UNIVERSITÀ DI PAVIA

Collegio Alessandro Volta
Via Adolfo Ferrata, 17, Pavia (PV)

# WEB DESIGN

Lecture 3 – Style HTML with CSS (and not only)


## Giovanni Nicola D'Aloisio

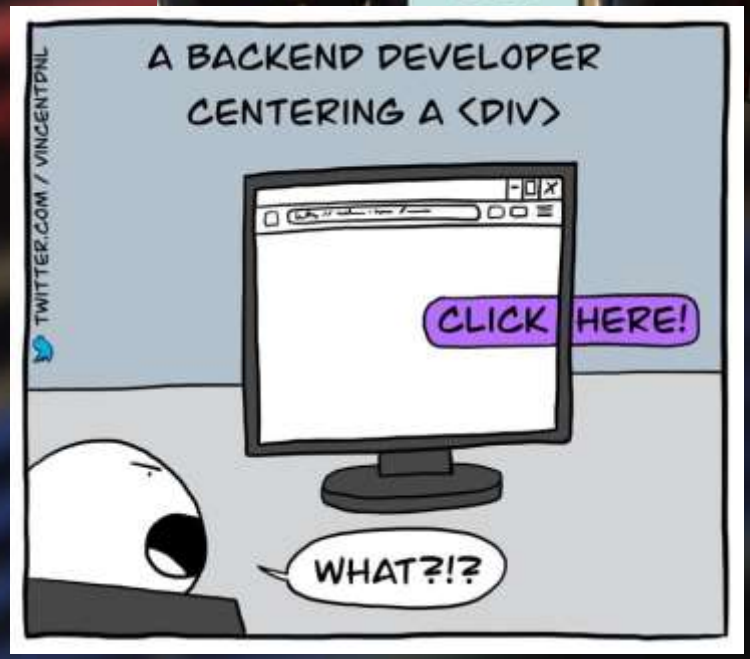
Department of Physics – University of Pavia
Classe di Scienze, Tecnologie e Società – IUSS Pavia

E-Mail: giovanninicola.daloisio01@universitadipavia.it

# What CSS is

Cascading Style Sheets — CSS — is the first technology you should start learning after HTML. While HTML is used to define the content, CSS is used to style it and lay it out.

From the beginning, you'll primarily apply colors to HTML elements and their backgrounds; change the size, shape, and position of elements; and add and define borders on elements. But there's not much you can't do once you have a solid understanding of even the basics of CSS. Once you know the fundamentals, usually you have a pretty good feel for what can and can't be done, even if you don't know how to do it yet!

# Starting with some HTML

Our starting point is an HTML document. Save the code below as `index.html` in a dedicated folder.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Getting started with CSS</title>
    <link rel="stylesheet" href="styles.css" />
  </head>

  <body>
    <h1>I am a level one heading</h1>

    <p>
      This is a paragraph of text. In the text is a
      <span>span element</span> and also a
      <a href="https://example.com">link</a>.
    </p>

    <p>
      This is the second paragraph. It contains an <em>emphasized</em> element.
    </p>

    <ul>
      <li>Item <span>one</span></li>
      <li>Item two</li>
      <li>Item <em>three</em></li>
    </ul>
  </body>
</html>
```

HTML

# Adding CSS to our document

The `<link>` element tells the browser that we have a stylesheet, using the `rel` attribute, and the location of that stylesheet as the value of the `href` attribute.

Now create a `styles.css` file inside the same folder, and put inside that file the following code:

```css
h1 {
  color: red;
}
```

Now save both `index.html` and `styles.css` files and open the index page using your browser.

# Styling HTML elements

By making your heading red, you have already demonstrated that you can target and style an HTML element using CSS technology. More exactly, you do this by targeting an element selector — a selector that directly matches an HTML element name.

To target all paragraphs in the document, you would use the selector `p`. To turn all paragraphs green, you would write then, inside your `styles.css` file:

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

You can target multiple selectors at the same time by separating the selectors with a comma.
As example, if you want all paragraphs and all list items to be green, your rule would look like this:

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

# Changing the default behavior of elements

When we look at a well-marked up HTML document we can see how the browser is making the HTML readable by adding some default styling. Headings are large and bold and our list has bullets. This happens because browsers have internal stylesheets containing default styles, which they apply to all pages by default; without them all of the text would run together in a clump and we would have to style everything from scratch. All modern browsers display HTML by default in pretty much the same way.

However, you can change the HTML element you want to change by using a CSS rule.

A good example is `<ul>`, an unordered list. It has list bullets. If you don't want those bullets, add:

```
li {
  list-style-type: none;
}
```
CSS

Or you can define other styles as those you can find here.

# Adding a class

Most of the time you don't want to make all the HTML tags in a single way, so you need to find a way to select a subset of the elements without changing the others. The most common way to do this is to add a class to your HTML element and target that class.

In your HTML document, add a class attribute to the second list item, like the example shown here. Then, in your CSS, you can target the class of special by creating a selector that starts with a full stop character.

Save, refresh and see the result.

HTML

```
<ul>
  <li>Item one</li>
  <li class="special">Item two</li>
  <li>Item <em>three</em></li>
</ul>
```

CSS

```
.special {
  color: orange;
  font-weight: bold;
}
```

# Styling things based on their location

There are times when you will want something to look different based on where it is in the document.

The selector on the right will select any `<em>` element that is inside an `<li>`. In your previous document, you should find that the `<em>` in the third list item is now purple, but the one inside the paragraph is unchanged.

Something else you might like to try is styling a paragraph when it comes directly after a heading at the same hierarchy level in the HTML. To do so, place a `+` (an adjacent sibling combinator) between the selectors.

CSS

```
li em {
  color: rebeccapurple;
}
```

CSS

```
h1 + p {
  font-size: 200%;
}
```

# Styling things based on state

- When we style a link, we need to target the <a> (anchor) element. This has different states depending on whether it is unvisited, visited, being hovered over, focused via the keyboard, or in the process of being clicked (activated). You can use CSS to target these different states — the CSS below styles unvisited links pink and visited links green.

CSS

```
a:link {
  color: pink;
}

a:visited {
  color: green;
}
```

- You can change the way the link looks when the user hovers over it, for example by removing the underline, which is achieved by the rule hover.

CSS

```
a:hover {
  text-decoration: none;
}
```


That's all Folks!

# Applying CSS to HTML – External stylesheets

First, let's examine three methods of applying CSS to a document:

- with an external stylesheet;
- with an internal stylesheet;
- with inline styles.

An external stylesheet contains CSS in a separate file with a `.css` extension. This is the most common and useful method of bringing CSS to a document. You can link a single CSS file to multiple web pages, styling all of them with the same CSS stylesheet. In the Getting started with CSS, we linked an external stylesheet to our web page.

index.html

```
<!DOCTYPE html>
<html lang="en-GB">
  <head>
    <meta charset="utf-8" />
    <title>My CSS experiment</title>
    <link rel="stylesheet" href="styles.css" />
  </head>
  <body>
    <h1>Hello World!</h1>
    <p>This is my first CSS example</p>
  </body>
</html>
```

styles.css

```
h1 {
  color: blue;
  background-color: yellow;
  border: 1px solid black;
}

p {
  color: red;
}
```

# Applying CSS to HTML – Internal stylesheets

An internal stylesheet resides within an HTML document. To create an internal stylesheet, you place CSS inside a `<style>` element contained inside the HTML `<head>`.

For sites with more than one page, an internal stylesheet becomes a less efficient way of working. To apply uniform CSS styling to multiple pages using internal stylesheets, you must have an internal stylesheet in every web page that will use the styling. The efficiency penalty carries over to site maintenance too.

index.html

```
<!DOCTYPE html>
<html lang="en-GB">
  <head>
    <meta charset="utf-8" />
    <title>My CSS experiment</title>
    <style>
      h1 {
        color: blue;
        background-color: yellow;
        border: 1px solid black;
      }

      p {
        color: red;
      }
    </style>
  </head>
  <body>
    <h1>Hello World!</h1>
    <p>This is my first CSS example</p>
  </body>
</html>
```

# Applying CSS to HTML – Inline styles

Inline styles are CSS declarations that affect a single HTML element, contained within a style attribute. The implementation of an inline style in an HTML document might look like the one on the right.

Avoid using CSS in this way, when possible, because:

- It is the least efficient implementation of CSS for maintenance;
- Inline CSS mixes CSS code with HTML and content, making everything more difficult to read and understand.

index.html

```
<!DOCTYPE html>
<html lang="en-GB">
  <head>
    <meta charset="utf-8" />
    <title>My CSS experiment</title>
  </head>
  <body>
    <h1 style="color: blue;background-color: yellow;border:
1px solid black;">
      Hello World!
    </h1>
    <p style="color:red;">This is my first CSS example</p>
  </body>
</html>
```

# Selectors

Each CSS rule starts with selectors which tell the browser which elements the rules should apply to. You can see them reported on the right box.

There may be two selectors selecting the same HTML element, like in the example here exposed. The class prevails on the element, because it is more specific, so it cancels the other conflicting declaration (specificity rule).

However, applying the stylesheet on the right, the <p> text will be blue, because the declaration that sets the paragraph text to blue appears later in the stylesheet (cascade rule).

CSS selectors

```
h1
a:link
.manythings
#onething
*
.box p
.box p:first-child
h1, h2, .intro
```

CSS (paragraph is red)

```
.special {
  color: red;
}

p {
  color: blue;
}
```

CSS (paragraph is blue)

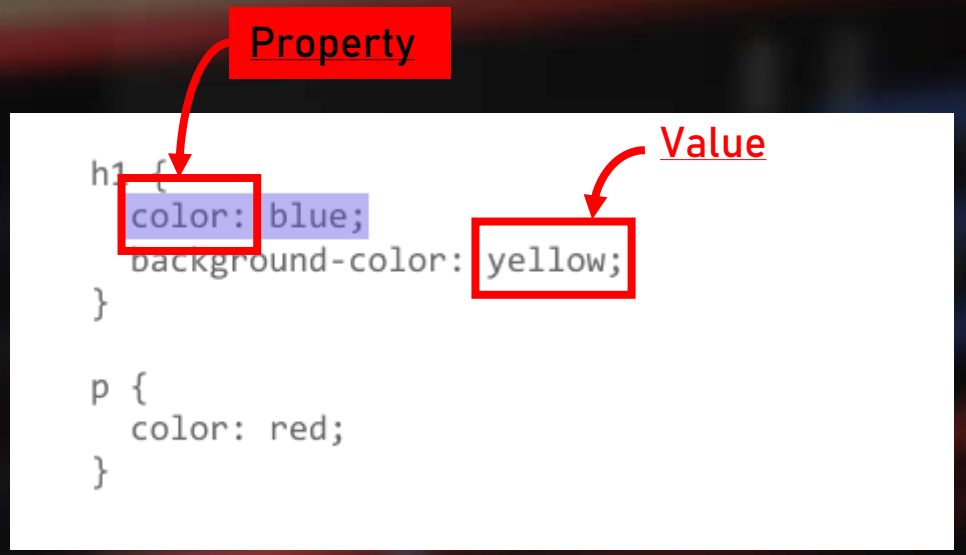
```
p {
  color: red;
}

p {
  color: blue;
}
```

# Properties and values

Most basically, CSS consists of two components:

- Properties: human-readable identifiers that indicate the stylistic features to modify;
- Values: indicate how to style the property.





# Functions

While most values are relatively simple keywords or numeric values, there are some values that take the form of a function.

An example would be the `calc()` function, which can do simple math within CSS. The values define the width of this box to be 90% of the containing block width, minus 30 pixels. The result of the calculation isn't something that can be computed in advance and entered as a static value.

A function consists of the function name, and parentheses to enclose the values for the function.

HTML

```
<div class="outer"><div class="box">The inner box is 90% - 30px.</div></div>
```

CSS

```
.outer {
  border: 5px solid black;
}

.box {
  padding: 10px;
  width: calc(90% - 30px);
  background-color: rebeccapurple;
  color: white;
}
```

# Functions

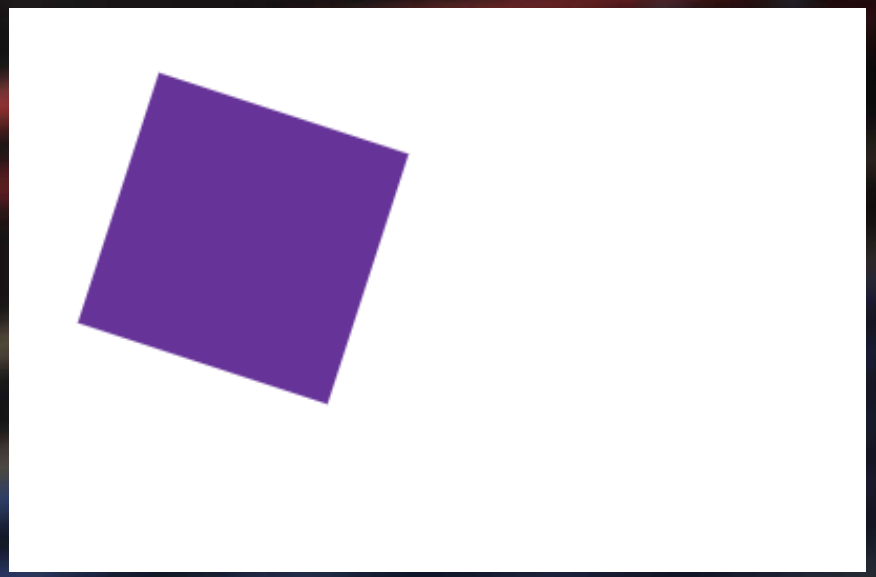
Another example would be the various values for transform, such as `rotate()`.

HTML

```
<div class="box"></div>
```

CSS

```
.box {
  margin: 30px;
  width: 100px;
  height: 100px;
  background-color: rebeccapurple;
  transform: rotate(0.8turn);
}
```



# @rules

CSS @rules (pronounced "at-rules") provide instruction for what CSS should perform or how it should behave. Some @rules are simple with just a keyword and a value. For example, @import imports a stylesheet into another CSS stylesheet.

One common @rule that you may encounter is @media, which is used to create media queries. Media queries use conditional logic for applying CSS styling.

In the second example, the stylesheet defines a default pink background for the <body> element. However, a media query follows that defines a blue background if the browser viewport is wider than 30em.

CSS

```
@import "styles2.css";
```

CSS

```
body {
  background-color: pink;
}

@media (min-width: 30em) {
  body {
    background-color: blue;
  }
}
```

# Shorthands

Some properties like font, background, padding, border, and margin are called shorthand properties. This is because shorthand properties set several values in a single line.

CSS

```
background: red url(bg-graphic.png) 10px 10px
repeat-x fixed;
```

CSS

```
background-color: red;
background-image: url(bg-graphic.png);
background-position: 10px 10px;
background-repeat: repeat-x;
background-attachment: fixed;
```

CSS

```
/* In 4-value shorthands like padding and margin, the
values are applied
   in the order top, right, bottom, left (clockwise from
the top). There are also other
   shorthand types, for example 2-value shorthands, which
set padding/margin
   for top/bottom, then left/right */

padding: 10px 15px 15px 5px;
```

CSS

```
padding-top: 10px;
padding-right: 15px;
padding-bottom: 15px;
padding-left: 5px;
```

# 

The  HTML element is the generic container for flow content. It has no effect on the content or layout until styled in some way using CSS. As a "pure" container, the  element does not inherently represent anything.

Instead, it's used to group content so it can be easily styled using the class or id attributes, marking a section of a document as being written in a different language (using the lang attribute), and so on.

Note: The align attribute is obsolete; do not use it anymore. Now we use Grid or Flexbox to align and position  elements on the page.

HTML

```
<div class="shadowbox">
  <p>Here's a very interesting note displayed in a lovely
shadowed box.</p>
</div>
```

CSS

```
.shadowbox {
  width: 15em;
  border: 1px solid #333;
  box-shadow: 8px 8px 5px #444;
  padding: 8px 12px;
  background-image: linear-gradient(180deg, #fff, #ddd 40%,
#ccc);
}
```

# 

The  HTML element is a generic inline container for phrasing content, which does not inherently represent anything.

It can be used to group elements for styling purposes (using the class or id attributes), or because they share attribute values, such as lang. It should be used only when no other semantic element is appropriate.

 is very much like a  element, but  is a block-level element whereas a  is an inline element.

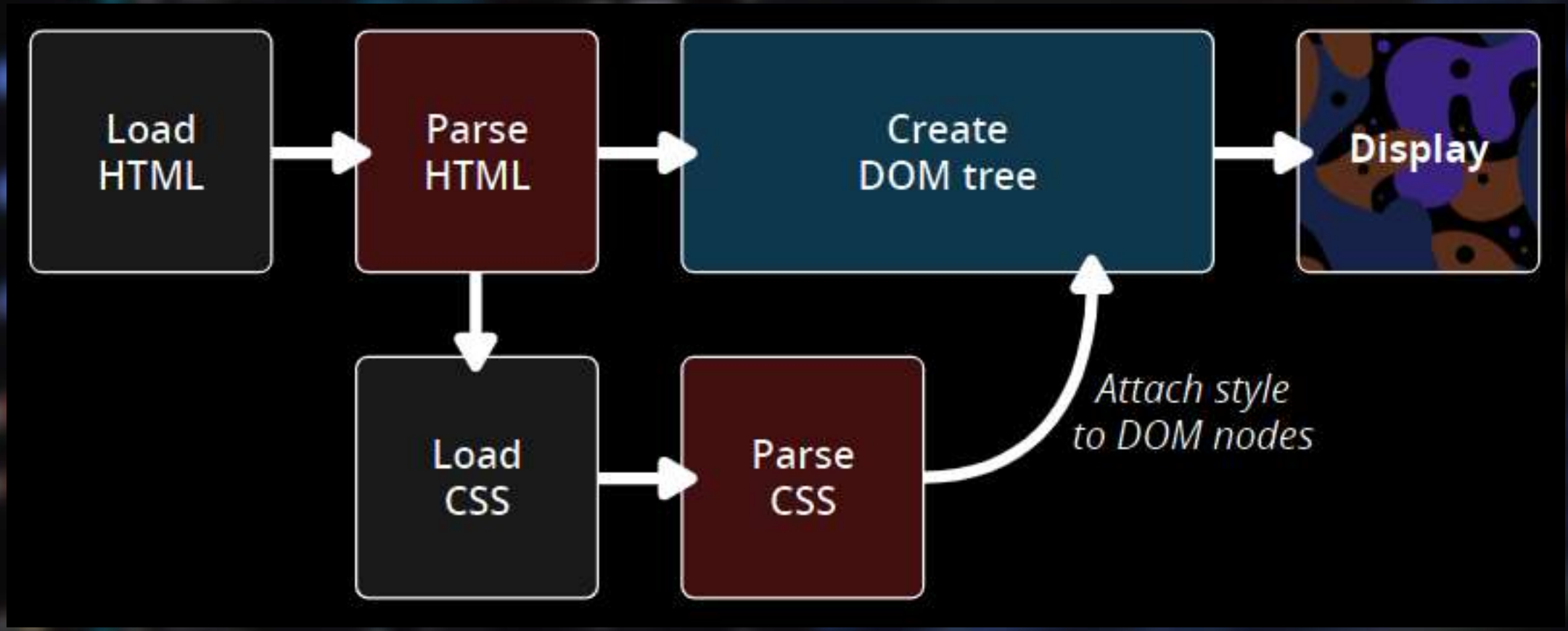
HTML

```
<li>
  <span>
    <a href="portfolio.html" target="_blank">See my
portfolio</a>
  </span>
</li>
```

CSS

```
li span {
  background: gold;
}
```

# How CSS works



# How are elements laid out by default?

## HTML

```
<h1>Basic document flow</h1>

<p>
  I am a basic block level element. My adjacent block level elements sit on new
  lines below me.
</p>

<p>
  By default we span 100% of the width of our parent element, and we are as tall
  as our child content. Our total width and height is our content + padding +
  border width/height.
</p>

<p>
  We are separated by our margins. Because of margin collapsing, we are
  separated by the width of one of our margins, not both.
</p>
<p>
  Inline elements <span>like this one</span> and <span>this one</span> sit on
  the same line along with adjacent text nodes, if there is space on the same
  line.
Overflowing inline elements will
<span>wrap onto a new line if possible (like this one containing text)</span>,
  or just go on to a new line if not, much like this image will do:
  <img src="long.jpg" alt="snippet of cloth" />
</p>
```

## CSS

```
body {
  width: 500px;
  margin: 0 auto;
}

p {
  background: rgba(255, 84, 104, 0.3);
  border: 2px solid rgb(255, 84, 104);
  padding: 10px;
  margin: 10px;
}

span {
  background: white;
  border: 1px solid black;
}
```

# How are elements laid out by default?

## Basic document flow

I am a basic block level element. My adjacent block level elements sit on new lines below me.

By default we span 100% of the width of our parent element, and we are as tall as our child content. Our total width and height is our content + padding + border width/height.

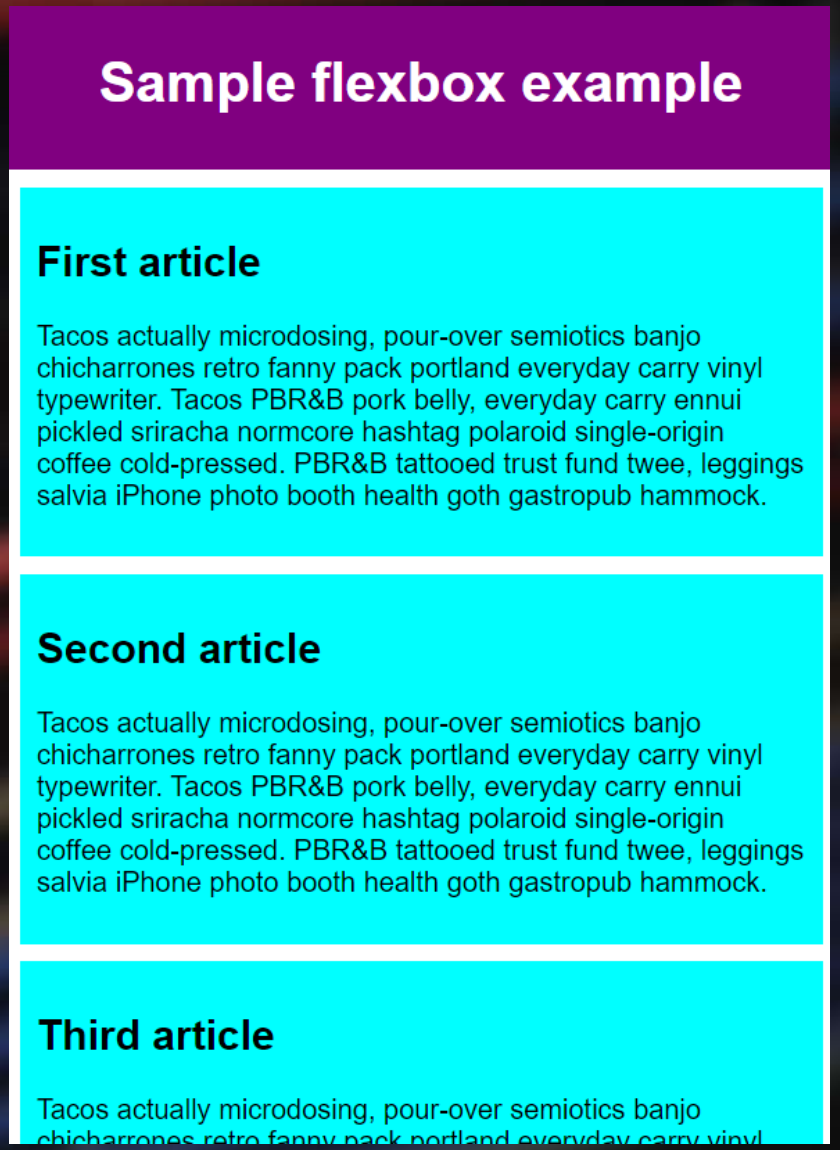
We are separated by our margins. Because of margin collapsing, we are separated by the width of one of our margins, not both.

Inline elements like this one and this one sit on the same line along with adjacent text nodes, if there is space on the same line. Overflowing inline elements will wrap onto a new line if possible (like this one containing text), or just go on to a new line if not, much like this image will do:

# Flexbox

For a long time, the only reliable cross-browser compatible tools available for creating CSS layouts were features like floats and positioning. These work, but in many times they're frustrating.

Vertically centering a block of content inside its parent, or making all the children of a container take up an equal amount of the available width/height, or regardless of how much width/height is available are either difficult or impossible to achieve with such tools in any kind of convenient, flexible way.



# Specifying what to lay out as flexible

Look at this file.

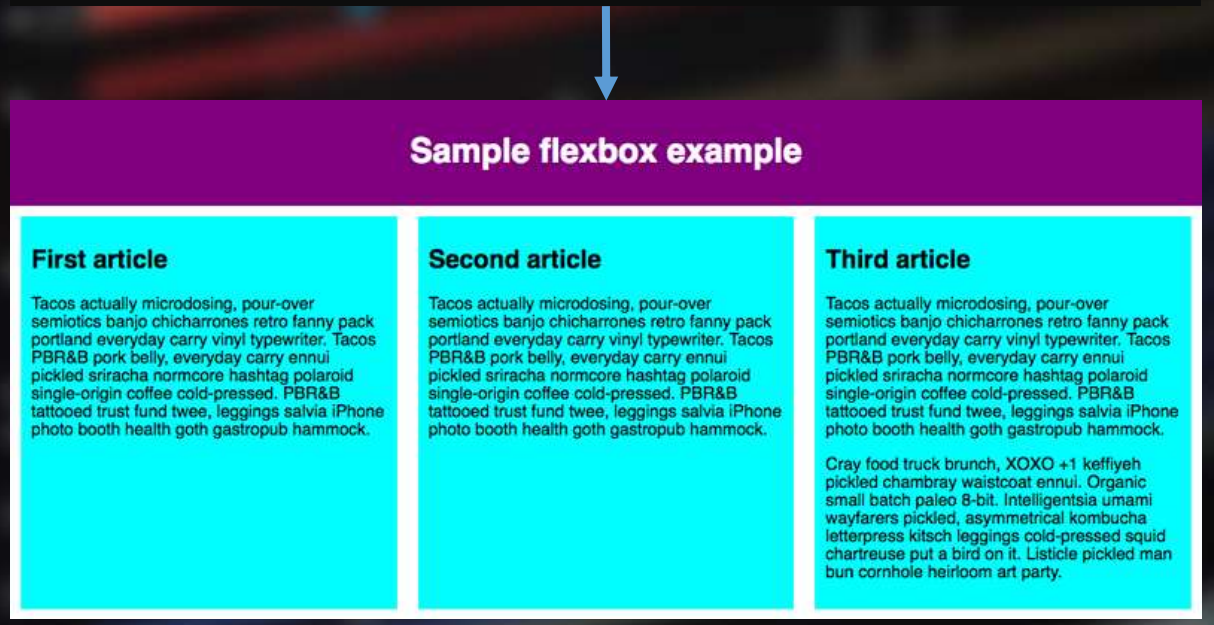
To start with, we need to select which elements are to be laid out as flexible boxes. To do this, we set a special value of display on the parent element of the elements you want to affect.

This causes the <section> element to become a flex container and its children to become flex items.

Using flex-direction property you can also specify which direction the main axis runs. By default this is set to row.

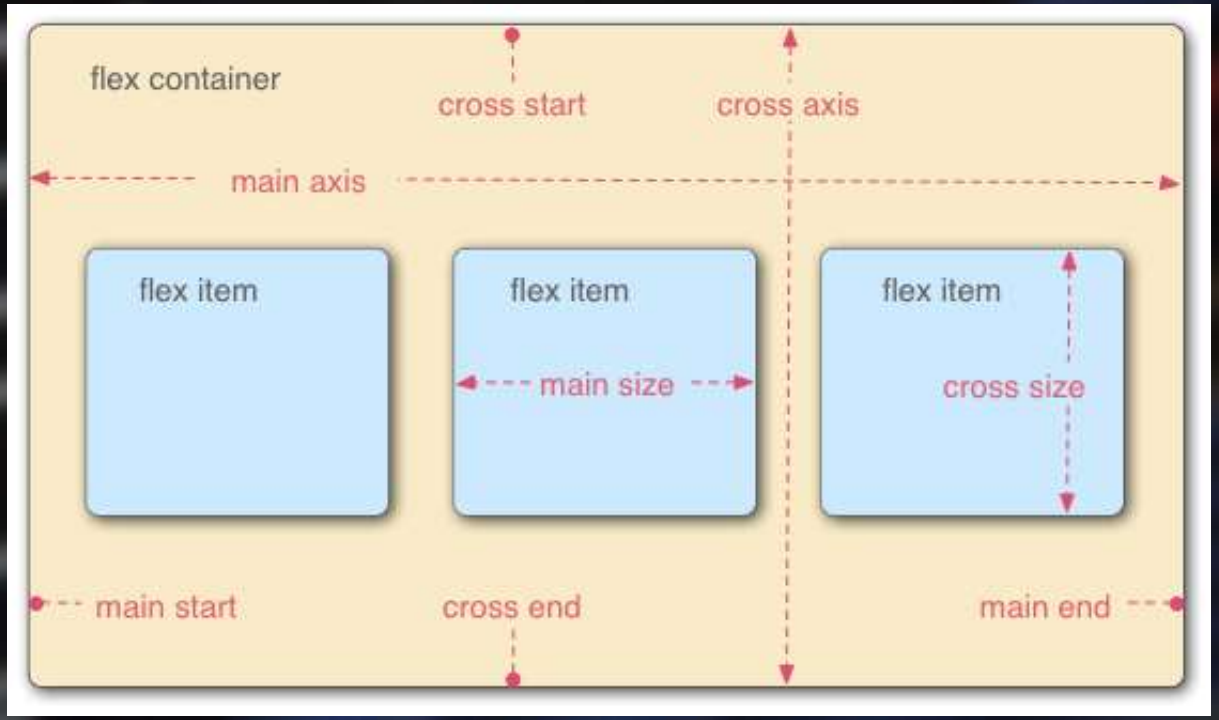
CSS

```
section {
  display: flex;
  flex-direction: row; \* Default – Try using column
instead ☺ *\
}
```



# The flex model

- Main axis: it runs in the direction the flex items are laid out in (for example, as rows across the page, or columns down the page.)
- Cross axis: it runs perpendicular to the direction the flex items are laid out in.
- Flex container: it has `display: flex` set on it (the `<section>` in our example).
- Flex items: the items laid out as flexible boxes inside the flex container.



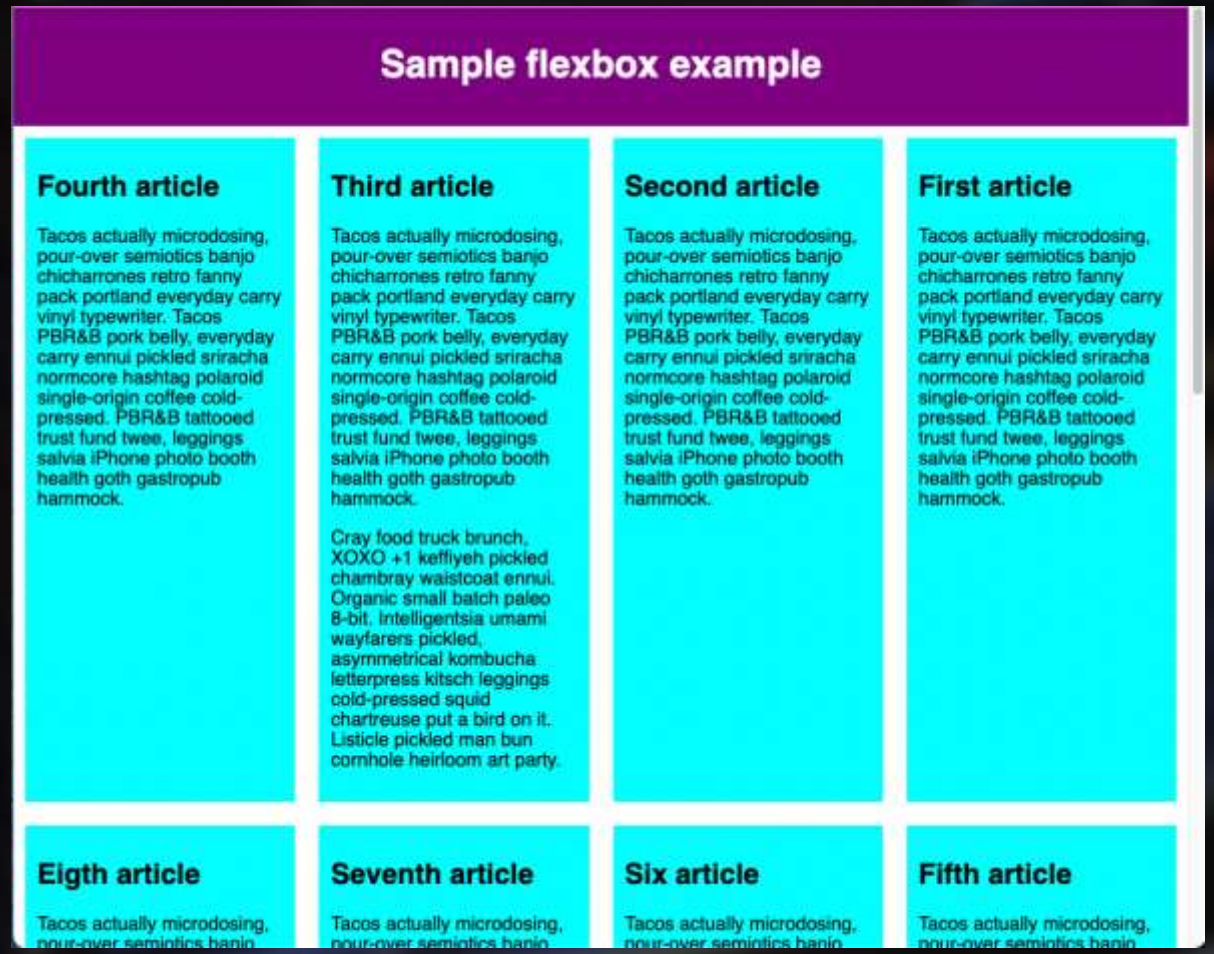
# Wrapping

Look at this file.

Try to apply the code below. Now you have multiple rows, and each one has as many flexbox children fitted into it as is sensible. Any overflow is moved down to the next line. The `flex: 200px` declaration sets that each article will be at least 200px wide. Also notice that the last few children on the last row are each made wider so that the entire row is still filled.

CSS

```
section {
  display: flex;
  flex-wrap: wrap;
  flex: 200px;
}
```



# Multiple-column layout

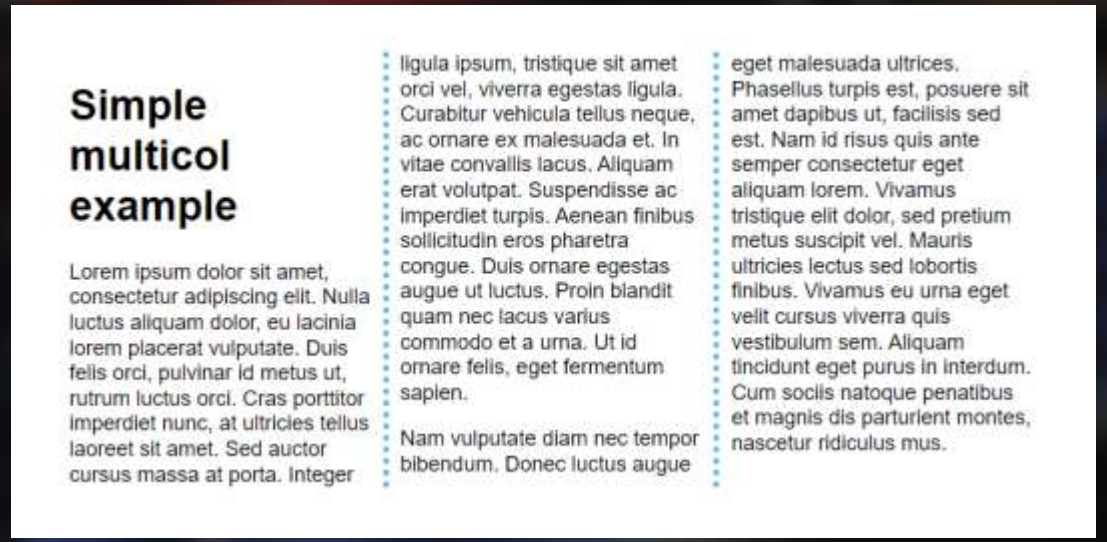
Look at this file.

This simple HTML contains a wrapper with a class of container, inside of which you have a heading and paragraphs.

Adding the CSS on the right, the  with a class of container will become a multicolumn container. With the column-width, gap and rule properties the browser will now give you as many columns as it can of the size that you specify, with the size of the gap between columns that you want, and a special kind of rule between the columns (in this case, dotted rules).

CSS

```
.container {
  column-count: 3;
  column-width: 200px;
  column-gap: 20px;
  column-rule: 4px dotted rgb(79, 185, 227);
}
```

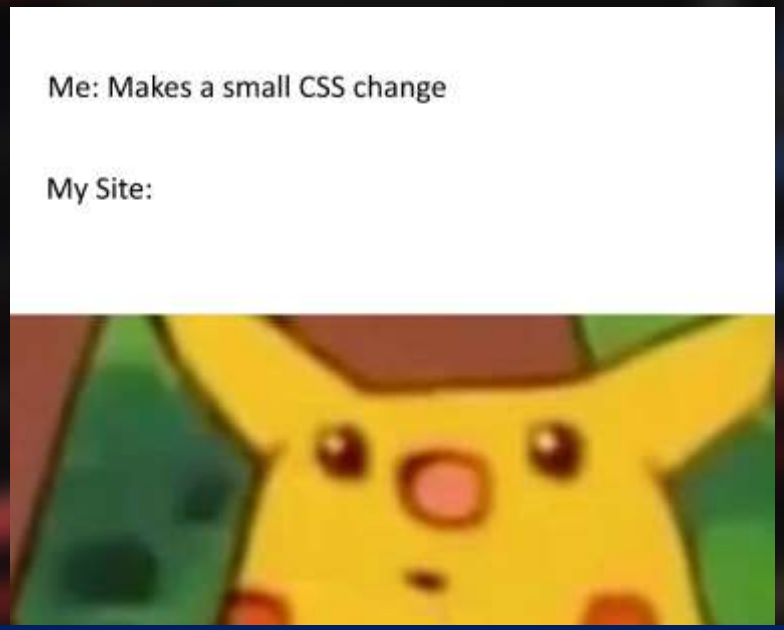


# Positioning

Positioning allows you to produce interesting results by overriding normal document flow.

There are a number of different types of positioning that you can put into effect on HTML elements. To make a specific type of positioning active on an element, you use the `position` property.

Static positioning is the default that every element gets. It just means "put the element into its normal position in the document flow — nothing special to see".




HTML

```
<p class="positioned">…</p>
```

CSS

```
.positioned {
  position: static;
  background: yellow;
}
```

# Relative positioning

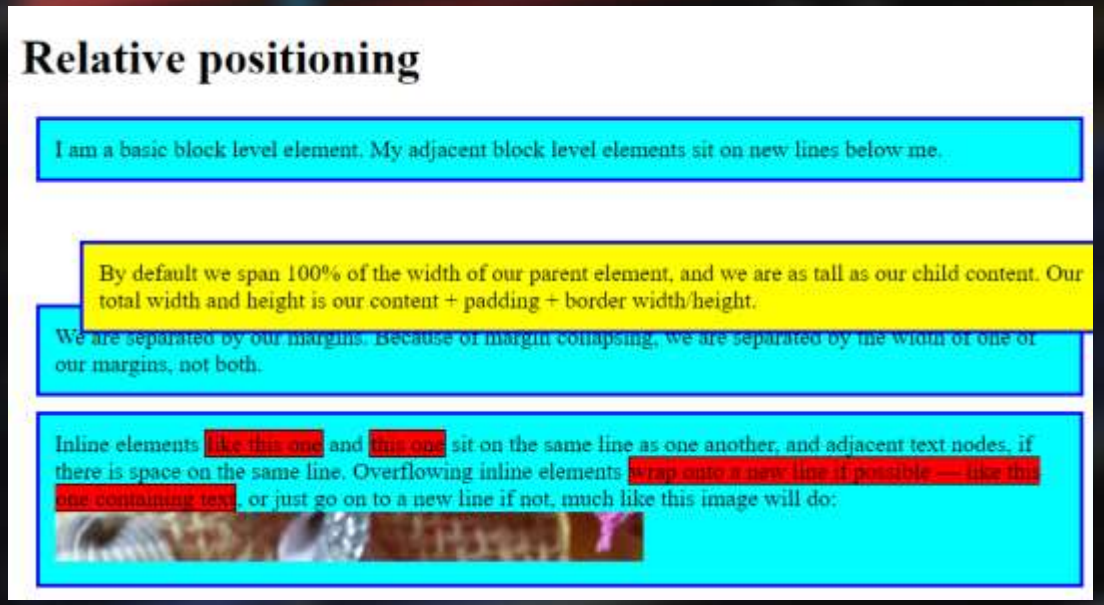
This is very similar to static positioning, except that once the positioned element has taken its place in the normal flow, you can then modify its final position, including making it overlap other elements on the page.

`top`, `bottom`, `left`, and `right` are used alongside position to specify exactly where to move the positioned element to.

You need to think of it as if there's an invisible force that pushes the specified side of the positioned box, moving it in the opposite direction. So, for example, if you specify `top: 30px;`, it's as if a force will push the top of the box, causing it to move downwards by 30px.

CSS

```
.positioned {
  position: relative;
  background: yellow;
  top: 30px;
  left: 30px;
}
```



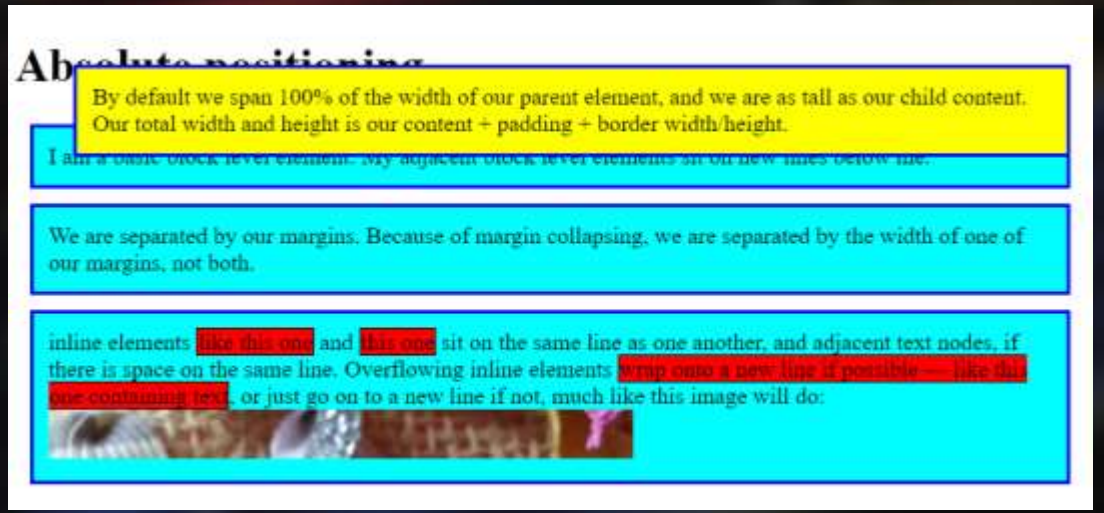
# Absolute positioning

An absolutely positioned element no longer exists in the normal document flow. Instead, it sits on its own layer separate from everything else. So we can create isolated UI features that don't interfere with the layout of other elements on the page. For example, popup information boxes, control menus, rollover panels, UI features that can be dragged and dropped anywhere on the page, and so on.

The position of the element has changed. This is because top, bottom, left, and right specify the distance the element should be from each of the containing element's sides.

CSS

```
.positioned {
  position: absolute;
  background: yellow;
  top: 30px;
  left: 30px;
}
```



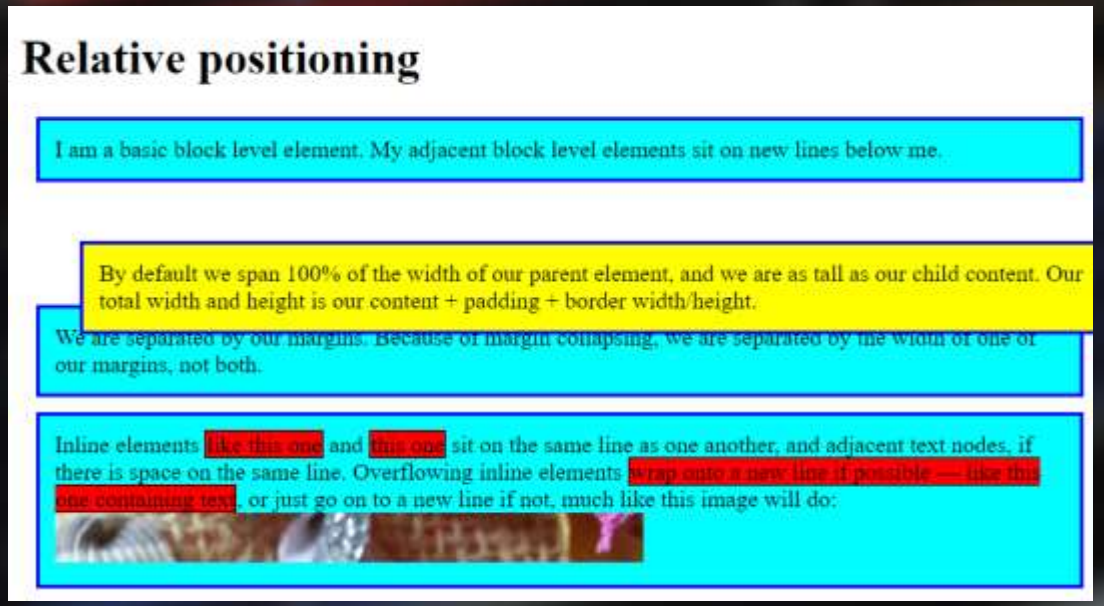
# Introducing z-index

For changing the stacking order you may use the z-index property. "z-index" is a reference to the z-axis. You may recall from previous points in the course where we discussed web pages using horizontal (x-axis) and vertical (y-axis) coordinates to work out positioning for things like background images and drop shadow offsets.

The z-axis is an imaginary line that runs from the surface of your screen towards your face. z-index values affect where positioned elements sit on that axis; positive values move them higher up the stack, negative values move them lower down the stack. By default, positioned elements all have a z-index of auto, which is effectively 0.

CSS

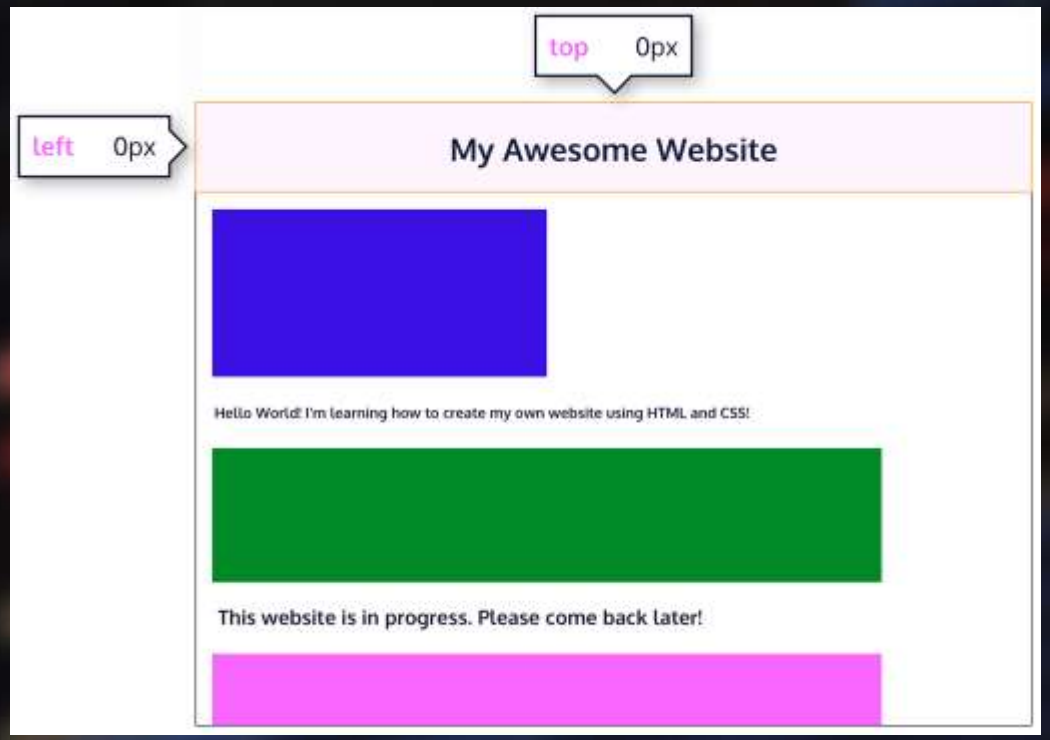
```
.positioned {
  position: relative;
  background: yellow;
  z-index: 1;
  top: 30px;
  left: 30px;
}
```



# Fixed positioning

This works in exactly the same way as absolute positioning, but it fixes an element in place relative to the visible portion of the viewport, except the case in which one of the element's ancestors is a fixed containing block, because its transform property has a value other than none.

This means that you can create useful UI items that are fixed in place, like persistent navigation menus that are always visible no matter how much the page scrolls.

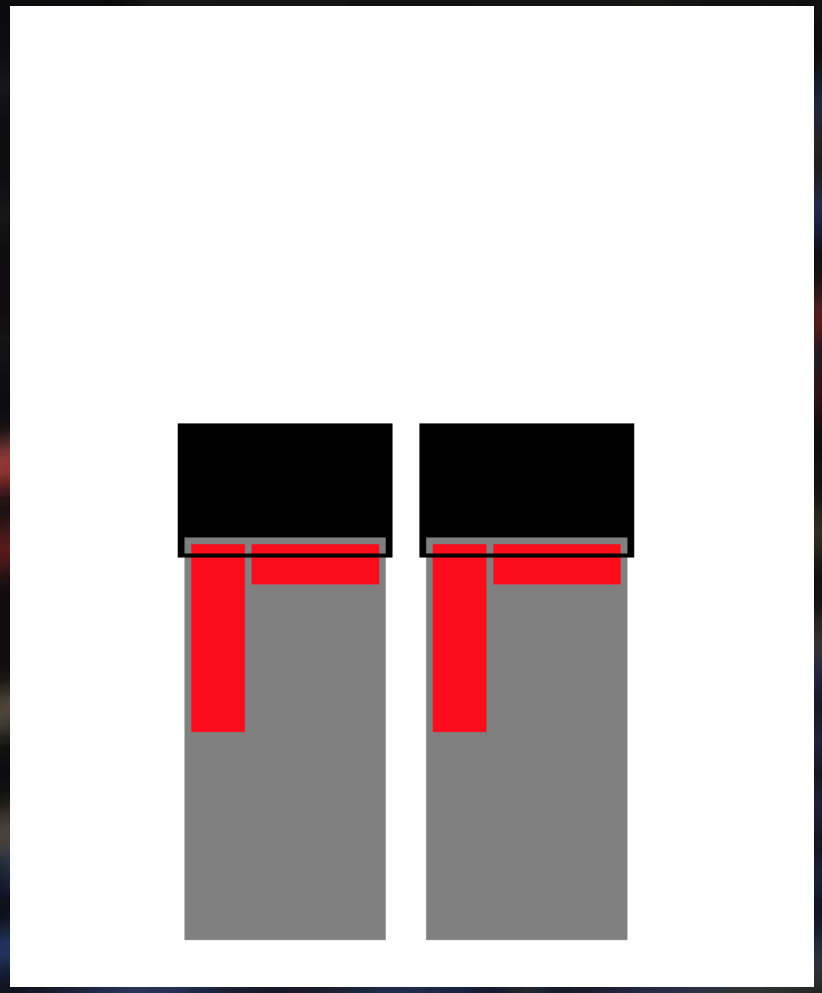


# Sticky positioning

There is another position value available called `position: sticky`, which is somewhat newer than the others.

This is basically a hybrid between relative and fixed position. It allows a positioned element to act like it's relatively positioned until it's scrolled to a certain threshold, after which it becomes fixed.

Sticky positioning can be used, for example, to cause a navigation bar to scroll with the page until a certain point and then stick to the top of the page.



# More than standard CSS