CSS

André Restivo

Index

Introduction

What are they?

- **C**ascading **S**tyle **S**heets
- A style sheet language used for describing the look and formatting of a document written in a markup language (like HTML).
- Based on two concepts: **selectors** and **properties**.

History

- 1996 **CSS 1** Limited and poorly supported by browsers
- 1998 **CSS 2**
- 1999 **CSS 1** Supported by browsers
- 2003 **CSS 2** Decently supported by browsers
- 2003 **CSS Zen Garden** (http://www.csszengarden.com/)
- 2011 **CSS 2.1**
- 2011-2012 **CSS 3**

Selectors

Allow us to select the HTML elements to which we want to apply some styles.

Properties

Define what aspect of the selected element will be changed or styled.

```css
p {              /* selector */
  color: red;    /* property: value */
}
```

Together, selectors and properties define CSS **rules**.

Linking to HTML

We can apply CSS styles to HTML documents in three different ways.

Inline

Directly in the HTML element:

```html
<p style="color: red">
  This is a red paragraph.
</p>
```

Internal Style Sheet

Using a stylesheet inside the HTML document:

```html
<head>
  <style>
  p {
    color: red;
  }
  </style>
</head>
<body>
  <p>This is a red paragraph.</p>
</body>
```

External Style Sheet

In a separate stylesheet:

```html
<head>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <p>This is a red paragraph.</p>
</body>
```

style.css

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

The preferred way. Allows for style **separation** and **reuse**.

Resources

- References:
  - https://developer.mozilla.org/en/docs/Web/CSS/Refere
  - http://www.w3.org/Style/CSS/specs.en.html
- Tutorials:
  - https://css-tricks.com/almanac/
  - http://www.htmldog.com/guides/css/

Selectors

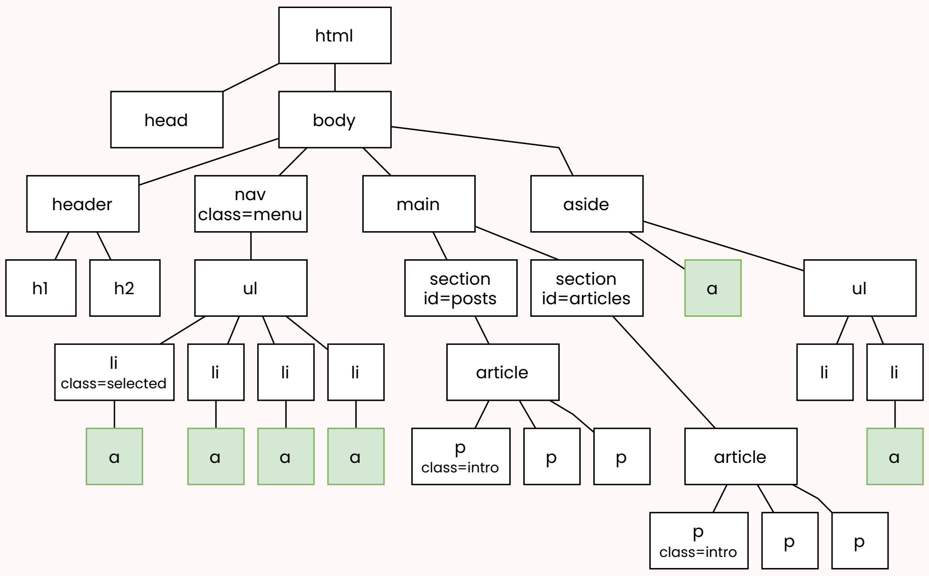
Selectors

- A selector defines a pattern-matching rule that determines which style rules apply to which elements in the document tree.
- There are several types of selectors:
  - The Universal(*) selector.
  - Type selectors.
  - Attribute([ ]) selectors.
  - Class(.) & Id(#) selectors.
  - Pseudo-classes(:) and Pseudo-elements(::).
- There are also ways to:
  - Group(,) selectors to reuse properties.
  - Combine(space, >, +, ~) selectors into more complicated ones.

Type Selectors

Select elements by their element type:

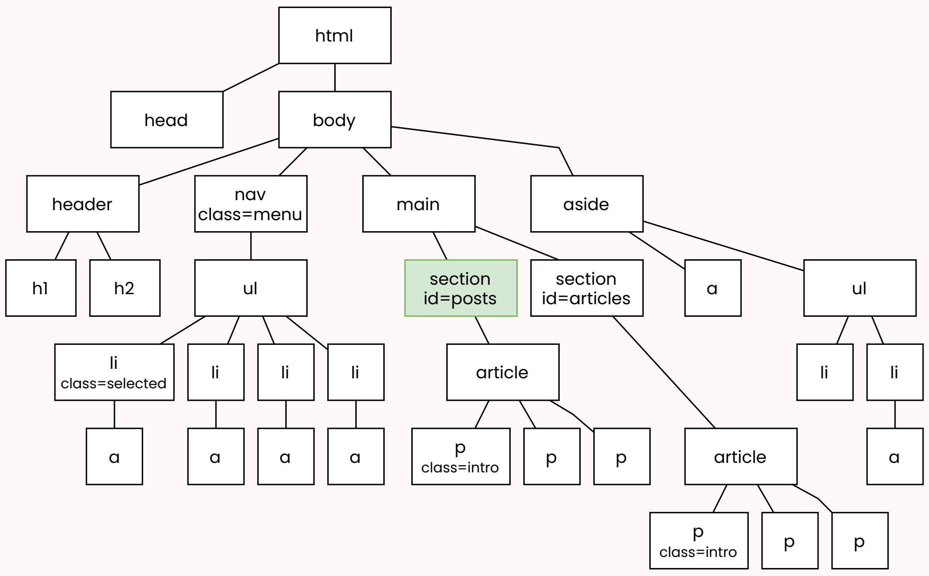
```
a
```



Id Selector

Selects element by their id (#):

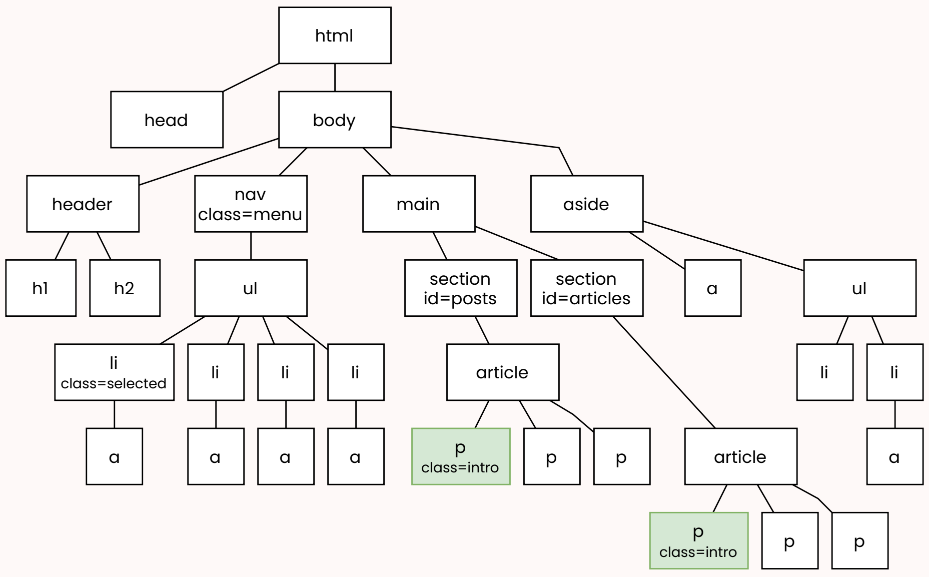
```
#posts
```



Class Selector

Selects element by their class (.):

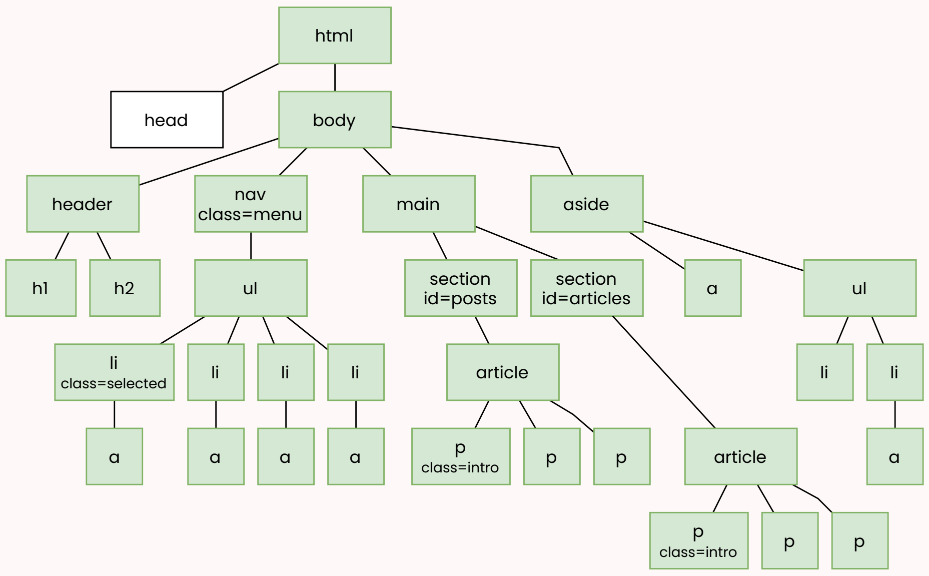
```
.intro
```



Universal Selector

Selects all elements (*):

```
*
```



Attribute Selectors

Select elements based on their attribute existence and values:

- **[attribute]** – exists.
- **[attribute=value]** – equals.
- **[attribute~=value]** – containing value (word).
- **[attribute|=value]** – starting with value (word).
- **[attribute^=value]** – starting with value.
- **[attribute$=value]** – ending with value.
- **[attribute*=value]** – containing value.

```
form[method=get] /* selects all forms with attrib
```

Combining Selectors

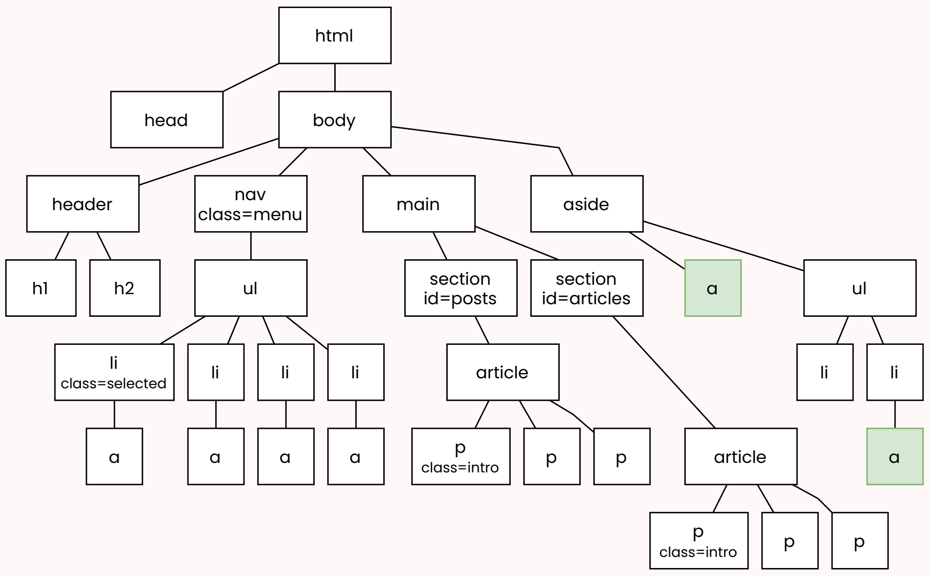
Combining Selectors

- Sometimes, we want to select elements based on their relationship with other elements.
- For this, we can use the following combinators:
  - Descendant combinator (space).
  - Child combinator (>).
  - Next-sibling combinator (+).
  - Subsequent-siblings combinator (~).
- In combinators, the **last selector** is the one that identifies the element we are selecting.

Descendant Combinator

Selects all descendants (space):

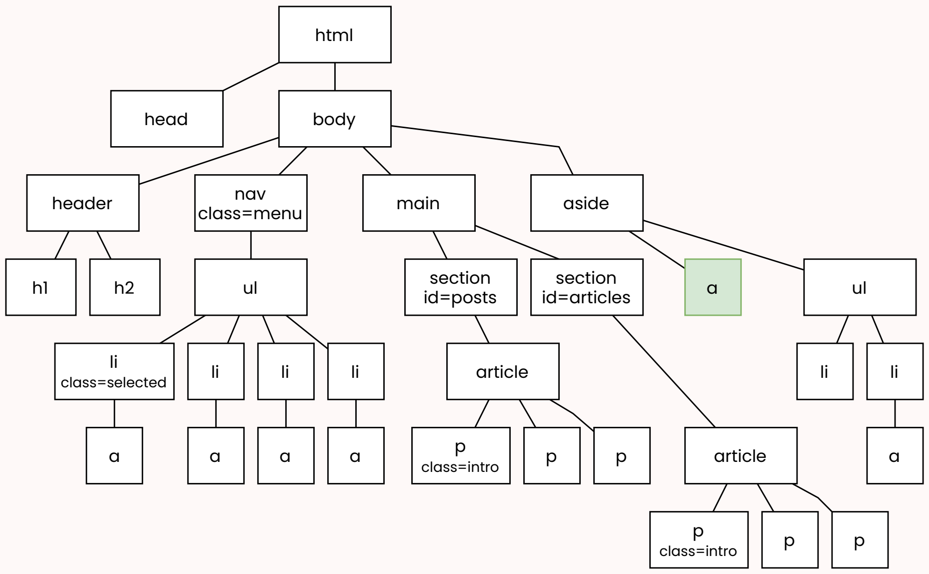
```
aside a
```



Child Selector

Selects all children (>); they have to be direct descendants:

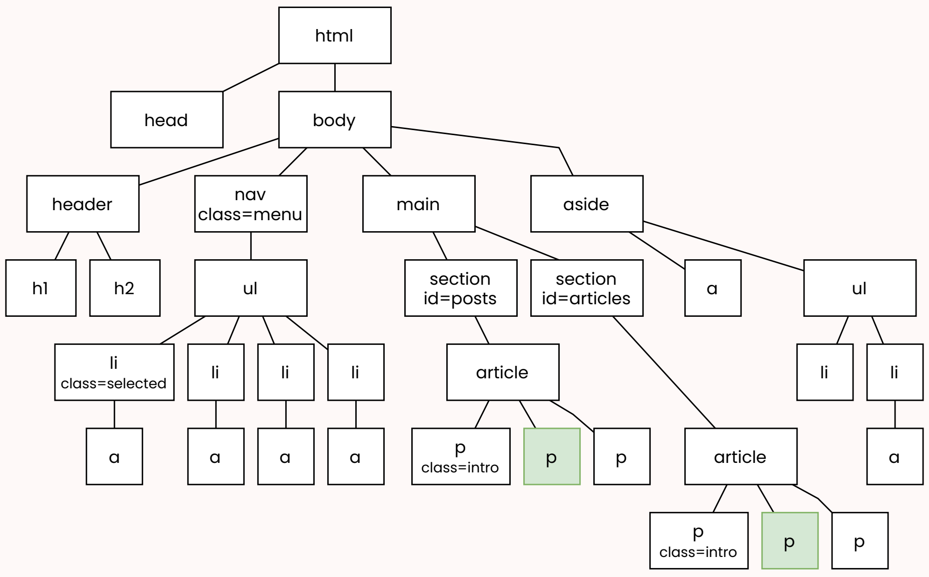
```
aside > a
```



Next-sibling Selector

Selects the next sibling (+); they have to be the next one:

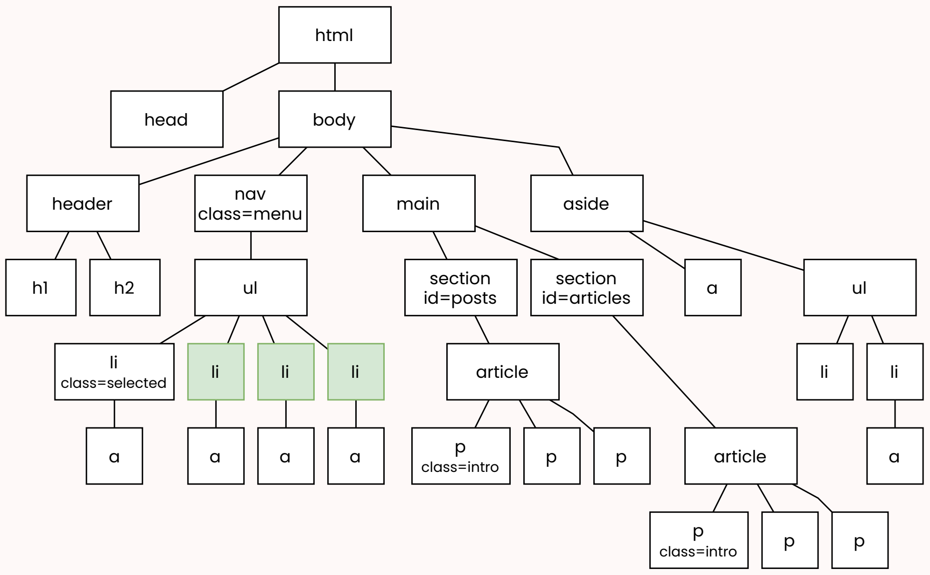
```
.intro + p
```



Subsequent-sibling Selector

Selects subsequent siblings (~). They just have to be after:

```
.selected ~ li
```

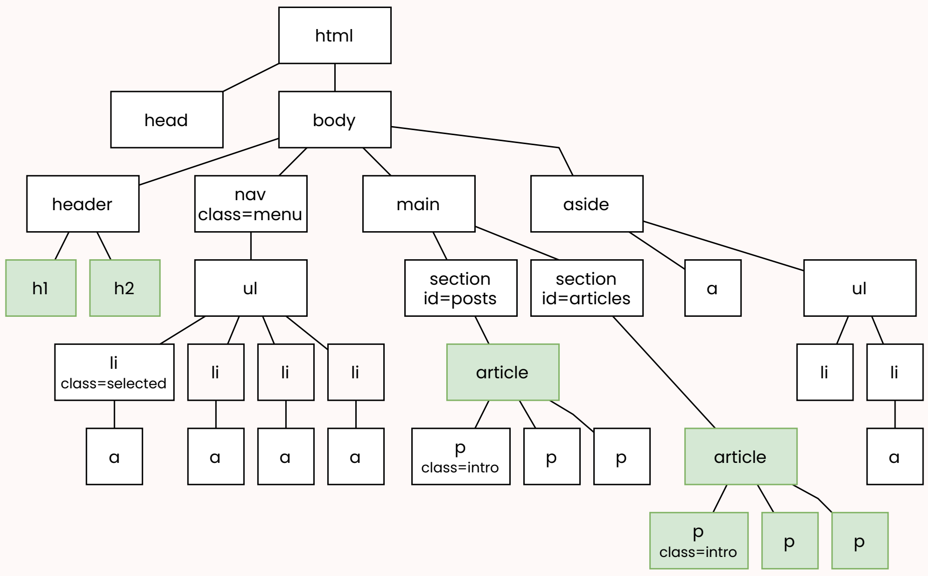


Grouping Selectors

Grouping Selectors

Selector groups (,) are just a way to simplify CSS rules:

```
header > *, main article, #articles p
```



Pseudo-selectors

Pseudo-classes and Pseudo-elements

- A pseudo-class(:) is a way of selecting **existing elements** based on their **state** as if it were a class (*e.g.*, all elements of the class *visited links*).
- A pseudo-element(::) allows logical, not actual, elements to be defined (*e.g.*, the first letter of a paragraph).

Anchor Pseudo-classes

Pseudo-classes that select anchors (links) based on their state:

```
a:visited /* selects all links that were visited
```

- **:link** – The link was never visited.
- **:visited** – The link was visited previously.
- **:active** – The link is active (being clicked).
- **:hover** – The mouse is over the link (also works on other element types):

```
img:hover /* selects images when the mouse pointe
```

Form Pseudo-classes

Selects form controls based on their state:

```
input:focus     /* the input is focused */

input:valid     /* the data in the input is valid
input:invalid   /* the data in the input is not v

input:required  /* the input is mandatory */
input:optional  /* the input is optional */

input:read-only  /* the input is read-only */
input:read-write /* the input is not read-only */

radio:checked    /* the radio button is checked *
```

The focus pseudo-class can be used in other elements (*cf.* accessibility).

Target Pseudo-class

The **target** pseudo-class selects the **unique** element, if any, with an *id* matching the URL's fragment identifier (the part after #).

If we have this HTML in our *news.html* page:

```
<section id="sports">...</section>
<section id="politics">...</section>
```

When the URL changes to *news.html#sports*, the page scrolls to the *section* with *id* "sports", and both these selectors then select that section:

```
:target
```

```
section:target
```

First and Last Pseudo-classes

Select elements based on their position in the tree:

```
/* any paragraphs that are the first child of the
p:first-child


/* any element that is the last child of their pa
:last-child
```

- **first-child** – Selects elements that are the first child of their parents.
- **last-child** – Selects elements that are the last child of their parents.
- **first-of-type** – Selects elements that are the first child of their parents having their type.
- **last-of-type** – Selects elements that are the last child of their parents having their type.

Nth Child Pseudo-classes

The **nth-child(an+b)** selector selects elements that are the **bth** child of an element after all its children have been split into groups of **a** elements each.

In other words, this class matches all children whose index fall in the set *{ an + b; n = 0, 1, 2, ... }*.

```
:nth-child(1)    /* is the same as :first-child *
:nth-child(2)    /* second child */
:nth-child(2n)   /* the even childs */
:nth-child(2n+1) /* the odd childs */
:nth-child(-n+3) /* one of the first three childr
```

The **nth-of-type(an+b)** selector does the same thing but counts only siblings with the same name.

Empty and Only-child Pseudo-classes

Select elements based on the **number of children** of an element:

```
/* paragraphs that are the only children of their
p:only-child

/* paragraphs that have no children (not even tex
p:empty
```

Not Pseudo-class

Represents elements that **do not match** a list of selectors:
Negation pseudo-class selectors cannot be nested.

```
:not(p) /* all elements that are not a paragraph

/* all paragraphs inside sections that are direct
/* children of an element that is not an article
section :not(article) > p
```

Be careful with some **pitfalls**:

```
<section><article><p>
  The quick brown fox jumps over the lazy dog
</p></article></section>
```

```
section :not(article) p /* does not select the pa
:not(article) > p       /* does not select the pa
:not(article) p         /* selects the paragraph,
```

Has Pseudo-class

Represents elements where any of the relative selectors passed as arguments match:
Also known as the "parent selector" because it allows you to select elements based on their descendants.

```
section:has(p)    /* all sections that contain a
h1:has(+ h2)      /* a h1 that is followed by a h
:has(p)            /* any element that contains a
```

Examples:

```
<section>    <!-- selected by :has(p) and section
  <article>  <!-- selected by :has(p) -->
    <p>The quick brown fox jumps over the lazy do
  </article>
</section>
```

What about "`section :has(p)`"?

Typographic Pseudo-elements

Select **parts** of elements based on their position in the element:

```
p::first-letter /* the first letter of any paragr
```

- **::first-line** – Selects the first line of the selector.
- **::first-letter** – Selects the first character of the selector.

A more complicated example:

```
/* the first letter of any paragraph that     */
/* is the first paragraph child of an article */
article > p:first-of-type::first-letter
```

Before and After Pseudo-elements

Before and after pseudo-elements can be combined with the **content** property to generate content around an element.

The **content** property can have the following values:

- **none** – The default value; adds nothing. Cannot be combined with other values: *none*.
- **a string** – Using single quotes. Adds the text to the element: *'Chapter'*.
- **an url** – An external resource (such as an image): *url('dog.png')*.
- **counter** – Variables maintained by CSS whose values may be manipulated by CSS rules to track how many times they're used: *counter(section)*.
- **open-quote** and **close-quote** – Open and close quotes: *open-quote*.

```
blockquote::before { content: open-quote;  }
blockquote::after  { content: close-quote; }
```

Complex Selectors

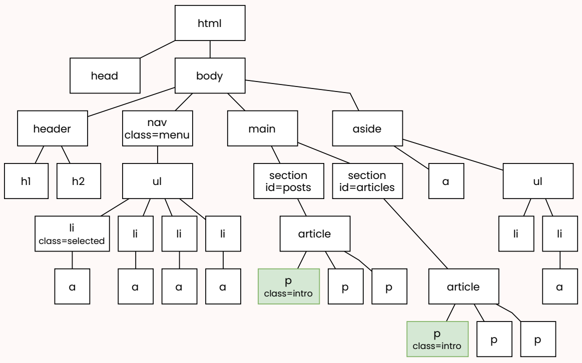
Complex Selectors

All these type of selectors can be combined to form complex selectors:

```
nav.menu + * > section :first-child p.intro
```

It's easier to read them from the **right to the left**:

> *Paragraphs with class "intro" that are descendants of elements that are the first child of their parents and are descendants of "sections" that are direct children of any element that is the next sibling of a "nav" with class "menu."*



Common Mistakes

Spaces are important when writing and parsing CSS selectors:

```
/* a paragraph with class "intro" */
p.intro

/* an element with class "intro" descendant of a
p .intro
```

```
/* a paragraph that is the first child of its par
p:first-child

/* an element that is the first child of its */
/* parent and a descendant of a paragraph */
p :first-child
```

Common Mistakes

And so is the whole **context**:

```
<nav>
  <ul>
    <li><a>...</a></li>
    <li><a>...</a></li>
    <li><a>...</a></li>
  </ul>
</nav>
```

```
/* selects all links of the list */
a:first-child

/* selects all links of the list */
:first-child a

/* selects the first link of the list */
li:first-child a

/* selects the first link of the list */
:first-child > a
```

Nesting Selectors

Nesting Selectors

Nested style rules **inherit** their parent rule's selector context, eliminating the need for repetition and allowing for further building upon the parent's selector context while still associating properties with elements via selectors.

These two are equivalent:

```css
.foo {
  color: blue;
  .bar {
    color: red;
  }
}
```

```css
.foo { color: blue; }

.foo .bar { color: red; }
```

❗ This recent development may not be universally supported across all browsers. It's essential to verify the percentage of users whose browsers can accommodate this feature before implementation.

Relative Selectors

Nested style rules can use relative selectors to specify relationships other than "descendant":

- The parent of the nested selector: &
- The child, next sibling, and next siblings combinators: >, +, ~

```css
form {
  color: blue;
  & input {
    color: red;
  }
}
```

```css
form { color: blue; }

form input { color: red; }
```

❗ A nested selector cannot start with an element selector (*i.e.*, an identifier), or it would be hard to distinguish between elements and properties.

Nested Examples

```css
header {
  color: blue;

  > h1 {
    color: red;
    + h2 {
      color: yellow;
    }
  }

  body > & {
    color: green;
  }
}
```

```css
header { color: blue; }

header > h1 { color: red; }

header > h1 + h2 { color: yellow;

body > header { color: green; }
```

Color

Text Color

We can set the text color of any element:

```css
p {
  color: green;
}
```

```html
<p>The quick brown fox jumps over the lazy dog</p
```

Results in:

The quick brown fox jumps over the lazy dog

Background Color

We can set the background color of any element:

```
p {
  background-color: yellow;
}
```

```
<p>The quick brown fox jumps over the lazy dog</p
```

Results in:

The quick brown fox jumps over the lazy dog

The default background color of most elements is **transparent**.

Color by Name

Colors can be referenced using one of these pre-defined names:

```
aqua, black, blue, fuchsia, gray, green,
lime, maroon, navy, olive, orange, purple,
red, silver, teal, white, and yellow.
```

```
p {
  background-color: fuchsia;
}
```

aqua black blue fuchsia gray green lime maroon navy olive orange purple red silver teal white yellow

Modern browsers support an extended set of these.

Color by Hexadecimal Value

A hexadecimal color is specified using #RRGGBB, where the RR (red), GG (green) and BB (blue) hexadecimal integers specify the components of the color. All values must be between 00 and FF.

```
p {
  background-color: #336699;
}
```

#RGB is a shorthand for #RRGGBB

```
p {
  background-color: #369;
}
```

Color by Decimal Value

An RGB color value can also be specified using: rgb(red, green, blue). Each parameter (red, green and blue) defines the intensity of the color and can be an integer between 0 and 255 or a percentage value (from 0% to 100%).

```css
p {
  background-color: rgb(50, 100, 200);
}
```

Or:

```css
p {
  background-color: rgb(25%, 50%, 75%);
}
```

Opacity

Opacity represents the transparency of an element. Values can go from 0.0 (completely transparent) to 1.0 (fully opaque).

```css
p {
  opacity: 0.5;
}
```

Fonts

Font Family

In CSS, there are two types of font family names:

- **generic family** - a group of font families with a similar look.
- **font family** - a specific font family (*e.g.*, Times New Roman).

Specific Font Family

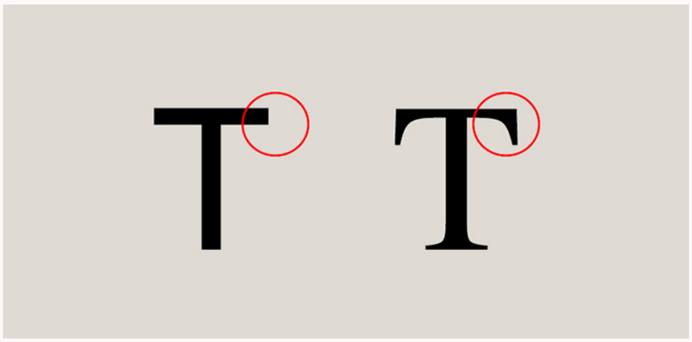
You can define a specific font family. Be careful as it might not exist in the target computer.

```css
p {
  font-family: "Arial";
}
```

Generic Font Family

Or a generic family like: **sans-serif**, **serif** and **monospace**.

```css
p {
  font-family: serif;
}
```



Sans-serif (left) vs. Serif (right)

Web Safe Fonts

- To ensure that websites look the same across different platforms, we should use *websafe* fonts like *Arial*, *Helvetica*, *Times New Roman*, *Times*, *Courier New* or *Courier*.
- You can specify several fonts. The browser will try to use the **first** and continue **down the list** if it doesn't exist.
- Start with the font you **want** and gradually **fall back** to platform defaults and finally **generic** defaults:

```css
p {
  font-family: 'Open Sans', 'Droid Sans', Arial,
}
```

Remote Fonts

- The *@font-face* rule specifies a custom font to display text.
- The font can be loaded from a remote server, making it possible to use all kinds of fonts.

```css
@font-face {
  font-family: "Open Sans";
  src: url("/fonts/OpenSans-Regular-webfont.woff2
       url("/fonts/OpenSans-Regular-webfont.woff"
}
```

- An easier way to use remote fonts is to use Google's Fonts.

```css
@import url('https://fonts.googleapis.com/css?fam
```

Font Weight

You can specify the weight of the font using the font-weight property. Values can be **normal**, **bold**, **bolder**, **lighter** or values from **100** to **900**.

```css
p.introduction {
  font-weight: bold;
}
```

Not all fonts support all weights.

Font Style

The font-style property allows you to specify if the font style should be *italic* or not. Values can be **normal**, **italic**, or **oblique**.

```
span.author {
  font-style: italic;
}
```

Font Size

To define the font size, you use the **font-size** property.

```
p.introduction {
  font-size: 1.2em;
}
```

Using **rem** or **em** units is a good idea for scalable layouts. More on this soon.

Text

Decoration

The **text-decoration** property is mostly used to remove underlines from links. But it has other possible values: **none**, underline, overline and ~~line-through~~.

```
#menu a {
  text-decoration: none;
}
```

Alignment

Text can be aligned **left**, **right**, **center** or justified (**justify**) using the **text-align** property. This property should be used for aligning text only.

```
p {
  text-align: center;
}
```

left
The quick brown
fox jumps over
the lazy dog

right
The quick brown
fox jumps over
the lazy dog

center
The quick brown
fox jumps over
the lazy dog

justified
The quick brown
fox jumps over
the lazy dog

Text Case

The **text-transform** property makes the text **uppercase**, **lowercase** or capitalized (**capitalize** first letter of each word).

```css
h1 {
  text-transform: capitalize;
}
```

The Quick Brown Fox Jumps Over The Lazy Dog

Indentation

The first line of each paragraph can be indented using the **text-indent** property. This property takes a length as its value.

```
.chapter p {
  text-indent: 10px;
}
```

"The quick brown fox jumps over the lazy dog" is an English-language pangram—a sentence that contains all of the letters of the English alphabet.

Length Units

Units

We can use several length units to change the dimension of elements in CSS. These units come in different flavors:

- Absolute units
- Font-relative units
- Viewport-percentage units
- Percentages

Absolute units

- Absolute length units represent a physical measurement.
- They are useful when the physical properties of the output medium are known, such as for print layout.

```
mm, cm, in, pt and pc
```

- **mm** One millimeter.
- **cm** One centimeter (10 millimeters).
- **in** One inch (2.54 centimeters).
- **pt** One point (1/72nd of an inch).
- **pc** One pica (12 points).

Pixel

- Also considered an **absolute length**.
- On low dpi screens, the **pixel (px)** represents one device pixel (**dot**).
- On higher dpi devices, most devices these days, a pixel represents an integer number of device pixels so that 1in ≈ 96px.

Font-relative units

Font relative length units are relative to the size of a particular character or font attribute in the font **currently in effect** in the element (or parent element in some cases).

They are useful when the physical properties of the output medium are unknown, such as for **screen layout**.

Units *rem* and *em* are used to create **scalable layouts**, which maintain the vertical rhythm of the page even when the user changes the font size.

- **rem** Represents the size of the root element font. If used to change the *font-size* in the root element, it represents the browser's initial (default or user-defined) value (typically 16px).

- **em** When used to change the *font-size*, it represents the size of the parent element font. When used to set the size of an element, it represents the size of the current element font.

Example (rem and em)

This example shows how changing font size in some elements affects the font size in others:

- Setting the font-size of the root element (**<html>**) to 2rem.
  For other elements, 1rem becomes 32px (if the user didn't change the default).
- Setting the font-size of other element to 2rem.
  The font-size of that element becomes 64px, twice the size of the root's font-size.
- Setting the font-size of the **<body>** element to 2em.
  The font-size of that element becomes 64px, twice its parent's font-size.

```
html { font-size: 2rem; } /* 32px */
p    { font-size: 2rem; } /* 64px regardless of i
body { font-size: 2em;  } /* 64px the parent is t
```

Viewport-percentage units

Define lengths relative to the viewport size (the visible part of the document):

- **vw** - 1% of the viewport width.
- **vh** - 1% of the viewport heigth.
- **vmin** - the smaller of *vw* and *vh*.
- **vmax** - the larger of *vw* and *vh*.

So, if the viewport is 600x400 pixels, vw = 6px, vh = 4px, vmin = 4px, vmax = 6px.

Percentage unit

- The *percentage* CSS data type represents a percentage value.
- A percentage consists of a *number* followed by the percentage sign %.
- There is no space between the symbol and the number.

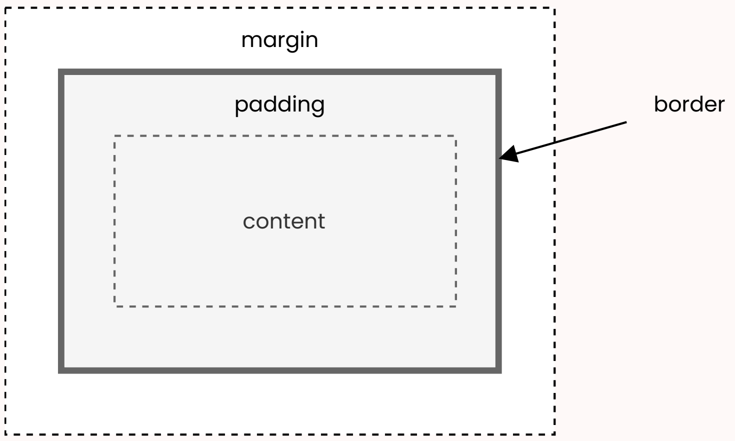
Many CSS properties (width, margin, padding, font-size, ...) can take *percentage* values to define a size relative to its parent object.

```
width: 50%;     /* width is 50% of the parent's
font-size: 80%; /* font-size is 80% of the parent
                /* the same as 0.8em
```

Box Model

Box Model

- All page elements are **rectangular**.
- They can have a **border**.
- Some **space** between themselves and that **border** (**padding**)
- And some **space** between themselves and the **next element** (**margin**).

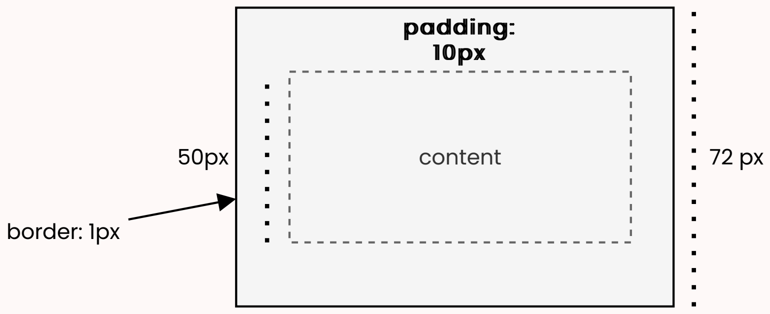


Width and Height

We can use the *width* and *height* properties to change the size of the **content area**:

- Values can be a **length**, a **percentage** or **auto** (the browser will automatically calculate a width/height).
- The *default* value is **auto**.

```
section {
  width: auto;
  height: 50px;
}
```

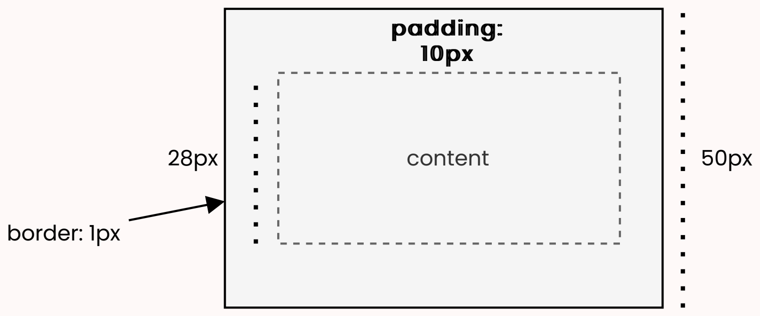


Box-sizing

We can change the behavior of the *width* and *height* properties, by changing the **box-sizing** property:

- **border-box** - the width and height properties include the **padding** and **border** (much easier to work with).
- **content-box** - the width and height properties refer to the **content area** only (the default).

```css
section {
  box-sizing: border-box;
  height: 50px;
}
```



Minimum and Maximum

- When the width/height is calculated depending on something else (e.g., the parent's size or the amount of content), we can set their minimum and maximum values using the **min-width**, **max-width**, **min-height**, and **max-height** properties.
- Values can be a **length**, a **percentage**, or **auto** (the default value).

```css
section {
  /* width is 50% of the parent's width but 40em
  width: 50%; max-width: 40em;

  /* height is automatically calculated but 100px
  height: auto; min-height: 100px;
}
```

Margin and Padding

We can use the *padding* and *margin* properties to change those two areas of the *box-model*:

```
padding: 20px;
margin: 1em;
```

But in reality, each one of these properties is a **shorthand** for four other properies:

- padding-**top**, padding-**right**, padding-**bottom** and padding-**left**.
- margin-**top**, margin-**right**, margin-**bottom** and margin-**left**.

```
margin-left: 1em;
margin-right: 2em;
padding-top: 100px;
```

Margin/Padding Shorthands

The **margin** and **padding** shorthands can take four forms:

- **One** value: changes **all four** sides of the area at once.
- **Two** values: the first is **top/bottom**, and the second is **left/right**.
- **Three** values: the first is **top**, then **left/right**, and then **bottom**.
- **Four** values: corresponding to **top**, **right**, **bottom**, **left**.

Some examples:

```
body { margin: 0 auto; } /* A common way to cente

#menu { padding: 1em; }  /* 1em padding all aroun

/* 1.5em top, 1em left/right, 3em bottom/ */
body > nav li { margin: 1.5em 1em 3em; }
```

Border

The border can be set using the **border** property:

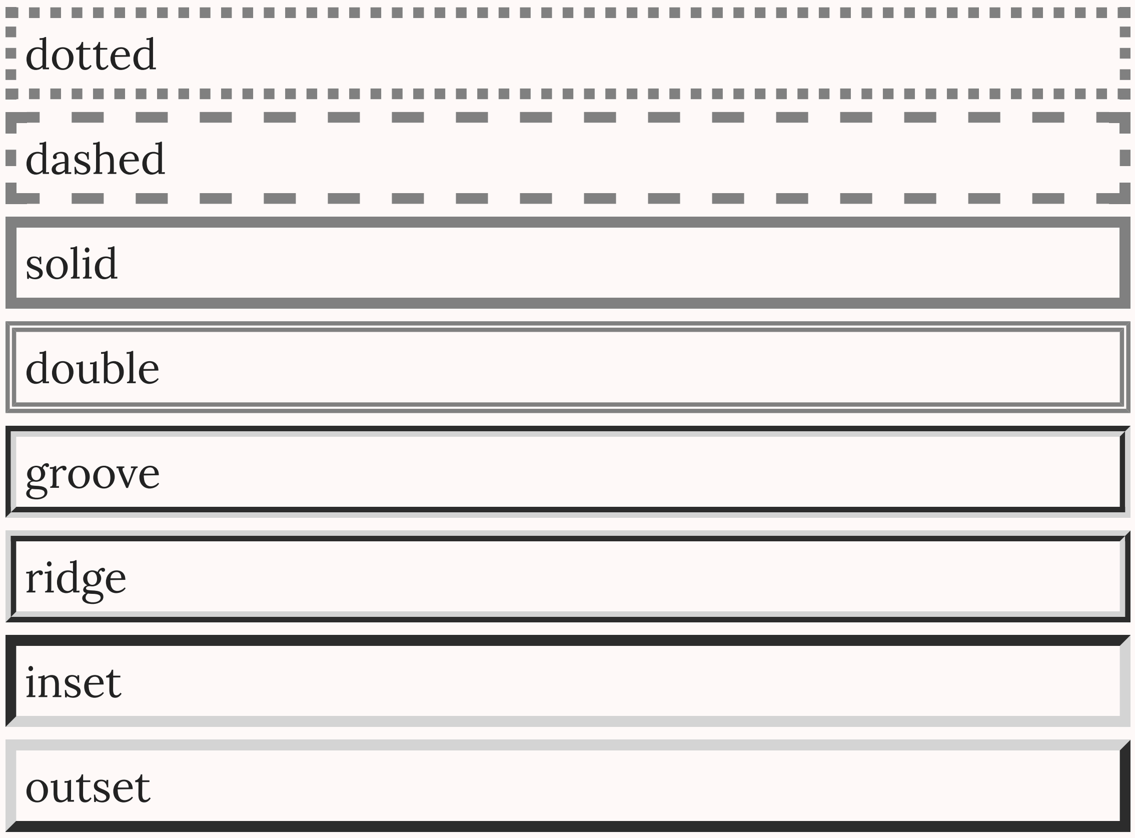
- It takes three values: the *width*, the *style*, and the *color*.
- The width is a length, but can also be *thin*, *medium* or *thick*.
- Style is one of the following: *none*, *hidden*, *dotted*, *dashed*, *solid*, *double*, *groove*, *ridge*, *inset*, and *outset*.
- And color is a color.

```
section#posts {
  border: 1px solid blue;
}
```

And is just a shorthand for three different properties: **border-width**, **border-style**, and **border-color**.

Border Styles

Border **style** examples (**5px, gray**):

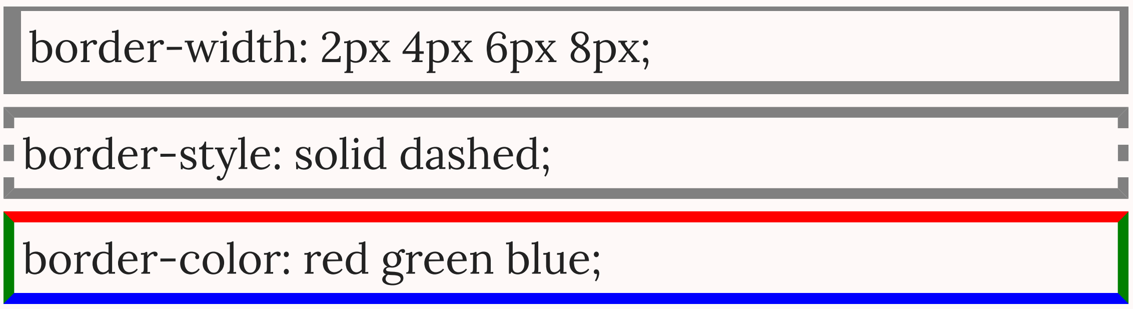


Border Shorthands

Each of the three border properties (*border-width*, *border-style*, and *border-color*) is also a shorthand to set all **four** borders at once.

This means what we really have are, for example, **border-bottom-width**, **border-top-style**, or **border-left-color**; 12 (3 × 4) properties on total.

Just like with *margin* and *padding*, there are also shorthands for setting different values for each property at once:



```
border-width: 2px 4px 6px 8px;
border-style: solid dashed;
border-color: red green blue;
```

Border Radius

- The **border-radius** property is used to define how rounded border corners are.
- The curve of each corner is defined using **one or two** *radii*, defining its shape: **circle** or **ellipse**.
- We can set different border radius for each corner using the properties:
    - **border-top-left-radius**
    - **border-top-right-radius**
    - **border-bottom-right-radius**
    - **border-bottom-left-radius**.
- Values can be a *length* or a *percentage*.
- If two radii are used, they are separated by a `/`.

Shorthands

As with other properties, we can use more than one value in the *border-radius* property to simultaneously change the *radius* of several corners.

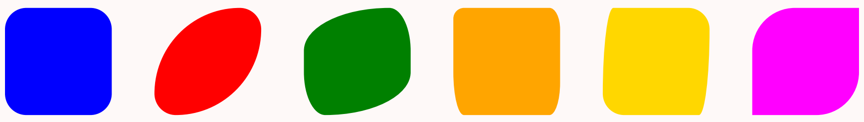
The possible combinations are as follows:

- One value: single radius for the whole element.
- Two values: **top-left-and-bottom-right** and **top-right-and-bottom-left**.
- Three values: **top-left**, **top-right-and-bottom-left** and **bottom-right**.
- Four values: **top-left**, **top-right**, **bottom-right**, **bottom-left**.

Examples

```html
<div id="a"></div><div id="b"></div><div id="c"></div>
<div id="d"></div><div id="e"></div><div id="f"></div>
```

```css
div {
  width: 50px; height: 50px;
}
#a { border-radius: 10px; background-color: blue;}
#b { border-radius: 40px 10px; background-color: red;}
#c { border-radius: 40px 10px / 20px 20px; background-color: green;}
#d { border-radius: 10% / 10% 20% 30% 40%; background-color: orange;}
#e { border-radius: 10% 20% / 40px 10px; background-color: gold;}
#f { border-radius: 20px 0; background-color: fuchsia;}
```



Margin Collapse

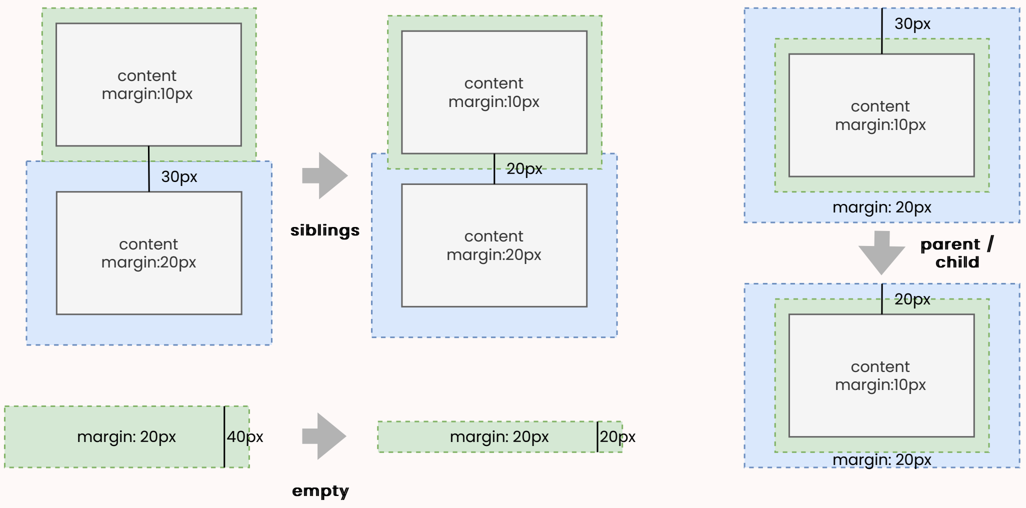
Adjacent margins collapse in three different cases:

- The margins of **adjacent siblings** are collapsed.
- The margin of **parents** and **descendants** with no separating content.
- The top and bottom margin of **empty** elements.

Margins collapse into a **single margin** with size equal to the **largest** of the individual margins.

Margin Collapse Examples

The three types of margin collapse, visualized:



Background

Image

- Besides having a background color, elements can also have an image as background using the **background-image** property.
- This property accepts an URL as its value.

```css
nav#menu {
  background-image: url('squares.png');
}
```

Position

The position of the background image can be set using the **background-position** property. This property receives two values:

- The first one can be **left**, **right**, **center** or a **length**.
- The second one can be **top**, **bottom**, **center** or a **length**.

```
nav#menu {
  background-image: url('squares.png');
  background-position: left top;
}
```

Attachment

- Using the **background-attachment** property, we can specify if the background should or not scroll with the page or element.
- Possible values are **fixed** (in relation to the viewport), **scroll** (in relation to the element) and **local** (in relation to the content).
- Scroll is the default value.

```
nav#menu {
  background-image: url('squares.png');
  background-position: left top;
  background-attachment: local;
}
```

https://css-tricks.com/almanac/properties/b/background-attachment/

Repeat

We can also define if the background repeats along one or both axis with the **background-repeat** property. Possible values are **no-repeat**, **repeat-x**, **repeat-y** and **repeat**.

```css
nav#menu {
  background-image: url('squares.png');
  background-position: left top;
  background-attachment: local;
  background-repeat: repeat;
}
```

Clipping

- By default, background properties, like **background-color**, apply to the content, padding, and border.
- This can be changed using the **background-clip** property.
- The possible values are: **border-box** (default), **padding-box** (only content and padding) and **content-box** (only content).

https://css-tricks.com/almanac/properties/b/background-clip/

Shorthands

- The **background** shorthand property sets all the background properties (including color) in one declaration.
- The properties that can be set, are: **background-color**, **background-position**, **background-size**, **background-repeat**, **background-origin**, **background-clip**, **background-attachment**, and **background-image**.
- It does not matter if one or more of the values above are missing.

```css
nav#menu {
  background: url('squares.png') repeat left top;
}
```

Lists

Markers

- In lists, each item has left markers that define their position.
- We can change the markers of both types of lists (ordered and unordered) using the **list-style-type** property.
- Some possible values for unordered lists are: **none**, **disc** (default), **circle** and **square**.
- For ordered lists we can use: **none**, **decimal** (default), **lower-alpha**, **lower-greek**, **lower-roman**, **upper-alpha** and **upper-roman**.

```css
#menu ul { list-style-type:none }
.article ol { list-style-type:lower-roman }
```

Images as Markers

It is also possible to use an arbitrary image as the list marker:

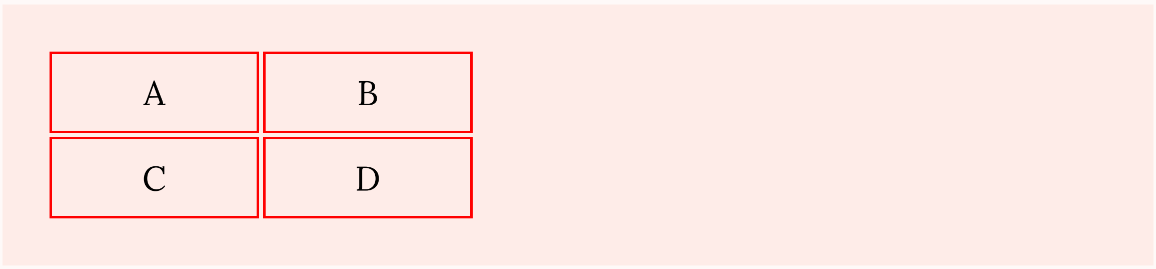
```css
div#menu ul{
  list-style-image: url('diamong.gif');
}
```

Tables

Borders

To draw border around table elements we can use the **border** property that we have seen before:

```
table, th, td {
    border: 1px solid red;
}
```



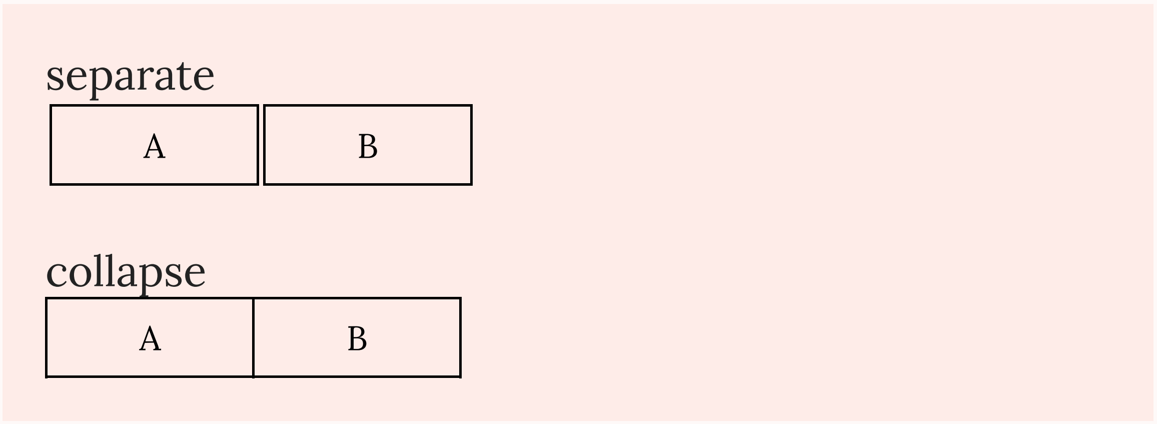
A	B
C	D

Collapse Borders

There is a gap between the borders of adjacent cells:

- To collapse borders into a single border, use the **border-collapse** property.
- The default value is **separate**.

```
table { border-collapse: collapse; }
td    { border: 1px solid; }
```



Transform

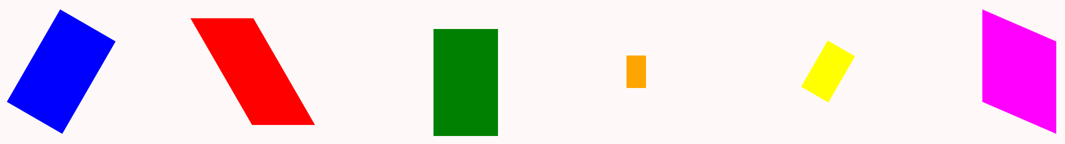
Transform

- The **transform** property modifies the coordinate space of the CSS visual formatting model:
    - A space-separated list of transforms applied one after the other.
- The **transform-origin** property specifies the origin of the transformation:
    - By default, it is at the center of the element.
    - It takes two values (x-offset and y-offset) that can be a length, a percentage, or one of *left*, *center*, *right*, *top*, and *bottom*.

Examples

```html
<div id="a"></div><div id="b"></div><div id="c"><
<div id="d"></div><div id="e"></div><div id="f"><
```

```css
div {
  margin: 30px;
  float: left;
  width: 50px; height: 50px;
}
#a {transform: rotate(30deg); background-color: blue;}
#b {transform: skew(30deg); background-color: red;}
#c {transform: translate(10px, 10%); background-color: green;}
#d {transform: scale(0.3); background-color: orange;}
#e {transform: rotate(30deg) scale(0.5); background-color: yellow;}
#f {transform: skew(30deg) rotate(30deg); background-color: fuchsia;}
```



Transition

Transition

- Provide a way to control **animation speed** when changing CSS properties
- Instead of having property changes take effect immediately, you can cause changes in a property over a period of time.
- CSS transitions let you decide:
    - which properties to animate (**list**)
    - when the animation will start (**delay**)
    - how long the transition will last (**duration**)
    - how the transition will run (**timing function**): ease, ease-in, ease-out, ease-in-out, linear, step-start, step-end

```
transition-property: opacity, left, top, height;
transition-duration: 3s, 5s; /* repeats (3s, 5s,
transition-delay: 1s;        /* same for all prop
transition-timing-function: ease-in;
```

Example

There is also a shorthand to set all these properties.

```css
.box {
    margin: 0 auto;
    border: 1px solid;
    width: 100px; height: 100px;
    background-color: #0000FF;
    transition: width 2s, height 2s, background-color 2s, transform 2
}

.box:hover {
    background-color: #FFCCCC;
    width: 150px; height: 150px;
    transform: rotate(180deg);
    border-radius: 50%;
}
```

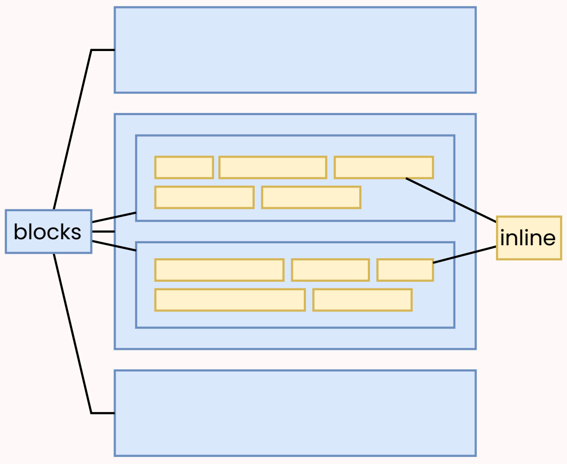
The Flow

Normal Flow

Normal flow, or flow layout, is how elements are placed on a page before any layout changes.

There are two primary types of elements contributing to this flow: **block** and inline **elements**.

- Block elements start on the **top** and move **down** the page.
- Inline elements start on the **left** (or the right, depending on the *locale*) and move to the **right**.



Display

- The display property controls if an element is **block** or **inline**.
- There are **many** possible values for this property.
- For example, **tables** (rows and cells) and **lists** (and items) have **specific** display values.
- But we will concentrate on four of them: **block**, **inline**, **inline-block**, and **none**.

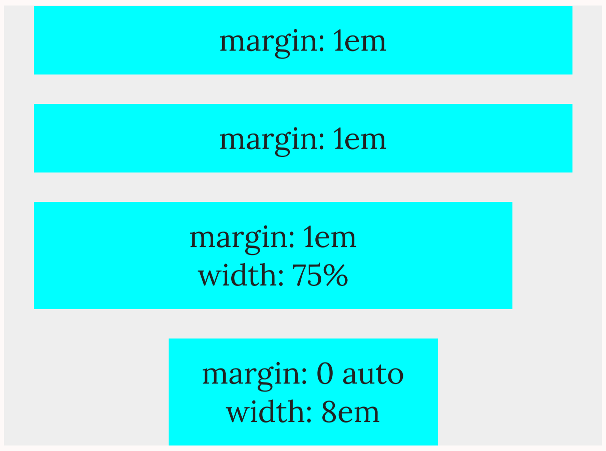
```css
img {
  display: block;
}
```

❗ We are simplifying some concepts. Here are all the nasty details!

Block

- Always take a **new line**.
- If its width is *auto*, it occupies the **entire horizontal space** of its parent element.
- If its height is *auto*, it will be **as tall as needed** to contain its child elements.
- Respects **margins** and **padding**.
- Can **contain** other *block-level* and *inline-level* elements.

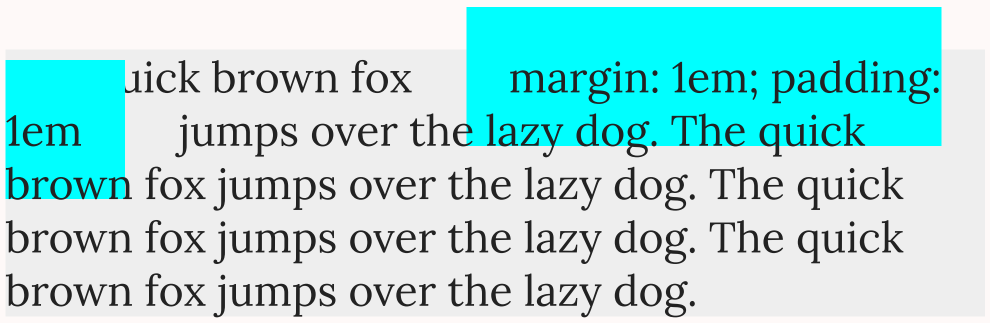
These are some *block* elements: *p*, *h1–h6*, *main*, *section*, *article*, *header*, *footer*, and *div*.



Inline

- Layed out **horizontally** one after the other.
- Ignore any width or height values. They only take as much space as necessary.
- **Top** and **bottom** margin and padding do not affect other elements.
- May be **aligned vertically** on their tops, bottoms, baselines, and others.
- Can break from one line to the next if there is no more space.

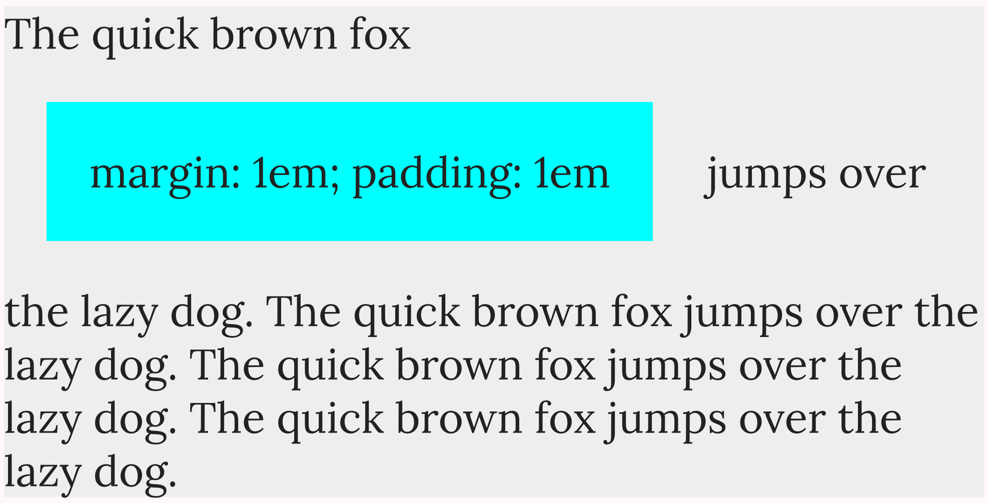
These are some *inline* elements: *a*, *strong*, *em*, *span*, and *text*.


uick brown fox margin: 1em; padding: 1em jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

A paragraph with a cyan span inside.

Inline-Block

- Inline elements that **behave** as block elements.
- Block elements that **stack** horizontally.


The quick brown fox

margin: 1em; padding: 1em

jumps over

the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.

A paragraph with a cyan span inside having **display: inline-block**.

None

- Setting the **display** property to none, **removes** the element from the page completely.
- This is different from making it invisible (with the *visibility* attribute).
- The element disappears and does not occupy any space on the page.

Changing the Flow

The **position** property allows the developer to alter how an element is positioned.

There are four possible values:

- static
- relative
- fixed
- absolute

Position Example

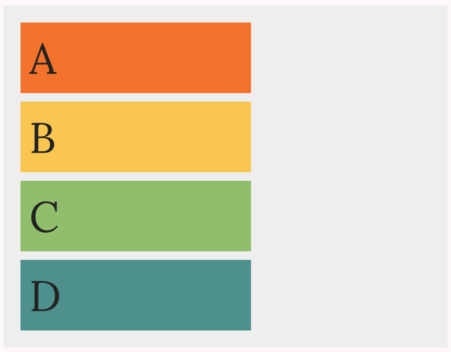
The next few pages will use the following example:

```
<section>
  <article id="a">A</article>
  <article id="b">B</article>
  <article id="c">C</article>
  <article id="d">D</article>
</section>
```

```
section { background-color: #eee; width: 10em; pa
article { width: 5em; margin: 0.2em; padding: 0.2
#a { background-color: #f3722c } #b { background-
#c { background-color: #90be6d } #d { background-
```

Static

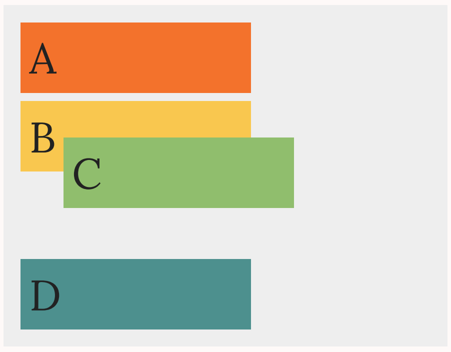
- The **default** value.
- The element keeps its place **in the document flow**.



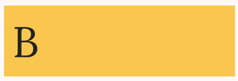
```
article {
  position: static;
}
```

Position Relative

- The element keeps its position **in the flow**.
- But can be moved relative to its static position using **top**, **right**, **bottom**, and **left**.

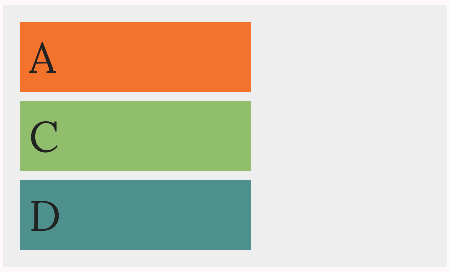


```
#b {
  position: relative;
}
#c {
  position: relative;
  left: 20px;
  top: -20px;
}
```



Position Fixed

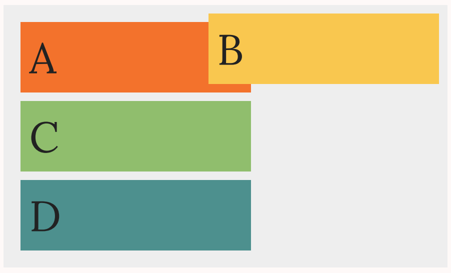
- The element is **no longer a part of the flow**.
- Can be positioned relative to the **browser window**.
- **Scrolling doesn't** change the element's **position**.



```css
#b {
  position: fixed;
  right: 1em;
  top: 1em;
}
```

Position Absolute

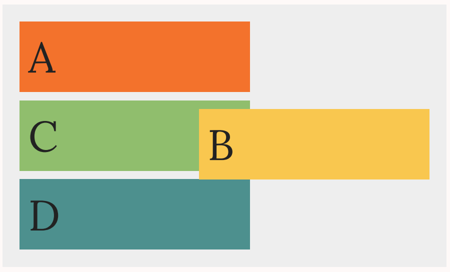
- The element is **no longer a part of the flow**.
- But it **still scrolls** with the page.
- Can be positioned relative to its **first positioned parent** (non-static).



```
section { position: relative }
#b {
  position: absolute;
  right: 0;
  top: 0;
}
```

Float

The float property removes an element from the document flow and shifts it to the **left** or to the **right** until it touches the edge of its containing box or another floated element.

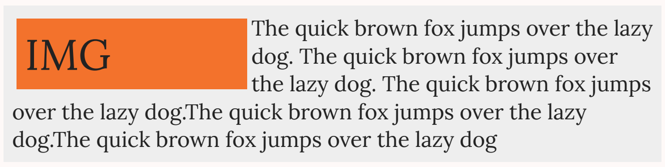


```
#b {
  float: right;
}
```

Articles "b" and "c" are misaligned due to a strange phenomenon. As "b" is no longer part of the flow, its top margin doesn't collapse anymore.

Floats and Text

Text always **flows around** floated elements. This is useful to make text that **flows around** images.

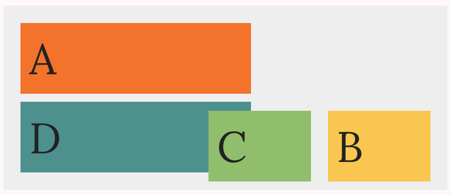

IMG
The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog.The quick brown fox jumps over the lazy dog.The quick brown fox jumps over the lazy dog

```
<section>
  <img id="a">
  <p>...text...</p>
</section>
```

```
#a {
  float: left;
}
```

Multiple Floats

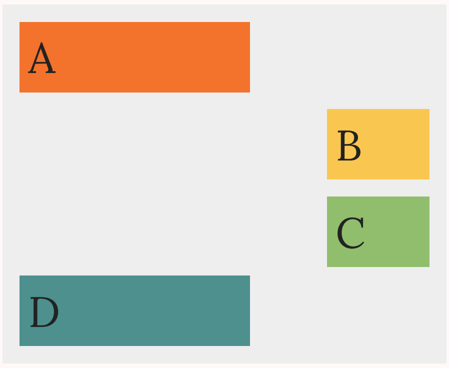
Floats go right or left until they find **another float** or the **parent container**.



```
#b, #c {
  width: 2em;
  float: right;
}
```

Clear

- The clear property indicates if an element can be next to floating elements that precede it or must be moved down.
- Values can be **left**, **right** or **both**.



```
#b, #c {
  width: 2em;
  float: right;
}
#c, #d {
  clear: right;
}
```

Ordering

- When elements are positioned outside the normal flow, they can **overlap** others.
- The **z-index** property specifies the stack order of an element and its descendants.
- But it can only be applied to **positioned elements** (non-static).
- An element with **greater** stack order is always in **front** of an element with a lower stack order.
- By **default**, the elements are stacked following the order they are declared in the HTML.

```css
#b {
  position: relative;
  z-index: -1;
}
```

Overflow

- The **overflow** property specifies the behavior of an element when its contents don't fit its specified size.
- Possible values are:
    - **visible**: The overflow is not clipped. It renders outside the element's box. This is the default.
    - **hidden**: The overflow is clipped, and the rest of the content will be invisible.
    - **scroll**: The overflow is clipped, but a scroll bar is added to see the rest of the content.
    - **auto**: If overflow is clipped, a scroll bar should be added to see the rest of the content.

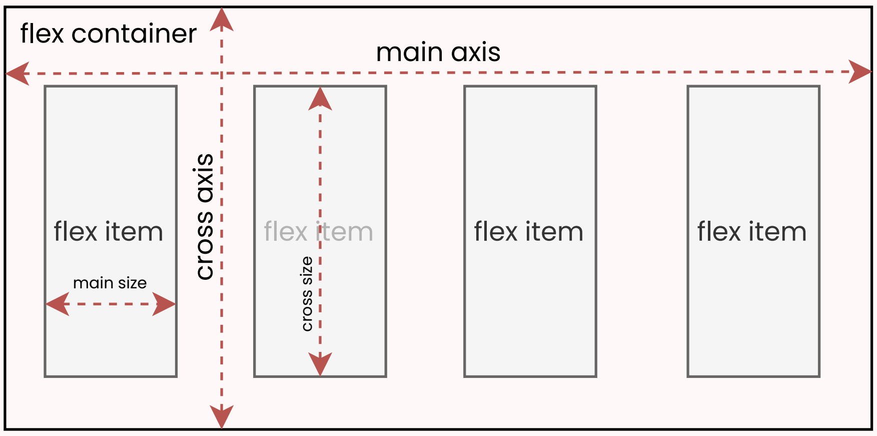
Flexbox

Flexbox

- A direction agnostic alternative to the box model layout model.
- Flexbox provides block level arrangement of **parent** and **child** elements that are **flexible** to adapt to display size.
- Flexbox items **cannot** be floated.
- The flex container's margins **do not collapse** with the margins of its contents.

A Complete Guide to Flexbox

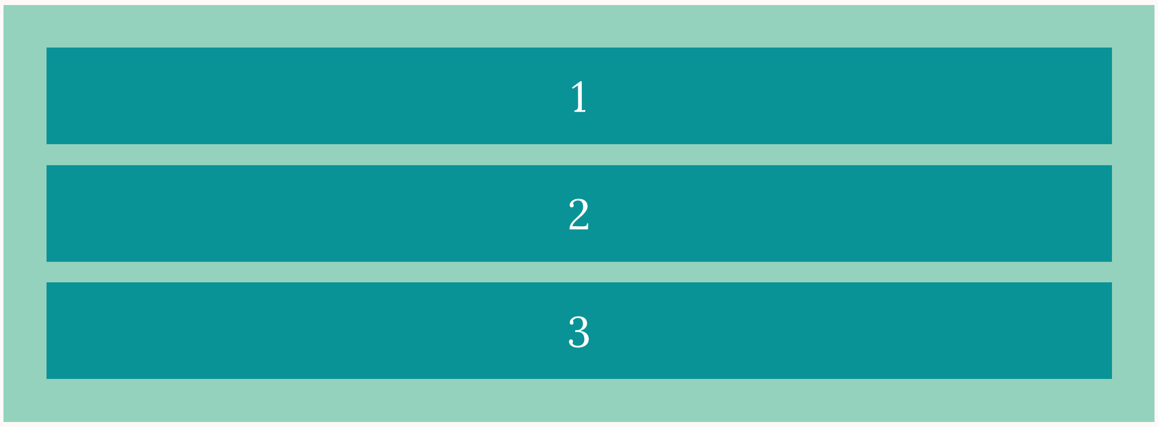
Flexbox Vocabulary



Running Example

```html
<div class="container">
  <div class="item">1</div>
  <div class="item">2</div>
  <div class="item">3</div>
</div>
```

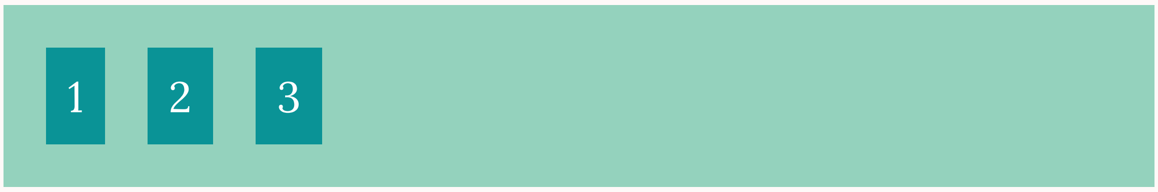
```css
.container {
  background-color: #94d2bd;
  padding: 0.5em;
}

.item {
  color: white; text-align: center;
  margin: 0.5em; padding: 0.5em;
  background-color: #0a9396;
}
```



Flex

Changing the display property of the container to *flex* transforms the contained items into flexboxes.

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

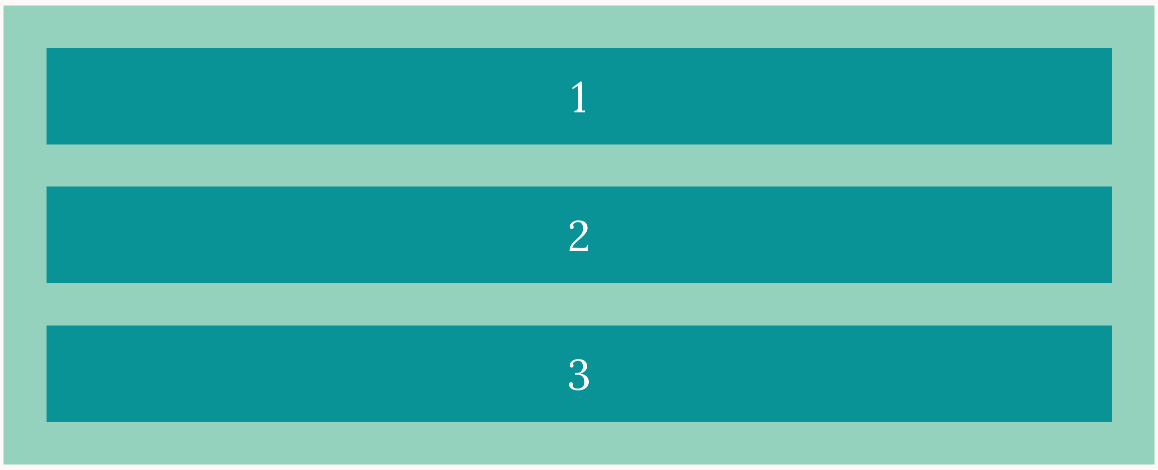


By default, the *main* axis is horizontal from left to right.

Flex Direction

We can change the direction of the *main* axis by changing the flex-direction property of the container to: **row**, **row-reverse**, **column** or **column-reverse**.

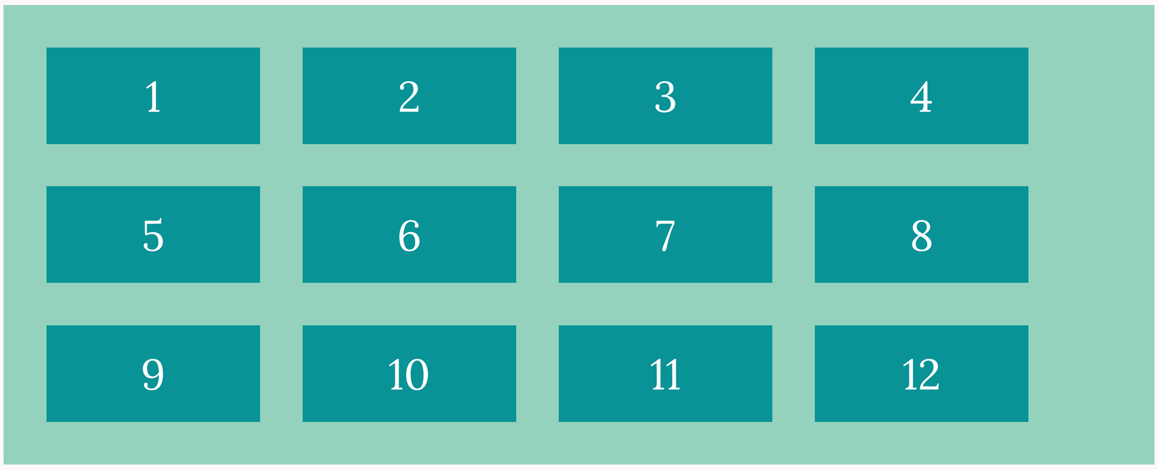
```
.container {
  flex-direction: column;
}
```



Flex Wrap

The flex-wrap property allows us to specify how items should wrap when changing lines: **nowrap**, **wrap**, **wrap-reverse**. The default is **nowrap**.

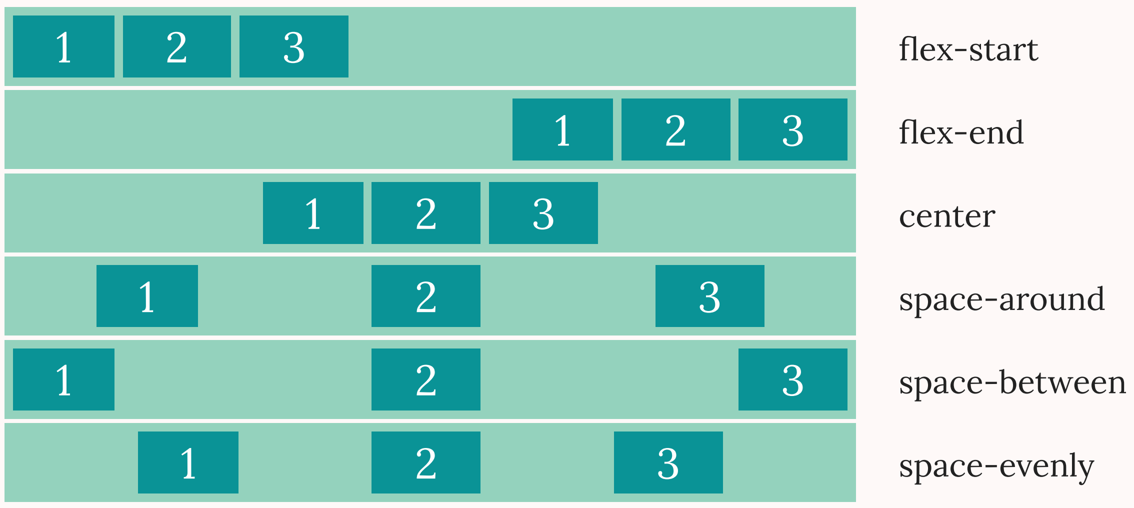
```css
.container {
  flex-wrap: wrap;
}
.item {
  width: 4em;
}
```



Justify Content

The justify-content property defines the alignment along the **main** axis allowing the distribution of extra space: **flex-start**, **flex-end**, **center**, **space-around**, **space-between**, **space-evenly**. The default is **flex-start**.

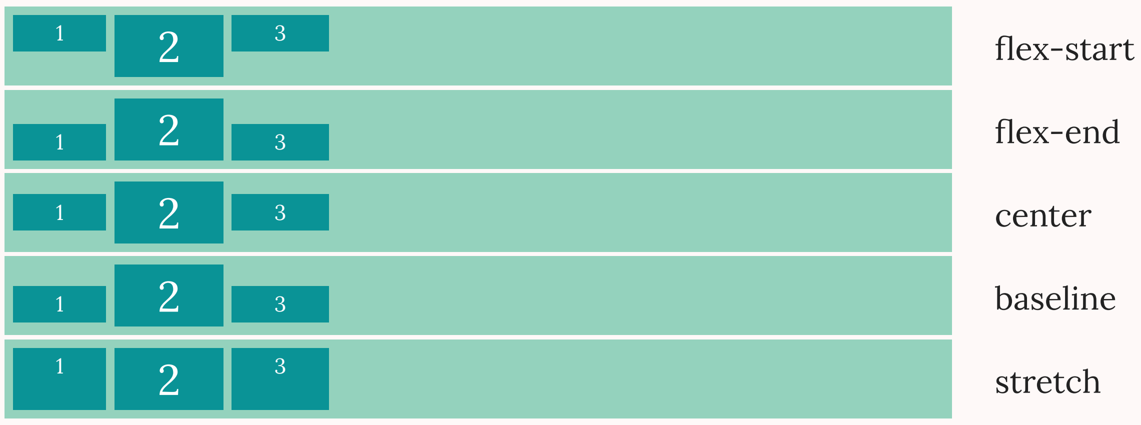
```
.container {
  justify-content: flex-start;
}
```



Align Items

The align-items property defines the default behaviour for how flex items are laid out along the **cross** axis on the current line: **flex-start**, **flex-end**, **center**, **baseline**, **stretch**. The default is **stretch**.

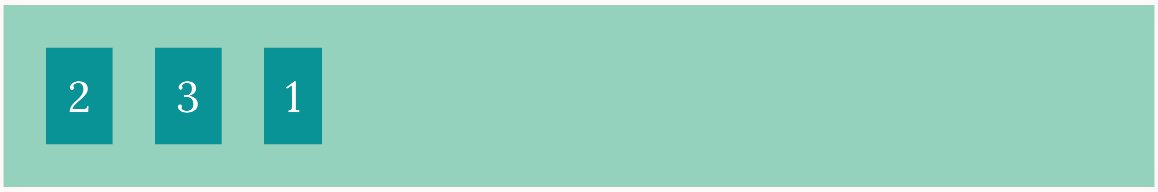
```
.container {
  align-items: flex-start;
}
```



Order

The order property alters the order in which a **flex item** is laid out in its container.

```
.item:first-child {
   order: 3;
}
```



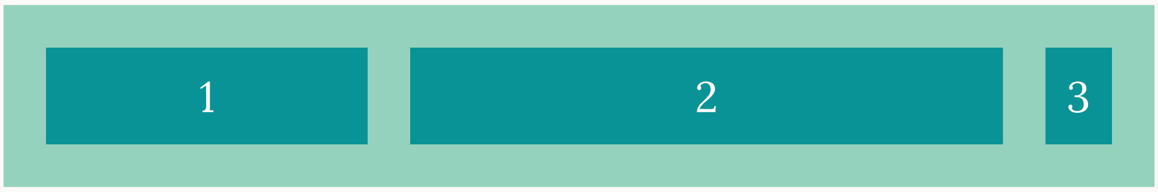
Notice that we are targetting the items now!

Grow and Shrink

The flex-grow and flex-shrink properties define the ability for a flex item to **grow** (if there is extra space) or **shrink** (if there isn't enough):

- They accept a unitless value that serves as a **proportion**.
- The default is **0** for *flex-grow*, which means items don't grow by default.
- The default is **1** for *flex-shrink*, which means items shrink equally.

```
.item:nth-child(1) {
  flex-grow: 1;
}

.item:nth-child(2) {
  flex-grow: 2;
}
```

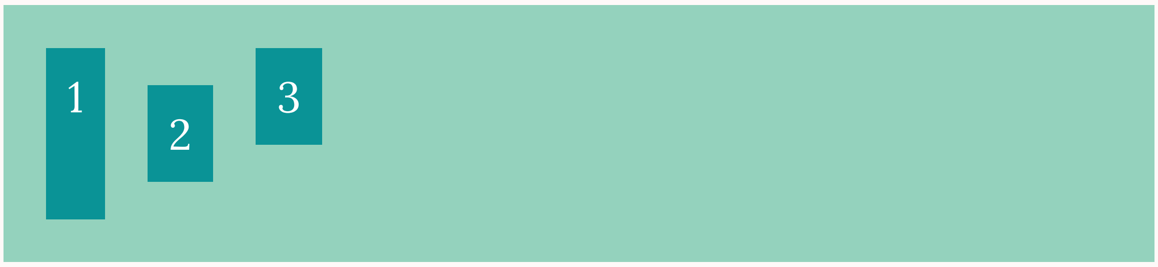


Align Self

The align-self property allows the alignment specified by *align-items* to be overridden for individual flex items. The default value is **auto**, meaning that items follow the alignment set by *align-items*.

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

.item:nth-child(1) { height: 3em; }
.item:nth-child(2) { align-self: center; }
```

All items aligned as **flex-start** except the second one that is **center**-aligned.



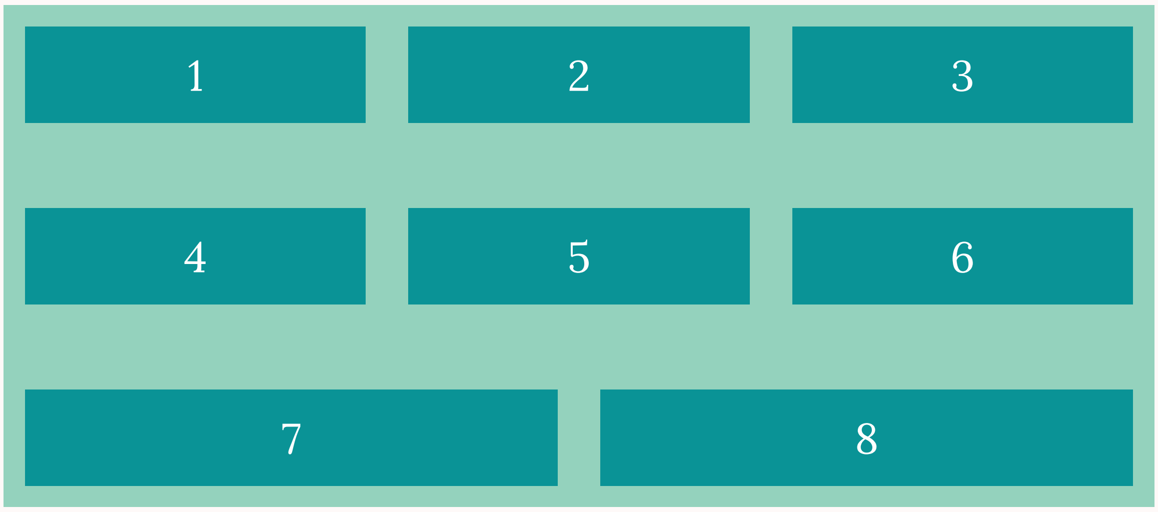
Gap

The gap property is a shorthand for two other properties:

- **row-gap**: a gap between every row in a flexbox.
- **column-gap**: a gap between every column in a flexbox.

You can either pass only one value (a length) and set both simultaneously or two values and set each one individually (*row-gap* first, then *column-gap*).

```
.container { gap: 2em 1em; }

.item { width: 6em; margin: 0; flex-grow: 1; }
```



Grid

Grid

A grid layout enables us to align elements into **columns** and **rows** of different sizes.

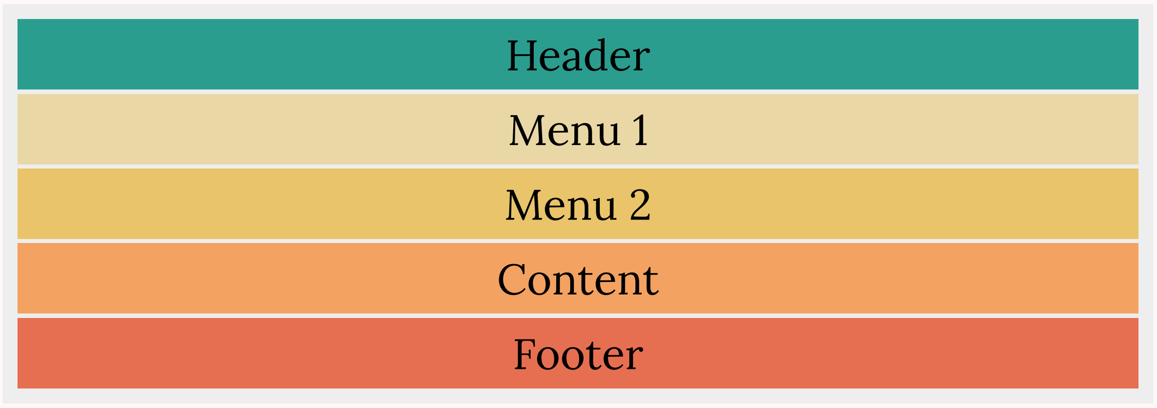
Elements in a grid layout can occupy the same cells as other elements, thus **overlapping** and creating **layers**.

A Complete Guide to Grid

Running Example

```html
<div class="container">
  <div class="item header">Header</div>
  <div class="item menu1">Menu 1</div>
  <div class="item menu2">Menu 2</div>
  <div class="item content">Lorem ipsum...</div>
  <div class="item footer">Footer</div>
</div>
```

```css
.container {
  background-color: #eee;padding: 5px;
}
.item {
  color: black; text-align: center; margin: 2px; padding: 0.2em; back
}
```

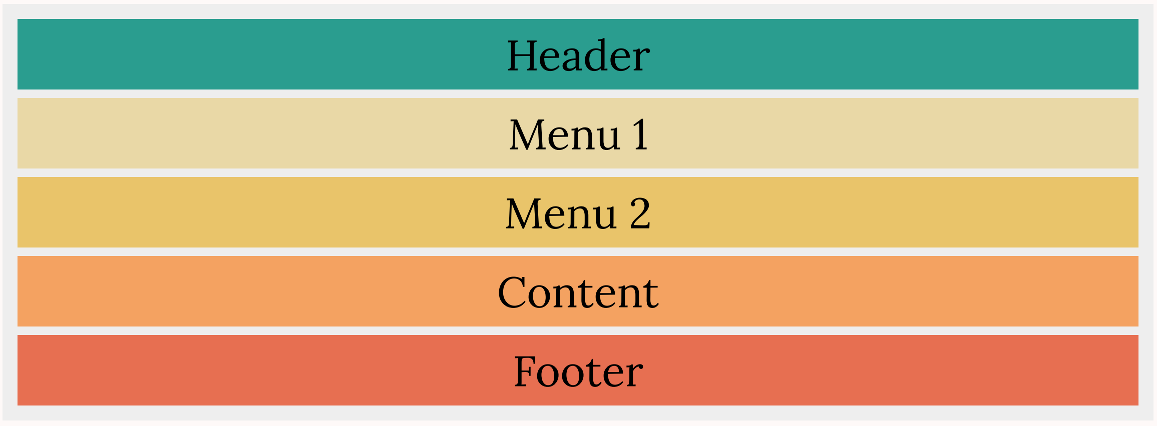


Grid

Changing the display property of the container to *grid* transforms it into a grid layout.

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

By default, there is only one column.

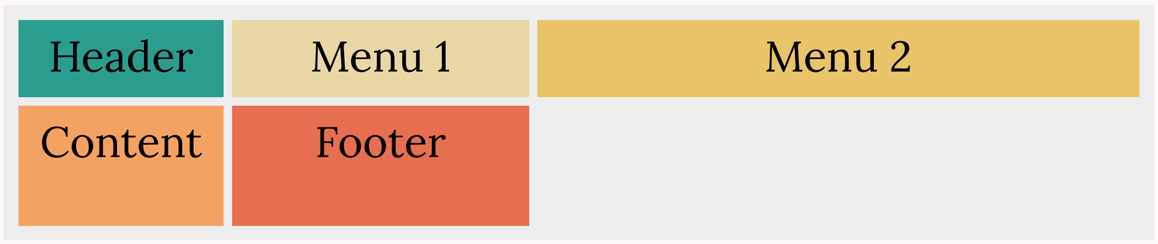


Grid Templates

The grid-template-columns and grid-template-rows properties allow us to define the number and size of the columns and rows of our table.

Sizes can be defined as **auto**, a **length**, a **percentage** or a **fraction** of the free space (using the *fr* unit).

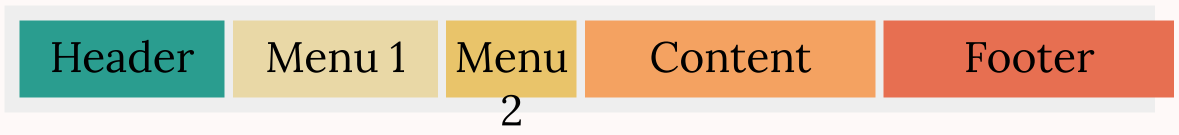
```
.container {
  grid-template-columns: 5em 1fr 2fr;
  grid-template-rows: 2em 3em;
}
```



Grid Templates Repeating

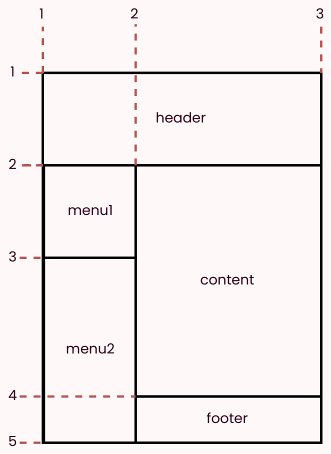
The repeat keyword can be used to simplify grid templates with many columns and rows of the same size.

```
.container {
  grid-template-columns: repeat(2, 5em) 1fr repea
  grid-template-rows: 2em;
}
```



Numerical Names

By default, gridlines are assigned a numerical value starting on *one*.



Assigning Location

We can assign a **location** to an item within the grid by referring to specific grid lines using the grid-column-start, grid-column-end, grid-row-start, and grid-row-end properties.

```
.header {
  grid-column-start: 1;
  grid-column-end: 3;
  grid-row-start: 1;
  grid-row-end: 2;
}
```

The *end* values can also be the number of **rows** or **columns** to **span**. By default, these values have a *span* of 1.

```
.header {
  grid-column-end: span 2;
  grid-row-end: span 1;    /* not needed - defaul
}
```

Location Shorthand

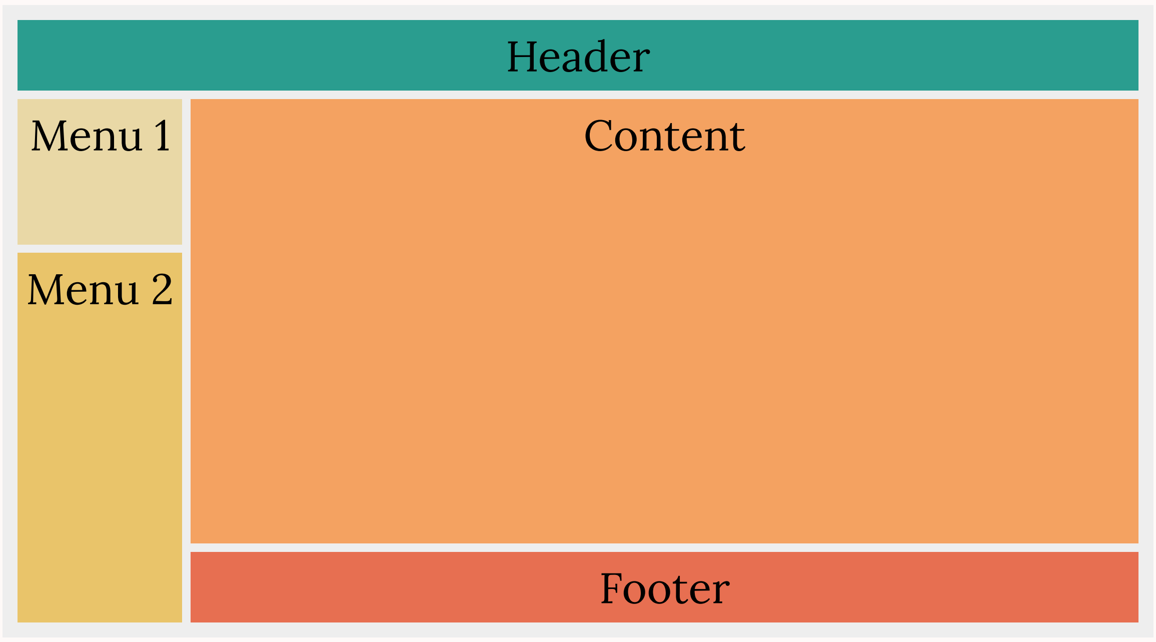
The grid-column and grid-row properties can be used as a **shorthand** for assigning the location of an item. Each receives **two values** separated by a **forward slash** (start / end).

The grid-area property can be used as a **shorthand** for the **four values** at once: *row-start / column-start / row-end / column-end.*

```
.header {
  grid-area: 1 / 1 / span 1 / span 2;
}

.menu1 {
  grid-column: 1; grid-row: 2;
}

.menu2 {
  grid-column: 1; grid-row: 3 / 5;
}

.content {
  grid-column: 2; grid-row: 2 / span 2;
}

.footer {
  grid-column: 2; grid-row: 4;
}
```

Location Result

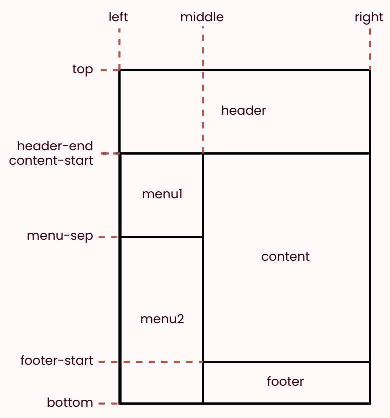
```
.container {
  grid-template-columns: auto 1fr;
  grid-template-rows: auto auto 1fr auto;
}
.menu1   { height: 3em;  } /* to simulate  */
.content { height: 10em; } /* some content */
```



Grid Line Names

We can assign **names** (more than one) to the grid lines.

```css
.container {
  grid-template-columns: [left] auto [middle] 1fr [right];
  grid-template-rows: [top] auto [header-end content-start] auto
                      [menu-sep] 1fr [footer-start] auto [bottom];
}
.content {
  grid-area: content-start / middle / footer-start / right;
}
```



Grid Template Areas

We can define a **grid template** more visually by giving names to items using the *grid-area* property.

Any number of **adjacent periods** can be used to declare a single empty cell.

```
.container {
  grid-template-columns: auto 1fr;
  grid-template-rows: auto auto 1fr auto;

  grid-template-areas: "header header"
                       "menu1  content"
                       "menu2  content"
                       "menu2  footer";
}

.header { grid-area: header; }

.menu1 { grid-area: menu1; }

.menu2 { grid-area: menu2; }

.content { grid-area: content; }

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

Aligning Items

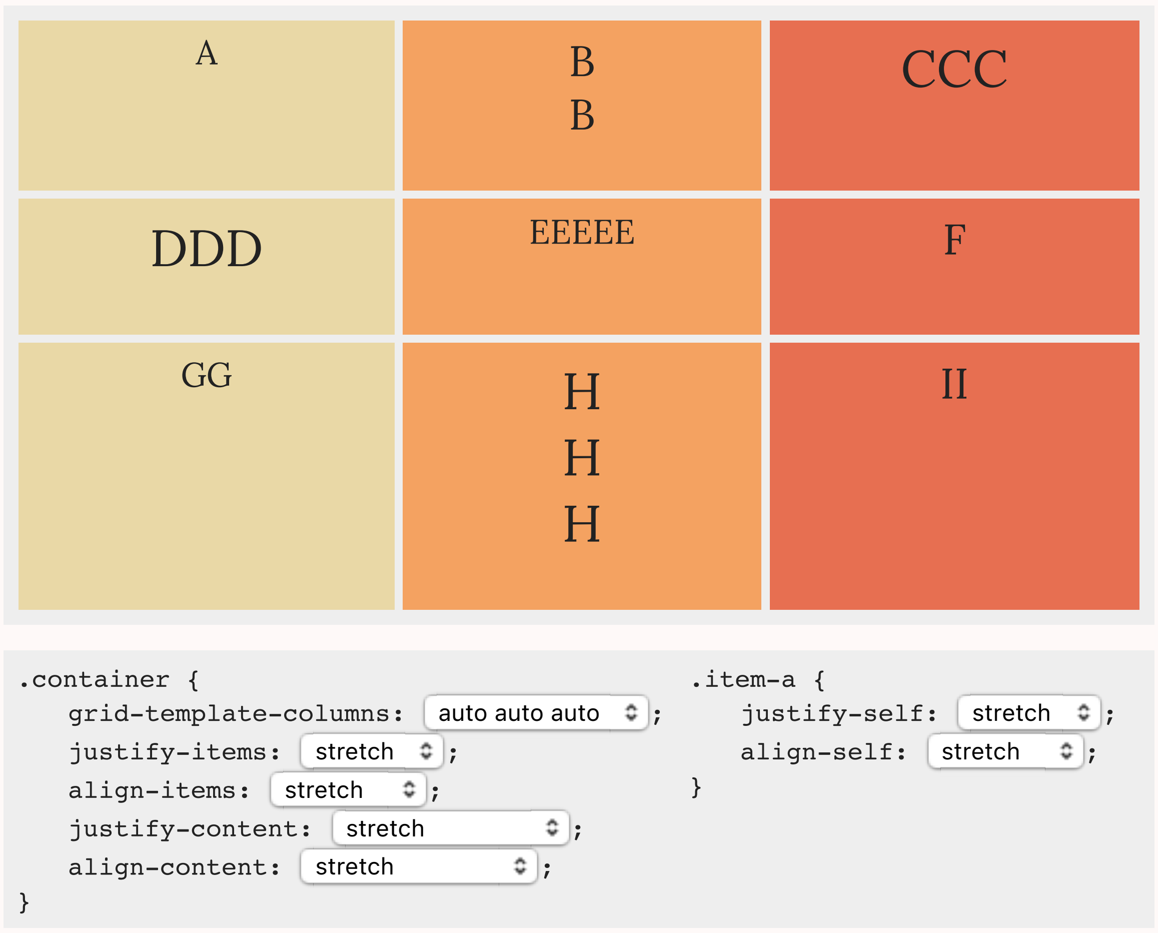
Like with *flexbox*, we can adjust the placement of each item inside its cell on the grid. For that, we use the following properties:

- column-gap, row-gap, and gap: they work just like in **flexbox**.
- justify-items: Align items along the **row** with: *start*, *end*, *center*, and *stretch*.
- align-items: Align items along the **column** with: *start*, *end*, *center*, *stretch*, and *baseline*.
- justify-content and align-content: Align the rows and columns inside the grid **container** (if they have room to spare) with: *start*, *end*, *center*, *stretch*, *space-around*, *space-between*, *space-evenly*.

There are also properties to adjust each **item individually**: justify-self and align-self.

Alignment Example

An interactive example to showcase different alignments:



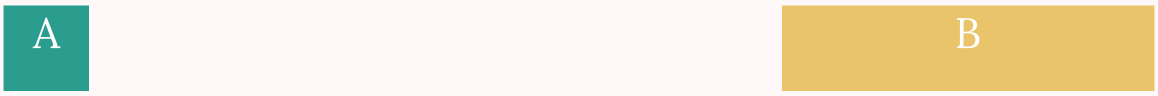
```
.container {                                             .item-a {
    grid-template-columns: auto auto auto ;                  justify-self: stretch ;
    justify-items: stretch ;                                 align-self: stretch ;
    align-items: stretch ;                               }
    justify-content: stretch ;
    align-content: stretch ;
}
```

Implicit Cells

You can assign an item to a location you **did not define** using grid-template-columns and grid-template-rows.

In that case, the needed column and rows are **automatically** added with size auto.

```
.container { grid-template-rows: 2em; grid-templa
.item_a { grid-column-start: 1 }
.item_b { grid-column-start: 4 }
```



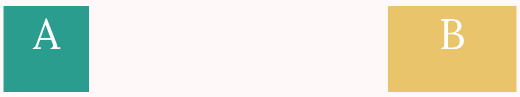
There are three **implicit** grid tracks in this example.

Auto Column and Rows

We can specify the **size** of any **implicit grid tracks** using *grid-auto-rows* and *grid-auto-columns*.

These properties can receive a **length** — or a series of lengths — that is **repeated** as needed.

```
.container {
  grid-template-rows: 2em; grid-template-columns: 2em; grid-auto-colu
}
.item_a { grid-column-start: 1 }
.item_b { grid-column-start: 4 }
```



The three **implicit** grid tracks in this example have sizes *3em*, *4em*, and *3em*. Only the last one has content.

Auto Flow

If some items are **not explicitly** assigned a location, then an *auto-placement* algorithm is used to place them.

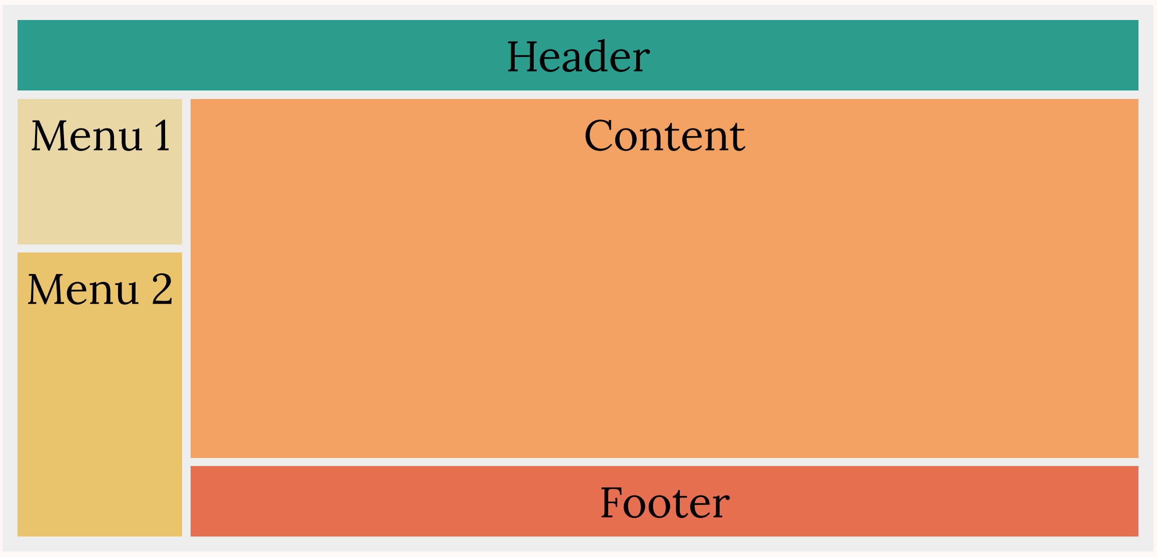
The behavior of this algorithm can be changed using the grid-auto-flow property:

- *row* - The **default** value; Fill empty spaces **row by row** (if the item fits) and add **new rows** if necessary.
- *column* - Fill empty spaces **column by column** (if the item fits) and add **new columns** if necessary.
- *dense* - If some spaces were left **empty**, see if items that appear **later** fit there and use those spaces. This **changes the order** of the items.

Simplified Positioning

We can use the **auto-placement** algorithm to **simplify** item positioning:

```
.container {
  grid-template-columns: auto 1fr;
  grid-template-rows: auto auto 1fr auto;
}
.header { grid-column-end: span 2 }
.content { grid-row: 2 / span 2; grid-column: 2 }
.menu2 { grid-row-end: span 2 }
```



Cascading

Example

What **color** will the **text** be? And what about the **link**?

```html
<section>
 <p>The quick brown fox <a href="#">jumps</a> ove
</section>
```

```css
section {
  color: red;
}
```

Example

What **color** will the **text** be? And what about the **link**?

```html
<section>
 <p>The quick brown fox <a href="#">jumps</a> ove
</section>
```

```css
section {
  color: red;
}
```

The quick brown fox jumps over the lazy dog

The text becomes red, but the link is still blue. Why?

Defaults

Each browser has **its own** set of default values for the properties of each HTML element.

These defaults are very similar between browsers, but slight differences can make cross-browser development harder.

Tip: There are several CSS stylesheets that normalize and reset each default value to mitigate this problem.

Inherit

A special value that can be used in almost every property:

- When a property is set to **inherit**, the value of that property is **inherited** from the element's **parent**.
- Most browser defaults have the value **inherit**.

```
<nav id="menu">
  <h1>Menu</h1> <!-- inherits the blue color from
</nav>
```

```
h1{
  color: inherit;
}

#menu {
  color: blue;
}
```

Example Revisited

- In most browsers, the **link** color is set as **blue**.
- On the other hand, the **paragraph** color is set as **inherit**.

```
<section>
 <p>The quick brown fox <a href="#">jumps</a> ove
</section>
```

```
a {
  color: blue;    /* Doesn't inherit the color */
}

p {
  color: inherit; /* Inherits the color from the
}

section {
  color: red;
}
```

Specificity

What about this example?

```html
<nav id="menu">
  <p>What is my color?</p>
</nav>
```

```css
#menu p {
  color: green;
}

nav p {
  color: red;
}
```

Specificity

What about this example?

```html
<nav id="menu">
  <p>What is my color?</p>
</nav>
```

```css
#menu p {
  color: green;
}

nav p {
  color: red;
}
```

Green! Because the first **rule** is **more specific** than the second one.

Calculating Specificity

A rule's specificity is defined as three values (a, b, c).

Each one of them is incremented when a certain type of selector is used:

- **a**: Id
- **b**: Class, Pseudo class, Attribute
- **c**: Element, Pseudo Element

Ordering:

- Rules with a bigger **a** value are **more specific**.
- If the **a** value is the same for both rules, the **b** value is used for comparison.
- If still needed, the **c** value is used.

Specificity Examples

- *: - (0, 0, 0)
- p: 1 element – (0, 0, 1)
- div: 1 element – (0, 0, 1)
- #sidebar: 1 id – (1, 0, 0)
- div#sidebar: 1 element, 1 id – (1, 0, 1)
- div#sidebar p: 2 elements, 1 id – (1, 0, 2)
- div#sidebar p.bio: 2 elements, 1 class, 1 id – (1, 1, 2)

Specificity Calculator: http://specificity.keegan.st

Cascading

- The rule to be applied is selected using the following rules in order:
    - **Origin** (author, user, default).
    - **Specificity** (bigger is better).
    - **Position** (last is better).
- **Origin** Explanation:
    - **author**: The CSS rules defined by the page developer.
    - **user**: User defined preferences.
    - **default**: Browser defaults.

Vars

Vars

Entities that contain **reusable values**. Set using a **custom property** notation:

```css
body {
  --main-bg-color: blue;
  --default-margin: 1em;
}
```

Accessed using the *var()* function:

```css
body header {
  margin: var(--default-margin);
}
```

CSS Vars Inheritance

CSS vars are also **inherited**.

If no value is set for a *var* on a given element, the value of its *parent* is used.

```html
<section>
  <header>
    <h1>Title</h1>     <!-- red -->
    <h2>Sub-title</h2> <!-- blue -->
  </header>
</section>
```

```css
section  { --text-color: blue; }
h1       { --text-color: red; }
header * { color: var(--text-color); }
```

Extra: What if the * is removed?

Responsive Design

Responsive Design

Responsive web design makes websites that work effectively on both desktop browsers and the myriad of mobile devices on the market.

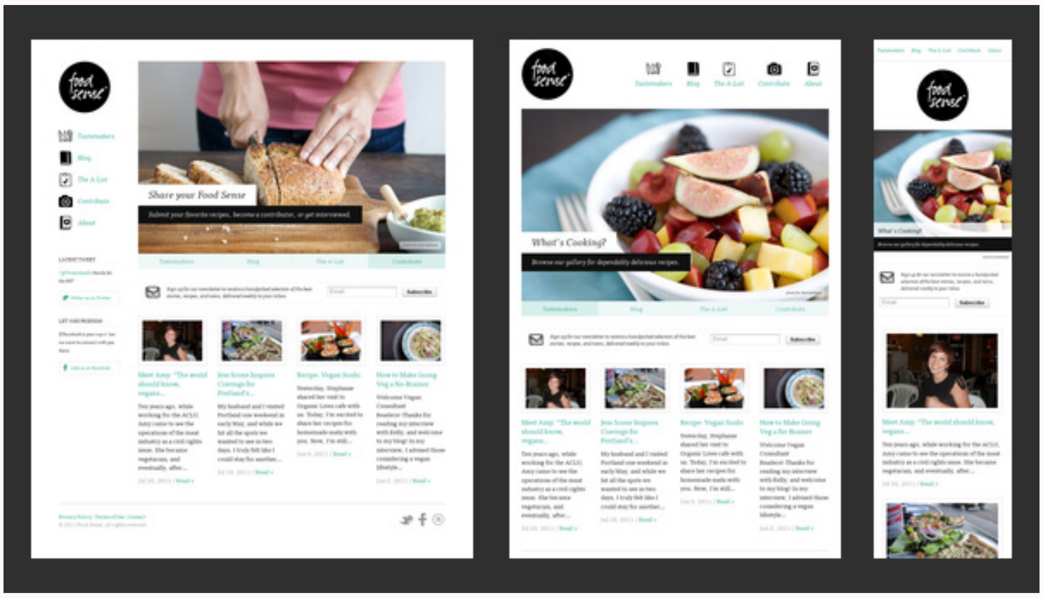


Image taken from http://designmodo.com/responsive-design-examples/

Responsive vs. Adaptative

Adaptive Design: Multiple **fixed** width layouts

Responsive Design: Multiple **fluid** grid layouts

Mixed Approach: Multiple fixed width layout for larger screens, multiple fluid layout for smaller screens.

Viewport

Pages optimized for various devices must include a meta viewport element in the head of the document.

A meta viewport tag gives the browser instructions on how to control the page's dimensions and scale.

```
<meta name="viewport"
      content="width=device-width, initial-scale=
```

- *width=device-width* matchs the screen's width in device independent pixels.
- initial-scale=1* establishs a 1:1 relationship between CSS pixels and device independent pixels.

Learn more: MDN and a tale of two viewports part 1 and part 2.

Media Queries

A **media-query** is composed of a **media type** and/or a number of **media features**.

They can be used when linking to a CSS file from HTML or directly in the CSS code (allowing dynamic changes).

```
<link rel="stylesheet"
      media="(min-width: 600px) and (max-width: 8
      href="medium.css" />
```

```
@media (max-width: 600px) {
  .sidebar {
    display: none;
  }
}
```

Media Types

The media type indicates in what media type the CSS is to be applied.

- **all** - suitable for all devices.
- **print** - intended for paged material and for documents viewed on screen in print preview mode.
- **screen** - intended primarily for color computer screens.
- **speech** - intended for speech synthesizers (aural in CSS2).

```
<link rel="stylesheet" media="print" href="print.
```

Media Features

- **min-width** width over the value defined in the query.
- **max-width** width under the value defined in the query.
- **min-height** height over the value defined in the query.
- **max-height** height under the value defined in the query.
- **orientation=portrait** height is greater than or equal to the width.
- **orientation=landscape** width is greater than the height.

```
<link rel="stylesheet"
      media="(min-width: 800px)"
      href="large.css" />
```

Parentheses are required around expressions; failing to use them is an error.

Logical Operators

- **and** used for combining multiple media features.
- **comma-separated** lists behave as the logical operator **or**.
- **not** applies to the whole media query and returns true if the media query would otherwise return false.

```
<link rel="stylesheet"
      media="(min-width: 800px) and screen, print
      href="large.css" />
```

Learn more: MDN.

Vendor Prefixes

Vendor Prefixes

While the specification of selectors, properties, and values are still being finalized, it is normal for browsers to go through an **experimentation** period.

Browsers might also have **proprietary** extensions to the CSS standard.

In order to accommodate the release of vendor-specific extensions, the CSS specifications define a specific format that vendors should follow:

```css
.round {
  -webkit-border-radius: 2px;
  -moz-border-radius: 2px;
  border-radius: 2px;
}
```

Prefixes: **-webkit-** (chrome, safari), **-moz-** (firefox), **-o-** (opera), **-ms-** (internet explorer), ...

Check browser support: http://caniuse.com/

Validation

http://jigsaw.w3.org/css-validator/

Extra stuff

- Frameworks: Ink, Bootstrap, Flat UI, Pure
- Advanced/Experimental: Shadows, Animations
- Playgrounds: JSFiddle, CodePen
- Pre-processors: Less, Sass
- Information: Google Web Essentials, Mozilla Developer Network
- Icons: Font Awesome