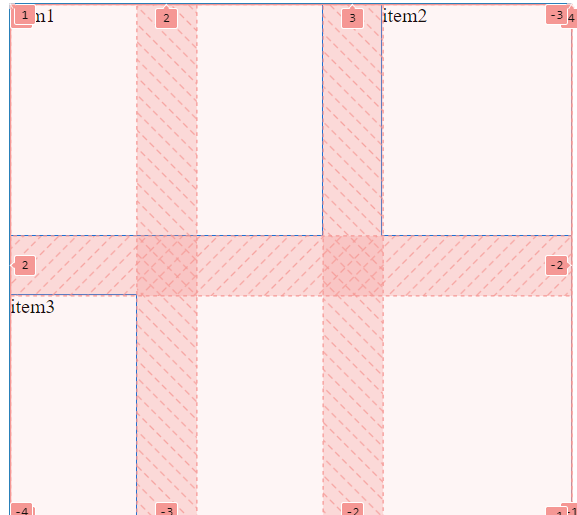
**Grid-Column / Span**

One way to merge two cells is by using grid-column and grid-row with span as value. You also have to specify how many cells you want the span to take, which in this case is 2.

|  |  |
| --- | --- |
| **CSS code** | **Result** |
| .container {     height: 100vh;     display: grid;     gap: 3rem;     grid-template-rows: 1fr 1fr;     grid-template-columns: 1fr 1fr 1.5fr;  }  .item1 {    grid-column: span 2;  } |  |

**grid-column-start / grid-column-end / grid-row-start / grid-row-end**

The same thing could be achieved with grid-column-start / end, which specify at which line the container will start and end.



|  |  |
| --- | --- |
| **Solution 1** | **Solution 2** |
| .item1 {     grid-column-start: 1;     grid-column-end: 3;  } | .item1 {    grid-column-start: 2 span;    grid-column-end: auto;  } |

**Order**

To get **item2** underneath **item1**, we have to swap the order (at the moment **item3** is the one that was pushed down).

By default, every item has order of 0. So, to set **item2** to last, we give it order of 1 (or any number bugger than 0).

|  |  |
| --- | --- |
| **CSS code** | **Result** |
| .container {     height: 100vh;     display: grid;     gap: 3rem;     grid-template-rows: 1fr 1fr;     grid-template-columns: 1fr 1fr 1.5fr;  }  .item1 {    grid-column: span 2;  }  .item2 {     order: 1;     grid-column: 2 span;  } |  |

Now to make **item3** occupy the cell below, we just make it span 2 rows.

|  |  |
| --- | --- |
| **CSS code** | **Result** |
| .container {     height: 100vh;     display: grid;     gap: 3rem;     grid-template-rows: 1fr 1fr;     grid-template-columns: 1fr 1fr 1.5fr;  }  .item1 {    grid-column: span 2;  }  .item2 {     order: 1;     grid-column: 2 span;  }  .item3 {    grid-row: 2 span;  } |  |

**Grid-area**

Another neat solution is to give the start and end line for each item. To do that, we don’t have to define all the 4 properties like below. Instead, we could just use **grid-area** property.

grid-area: grid-row-start / grid-column-start / grid-row-end / grid-column-end | itemname;

|  |  |
| --- | --- |
| **Solution 1 - long** | **Solution 2 - short** |
| .item3 {      grid-column-start: 1;      grid-column-end: 3;      grid-row-start: 2;      grid-row-end: 3;   } | .item3 {  grid-area: 2 / 1 / 3 / 3   } |