BRO Academy

Manual

Lesson 7

document flow

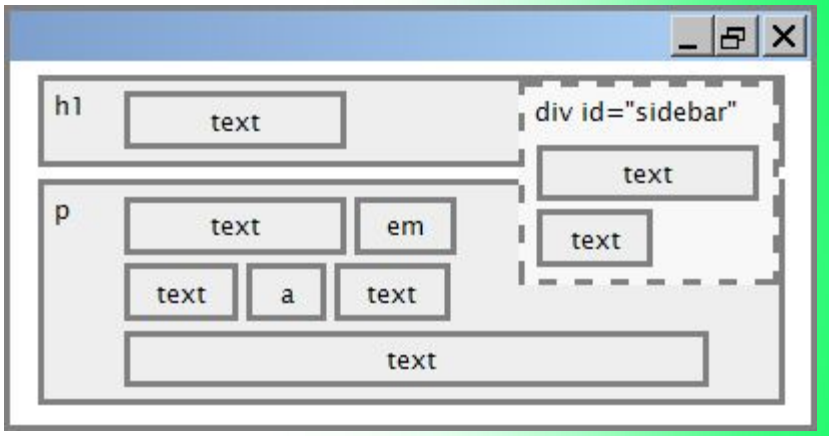
– is an abstract model of how elements on a webpage are arranged and interact with each other. It determines the order in which elements will be displayed on the page.
Browsers use special rules to define how items are displayed on the screen, which makes this possible.

- The standard flow runs from left to right, top to bottom. The flow is maintained in any element.

- The flow consists of two types of elements:
  – **inline**
  – **block**

Properties Affecting Document Flow

- display
- width, height
- position
- float
- column-width, column-count
- gap, column-gap, row-gap
- margin, padding, border



Content

block-level boxes – Occupy the entire available horizontal width and start on a new line

`display: block, list-item, table, flex, grid`

<blockquote>	<fieldset>	<h1>-<h6>			<section>
<dd>	<figcaption>	<header>		<p>	<main>
<dl>	<figure>	<footer>		<table>	<article>
<dt>	<form>	<aside>	<hr>	<nav>	

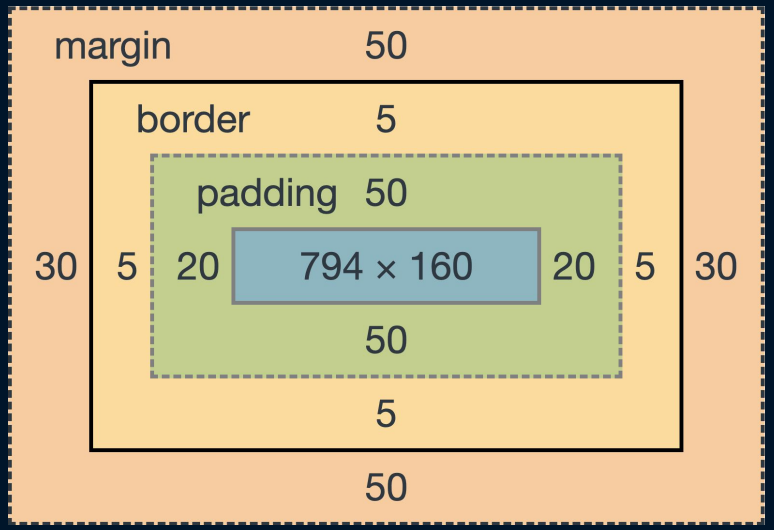
inline-level boxes – Occupy only the necessary horizontal space and are placed on the same line. They don't start on a new line and take up only as much space as they need.

`display: inline, inline-block, inline-table, inline-flex, inline-grid`

<a>	<time>	<i>	<label>		<cite>
	<kbd>	<q>	<textarea>	<sub>	<abbr>
<button>	<code>	<b>	<select>	<sup>	
<img>	 		<input>		

Box model – is essentially a box that wraps around every HTML element.

- content
- padding – The space inside the element, adding space within the element
- border
- margin – The space outside the element, pushing neighboring elements away.



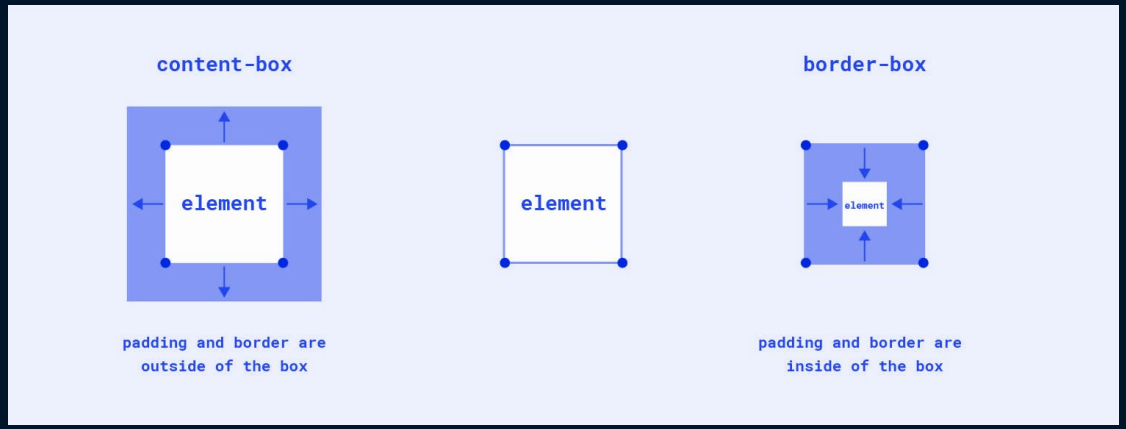
Demo

box-sizing property

controls how the box model is handled for the element it applies to.

Default:
`content-box` – width and height properties include the `content`, but **does not include** the `padding`, `border`, or `margin`.

`border-box` – width and height properties **include** the `content`, `padding`, and `border`, but **do not include** the `margin`.



padding

The padding CSS property is commonly used in CSS to add space in the inner side of an element.

Remember:

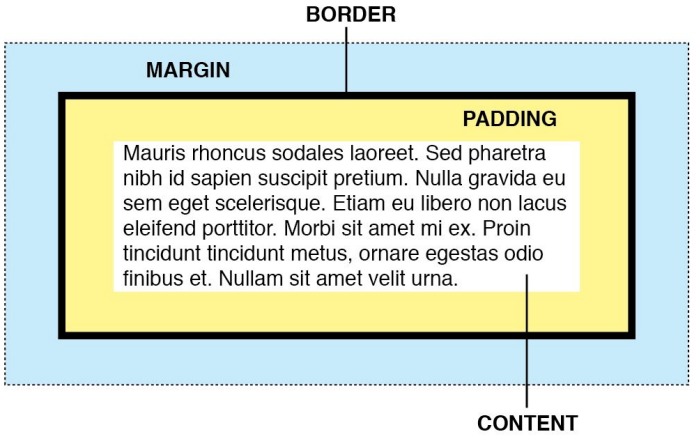
- margin adds space outside an element border
- padding adds space inside an element border

padding has 4 related properties that alter the padding of a single edge at once:

- `padding-top`
- `padding-right`
- `padding-bottom`
- `padding-left`

Units of measurement:

- `px`
- `%`



Example

```
padding-left: 30px;
padding-right: 3em;
```

`padding` is a shorthand to specify multiple padding values at the same time, and depending on the number of values entered, it behaves differently:

padding shorthands

1 value
Using a single value applies that to **all** the paddings:
top, right, bottom, left.

```
padding: 30px;
```

2 values
the first to **bottom & top**,
and the second to **left & right**.

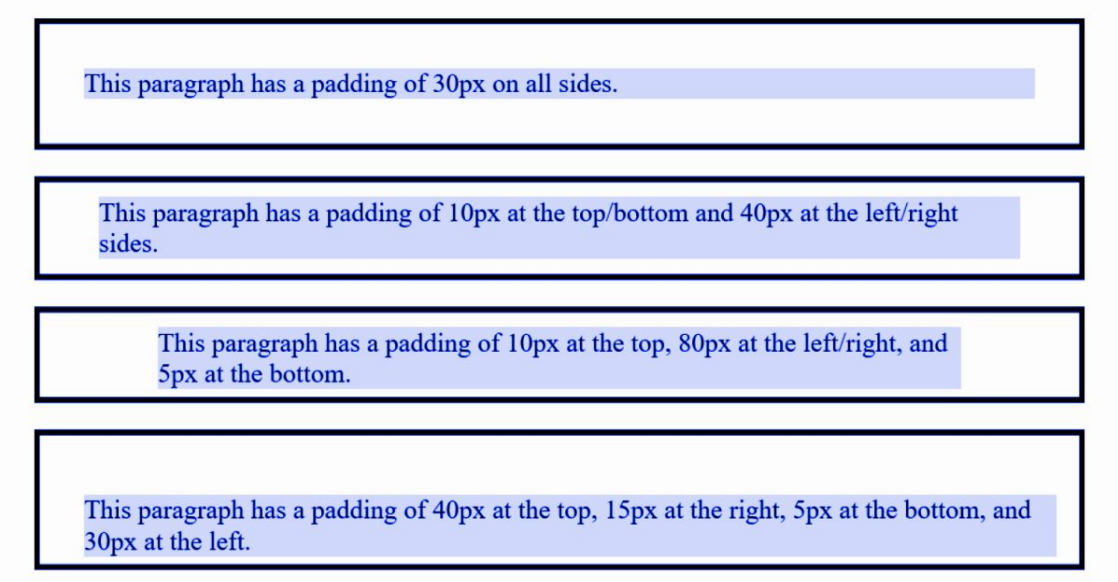
```
padding: 10px 40px;
```

3 values
the first to **top**,
the second to **left & right**,
the third to **bottom**.

```
padding: 10px 80px 5px;
```

4 values
the first to **top**,
the second to **right**,
the third to **bottom**,
the fourth to **left**.

```
padding: 40px 15px 5px 30px;
```



margin

The margin CSS property is commonly used in CSS to add space around an element.
Remember:

- margin adds space outside an element border
- padding adds space inside an element border

`margin` has 4 related properties that alter the margin of a single edge at once:

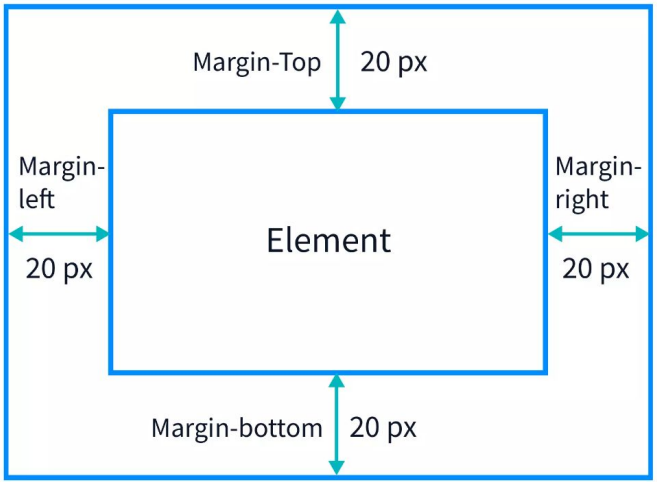
- `margin-top`
- `margin-right`
- `margin-bottom`
- `margin-left`

Units of measurement:

- **px**
- **%**
- **auto**

Can have negative values.

`margin-top` and `margin-bottom` do not work on inline-box



Example

```
margin-left: 30px;
margin-right: 3em;
```

margin

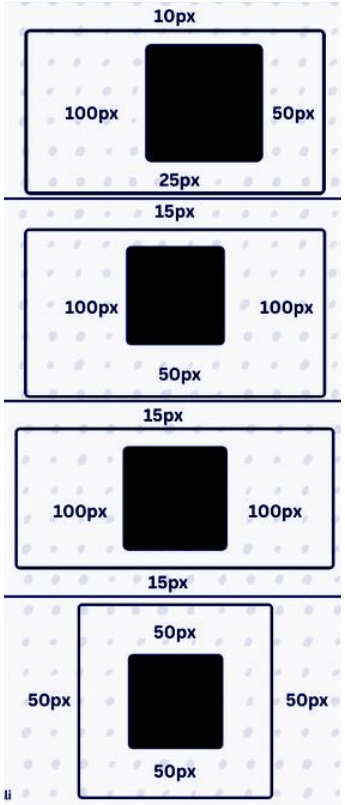
`margin` is a shorthand to specify multiple margins at the same time, and depending on the number of values entered, it behaves differently:

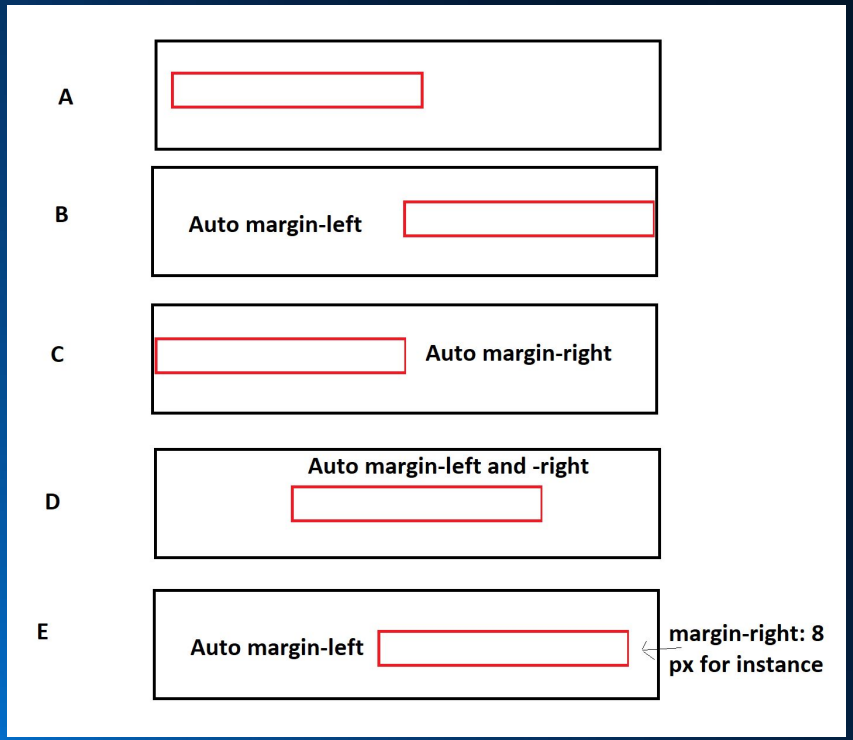
1 value
Using a single value applies that to **all** the paddings:
top, right, bottom, left.
```
margin: 50px;
```

2 values
the first to **bottom & top**,
and the second to **left & right**.
```
margin: 15px 100px;
```

3 values
the first to **top**,
the second to **left & right**,
the third to **bottom**.
```
margin: 15px 100px 50px;
```

4 values
the first to **top**,
the second to **right**,
the third to **bottom**,
the fourth to **left**.
```
margin: 10px 50px 25px 100px;
```





margin: auto

```
margin: auto;
margin-right: auto:
margin: 0 auto;
```

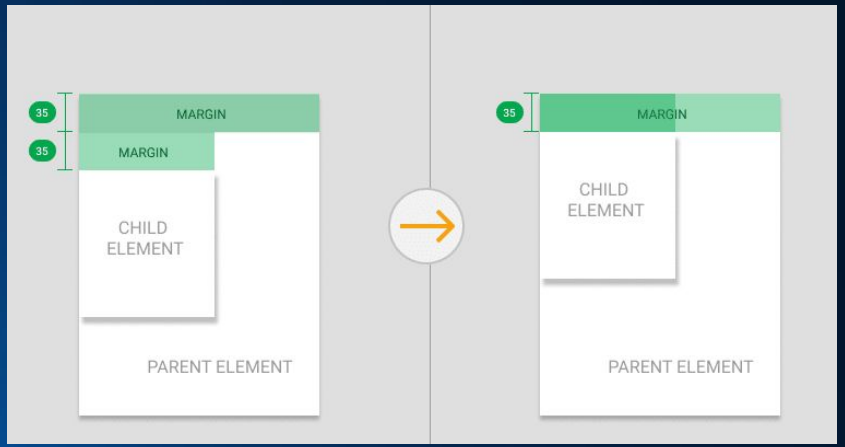
The width of the margin is automatically calculated by the browser and occupies all available space between elements and margins within the container.

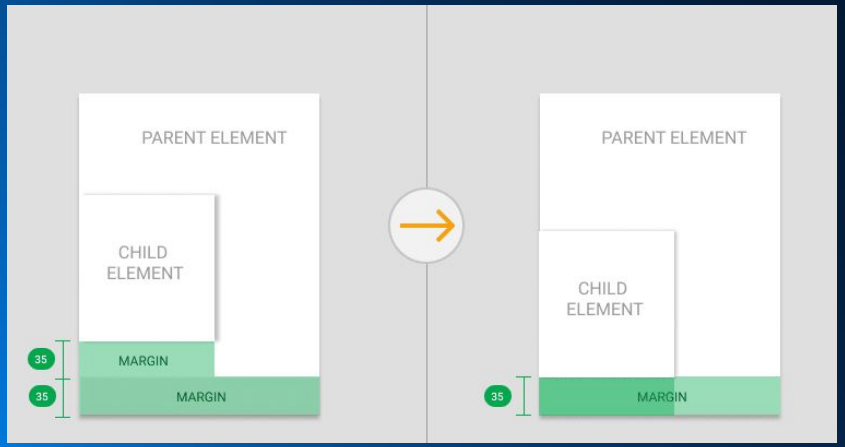
Collapsing Margins

- The concept of margin collapsing comes from graphic design. There, you have margins to title and subtitles, but when a subtitle comes just after the title you should not double the margins. This is why they developed the concept of collapsed margins and this is why it happens just on vertical margins.

- **When?**
  If there is one thing to remember when you leave this page is that collapsing only happens on vertical margins.

- In elements with `position: absolute` and `position: fixed`, `display: flex`, `display: grid`; `float` margins do not collapse or cause margin drop.



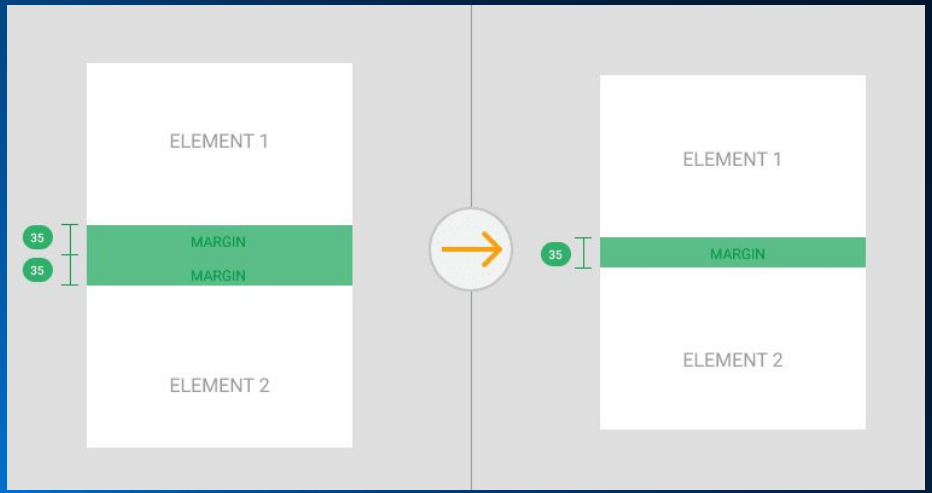


Collapsing Margins

- If **no** border, padding, inline part or clearance separate the margin-top of a parent to his first child, then it is going to collapse.
- The same happens with the last child of the parent.
- Works for the `margin-top` of the first child inside a parent and for the `margin-bottom` of the last child inside a parent.
- The child's `margin` becomes the parent's `margin` or the larger of the two applies.

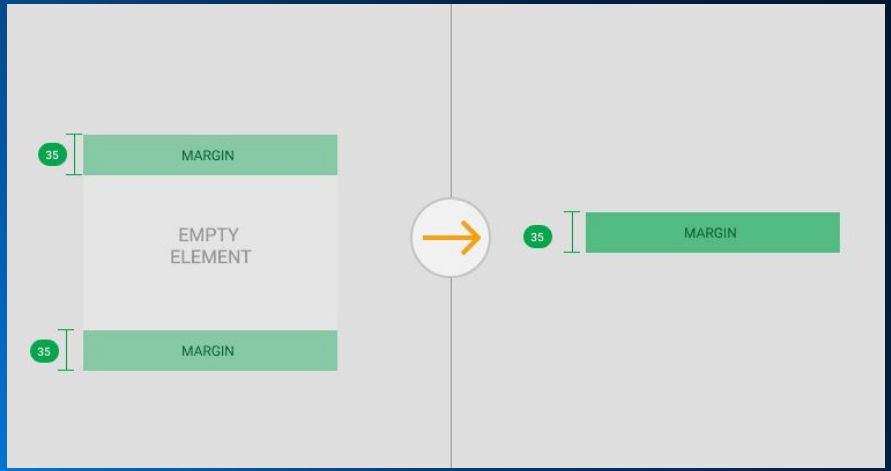
Collapsing Margins: Adjacent Siblings

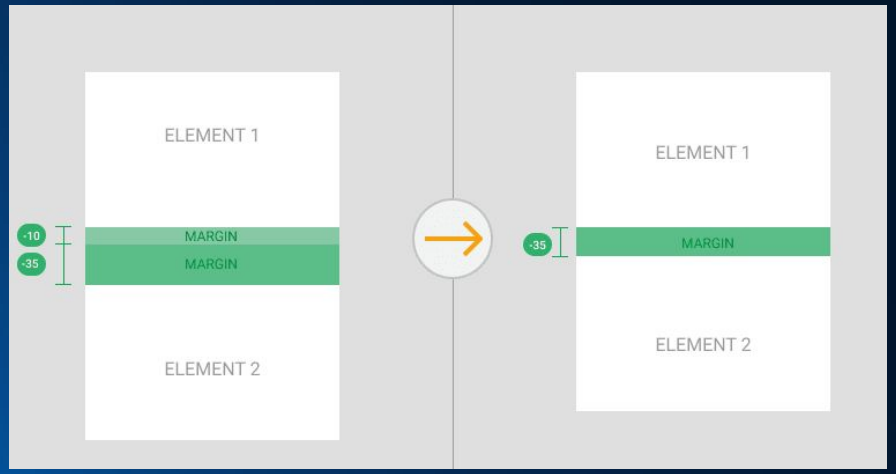
- When there are adjacent siblings, the margin-bottom of the top element collapses with the margin-top of the bottom element. The resulting size is the biggest between the two.

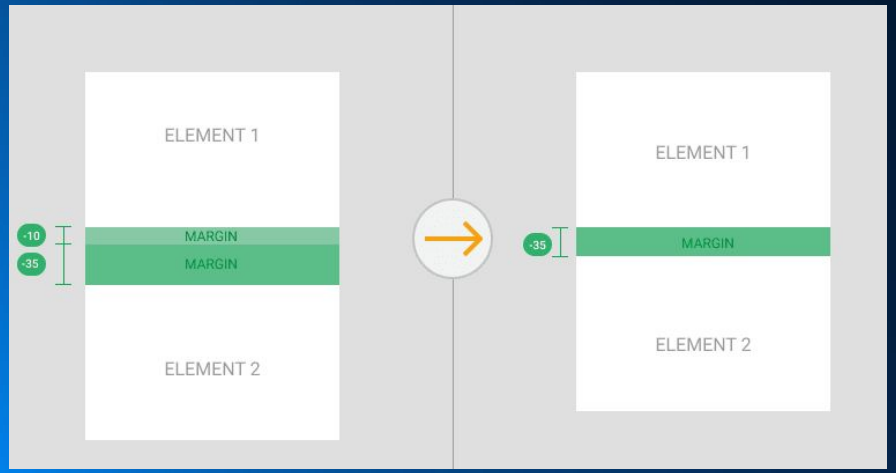


Collapsing Margins: Empty Blocks

- The margin-top and margin-bottom of an empty block collapse if there are no padding, border, inline content avoiding it. Also, it does not collapse if the block does not have a height or min-height attribute.







Collapsing Margins:

Negative Margins

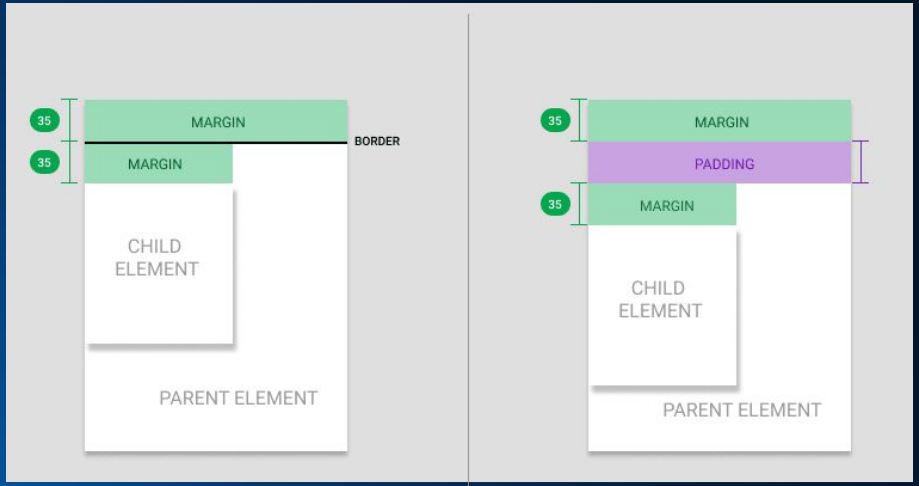
When it comes to negative margins, the resulting size depends on the signals involved.

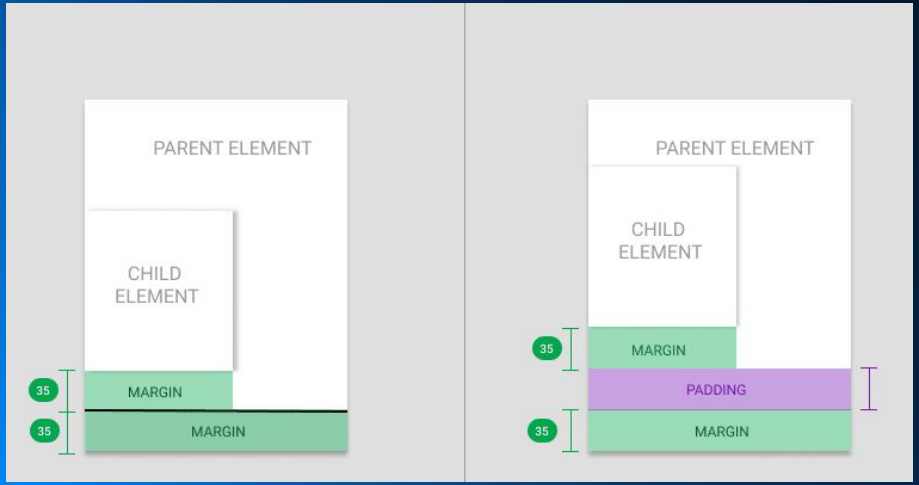
Negative + Positive

The resulting size of a negative margin and a positive margin is their algebric sum.

Negative + Negative

When both margins are negative, the resulting size is similar when there are positive + positive. In this, the most negative one wins.





Collapsing Margins:

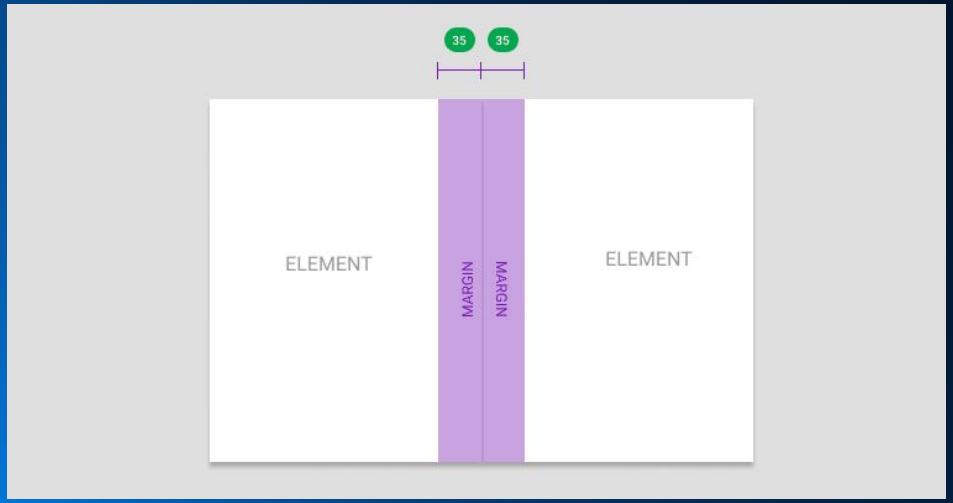
Border or Padding Property on the Parent Element

If there is a **border-top** or **padding-top** between the parent and the first-child.

The same happens if there is a **border-bottom** or **padding-bottom** between the parent and its last-child.

Collapsing Margins: horizontal margins

The horizontal margins never collapse.



logical properties

The CSS logical properties and values module defines logical properties and values that can control layout through logical rather than physical direction and dimension mappings. Logical properties define direction-relative equivalents to their corresponding physical properties. This means content translated into languages with different writing modes will be rendered as intended.

The start of a line is not always the left side of a line. Different writing systems operate in various directions.

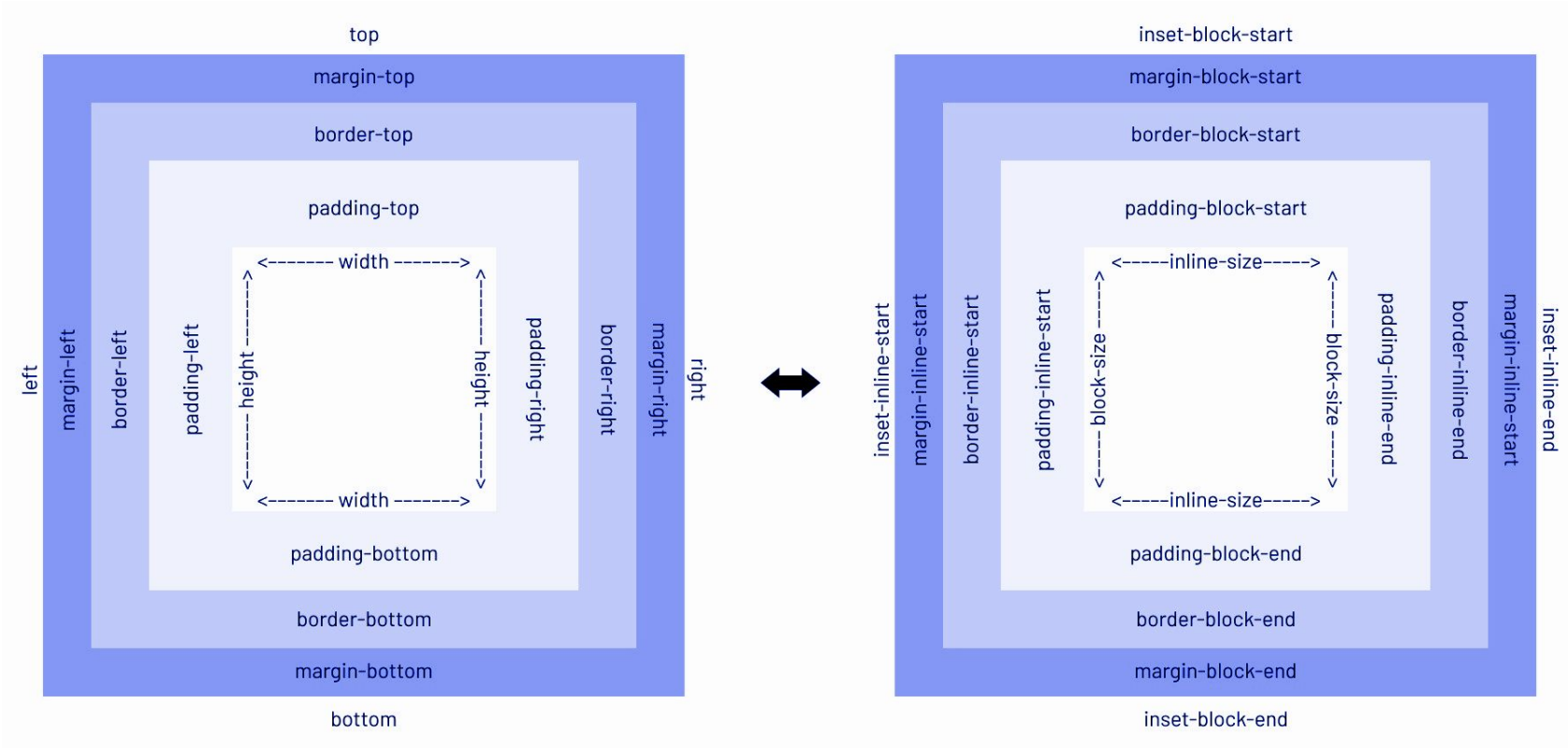
- English and Portuguese are written from left to right with new lines added below the previous ones.
- Hebrew and Arabic are right-to-left languages with new lines again being added below the previous ones.
- In some writing modes, the text lines are vertical, written from top to bottom. Chinese, Vietnamese, Korean, and Japanese are traditionally written vertically, from top to bottom, with each new vertical line added to the left of the previous one.
- Traditional Mongolian is also a top-to-bottom language, but new lines are to the right of previous ones.

The logical properties defined in this module enable defining properties relative to the content's writing direction, rather than a physical direction. This means content translated into languages with different writing modes will be rendered as intended.

The block dimension is perpendicular to the flow of text within a line, i.e., the vertical dimension in horizontal writing modes, and the horizontal dimension in vertical writing modes. For standard English text, it is the vertical dimension.

The inline dimension is parallel to the flow of text within a line, i.e., the horizontal dimension in horizontal writing modes, and the vertical dimension in vertical writing modes. For standard English text, it is the horizontal dimension.

physical / logical



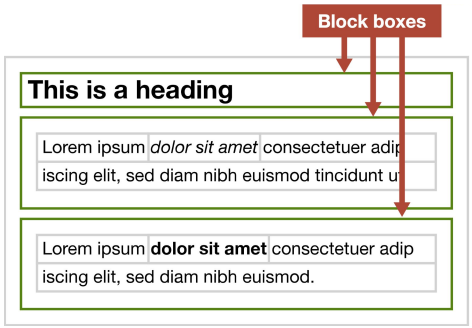
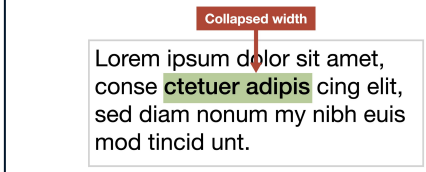
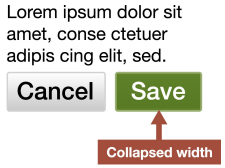
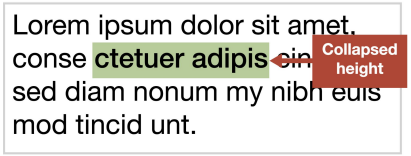
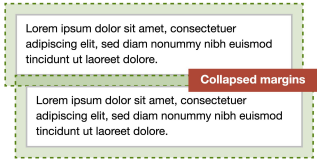
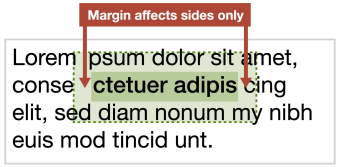
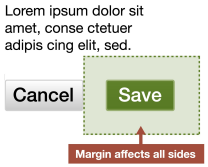
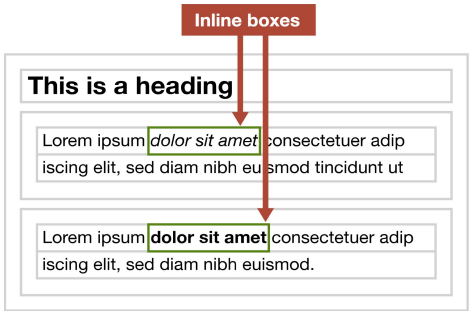
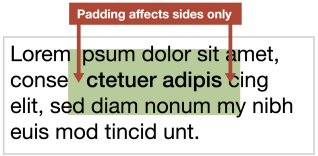
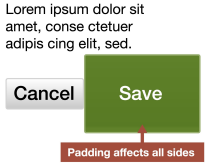
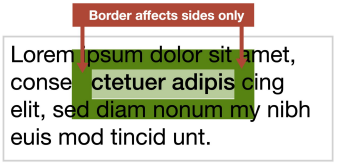
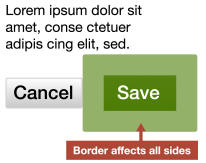
logical properties shorthands

Vertical
Using a single or double value applies that to **vertical** paddings/margins:
top + bottom

```
padding-block: 30px;
margin-block: 30px 10px;
```

Horizontal
Using a single or double value applies that to **horizontal** paddings/margins:
left + right.

```
margin-inline: 10px;
padding-inline: 10px 40px;
```

	block boxes	inline boxes	inline-block boxes
Default width			
Default height			
Margins			
Padding			
Border			

Demo

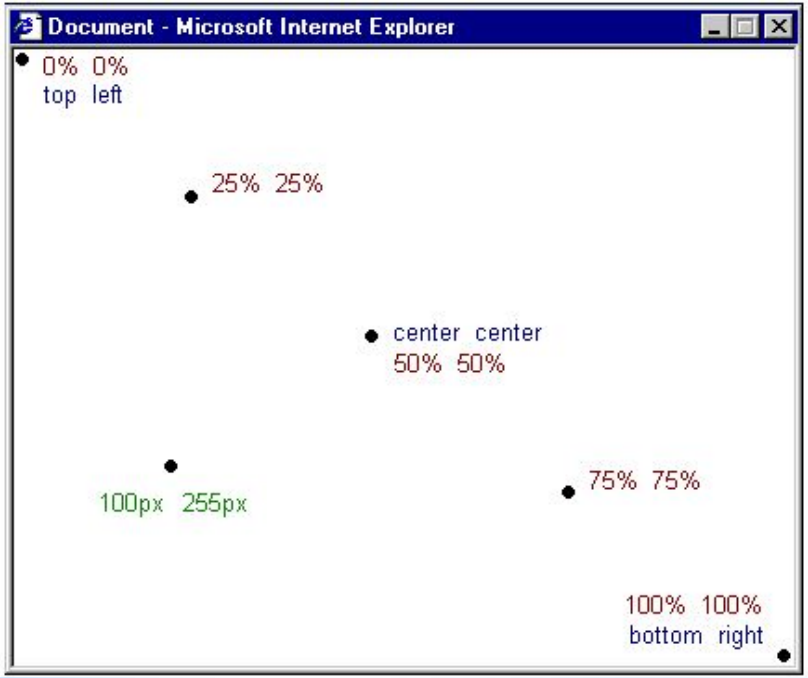
The `position` property specifies the type of positioning method used for an element:

- static
- relative
- absolute
- fixed
- sticky

- **Positioning should not be used for creating layouts, only for decorative purposes**

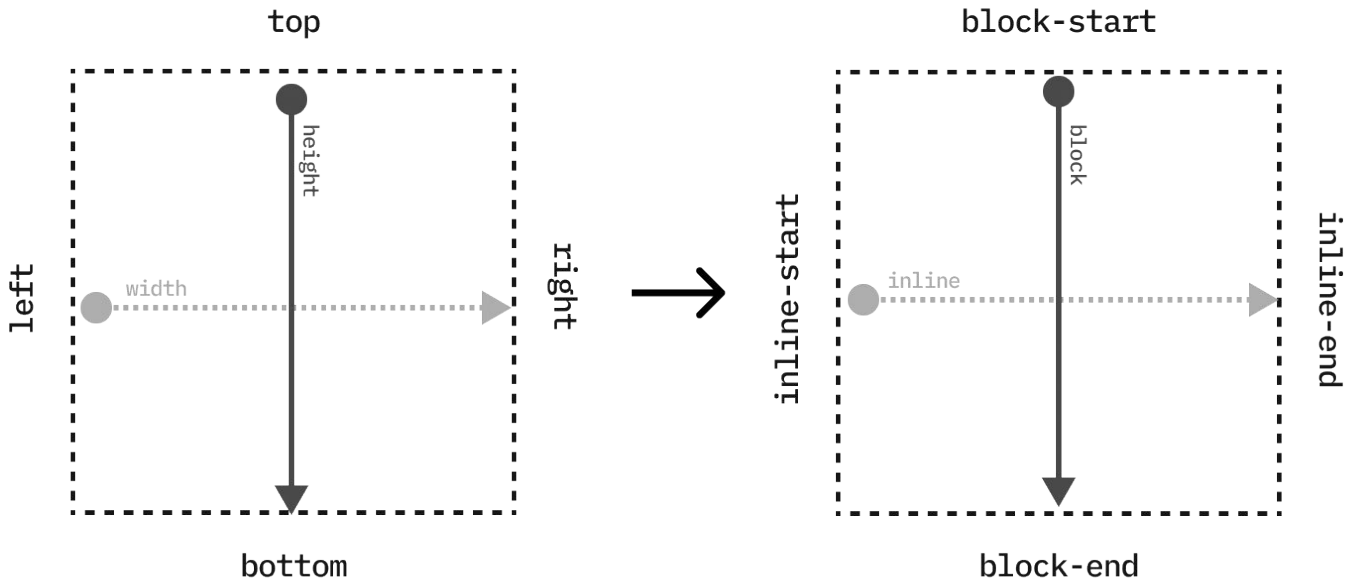
CSS POSITION

- **Do not use margin** to set the position of such an element.
- Units of measurement:
  - px
  - %
  - **top, right, bottom, left** (can be negative.)
  - **inset** – shorthand property for **top, right, bottom, left**
- should **always** have at least **one horizontal and one vertical** coordinate specified
- You can specify all four coordinates simultaneously, but there's no guarantee that they will all work together as expected.



CSS COORDINATE SYSTEM

logical properties coordinates



position:

static

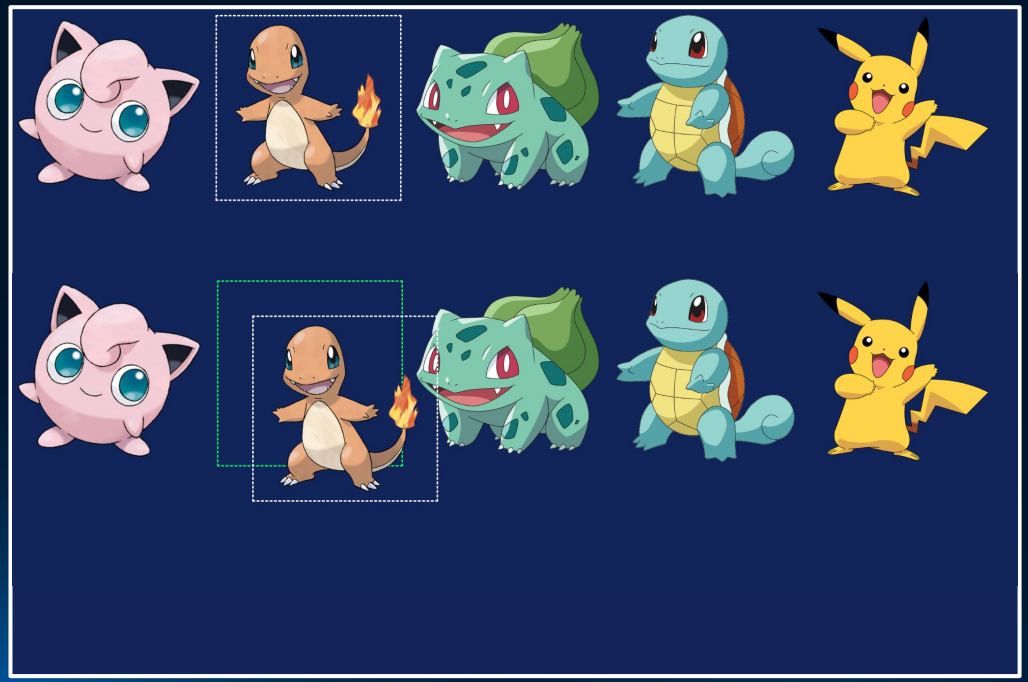
- This is the default value for all elements.
- Elements take their place in the flow depending on the box type and the amount of content.
- top, right, bottom, left, z-index don't work
- **normal document flow**

```
position: static;
```

position:

relative

- The element is **removed from the normal document flow.**
- It is positioned relative to its **original position in the flow.**
- Space for the element is **still reserved** in the document flow.
- The height is reset to height: auto
- normal document flow



```
position: relative;
left: 20px;
top: 20px;
```

position:

absolute

- The element is **removed from the document flow**, and neighboring elements completely ignore it.
- It is positioned **relative to its parent block** or
  the nearest ancestor with **position: relative**.
- The width defaults to width: auto, making it as wide as its content
  Padding and border are added to this width.
- If both horizontal or both vertical coordinates are specified, the element will stretch between them.

```
position: absolute;
left: 0;
top: 10px;
```

position:

fixed

- The element is removed from the document flow, and neighboring elements completely ignore it.
- It is positioned relative to the **viewport.**
- It does not move when **the page is scrolled.**
- The width defaults to width: auto, and the height to height: auto, making it sized according to its content. Note: padding, border are added to these dimensions.
- If both horizontal or both vertical coordinates are specified, the element will stretch between them.

```
position: fixed;
left: 20px;
top: 20px;
```

position:

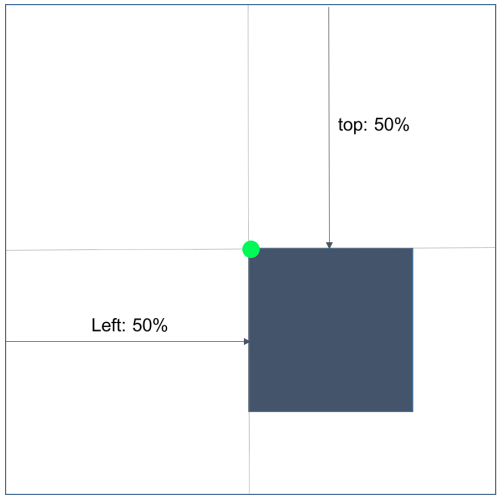
sticky

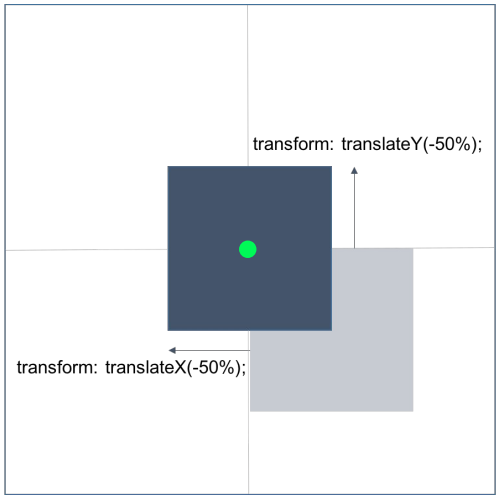
- The element is removed from the document flow.
- Neighboring elements behave as if the element is still in its original position, and space for the element is still reserved in the document flow.
- The element does not move when its containing **block is scrolled**.
- It behaves like position: relativeuntil the block is scrolled, after which it behaves like position: fixed
- The behavior of padding, border, margin depends on the box type—whether it is block-level or inline-level.

```
position: sticky;
right: 0;
top: 10px;
```

center position

Demo





Adds an offset upward by half the height of the block using:
`transform: translateY(-50%);`

Adds an offset to the left by half the width of the block using:
`transform: translateX(-50%);`

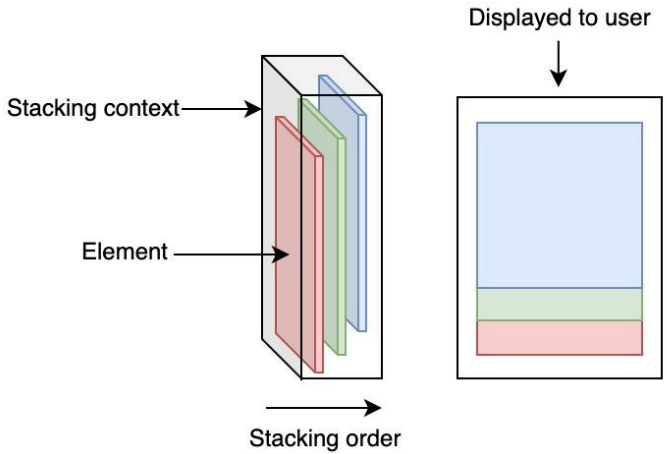
or

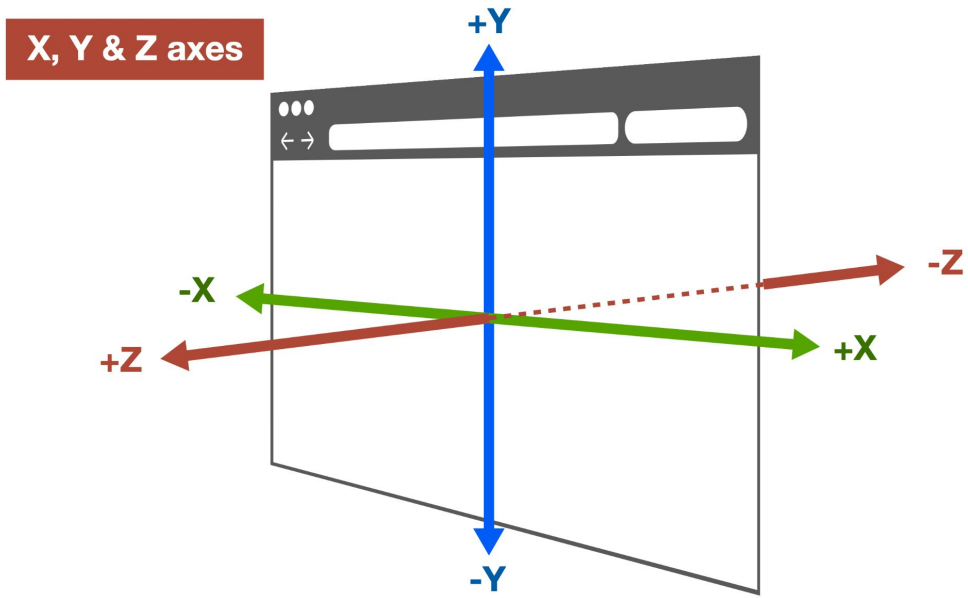
`translate: -50% -50%;`

stacking context

Demo

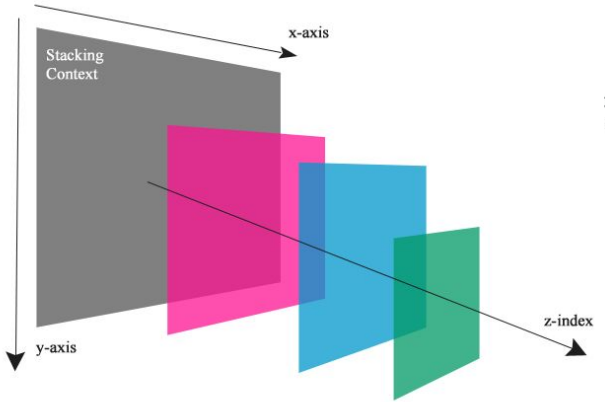
– is a three-dimensional conceptualization of HTML elements along an imaginary z-axis relative to the user, who is assumed to be facing the viewport or the webpage. HTML elements occupy this space in priority order based on element attributes.





z-index

Demo



– Controls the stacking order of elements within a stacking context.

- Applies only to elements with an explicitly defined `position`
- If positioning is not needed but you still require a z-index, use `position:relative` without coordinates.
- Values can be positive or negative numbers.
- The higher the number, the more visible the layer.
- By default, `z-index: auto` is used, and visibility is determined by the element's position in the DOM tree: the lower it is in the code, the more visible it is.

stacking context

`root stacking context` – **`<html>`** tag creates the root stacking context, unless other conditions are specified:

- **`position: absolute`** with a **`z-index`** other than **`auto`**
- **`position: relative`** with a **`z-index`** other than **`auto`**
- **`position: fixed`**
- **`position: sticky`** (only on mobile devices)
- A child element of a **`flex`** container with a **`z-index`** other than **`auto`**
- **`opacity`** less than **`1`**
- **`transform`** with a value other than **`none`**
- **`filter`** with a value other than **`none`**
- **`perspective`** with a value other than **`none`** or **`0`**

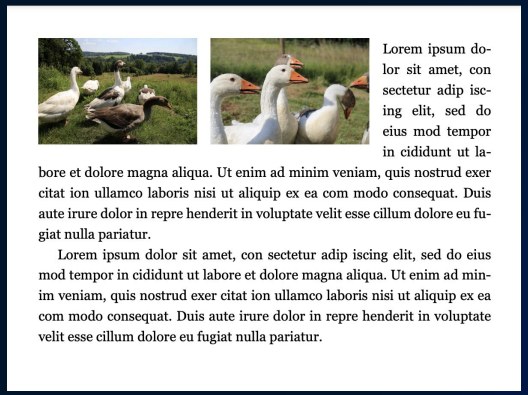
float

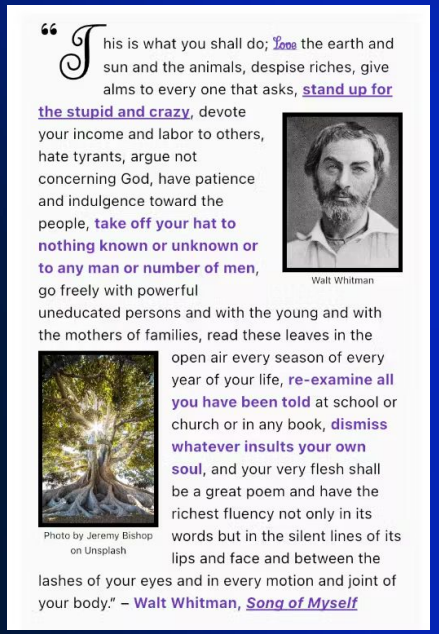
allows content to flow around a block element.

When an element is floated, it is removed from the normal document flow and pushed as far as possible to the specified side (left or right) within its containing element.

```
float: left;
     : right;
     : both;
     : none;
```

Do not use float for building layouts!






All three elements in normal flow
This is a heading
Lorem ipsum dolor sit amet, consect etuer adipisc
elit sed diam nibh euismod tincidunt ut laoreet
dolore magna aliquam erat volutpat.
Line boxes inside paragraph
This is a heading
Lorem ipsum dolor sit amet, consect etuer adipisc
elit sed diam nibh euismod tincidunt ut laoreet
dolore magna aliquam erat volutpat.
Heading not affected by floated element below
This is a heading
Lorem ipsum dolor sit amet, consect etuer adipisc
elit sed diam nibh euismod tincidunt ut laoreet
dolore magna aliquam erat volutpat.
1
Image moves out of flow
This is a heading
Lorem ipsum dolor sit amet, consect etuer adipisc
elit sed diam nibh euismod tincidunt ut laoreet
dolore magna aliquam erat volutpat.
2
Paragraph slides under the image
This is a heading
m ipsum dolor sit amet, consect etuer adipisc
elit sed diam nibh euismod tincidunt ut laoreet
dolore magna aliquam erat volutpat.
3
Line boxes inside paragraph will shorten
This is a heading
Lorem ipsum dolor sit amet, consect etuer
elit sed diam nibh euismod tincidunt ut
laoreet dolore magna aliquam erat volutpat.

clear float

is used to stop elements from being wrapped by floated elements.

It ensures that the element to which it is applied does not sit next to or wrap around any floated.
When you have an element that you want to appear beneath floated elements (rather than beside them), you apply **clear** to that element.
The most common scenario is applying clear to the element that follows a floated element to ensure that it does not wrap around the float but instead starts on a new line below it.

```
clear: left;
     : right;
     : both;
     : none;
```

Demo

It can be applied to:

1. An element that follows a floated element - `float`

The floating element is floated to the left, which causes the content to wrap around it.

The current element is floated to the right but has the clear: left; property applied. This causes the current element to drop down below the left-floated element, preventing any content from being next to it on the left side. However, since it is also floated right, it aligns itself to the right edge after moving down.

The floating element is again floated to the left, causing content to wrap around it.

The current element is also floated to the left and has clear: right; applied. However, because clear: right; doesn't affect left-floated elements, the current element aligns itself to the left directly next to the first element, maintaining the float on the same side. The clear: right; has no visible effect in this scenario because there's no element floated to the right.

It can be applied to:

1. An element that follows a floated element - `float`

In this scenario, the floating element is floated to the left.

The current element is also floated to the left but has the clear: left; property applied. This causes the current element to drop down below the left-floated element, clearing the float on the left side. The text and other content no longer wrap around the floating element, and the current element starts on a new line directly below the first floating element.

It can be applied to:
2. An element that contains a floated element

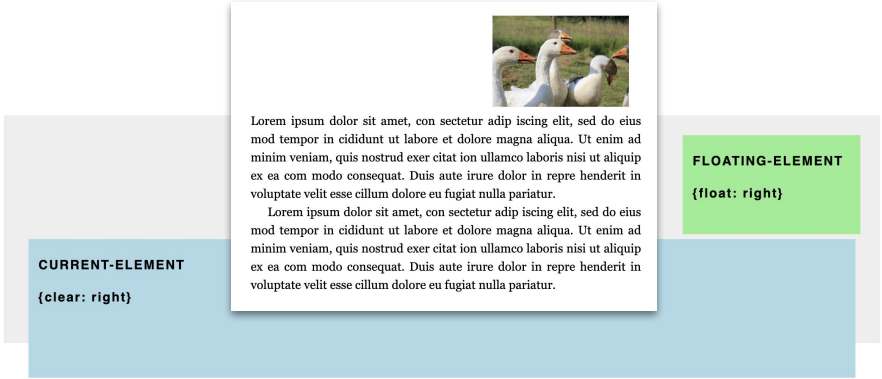
In this example, the current element has clear: left; applied. This ensures that the current element does not allow any left-floating elements to sit beside it, so it starts on a new line.

The floating element is floated to the right, causing the content to wrap around it on the left side.

Since there is no left-floating element before the current element, the clear: left; rule has no effect on the floating element positioned on the right side.

Here, the current element has clear: right; applied, which ensures that no elements floated to the right can be next to it. The current element is forced to the next line beneath the right-floating element.

The floating element is also floated to the right, causing content to wrap around it on the left. But because of the clear: right; property on the current element, it is pushed below the floated element and does not allow any content to be next to it on the right.



clear float

Alternatives:

`overflow: hidden;`

This technique is often used to contain floated elements within a parent element, clearing the floats without the need for additional markup or clearing elements.

`display: flow-root;`

A modern CSS property that establishes a new block formatting context, effectively containing floats and clearing them automatically.

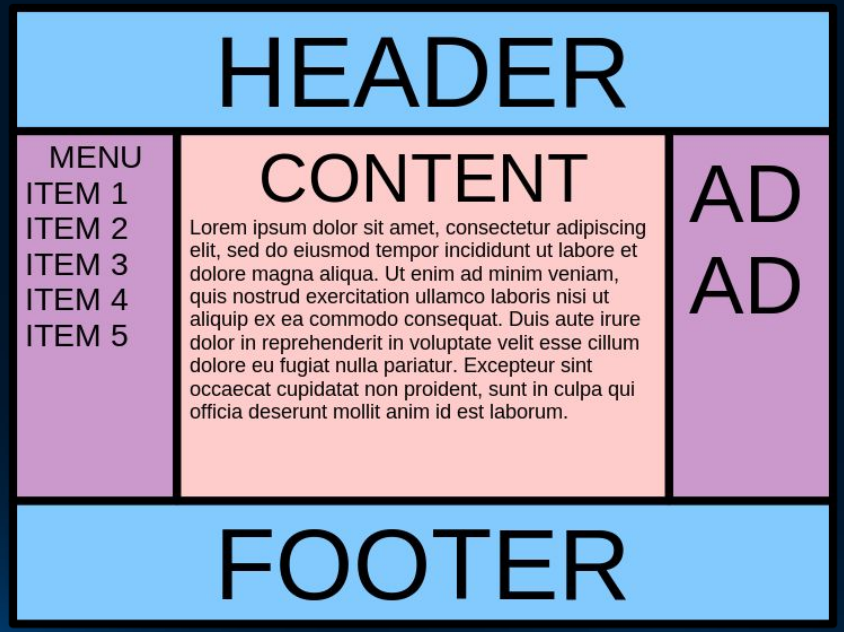
Demo

flexbox

Flexbox allows control over:

- Size
- Order
- Alignment of elements across multiple axes
- Distribution of free space between elements
- and more

- ★ Ease in creating complex layouts
- ★ Automatic alignment
- ★ Responsiveness



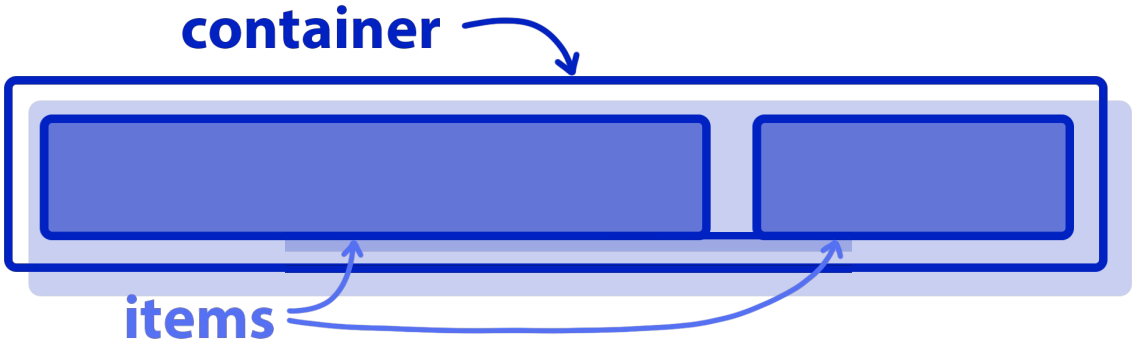
flex container and items

The container is given the property:

```
display: flex;
display: inline-flex;
```

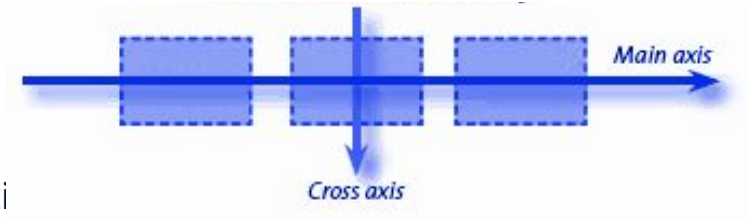
Style properties are applied to the **elements** inside the container.

The elements adjust to the width of their content and are arranged in a row by default.



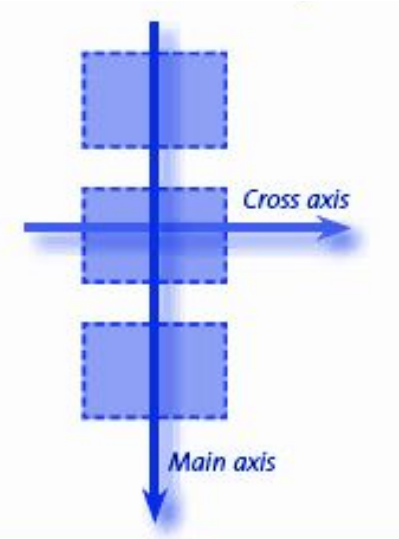
[Demo]

flex-direction



This establishes the main-axis, thus defining the directi
flex items are placed in the flex container. Flexbox is (aside
from optional wrapping) a single-direction layout concept.
Think of flex items as primarily laying out either in
horizontal rows or vertical columns.

main axis ~~horizontal~~
cross axis ~~vertical~~



Default:
```
flex-direction: row;
```

To arrange elements in a column:
```
flex-direction: column;
```

Demo

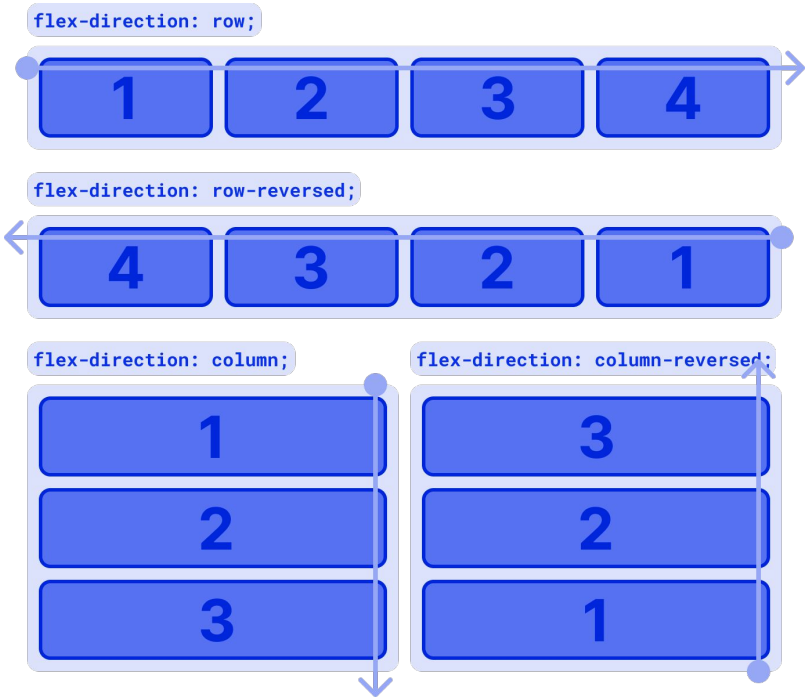
flex-direction: reversed

Demo

Using Reverse Direction:

```
flex-direction: row-reversed;
```

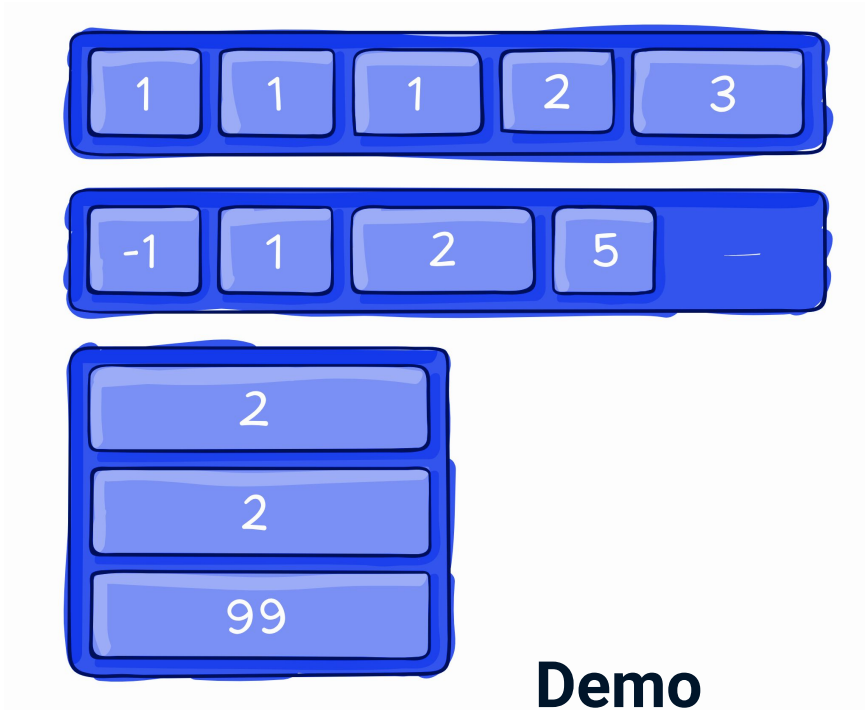
```
flex-direction: column-reversed;
```



flex order

By default, flex items are laid out in the source order. However, the **order** property controls the order in which they appear in the flex container.

`order: 4;`
Accepts positive and negative values.
Works correctly when the **order** property is set for all elements within the flex container.
Items with the same order revert to source order.



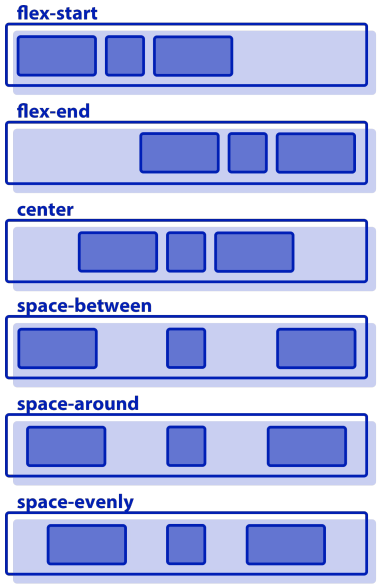
Demo

justify-content

Demo

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.

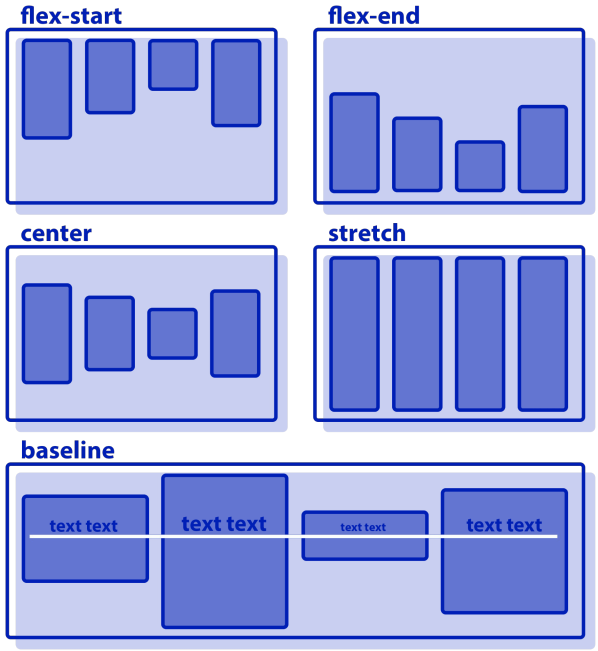


align-items

Demo

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).

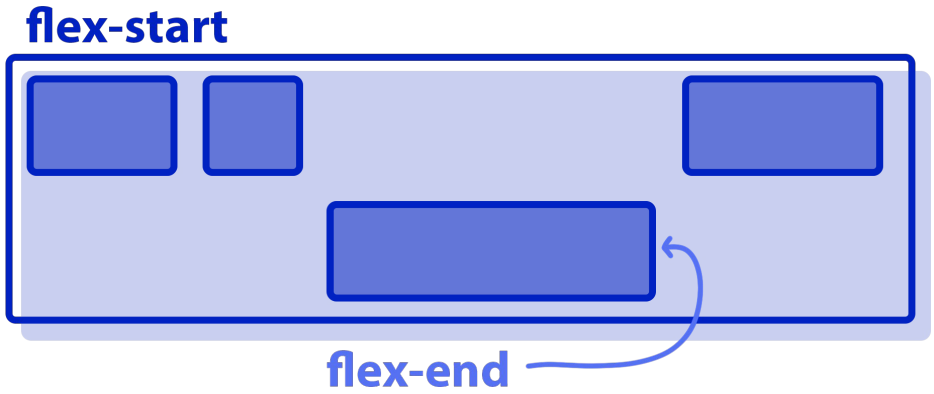
- `stretch` (default): stretch to fill the container (still respect min-width/max-width)
- `flex-start`: items are placed at the start of the cross axis. The difference between these is subtle, and is about respecting the flex-direction rules or the writing-mode rules.
- `flex-end`: items are placed at the end of the cross axis. The difference again is subtle and is about respecting flex-direction rules vs. writing-mode rules.
- `center`: items are centered in the cross-axis
- `baseline`: items are aligned such as their baselines align



`align-self`

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

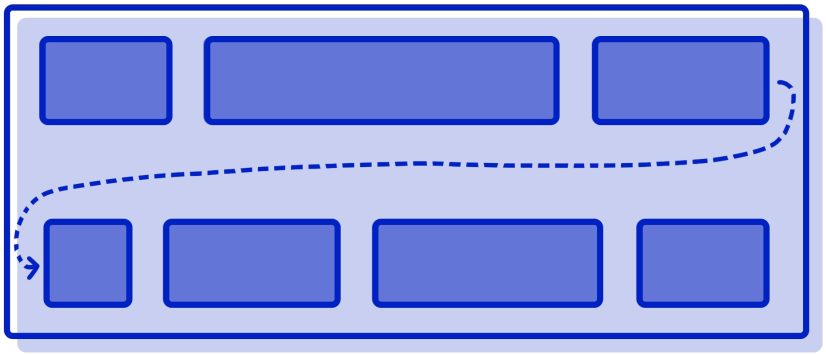
The values are the same as those for **`align-items`**



Demo

flex-wrap

By default, flex items will all try to fit onto one line. You can change that and allow the items to wrap as needed with this property.



Flexbox can wrap items to new rows using:

```
flex-wrap: wrap;
```

Default (all flex items will be on one line):

```
flex-wrap: no-wrap;
```

flex items will wrap onto multiple lines from bottom to top:

```
flex-wrap: wrap-reverse;
```

Demo

align-content

Demo

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

- `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

gap

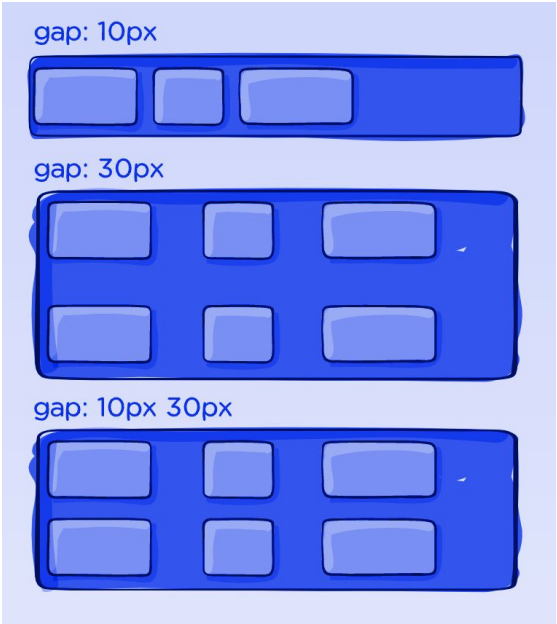
– explicitly controls the space between flex items. It applies that spacing only between items not on the outer edges.

You can set `margin` for each element individually, but it's much easier to set spacing for the entire container in one place using the `flex` container.
The behavior could be thought of as a minimum gutter, as if the gutter is bigger somehow (because of something like justify-content: space-between;) then the gap will only take effect if that space would end up smaller.

It is not exclusively for flexbox, gap works in grid and multi-column layout as well.

`gap` – Sets spacing between items

```
gap: 10px 15px;
column-gap: 10px;
row-gap: 15px;
```



flex-grow

Demo

This defines the ability for a flex item to grow if necessary. It accepts a unitless value that serves as a proportion. It dictates what amount of the available space inside the flex container the item should take up.

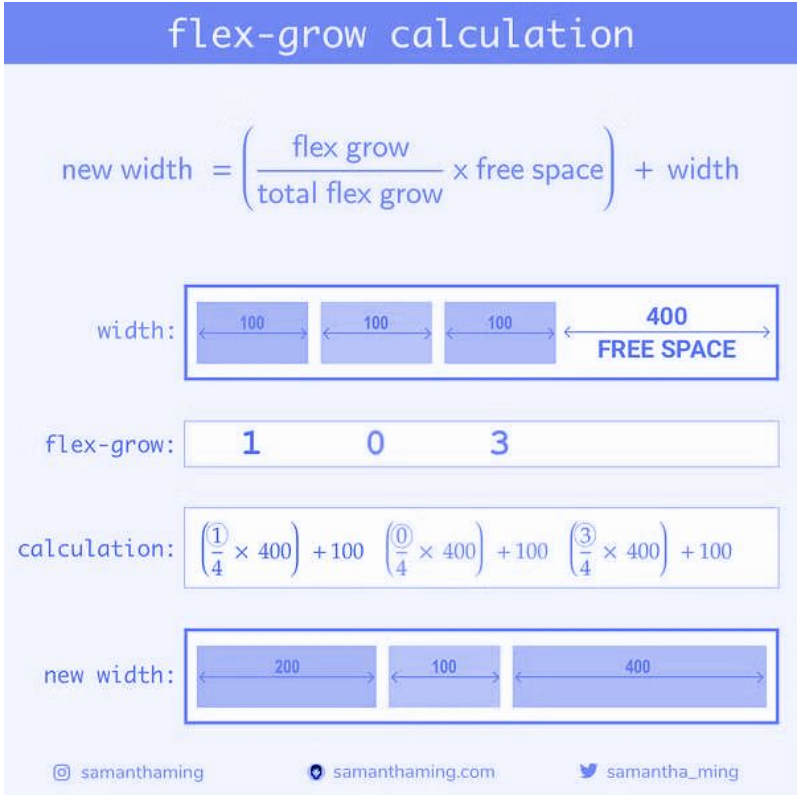
Applied along the **main axis** for `flex-item`.

Values are positive integers greater than 0.

```
flex-grow: 2;
```

The distribution is calculated based on a specific formula –>

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 as either one of the others (or it will try, at least).



Demo

flex-shrink

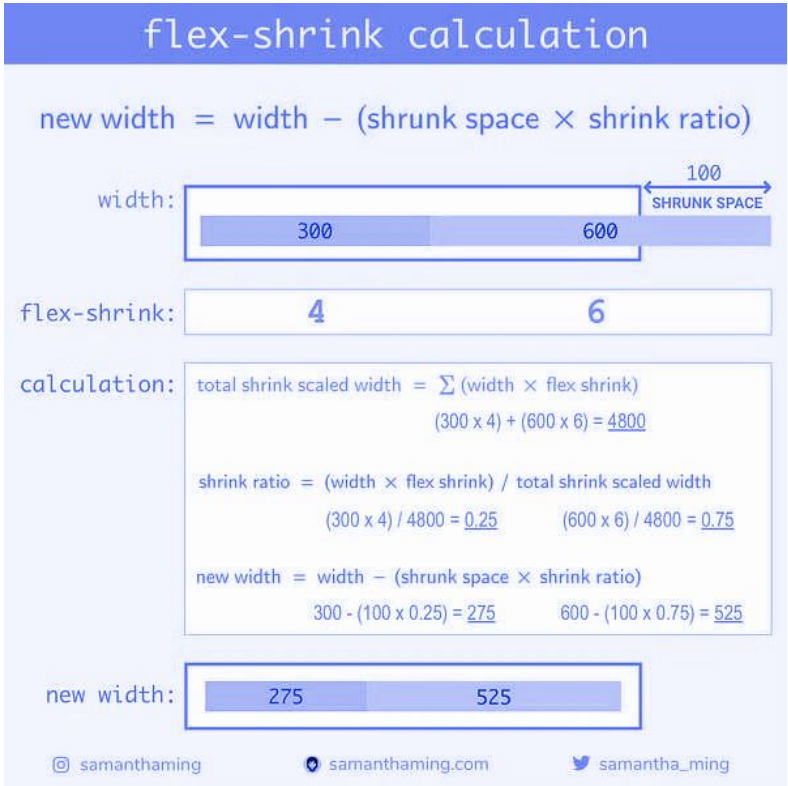
This defines the ability for a flex item to shrink if necessary.
Applied along the **main axis**.

Values are positive integers greater than 0.

`flex-shrink: 1;`

The shrinkage is calculated based on a specific formula →

`flex-shrink: 0;` – The element will maintain the width specified by `width` or `flex-basis`, even if this causes other elements to overflow.



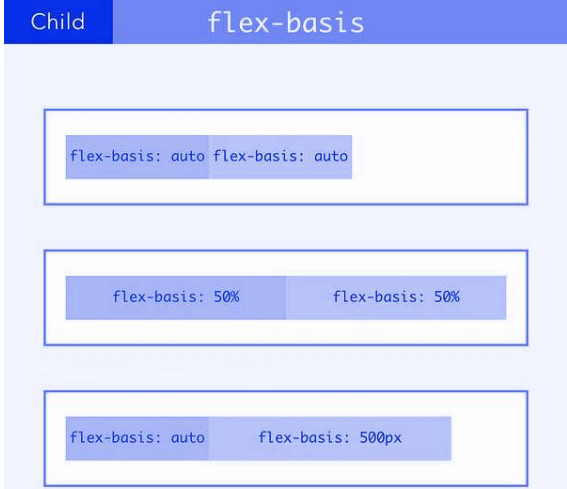
flex-basis

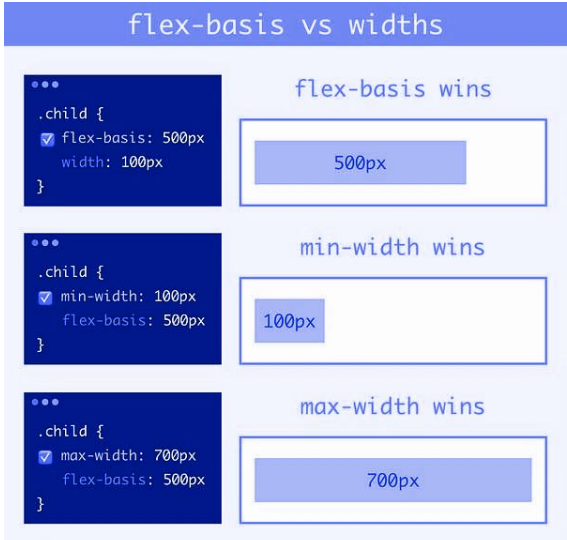
`flex-basis` defines the size of an element along the **main axis** before considering `flex-grow` and `flex-shrink`.

It can be a length (e.g. 20%, 50px, etc.) or a keyword:

- The **auto** keyword means “look at my width or height property” (which was temporarily done by the main-size keyword until deprecated).
- The **content** keyword means “size it based on the item’s content” – this keyword isn’t well supported yet, so it’s hard to test and harder to know what its brethren max-content, min-content, and fit-content do.

If set to 0, the extra space around content isn’t factored in. If set to auto, the extra space is distributed based on its flex-grow value. See this graphic.





`flex:` shorthand property

`flex-grow / flex-shrink / flex-basis`

```
flex: 1 0 300px;
```

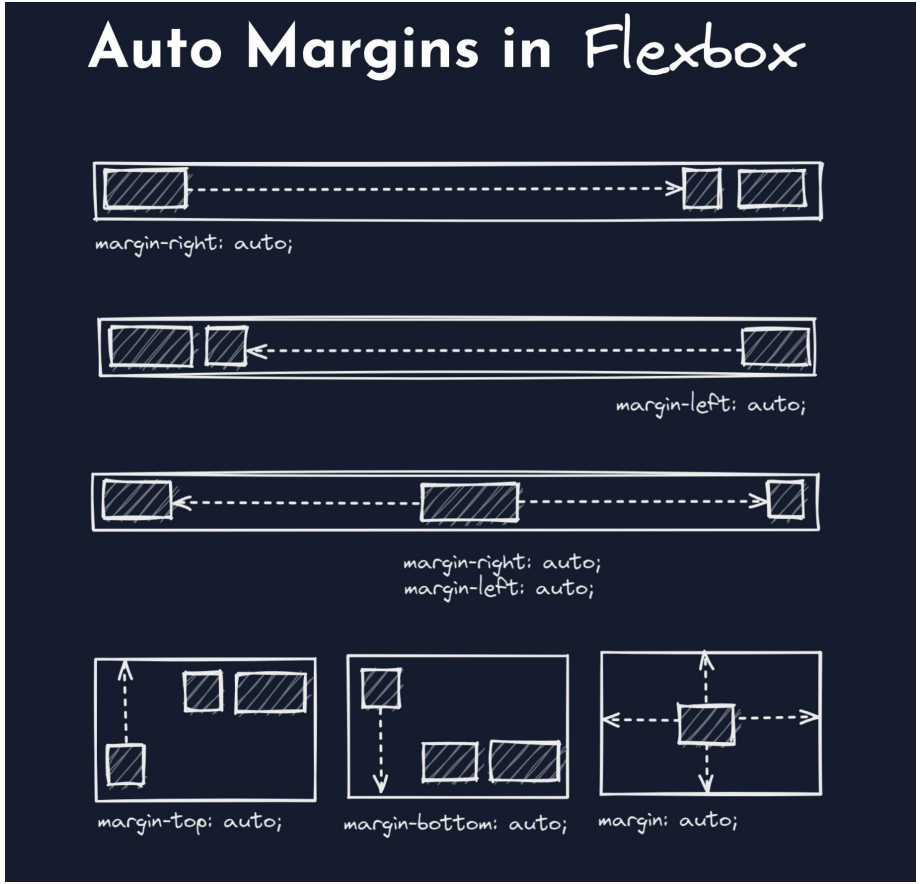
In Flexbox, **`auto`** margins can be used both horizontally and vertically to control the positioning of flex items within the container.

Horizontal Alignment:

- **`margin-left: auto;`** or **`margin-right: auto;`**: This pushes the flex item to the opposite side, allowing for easy alignment.

Vertical Alignment:

- **`margin-top: auto;`** and **`margin-bottom: auto;`**: These work similarly to horizontal margins, pushing the item either downward (using `margin-top: auto;`) or upward (using `margin-bottom: auto;`). This is particularly useful for vertically centering items when the flex container has a set height.



Centering Content, Full-Width Background

The background of the website should always stretch across the full width of the page.

The content has a width restriction (set by width/max-width) as per the design layout and is centered on the page.

```
<section>
     <div class=”content-wrapper”>

section { background-color: grey; }

.content-wrapper {
     width: 1400px;
     margin: 0 auto;
}
```

Demo