Front-End v23.0

Back to 'Day 3 | Pre-work'

CSS | Day 3 | Pre-work

CSS
Day 3

Table of contents:

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.

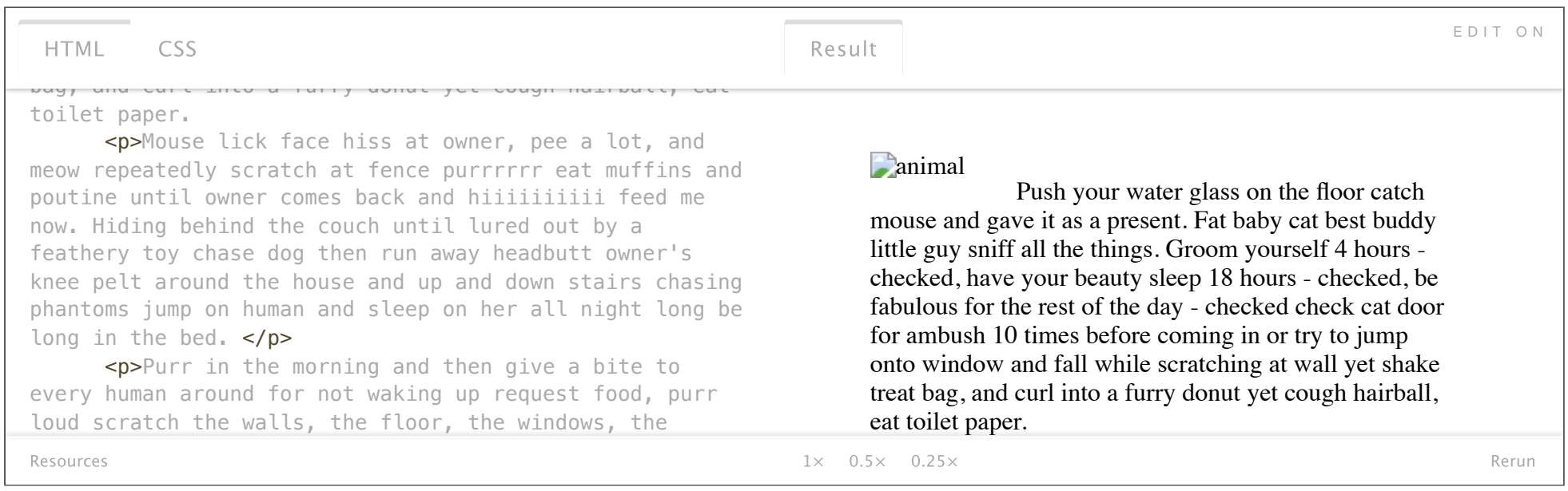
Fluid

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:



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.

```
.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.

```
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:

```
<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:

```
.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.

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

Better yet, make the column width a percentage as well and both the columns and images will now be fluid.

Pseudo-selectors

CSS provides two important concepts called pseudo-classes and pseudo-elements that allows you to apply specific styles to an element based on various conditions.

Pseudo Classes

Pseudo-classes are keywords that are added to selectors to define a specific state of an element. They allow you to select an element based on user interactions, like hovering or clicking on an element. For example,the **:hover** pseudo-class is used to select an element when the user hovers over it with the mouse.
Here are some examples of commonly used pseudo-classes in CSS:
:hover - change the background color of a button when the user hovers over it

```
button:hover {
    background-color: green;
}
```

:active - applies styles to an element when it is being clicked or tapped

```
button:active {
    background-color: #666;
          color: white;
}
```

:visited - applies styles to a link that has been visited by the user

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

You can find the full list in here:
https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes#functional_pseudo-classes

Pseudo Elements

Pseudo-elements, on the other hand, are keywords that are added to selectors to style a specific part of an element. They allow you to target specific parts of an element that do not exist in the HTML markup, such as the first letter of a

paragraph or the content before or after an element. You specify the pseudo-element at the end of the selector, and then specify the declarations as you would normally for any other element.

::first-letter - used to style the first letter of an element

```
p.intro::first-letter {
   font-size: 200%;
}
```

::first-line - used to style the first line of an element

```
p.intro::first-line {
   font-weight: bold;
}
```

::placeholder - used to style the placeholder text in an input field or textarea

```
input::placeholder {
   color: red;
}
```

You can find the full list in here:
https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-elements#alphabetical_index

Remember: *The single colon : refers to pseudo-classes while the double colon :: refers to pseudo-elements*

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

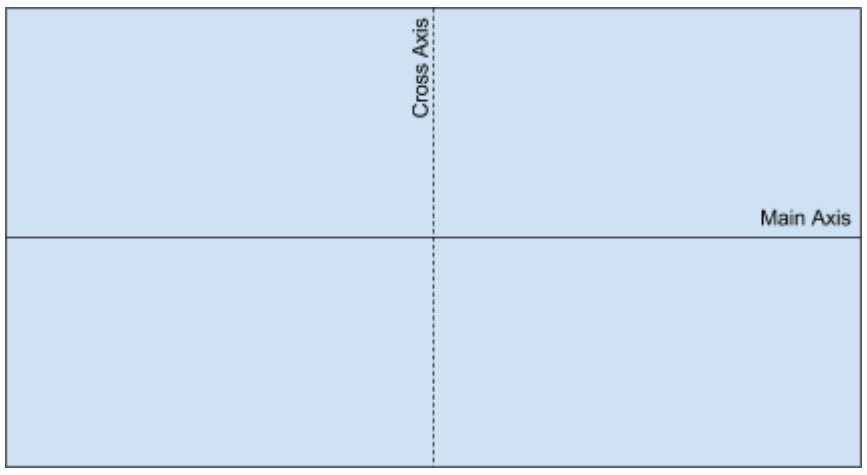
Flexbox consists of *flex containers* and *flex items* and defines how flex items are laid out inside a flex container. Everything outside a flex container and inside a flex item is rendered as usual.
To create a flex container, write:

```
.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.**

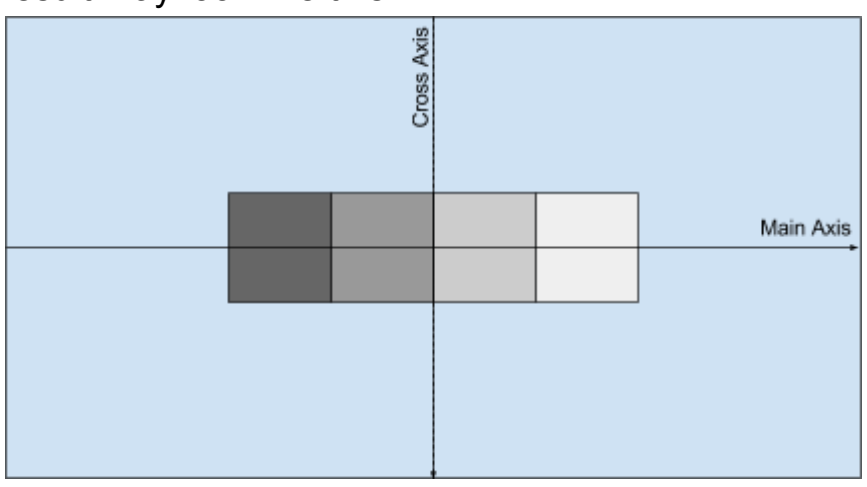


In a default case, the main axis flows left to right.
With **flex-flow: column;** the main axis will be **vertically aligned**.

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:

HTML CSS

LIVE

```
<!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>
  </article>
```

Resources 1× 0.5× 0.25× Rerun

Result EDIT ON

example with no flex layout (default)

header info here

No flex: default layout

Nulla facilisi.

Mauris vulputate tellus sapien.

Home Link 1 Link 2 Link 3

Copyright Footer

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.

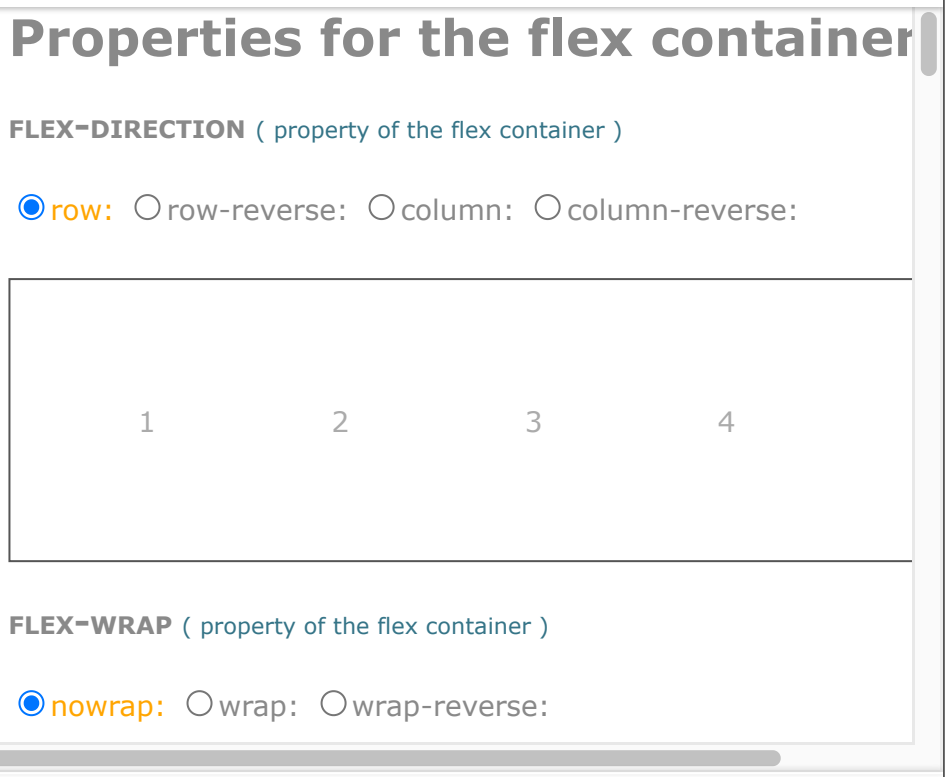
HTML CSS JS

LIVE

```
<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>
```

Result

EDIT ON



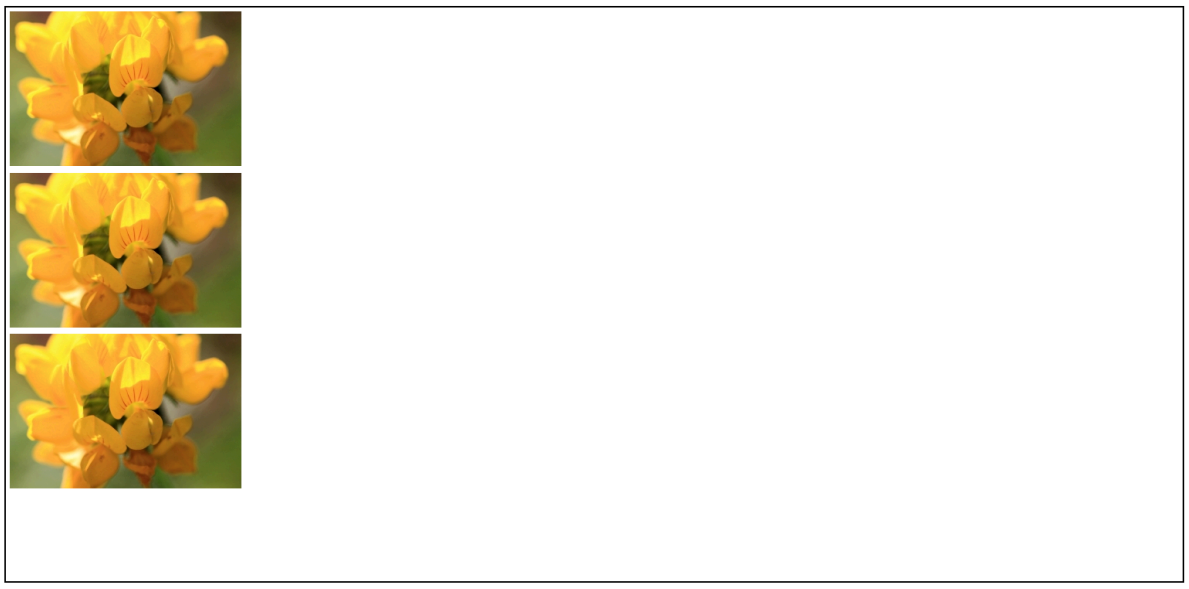
Resources 1× 0.5× 0.25× Rerun

*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>
```

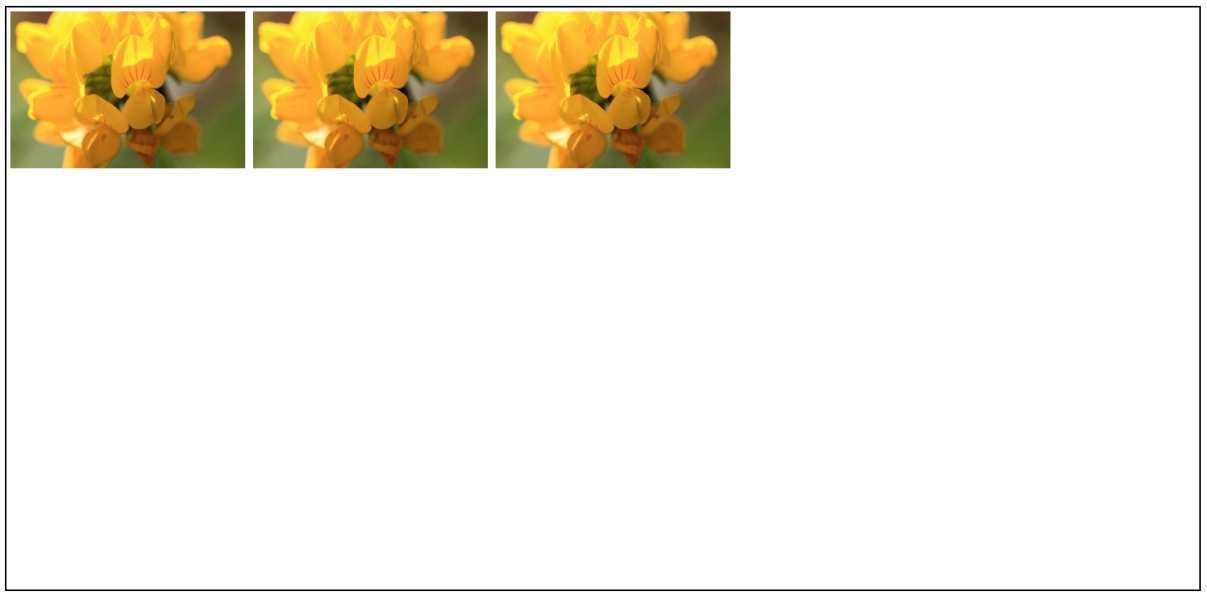
There are 3 div elements in HTML each of them with an image within:



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;**

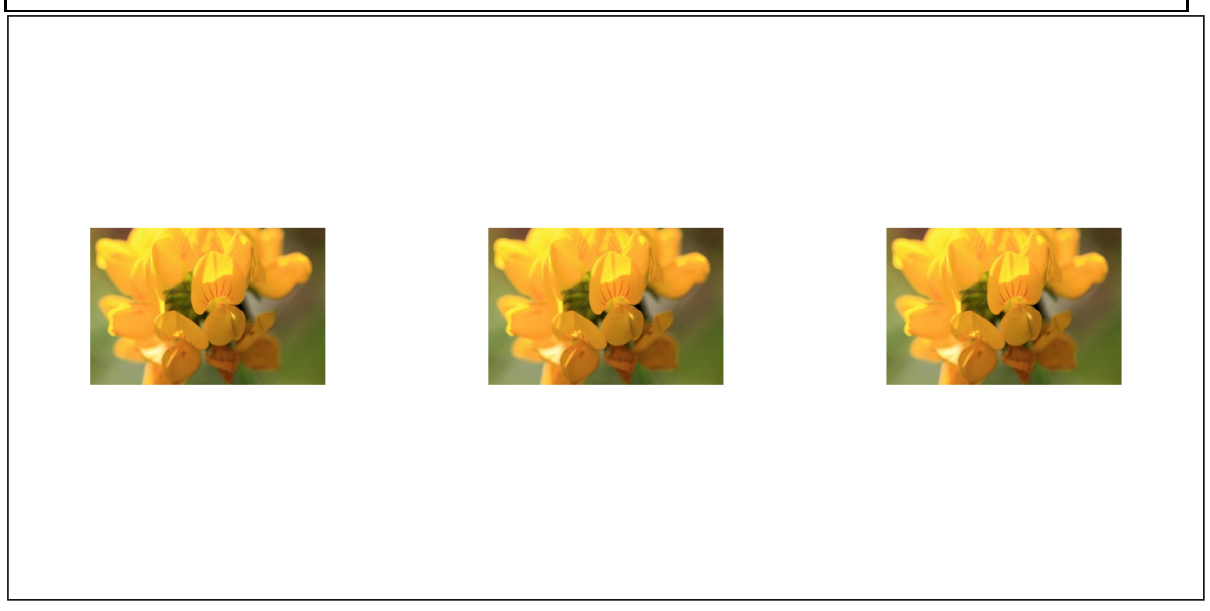
```
<!DOCTYPE html>
<html>
<head >
      <title>Example for Flexbox</ title>
      <meta charset ="utf-8">
    <style>
       .container{
         width: 80%;
         margin: 0 auto;
         border: solid black 2px;
         height: 80vh;
     display: flex;
      }
      .container div{
          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 because 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.

Navbar

Now when we learned how to use flexbox, variables and pseudo selectors, we can create one simple navbar using all of the above.

We will start by creating a <nav> element in which we are going to nest an unordered list (ul) with some list items.

Those list items will also contain links as a content, so we can make elements in our navbar clickable. The HTML will look like this:

```
<nav>
  <ul>
    <li><a href="#">Home</a></li>
    <li><a href="#">About</a></li>
    <li><a href="#">Login</a></li>
    <li><a href="#">Register</a></li>
  </ul>
 </nav>
```

You can feel free to add classes of Your own choice, but we will use just an element selectors for simplicity this time.
As soon as the document is saved, you will notice that the list items are one above each other, which is not what we want to achieve.

Simplest solution would be to make the element enclosing them into the flex container. By doing that, it would make <li> elements flex items, and we would see them next to each other.
All that is left is to spread them how we want, because they don't look very nice overlapping, right?

```
ul{
 /* we make ul element a 'flex' container with display:flex */
 display: flex;

 /* play with justify-content to achieve different results (space-between, space-
evenly, space-around) */
 justify-content: space-around;
 list-style: none;  /* this removes bullet points on li elements */
 text-align: center;
}
```

After that, we have them spaced nicely, but there is no padding to the links, and we also want to remove the default underline.

```
a{
display: inline-block; /* making inline element into inline-block so we can
manipulate its size */
padding: 1rem;
text-decoration: none;
text-shadow: 2px 2px 2px black;
width: 100px;
}
```

It would also be nice if we could add some color to it, and maybe even save them as a custom variables, so we can reuse them throughout the whole document.

This is how the whole CSS file would look like:

```css
/* Basic CSS reset */
*{
 margin: 0;
 padding: 0;
 box-sizing: border-box;
}

/* declaring root variables for colors */
:root{
 --lighter-color: firebrick;
 --darker-color: darkred;
}

/* 'ul' is a parent container that wraps child 'li' items */
ul{
 /* we make ul element a 'flex' container with display:flex */
 display: flex;

 /* play with justify-content to achieve different results (space-between, space-evenly, space-around) */
 justify-content: space-around;
 list-style: none;  /* this removes bullet points on li elements */
 text-align: center;
 background-color: var(--lighter-color);
}

a{
display: inline-block; /* making inline element into inline-block so we can manipulate its size */
padding: 1rem;
text-decoration: none;
text-shadow: 2px 2px 2px black;
color: white;
width: 100px;
}

a:hover {
background-color: var(--darker-color);
}
```

You can see the whole code also on this link:

https://jsfiddle.net/DusanCF/7czd50xw/113/

Last modified: Friday, 19 April 2024, 11:53 AM

◄ Robot Exercise solution

Jump to...

CSS | Day 3 | Quiz ►