Front-End 18.0

Back to 'Day 3 | Pre-work'

CSS3 | Day 3 | Pre-work

CSS
Day 3

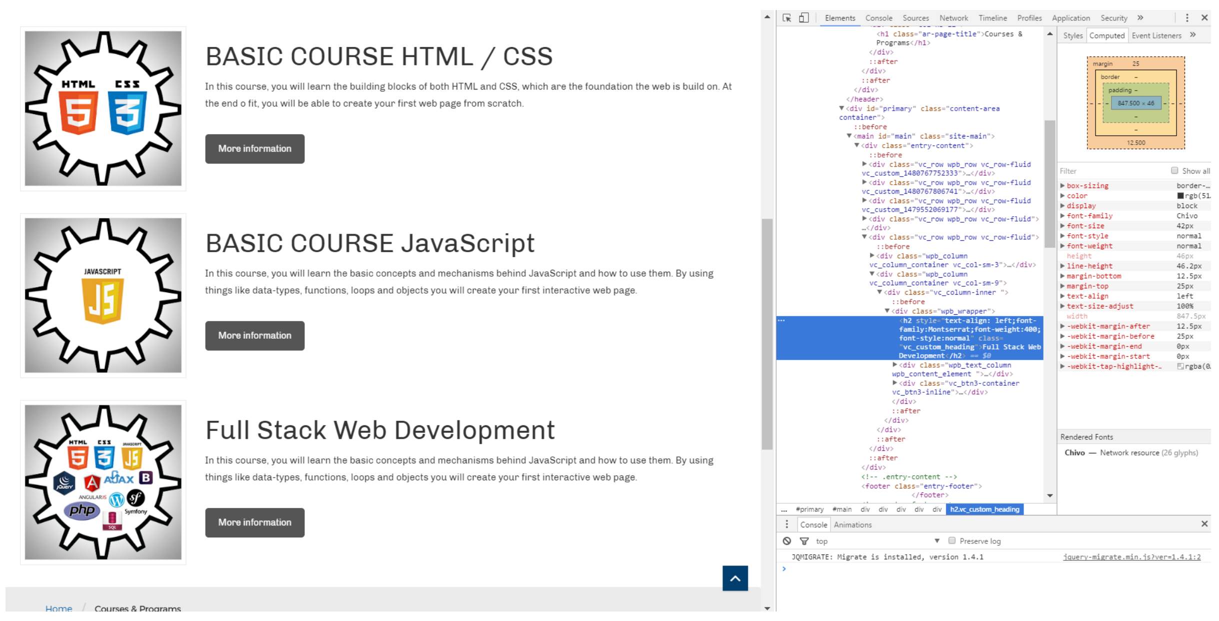
Table of contents:

- Box Model
  - Dimensions
  - Margin
  - Padding
  - Border
  - Object Content
- Creating Fluid Layouts & Images
  - Fixed vs Fluid layouts
    - Fixed layout
    - Fluid
    - Max-width
    - Fluid images
- Flexbox
  - The Flex Container: main & cross axis
  - Align flex box items with justify-content & align-items
  - Flex-container: axis direction & wrapping items
  - The Flex Items
- Variables - custom properties

Box Model

Even if it does not look like it, **every element in web design is a rectangular box**. The CSS properties affecting the layout of a page are based around the box model. Virtually all elements have (or can have) these properties, including the document body.

To understand visually how each element within the web page is wrapped inside a rectangular box, open your Google Chrome browser, use the shortcut Ctrl + Shift + C and click on any element on any web page, you should get something like this:



BASIC COURSE HTML / CSS

In this course, you will learn the building blocks of both HTML and CSS, which are the foundation the web is build on. At the end o fit, you will be able to create your first web page from scratch.

More information

BASIC COURSE JavaScript

In this course, you will learn the basic concepts and mechanisms behind JavaScript and how to use them. By using things like data-types, functions, loops and objects you will create your first interactive web page.

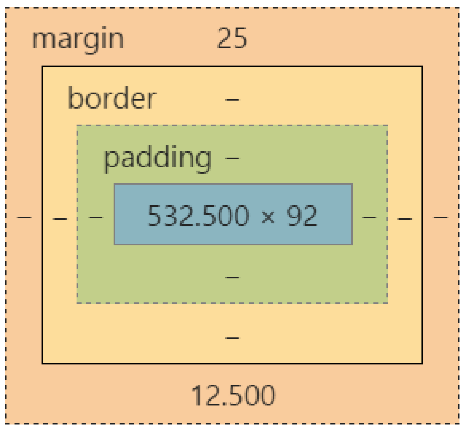
More information

Full Stack Web Development

In this course, you will learn the basic concepts and mechanisms behind JavaScript and how to use them. By using things like data-types, functions, loops and objects you will create your first interactive web page.

More information

Home / Courses & Programs



In the right-top corner you can see the box model (the orange-green-blue box) which starts on the outside with the object's margin. Inside this is the border, then there is padding between the border and the inner contents, and finally there's the object's contents.

These are the properties that affect the box model:

1. **Dimensions**
2. **Margin**
3. **Border**
4. **Padding**
5. **Content**

Once you have the hang of the box model, you will be well on your way to creating professionally laid-out pages, since these properties alone will make up much of your page styling.
It is especially important to control the appearance of each box on the web page once you start creating the page layout.

Dimensions

A box is by default designed to just hold its contents. To set your own dimensions for a box/element you can use the **height** and **width** properties.

```
width: 500px;
height: 600px;
```

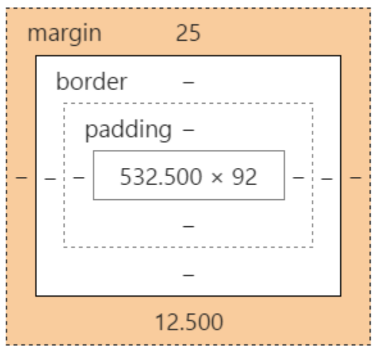
These are absolute values, and the element will always have the given size, no matter the size of its content. If the content is smaller it doesn't affect the container but if it is bigger, it will overflow. Though a limit could be given either for maximum or minimum size, just by adding the words **-min** or **-max** to the attributes width or height. When setting a -min value, the element will respect that size if their content is smaller otherwise, it will expand to accommodate the content. The -max attribute is the opposite: it will overflow immediately if the content is bigger than the container. Setting a min-height is recommendable for dynamic content.

```
max-width: 500px;
min-height: 600px;
```

The most popular ways to specify the size of a box are to use pixels, percentages, or ems. Traditionally, pixels have been the most popular method because they allow designers to accurately control their size.
When you use percentages, the size of the box is relative to the size of the browser window or, if the box is encased within another box, it is a percentage of the size of the containing box.
When you use ems, the size of the box is based on the size of text within it.

Margin

The margin property controls the gap between boxes. Its value is commonly given in pixels, although you may also use percentages or ems.

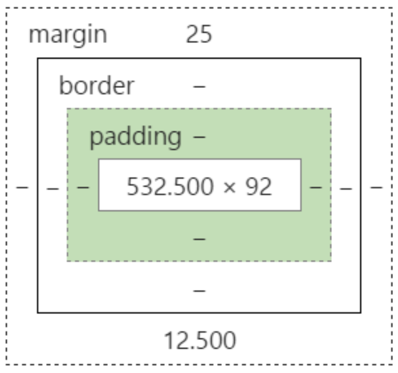


The margins of an element can be changed en masse with the margin property, or individually with margin-left, margin-top, margin-right, and margin-bottom. When setting the margin property, you can supply one, two, three, or four arguments, which have the effects commented in the following rules:

```
/* Set all margins to 1 pixel */
margin:1px;
/* Set top and bottom to 1 pixel, and left and right to 2 */
margin:1px 2px;
/* Set top to 1 pixel, left and right to 2, and bottom to 3 */
margin:1px 2px 3px;
/* Set top to 1 pixel, right to 2, bottom to 3, and left to 4 */
margin:1px 2px 3px 4px;
```

Padding

The **padding** property allows you to specify how much space should appear **between the content of an element and its border**.



The deepest of the box model levels (other than the contents of an element) is the padding, which is applied inside any borders and/or margins. The main properties used to modify padding are **padding, padding-left, padding-top, padding-right, and padding-bottom**.
The four ways of accessing individual property settings used for the margin and the border properties also apply with the padding property, so all the following are valid rules:

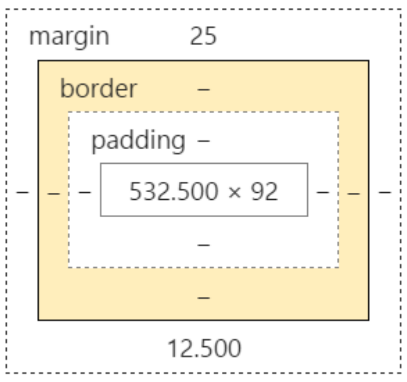
```
/* All padding */
padding: 1px;
/* Top/bottom and left/right */
padding: 1px 2px;
/* Top, left/right and bottom */
padding: 1px 2px 3px;
/* Top, right, bottom and left */
padding: 1px 2px 3px 4px;
```

Or you can use a shorthand (where the values are in **clockwise order: top, right, bottom, left**):

```
padding: 10px 5px 3px 1px;
```

Border

Every box has borders (even if it is not visible or is specified to be 0 pixels wide). The borders separate the edge of one box from another. Every element has 4 borders: **top, right, bottom** and **left.**



Each of those directions has the main properties: **width, style and color.**
The shorthand border property that includes the most popular attributes in only one line of css:

```
border: 1px dashed green;
```

You can also control each border of an element separately, targeting them by specifying the directions:

```
border-bottom: 3px dashed green;
```

```
border-top: 5px solid blue;
```

You can control the style of a border using the border-style property. This property can take the following values:

- **solid** a single **solid line**
- **dashed** a series of **short lines**
- **dotted** a series of **square dots** (if the border is 4px wide, then the dots are 4px squared with a 4px gap between them)
- **double** two **solid lines** (the value of the border-width property creates the total width of the two lines)
- **groove** looks to be **carved** into the page
- **hidden** means **no border** is displayed
- **inset** looks **embedded** into the page
- **outset** looks like it is **coming out** of the page
- **ridge** looks to **stick out** from the page

The three attributes **Margin**, **Padding** or **Border-width** can be specified giving 4 different values at once:

```
margin: 10px 5px 6px 2px;
```

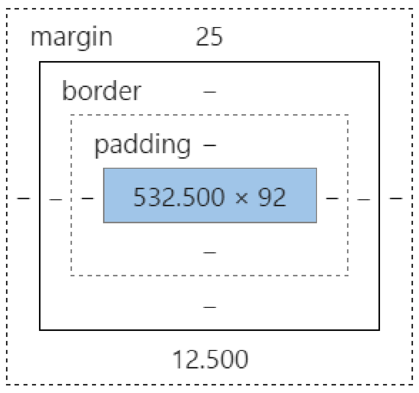
```
padding: 8px 10px 5px 3px;
```

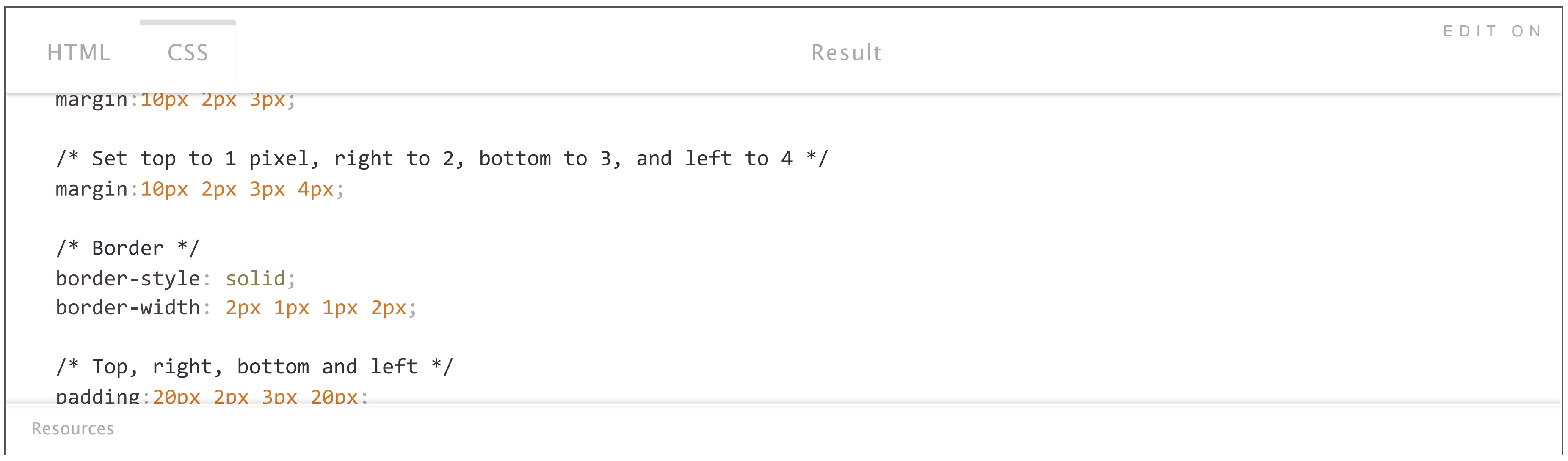
```
border-width: 2px 1px 4px 2px;
```

The sequence that these values are given has its importance, they correspond to a clockwise order: **top, right, bottom, left.**

Object Content

Deep within the box model levels, at its center, lies an element that can be styled in all the ways discussed in this chapter, and which can (and usually will) contain further subelements, which in turn may contain their own subelements, and so on, each with its own styling and box model settings.





```
margin:10px 2px 3px;

/* Set top to 1 pixel, right to 2, bottom to 3, and left to 4 */
margin:10px 2px 3px 4px;

/* Border */
border-style: solid;
border-width: 2px 1px 1px 2px;

/* Top, right, bottom and left */
padding:20px 2px 3px 20px;
```

Creating Fluid Layouts & Images

Before adding responsive techniques, it's important to create a **fluid layout** to make the transitions between different screen sizes easier to manage and require fewer breakpoints for making changes.

Fixed vs Fluid layouts

Fixed layout

Fixed layouts use exact pixel widths which means that the size of the page components will be the same for all resolutions.

HTML

```
<section class="wrapper">
  content goes here
</section>
```

CSS

```
.wrapper {
  width: 800px;
}
```

Optionally, a margin: 0 auto; is applied to automatically **center align** the block of content.

```
.wrapper {
  width: 800px;
  margin: 0 auto;
}
```

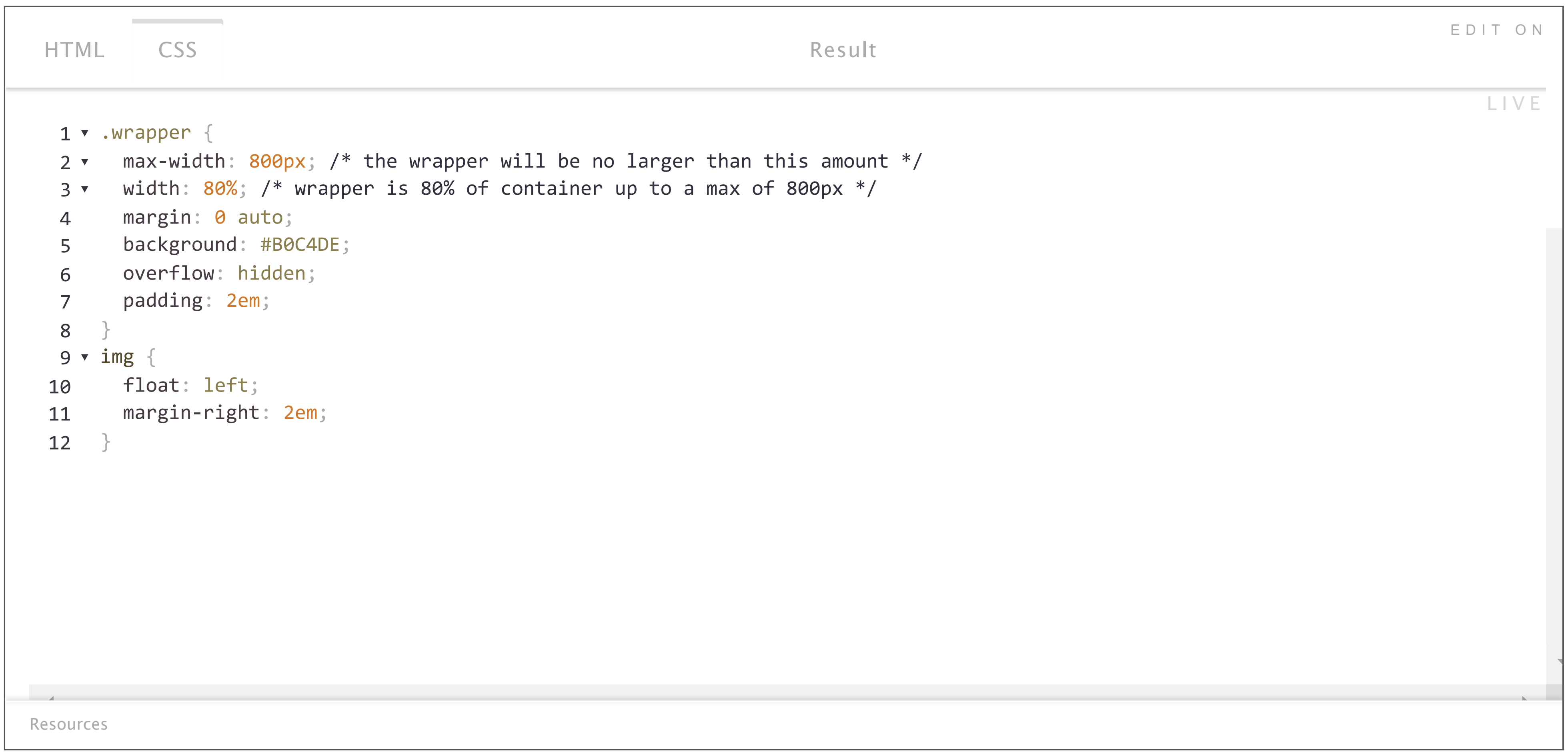
But, if the browser window is smaller than the width of the layout, a horizontal scrollbar will show and the elements itself won’t change in width.

Fluid

Most of the page components in a fluid page layout **adjust to the user’s screen size by using percentage** widths rather than fixed pixel widths. The previous example could be revised to use a percentage width instead.

```
.wrapper {
  width: 80%;
  margin: 0 auto;
}
```

No matter the size of the browser window, the wrapper is now 80% of its container, the body element/browser viewport in this example.
Here’s a comparison of a fixed width vs percentage width content wrapper. Notice that when using a percentage for the width, the wrapper element remains at an 80% width of its current container size and the content also stays within wrapper:



```css
.wrapper {
  max-width: 800px; /* the wrapper will be no larger than this amount */
  width: 80%; /* wrapper is 80% of container up to a max of 800px */
  margin: 0 auto;
  background: #B0C4DE;
  overflow: hidden;
  padding: 2em;
}
img {
  float: left;
  margin-right: 2em;
}
```

This technique is great for small resolutions, but what happens when the resolution of the screen goes wider than your desired width?
Remember, the percentage width is relative to the size of its container so when the resolution is bigger, the desired wrapper width is now too wide!

Max-width

Use the CSS property **max-width** to create boundaries for a fluid page.

```css
.wrapper {
  max-width: 800px;
  width: 80%;
  margin: 0 auto;
}
```

By setting a percentage width and a fixed pixel max-width, the content wrapper will always be relative to the screen size, **until it reaches the maximum** size, 800px in this example, to keep the content wrapper from extending beyond that.

Fluid images

You can also use percentages to create flexible images. Width and max-width can both be used, just like in the previous example.

```css
img {
  /* image stretches to 100% of its container */
  width: 100%;
  /* image will stretch 100% of its container until it
  reaches 100% of the width of the image file itself */
  max-width: 100%;
}
```

Let's say you have three columns of content, containing an image that is bigger than the width of the columns.

HTML:

```html
<div class="column"><img src="dog.jpg" alt="dog"></div>
<div class="column"><img src="dog.jpg" alt="dog"></div>
<div class="column"><img src="dog.jpg" alt="dog"></div>
```

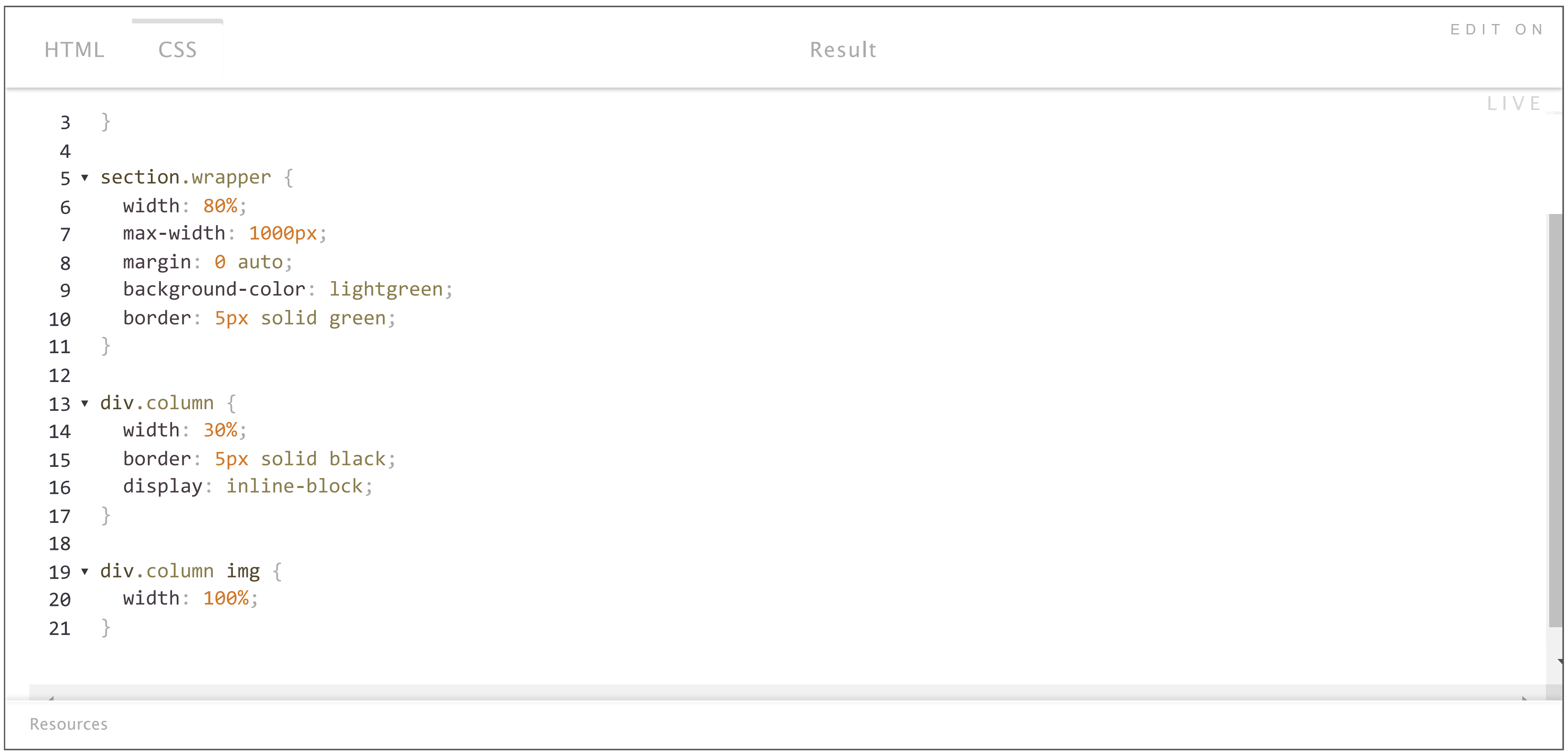
CSS:

```css
.column {
   width: 250px;
   border: 5px solid black;
   display: inline-block;
}
```

If the image is bigger than its container, it will “spill out” of its container.
Setting the image width to 100% will change that! This will make the images 100% of their container.

```css
.column img {
   width: 100%
}
```

Better yet, make the column width a percentage as well and both the columns and images will now be fluid. Refer to the CodePen example below to look at the final CSS and try changing the browser window size to see how the columns and images scale together.



```css
}

section.wrapper {
  width: 80%;
  max-width: 1000px;
  margin: 0 auto;
  background-color: lightgreen;
  border: 5px solid green;
}

div.column {
  width: 30%;
  border: 5px solid black;
  display: inline-block;
}

div.column img {
  width: 100%;
}
```

Flexbox

The main idea behind the flex layout is to give the container the ability to alter its items' width/height (and order) to best fill the available space, a very practical ability when it comes to displaying on different screen sizes e.g. in responsive design.
A flex container expands items to fill available free space or shrinks them to prevent overflow.
-Chris Coyier, CSS-Tricks

Most importantly, the flexbox layout is direction-focused as opposed to the regular layouts of block elements (vertically oriented) or inline elements (horizontally oriented)

Basically, we are speaking about two different types of items in flexbox:

- The **flex container** (parent)
- The **flex items** (children)

The Flex Container: main & cross axis

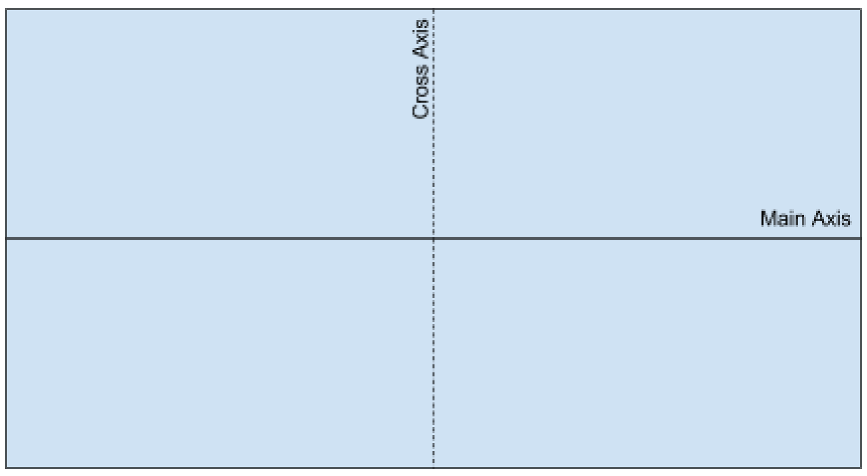
Flexbox consists of *flex containers* and *flex items* and defines how flex items are laid out inside a flex container.
Everything outside a flex container and inside a flex item is rendered as usual.
To create a flex container, write:

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

Inside a flex container there can be one or more flex items. Once an element is inside a flexbox container, it will follow the flexbox layout rules (instead of standard block & inline rules).
Flex items will flow inside a container along the *main-axis*. With

```
 flex-flow: row;
```

you are setting the **main axis horizontally.**

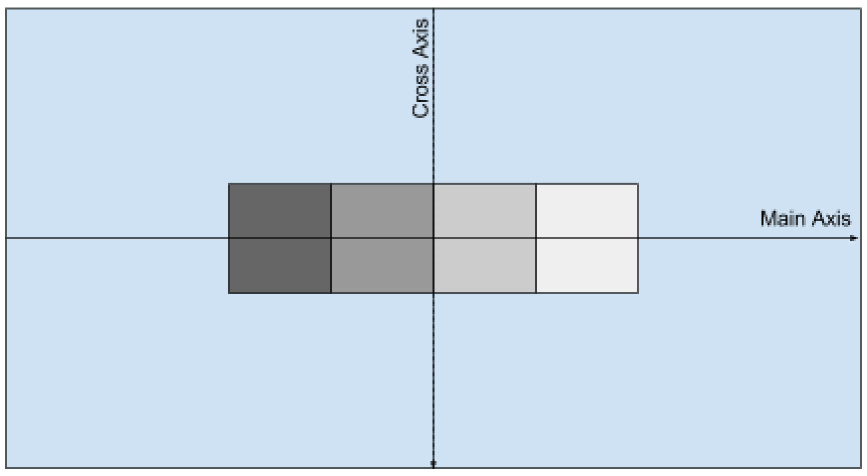


In a default case, the main axis flows left to right.
With **flex-flow: column;** the main axis will be **vertically aligned**.

Align flex box items with justify-content & align-items

The **justify-content** attribute defines the alignment of the flex-elements along the main axis, both when they are in one line and when they are overflowing.

For **justify-content: center;** the result may look like this:



Some of the most commonly used justify-content values are:

1. **flex-start** (default): items are packed toward the **start** of main axis
2. **flex-end:** items are packed toward to **end** of main axis
3. **center:** items are **centered** along the main axis
4. **space-between:** items are distributed **along the main axis**; first item is on the start of the main axis, the last item on the end of the main axis. The space between the items always looks the same.
5. **space-around:** items are evenly distributed in the line with equal space around them (think of it as magic that makes the margin-left=margin-right for all flexbox items, regardless of the size of the container). Note however that visually the spaces aren't equal.

Comparatively, **align-items** align elements along the cross-axis, similarly to how **justify-content** aligns those items along the main axis.

Some of the most commonly used align-items properties are:

1. **stretch** default. Items are stretched to fit the flex-container along the cross-axis.
2. **center** items are positioned at the **center** of the container
3. **flex-start** items are positioned at the **beginning** of the container
4. **flex-end** items are positioned at the **end** of the container
5. **baseline** items are positioned at the **baseline** of the container
6. **space-between** lines evenly distributed; the first line is at the start of the container while the last one is at the end
7. **space-around:** lines evenly distributed with equal space around each line

Flex-container: axis direction & wrapping items

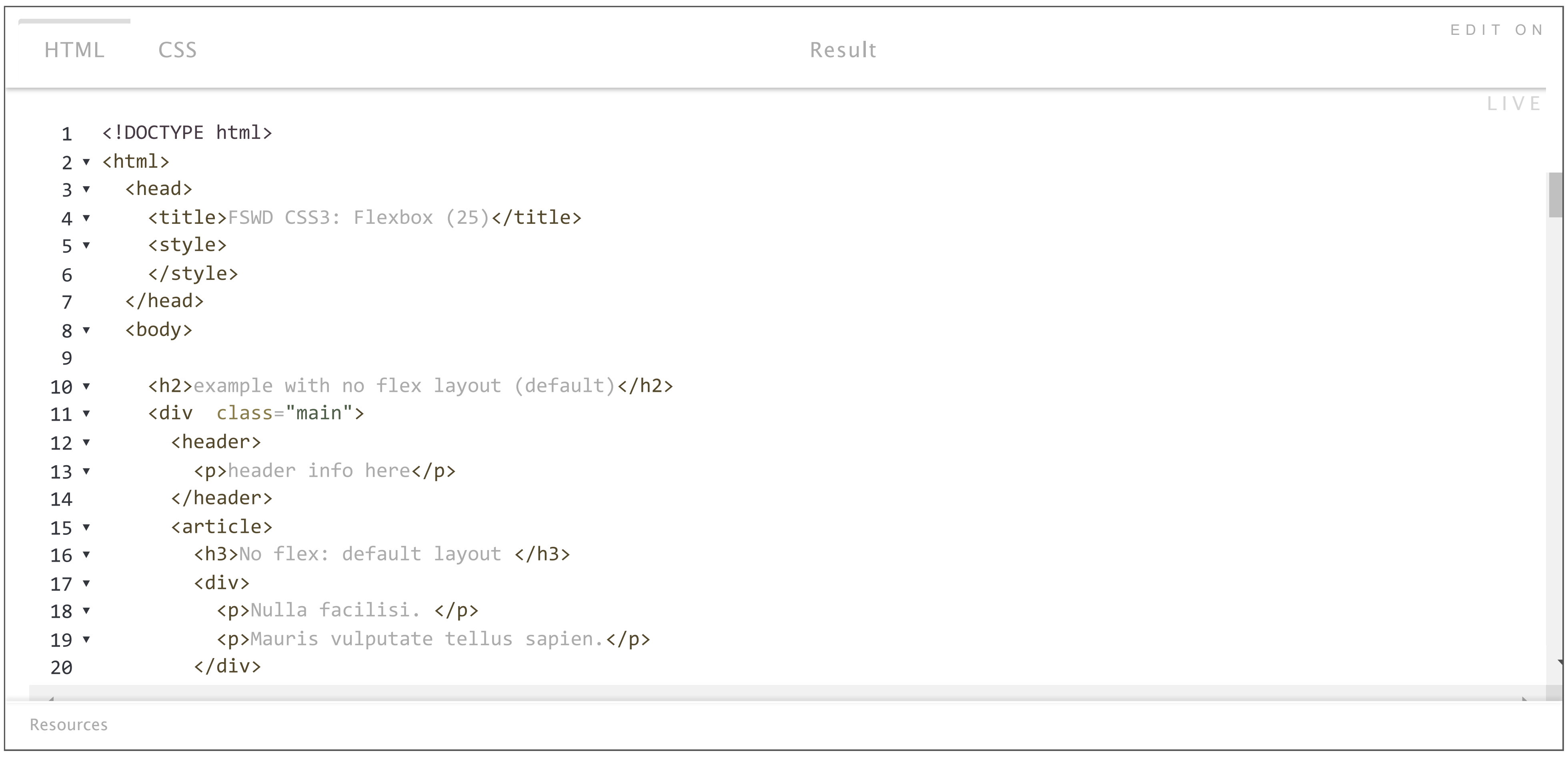
To change the main axis orientation and direction, use **flex-direction.** The **flex-direction** attribute establishes the main-axis on which the flex items are placed in the container. This can be **row** (default) or **row-reverse**, for left-to-right or right-to-left ordering, as well as **column** and **column-reverse** for top-to-bottom and bottom-to-top ordering. The **flex-wrap** attribute comes in handy when creating responsive layouts. **flex-wrap** sets the wrapping of flexbox items inside a container if the items will all try to fit into one line (**nowrap**, which is the default), **wrap** (which allows items to wrap onto the next line from left to right) or **wrap-reverse** (which does the same thing, only in opposite direction).

```
.container {
   flex-direction: row | row-reverse | column | column-reverse;
   flex-wrap: nowrap | wrap | wrap-reverse;
}
```

There is, of course, a shorthand variant of this, called **flex-flow**. This sets the flex-direction and flex-wrap in one line, like this:

```
.container {
   flex-flow: <'flex-direction'> || <'flex-wrap'>
}
```

To see some of those properties in action, check out this example:



```
<!DOCTYPE html>
<html>
  <head>
    <title>FSWD CSS3: Flexbox (25)</title>
    <style>
    </style>
  </head>
  <body>

    <h2>example with no flex layout (default)</h2>
    <div  class="main">
      <header>
        <p>header info here</p>
      </header>
      <article>
        <h3>No flex: default layout </h3>
        <div>
          <p>Nulla facilisi. </p>
          <p>Mauris vulputate tellus sapien.</p>
        </div>
```

The Flex Items

The **flex-grow** attribute (an integer) dictates the amount of the available space inside the flex container that an item should take up:

- If only one item has **flex-grow:1;** and all other **flex-grow: 0;** the “item with 1” will take all available space along the main-axis;
- If all flex items have **flex-grow:1;** all will take up the space equally.
- If one of the flex-items has **flex-grow:2;** and all others **flex-grow:1;** the “item with 2” will try to take up twice the remaining space compared to the others.

flex-shrink has the same functionality, just in reverse where the flex-items are pressed to shrink in size. Combining **flex-grow** and **flex-shrink** with **wrap** is a simple way for creating responsive logic in CSS.

An **order** attribute sets the order in which they appear in the flex container:

```
.item {
 order: <integer>;
}
```

Default value is 0;

The **flex-basis** defines the default-size of an element before the remaining space is distributed. Default is **auto** (determine the width based on the content).

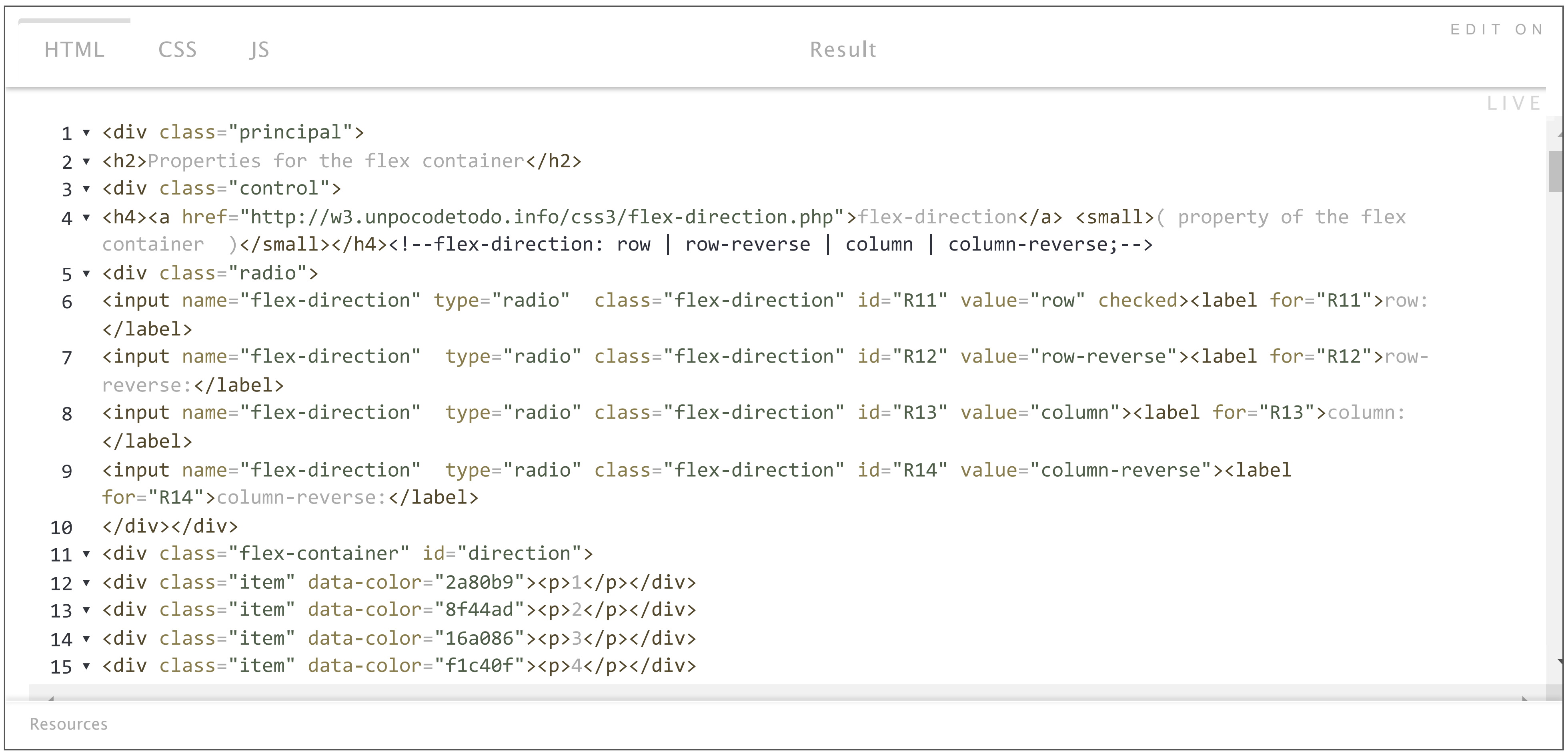
```
.item {
 flex-grow: <number>; /* default 0 */
 flex-shrink: <number>; /* default 1 */
 flex-basis: <length> | auto; /* default auto */
}
```

There is a shorthand for this, appropriately just named “flex” that combines flex-grow, flex-shrink and flex-basis.

```
.item {
 flex: none | [ <'flex-grow'> <'flex-shrink'>? || <'flex-basis'> ]
}
```

In many cases, this shorthand is even superior to setting individual properties, as the second and third parameters are optional and will be set automatically.

The **align-self** attribute allows a flex-item to override the general alignment as given through **align-items** on the flex container level.



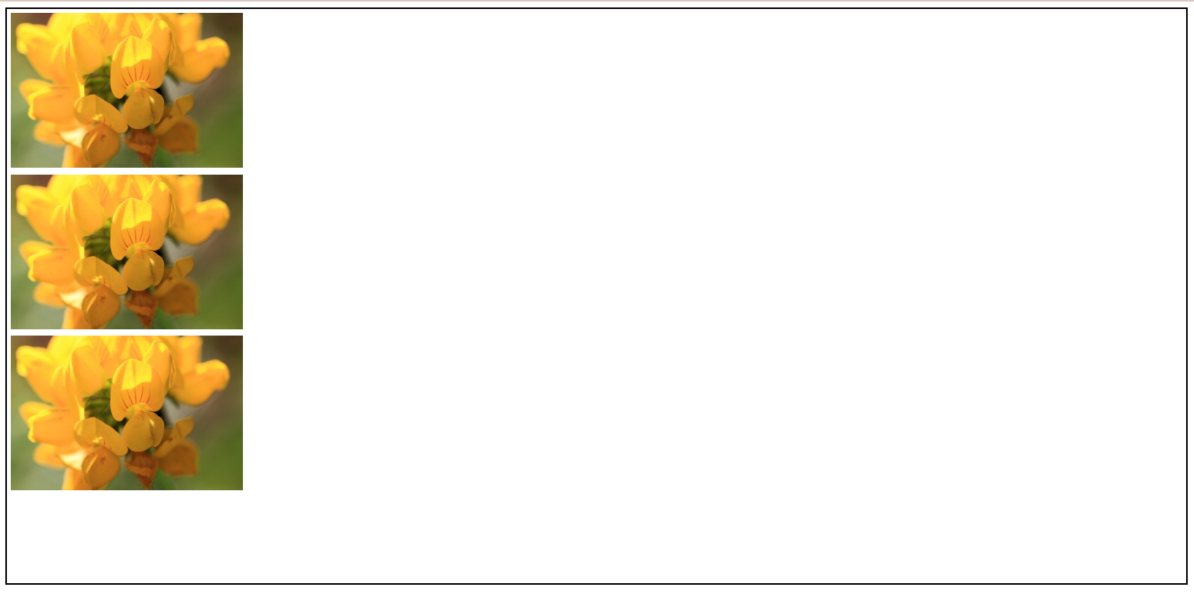
```
<div class="principal">
<h2>Properties for the flex container</h2>
<div class="control">
<h4><a href="http://w3.unpocodetodo.info/css3/flex-direction.php">flex-direction</a> <small>( property of the flex container   )</small></h4><!--flex-direction: row | row-reverse | column | column-reverse;-->
<div class="radio">
<input name="flex-direction" type="radio"  class="flex-direction" id="R11" value="row" checked><label for="R11">row:</label>
<input name="flex-direction"  type="radio" class="flex-direction" id="R12" value="row-reverse"><label for="R12">row-reverse:</label>
<input name="flex-direction"  type="radio" class="flex-direction" id="R13" value="column"><label for="R13">column:</label>
<input name="flex-direction"  type="radio" class="flex-direction" id="R14" value="column-reverse"><label for="R14">column-reverse:</label>
</div></div>
<div class="flex-container" id="direction">
<div class="item" data-color="2a80b9"><p>1</p></div>
<div class="item" data-color="8f44ad"><p>2</p></div>
<div class="item" data-color="16a086"><p>3</p></div>
<div class="item" data-color="f1c40f"><p>4</p></div>
```

*Reference: http://w3.unpocodetodo.info/css3/flex-direction.php

To begin implementing the idea of using Flexbox, we will start with this example :

```
<!DOCTYPE html>
<html>
<head>
        <title>Example for Flexbox</title>
        <meta charset="utf-8">
     <style>
        .container{
           width: 80%;
           margin: 0 auto;
           border: solid black 2px;
           height: 80vh;
        }
        .container div{
           margin: 5px;
        }
        </style>
</head>
<body>
        <div><img src="https://placeimg.com/300/200/nature" alt="">
</div>
           <div><img src="https://placeimg.com/300/200/nature" alt="">
</div>
           <div><img src="https://placeimg.com/300/200/nature" alt="">
</div>
</body>
</html>
```

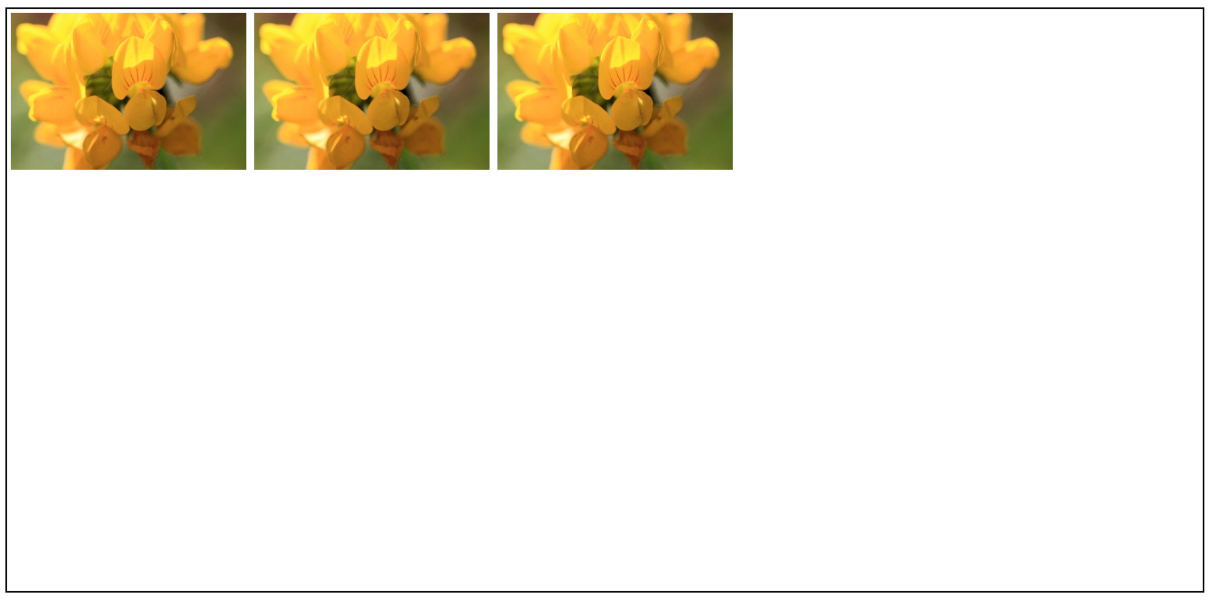
There are 3 div elements in HTML each of them with an image within:



In order to display the 3 images horizontally, a wrapper or container must be created. Just create another div element, moving those 3 elements into it. Assign it a class container. This class will hold the attribute **display: flex;**

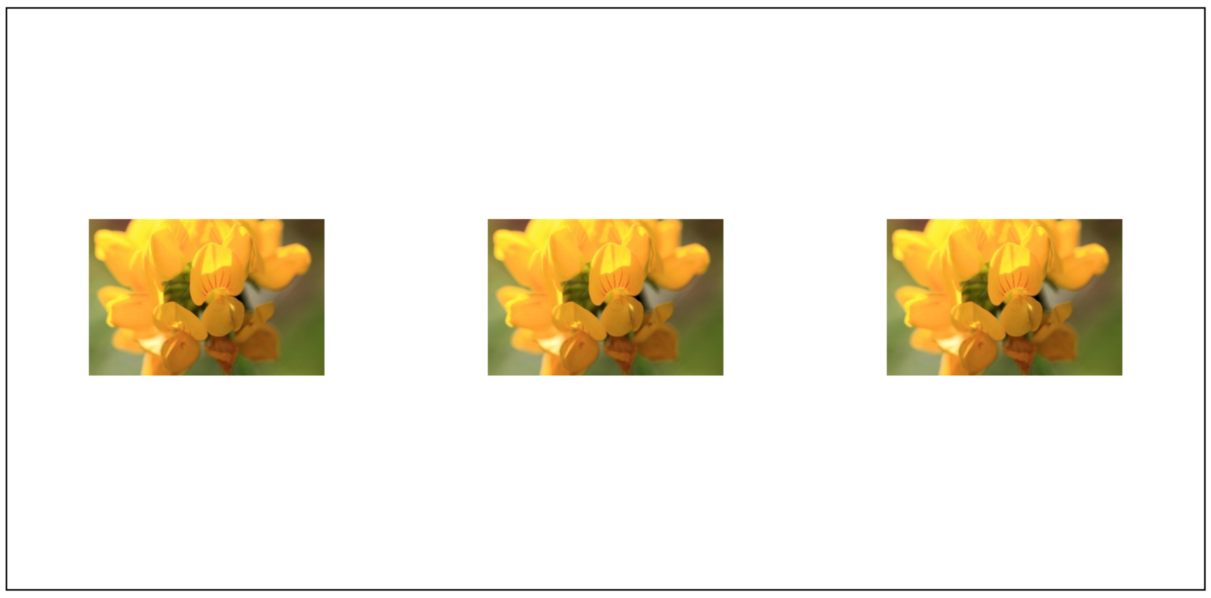
```
<!DOCTYPE html>
<html>
<head>
        <title>Example for Flexbox</title>
        <meta charset="utf-8">
     <style>
        .container{
           width: 80%;
           margin: 0 auto;
           border: solid black 2px;
           height: 80vh;
      display: flex;
        }
        .container div{
           margin: 5px;
        }
        </style>
</head>
<body>
   <div class="container">
        <div><img src="https://placeimg.com/300/200/nature" alt="">
</div>
           <div><img src="https://placeimg.com/300/200/nature" alt="">
</div>
           <div><img src="https://placeimg.com/300/200/nature" alt="">
</div>
   </div>
</body>
</html>
```

The result will look like as below :



All elements are in the same line. The positioning of the elements will change according to the use of **justify-content** attribute that has the value options (space-between, space-around, center, flex-start and flex-end); Justify-content will distribute the elements on the main-axis according to the value chosen. And considering that the container has the height bigger than the content, the attribute **align-item**: center could be used to bring the flex-elements to the center of the cross-axis.

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



For further information and from the resources used, we highly recommend taking a look at: https://css-tricks.com/snippets/css/a-guide-to-flexbox/

Variables - custom properties

Variables are elements that can "hold" information and allow this information to be used whenever and as many times as we want. The concept of variables is largely used not only in CSS but for programming in general. Think of it as a box with some content that can be used. You may hear the terminology custom property referring to them as well. Variables will help mainly to avoid repeating the same information many times, and as well will help you to name certain properties, making styling easier. For example:

Whenever the colors of a website are defined, you will probably have to repeat them many times over:

```
 .elementOne{
              color: #636318;
              background-color: #4343b8;
              margin: 10px;
              display: inline-block;
          }
.elementTwo{
              color: #10164e;
              background-color: #c75e21;
              margin: 10px;
              display: inline-block;
          }
.elementThree{
              color: #636318;
              background-color: #4343b8;
              margin: 10px;
              display: inline-block;
          }
```

What if instead, you could name the color values and whenever you want to use them, you only need to call their names? Let's see the syntax:

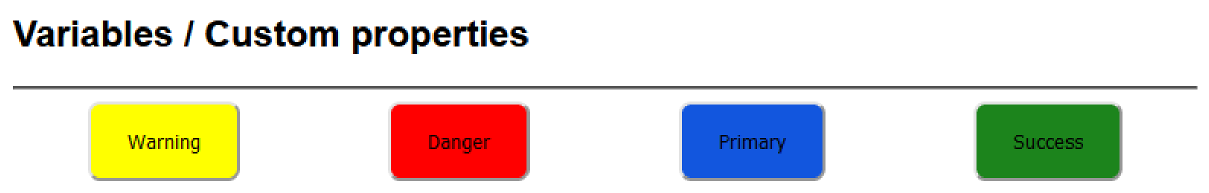
```
:root{
        --variable-name: value;
}
```

Declaring the variable into the pseudo-class **root** will make it global to be used anywhere in the document.

```
:root{
        --primary-bg: #4343b8;
        --secondary-bg:  #c75e21;
        --primary-text: #636318;
     --secondary-text: #10164e;
}
.elementOne{
               color: var(--primary-text);
               background-color: var(--primary-bg);
               margin: 10px;
               display: inline-block;
           }
.elementTwo{
               color:var(--secondary-text);
               background-color: var(--secondary-bg);
               margin: 10px;
               display: inline-block;
           }
.elementThree{
               color: var(--primary-text);
               background-color: var(--primary-bg);
               margin: 10px;
               display: inline-block;
           }
```

In order to bring the value from the variables, the function var() must be used.

Let's see a more practical example:



See the HTML and CSS below:

```
<!DOCTYPE html>
<html lang="en">

<head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <title>Variables CSS</title>
<body>
        <h2>Variables / Custom properties</h2>
        <hr>
        <div class="container">
           <button class="btn bg-warning">Warning</button>
           <button class="btn bg-danger">Danger</button>
           <button class="btn bg-primary">Primary</button>
           <button class="btn bg-success">Success</button>
        </div>
</body>
</html>
```

CSS:

```
body {
            font-family: Arial, Helvetica, sans-serif;
        }

        :root {
            --color-danger: #ff0000;
            --color-warning: #ffff00;
            --color-success: #1c841c;
            --color-primary: #1256de;
        }

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

        .btn {
            padding: 1rem 1.5rem;
            background: transparent;
            font-weight: 500;
            border-radius: 0.5rem;
            cursor: pointer;
            outline: none;
        }

        .bg-danger {
            background-color: var(--color-danger);
        }

        .bg-warning {
            background-color: var(--color-warning);
        }

        .bg-success {
            background-color: var(--color-success);
        }

        .bg-primary {
            background-color: var(--color-primary);
        }
```

The .btn class has the general button style so all the buttons will look the same. The class .bg-primary for example is a modifier class, it carries only the background and the color stored in a variable. This background color class can be reused anywhere in the code and if you decide to change the color bcz maybe the blue is too bright or the red is too dark you only need to go into the variables and by changing the color, it will affect all elements with that class.

Last modified: Monday, 12 September 2022, 8:24 AM

◄ CSS3 | Advanced

Jump to...

CSS | Quiz | Day 03 ►