Agenda:

- Flex Box Layout
- Grid Layout
- Debugging in css

In the last module, we have learnt about some advance topics in css like typography, box model, positioning and display properties.

In this module we will continue to learn about advance topics in css and we will start understanding that how we can layout our web page as per o requirements using css.

We will start with layout with Flexbox:

Flexbox:

We have learned that we can layout our elements on a web page with the help of display and position properties of css as we have seen that in previou module.

Actually, flexbox or layouting with flexbox, is a tool that greatly simplifies the positioning of our elments on our web page.

There are two important components to a flexbox layout: *flex containers* and *flex items*. A flex container is an element on a page that contains flex item All direct child elements of a flex container are flex items. This distinction is important because some of the properties you will learn in this lesson apply flex containers while others apply to flex items.

To designate an element as a flex container, set the element's `display` property to `flex` or `inline-flex` . Once an item is a flex container, the are several properties we can use to specify how its children behave. we are going to cover these properties:

1. `justify-content`
2. `align-items`
3. `flex-grow`
4. `flex-shrink`
5. `flex-basis`
6. `flex`
7. `flex-wrap`
8. `align-content`
9. `flex-direction`
10. `flex-flow`

Let's start with understanding that how we can create a container a flexbox container:

Display: flex

Any element can be a flex container. Flex containers are helpful tools for creating websites that respond to changes in screen sizes. Child elements flex containers will change size and location in response to the size and position of their parent container.

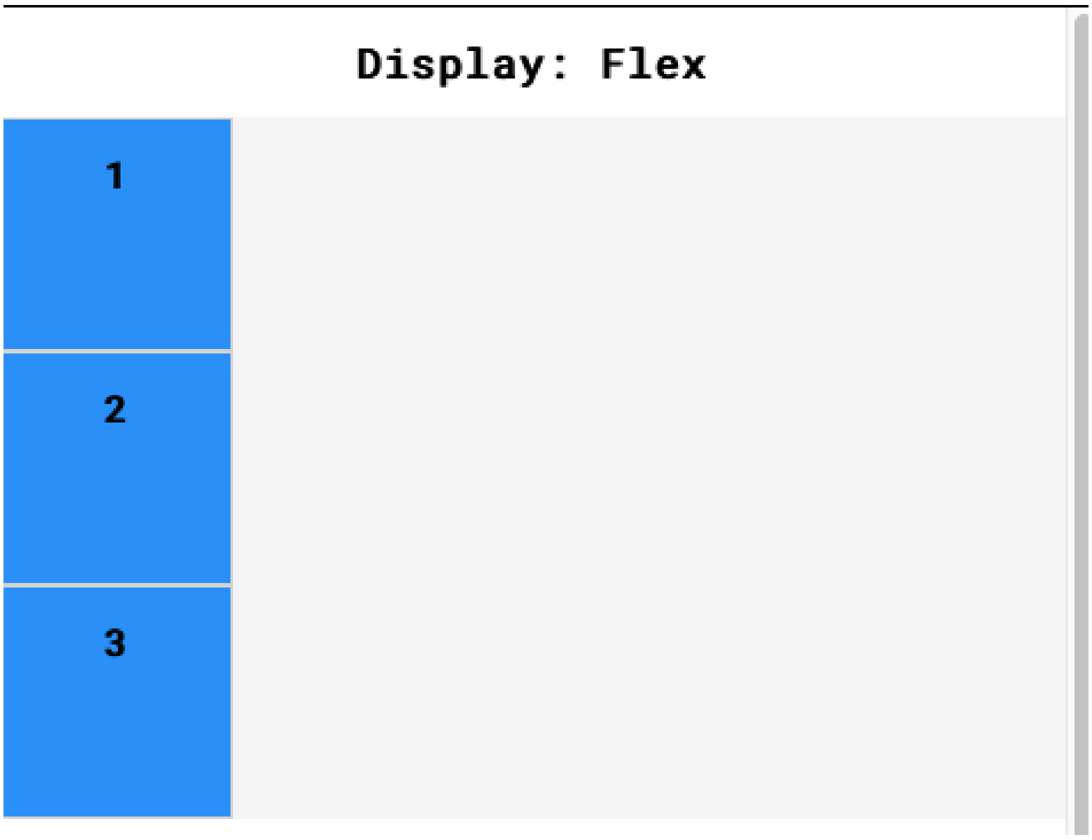
For an element to become a flex container, its `display` property must be set to `flex` .

Consider the example shown below:

```
<!DOCTYPE html>
<html>

<head>
  <style>
        .container {
                  background-color: whitesmoke;
                    display: flex;
                }
        .box {
          background-color: blue;
          width: 50px;
          height: 50px;
        }
      </style>
```

```
</head>

<body>
  <h1>Display: Flex</h1>
  <div class='container' id='flex'>
    <div class='box'>
      <h2>1</h2>
    </div>
    <div class='box'>
      <h2>2</h2>
    </div>
    <div class='box'>
      <h2>3</h2>
    </div>
  </div>
</body>
</html>
```

Output:



In the example above, all divs with the class `container` are flex containers. If they have children, the children are flex items. A div with th declaration `display: flex;` will remain block level — no other elements will appear on the same line as it.

However, it will change the behavior of its child elements. Child elements will not begin on new lines.

Now, we can also make a block level element inline as:

Display: inline-flex

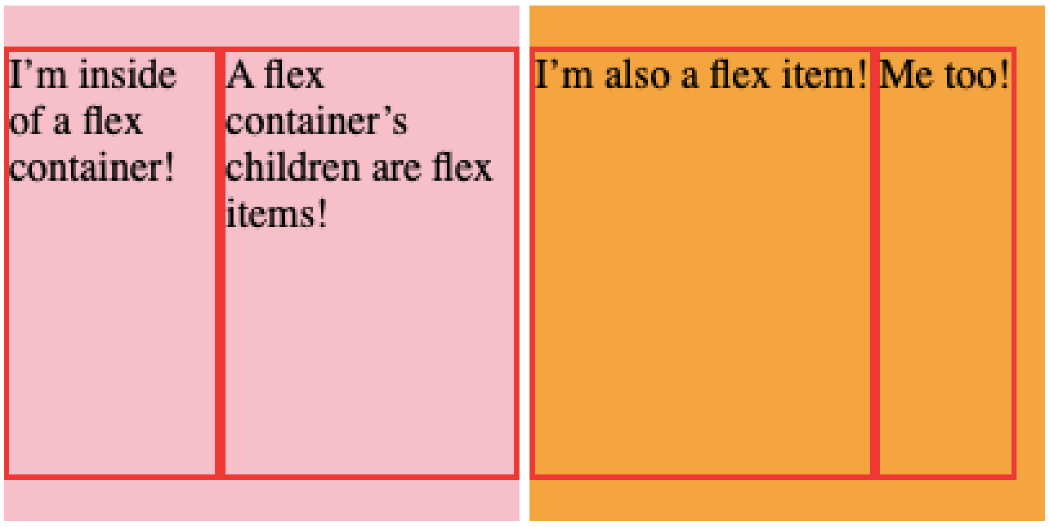
If we didn't want div elements to be block-level elements, we would use `display: inline` . Flexbox, however, provides the `inline-flex` value f the `display` property, which allows us to create flex containers that are also inline elements.

For example:

```
<!DOCTYPE html>
<html>

<head>
```

```
    <style>
      .container {
    width: 200px;
    height: 200px;
    display: inline-flex;
  }
    .containerOne {
      background-color: pink;
    }
    .containerTwo {
      background-color: orange;
    }
    p {
      border: 2px solid red;
    }
    </style>
  </head>

  <body>
    <div class='container containerOne'>
    <p>I'm inside of a flex container!</p>
    <p>A flex container's children are flex items!</p>
  </div>
  <div class='container containerTwo'>
    <p>I'm also a flex item!</p>
    <p>Me too!</p>
  </div>
  </body>

  </html>
```

Output:


I'm inside of a flex container!
A flex container's children are flex items!
I'm also a flex item!
Me too!

In the example above, When we change the value of the `display` property to `inline-flex`, the divs will display inline with each other if the page wide enough.

Now, let's understand the properties that can be applied on the child elements of the flex containers:

justify-content:

when we changed the `display` value of parent containers to `flex` or `inline-flex`, all of the child elements (flex items) moved toward the upp left corner of the parent container. This is the default behavior of flex containers and their children. We can specify how flex items spread out from left right, along the *main axis*. We will learn more about axes in a later exercise.

To position the items from left to right, we use a property called `justify-content`.

Below are five commonly used values for the `justify-content` property:

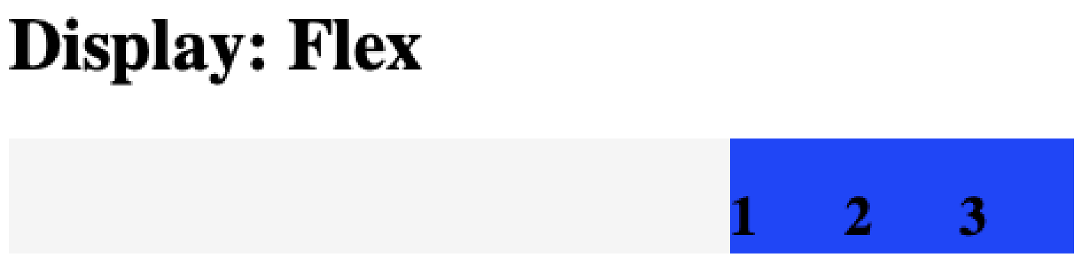
- `flex-start` — all items will be positioned in order, starting from the left of the parent container, with no extra space between or before them.
- `flex-end` — all items will be positioned in order, with the last item starting on the right side of the parent container, with no extra space between or after them.
- `center` — all items will be positioned in order, in the center of the parent container with no extra space before, between, or after them.
- `space-around` — items will be positioned with equal space before and after each item, resulting in double the space between elements.
- `space-between` — items will be positioned with equal space between them, but no extra space before the first or after the last elements.

Consider the example :

This is code is applied to the first example:

```
<!DOCTYPE html>
<html>

<head>
  <style>
        .container {
                  background-color: whitesmoke;
                    display: flex;
                    justify-content: flex-end;
                }
            .box {
          background-color: blue;
          width: 50px;
          height: 50px;
        }
    </style>
</head>

<body>
  <h1>Display: Flex</h1>
  <div class='container' id='flex'>
    <div class='box'>
      <h2>1</h2>
    </div>
    <div class='box'>
      <h2>2</h2>
    </div>
    <div class='box'>
      <h2>3</h2>
    </div>
  </div>
</body>
</html>
```

Output:



Display: Flex

In the example above, we set the value of `justify-content` to `flex-end` . This will cause all of the flex items to shift to the right side of the fle
container.

align-items:

It is also possible to align flex items vertically within the container. The `align-items` property makes it possible to space flex items vertically.

Below are five commonly used values for the `align-items` property:

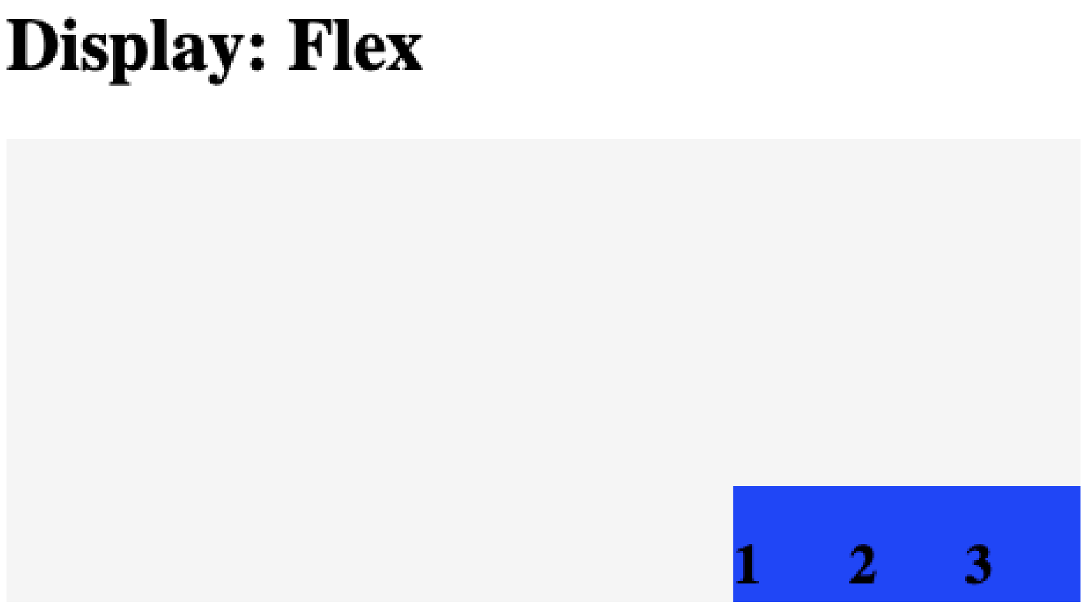
- `flex-start` — all elements will be positioned at the top of the parent container.
- `flex-end` — all elements will be positioned at the bottom of the parent container.
- `center` — the center of all elements will be positioned halfway between the top and bottom of the parent container.
- `baseline` — the bottom of the content of all items will be aligned with each other.
- `stretch` — if possible, the items will stretch from top to bottom of the container (this is the default value; elements with a specified height will not stretch; elements with a minimum height or no height specified will stretch).

Example:

```
<!DOCTYPE html>
<html>

<head>
  <style>
        .container {
                  background-color: whitesmoke;
          height: 200px;
                    display: flex;
                    justify-content: flex-end;
          align-items: flex-end;
                }
            .box {
          background-color: blue;
          width: 50px;
          height: 50px;
        }
    </style>
</head>

<body>
  <h1>Display: Flex</h1>
  <div class='container' id='flex'>
    <div class='box'>
      <h2>1</h2>
    </div>
    <div class='box'>
      <h2>2</h2>
    </div>
    <div class='box'>
      <h2>3</h2>
    </div>
  </div>
```

```
</body>
</html>
```

Output:


Display: Flex

In the above example, we have set the align-items property to flex-end that has sent the child elements in the bottom of the container.

flex-grow:

The `flex-grow` property allows us to specify if items should grow to fill a container and also which items should grow proportionally more or less tha
others.

consider the example shown below:

```
<div class='container'>
  <div class='side'>
    <h1>I’m on the side of the flex container!</h1>
  </div>
  <div class='center'>
    <h1>I'm in the center of the flex container!</h1>
  </div>
  <div class='side'>
    <h1>I'm on the other side of the flex container!</h1>
  </div>
</div>
```

```
.container {
  display: flex;
}

.side {
  width: 100px;
  flex-grow: 1;
}

.center {
  width: 100px;
  flex-grow: 2;
}
```

In the example above, the `.container` div has a `display` value of `flex` , so its three child divs will be positioned next to each other. If there additional space in the `.container` div (in this case, if it is wider than 300 pixels), the flex items will grow to fill it. The `.center` div will stretch twic as much as the `.side` divs.

flex-shrink:

Just as the `flex-grow` property proportionally stretches flex items, the `flex-shrink` property can be used to specify which elements will shrink ar in what proportions.

Consider the example shown below:

```
<div class='container'>
  <div class='side'>
    <h1>I'm on the side of the flex container!</h1>
  </div>
  <div class='center'>
    <h1>I'm in the center of the flex container!</h1>
  </div>
  <div class='side'>
    <h1>I'm on the other side of the flex container!</h1>
  </div>
</div>
```

```
.container {
  display: flex;
}

.side {
  width: 100px;
  flex-shrink: 1;
}

.center {
  width: 100px;
  flex-shrink: 2;
}
```

In the example above, the `.center` div will shrink twice as much as the `.side` divs if the `.container` div is too small to fit the elements within it. the content is 60 pixels too large for the flex container that surrounds it, the `.center` div will shrink by 30 pixels and the outer divs will shrink by pixels each. Margins are unaffected by `flex-grow` and `flex-shrink` .

flex-basis:

We seen that till now the dimensions of the divs were determined by heights and widths set with CSS. Another way of specifying the width of a flex ite is with the `flex-basis` property. `flex-basis` allows us to specify the width of an item before it stretches or shrinks.

Example:

```
<div class='container'>
  <div class='side'>
    <h1>Left side!</h1>
  </div>
  <div class='center'>
    <h1>Center!</h1>
  </div>
  <div class='side'>
    <h1>Right side!</h1>
  </div>
</div>
```

```
.container {
  display: flex;
}

.side {
  flex-grow: 1;
  flex-basis: 100px;
}

.center {
  flex-grow: 2;
  flex-basis: 150px;
}
```

In the example above, the `.side` divs will be 100 pixels wide and the `.center` div will be 150 pixels wide if the `.container` div has just the rig amount of space (350 pixels, plus a little extra for margins and borders). If the `.container` div is larger, the `.center` div will absorb twice as muc space as the `.side` divs.

flex-wrap:

Sometimes, we want flex items to move to the next line when necessary. This can be declared with the `flex-wrap` property. The `flex-wrap` property can accept three values:

1. `wrap` — child elements of a flex container that don't fit into a row will move down to the next line
2. `wrap-reverse` — the same functionality as `wrap`, but the order of rows within a flex container is reversed (for example, in a 2-row flexbox, the first row from a `wrap` container will become the second in `wrap-reverse` and the second row from the `wrap` container will become the first in `wrap-reverse`)
3. `nowrap` — prevents items from wrapping; this is the default value and is only necessary to override a wrap value set by a different CSS rule.

Consider the example:

```
<div class='container'>
  <div class='item'>
    <h1>We're going to wrap!</h1>
  </div>
  <div class='item'>
    <h1>We're going to wrap!</h1>
  </div>
  <div class='item'>
    <h1>We're going to wrap!</h1>
  </div>
</div>
```

```
.container {
  display: inline-flex;
  flex-wrap: wrap;
  width: 250px;
}

.item {
  width: 100px;
  height: 100px;
}
```

In the example above, three flex items are contained by a parent flex container. The flex container is only 250 pixels wide so the three 100 pixel wide fle items cannot fit inline. The `flex-wrap: wrap;` setting causes the third, overflowing item to appear on a new line, below the other two items.

align-content:

Sometimes we have multiple rows of flex items within the same flex container and if we want to change the position of these flex items from top to botto of the container, we can use `align-content` to space the rows from top to bottom.

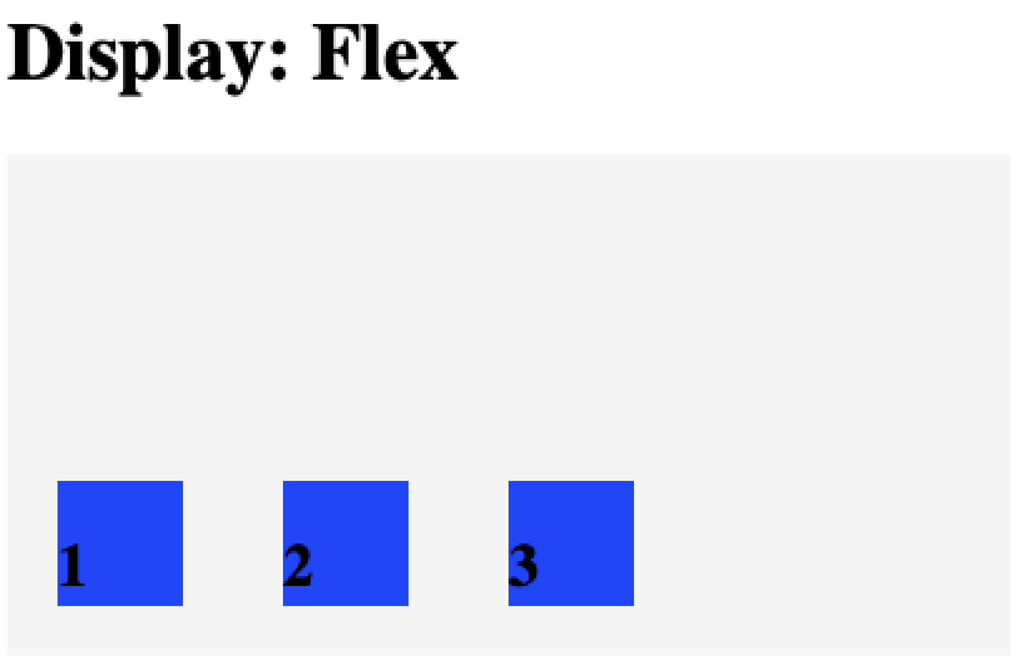
Below are some of the more commonly used `align-content` values:

- `flex-start` — all rows of elements will be positioned at the top of the parent container with no extra space between.
- `flex-end` — all rows of elements will be positioned at the bottom of the parent container with no extra space between.
- `center` — all rows of elements will be positioned at the center of the parent element with no extra space between.
- `space-between` — all rows of elements will be spaced evenly from the top to the bottom of the container with no space above the first or below the last.
- `space-around` — all rows of elements will be spaced evenly from the top to the bottom of the container with the same amount of space at the top and bottom and between each element.
- `stretch` — if a minimum height or no height is specified, the rows of elements will stretch to fill the parent container from top to bottom (default value).

Consider the example:

```
<!DOCTYPE html>
<html>

<head>
  <style>
        .container {
                  background-color: whitesmoke;
                    display: flex;
                    width: 400px;
          height: 200px;
          flex-wrap: wrap;
          align-content: flex-end;
                }
            .box {
          background-color: blue;
          width: 50px;
          height: 50px;
          margin: 20px;
        }
    </style>
</head>

<body>
  <h1>Display: Flex</h1>
  <div class='container' id='flex'>
    <div class='box'>
      <h2>1</h2>
    </div>
    <div class='box'>
      <h2>2</h2>
    </div>
    <div class='box'>
      <h2>3</h2>
    </div>
  </div>
</body>
</html>
```

Output:



In the above example, flex container contains three flex items and these flex items are space at the bottom of the container using align-content proper with the value flex-end.

flex-direction:

Flex containers have two axes: a *main axis* and a *cross axis*. By default, the main axis is horizontal and the cross axis is vertical.

The main axis is used to position flex items with the following properties:

1. `justify-content`
2. `flex-wrap`
3. `flex-grow`
4. `flex-shrink`

The cross axis is used to position flex items with the following properties:

1. `align-items`
2. `align-content`

The main axis and cross axis are interchangeable. We can switch them using the `flex-direction` property. If we add the `flex-direction` proper and give it a value of `column` , the flex items will be ordered vertically, not horizontally.

The `flex-direction` property can accept four values:

1. `row` — elements will be positioned from left to right across the parent element starting from the top left corner (default).
2. `row-reverse` — elements will be positioned from right to left across the parent element starting from the top right corner.
3. `column` — elements will be positioned from top to bottom of the parent element starting from the top left corner.
4. `column-reverse` — elements will be positioned from the bottom to the top of the parent element starting from the bottom left corner.

Example:

```
<div class='container'>
  <div class='item'>
    <h1>1</h1>
  </div>
```

```
    <div class='item'>
      <h1>2</h1>
    </div>
    <div class='item'>
      <h1>3</h1>
    </div>
    <div class='item'>
      <h1>4</h1>
    </div>
    <div class="item">
      <h1>5</h1>
    </div>
  </div>
```

```
.container {
  display: flex;
  flex-direction: column;
  width: 1000px;
}
.item {
  height: 100px;
  width: 100px;
}
```

In the example above, the five divs will be positioned in a vertical column. All of these divs could fit in one horizontal row. However, the `column` valu tells the browser to stack the divs one on top of the other.

So, till now we have seen that we can change the layout as we want with the help of flexbox, now we are going to see that how we can do the same wi Grid:

Layouting with Grid:

So, here we introduce a powerful tool called **css grid** to elegantly layout elements on a web page. The grid can be used to layout the entire web pag Whereas **Flexbox** is mostly useful for positioning items in a one-dimensional layout, CSS grid is most useful for two-dimensional layouts, providing mar tools for aligning and moving elements across both rows and columns.

We are going to see different grid properties shown below as we go along:

- `grid-template-columns`
- `grid-template-rows`
- `grid-template`
- `grid-template-area`
- `row-gap` / `column-gap` / `gap`
- `grid-row-start` / `grid-row-end`
- `grid-column-start` / `grid-column-end`
- `grid-area`

Let’s start with understanding how to create a grid container:

Creating a Grid:

To set up a grid, you need to have both a *grid container* and *grid items*. The grid container will be a parent element that contains grid items as childre and applies overarching styling and positioning to them.

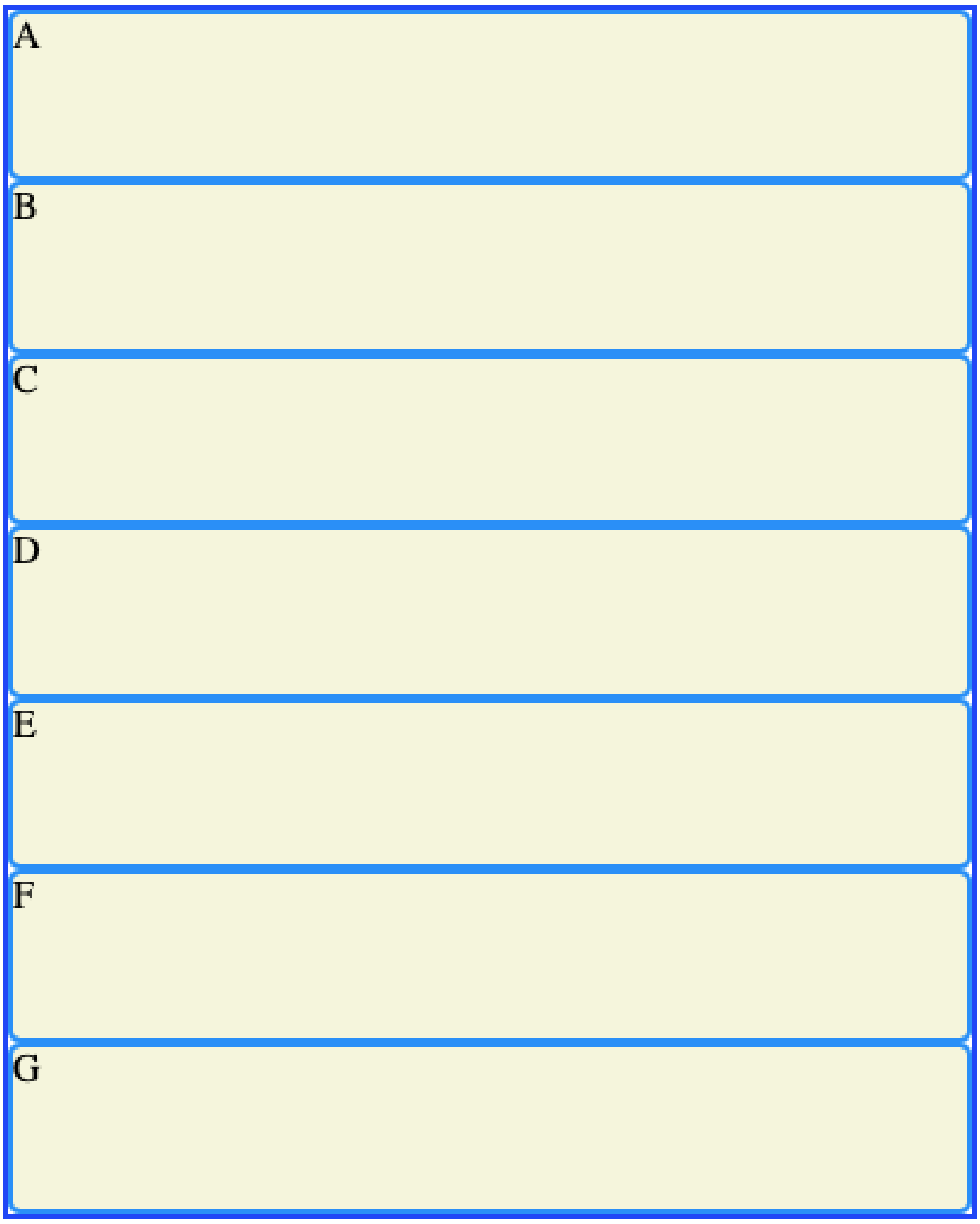
To turn an HTML element into a grid container, you must set the element’s `display` property to one of two values:

- `grid` — for a block-level grid.
- `inline-grid` — for an inline grid.

Consider the example:

```
<!DOCTYPE html>
<html>
<head>
  .grid {
  border: 2px blue solid;
  width: 400px;
  height: 500px;
  display: grid;
}

.box {
  background-color: beige;
  color: black;
  border-radius: 5px;
  border: 2px dodgerblue solid;
}
</head>

<body>
  <div class="grid">
    <div class="box a">A</div>
    <div class="box b">B</div>
    <div class="box c">C</div>
    <div class="box d">D</div>
    <div class="box e">E</div>
    <div class="box f">F</div>
    <div class="box f">G</div>
  </div>
</body>

</html>
```

Output:



Creating a Column:

By default, grids contain only one column. If you were to start adding items, each item would be put on a new row; that's not much of a grid! To chang
this, we need to explicitly define the number of rows and columns in our grid.

We can define the columns of our grid by using the CSS property `grid-template-columns` .

This property creates two changes. First, it defines the number of columns in the grid. Second, it sets the width of each column.

For example:

```
.grid {
  display: grid;
  width: 500px;
  grid-template-columns: 100px 200px;
}
```

In the example above, we have created two columns using grid-template-columns.The first column will be 100 pixels wide and the second column will b
200 pixels wide.

Creating a Row:

To specify the number and size of the rows, we are going to use the property `grid-template-rows` .

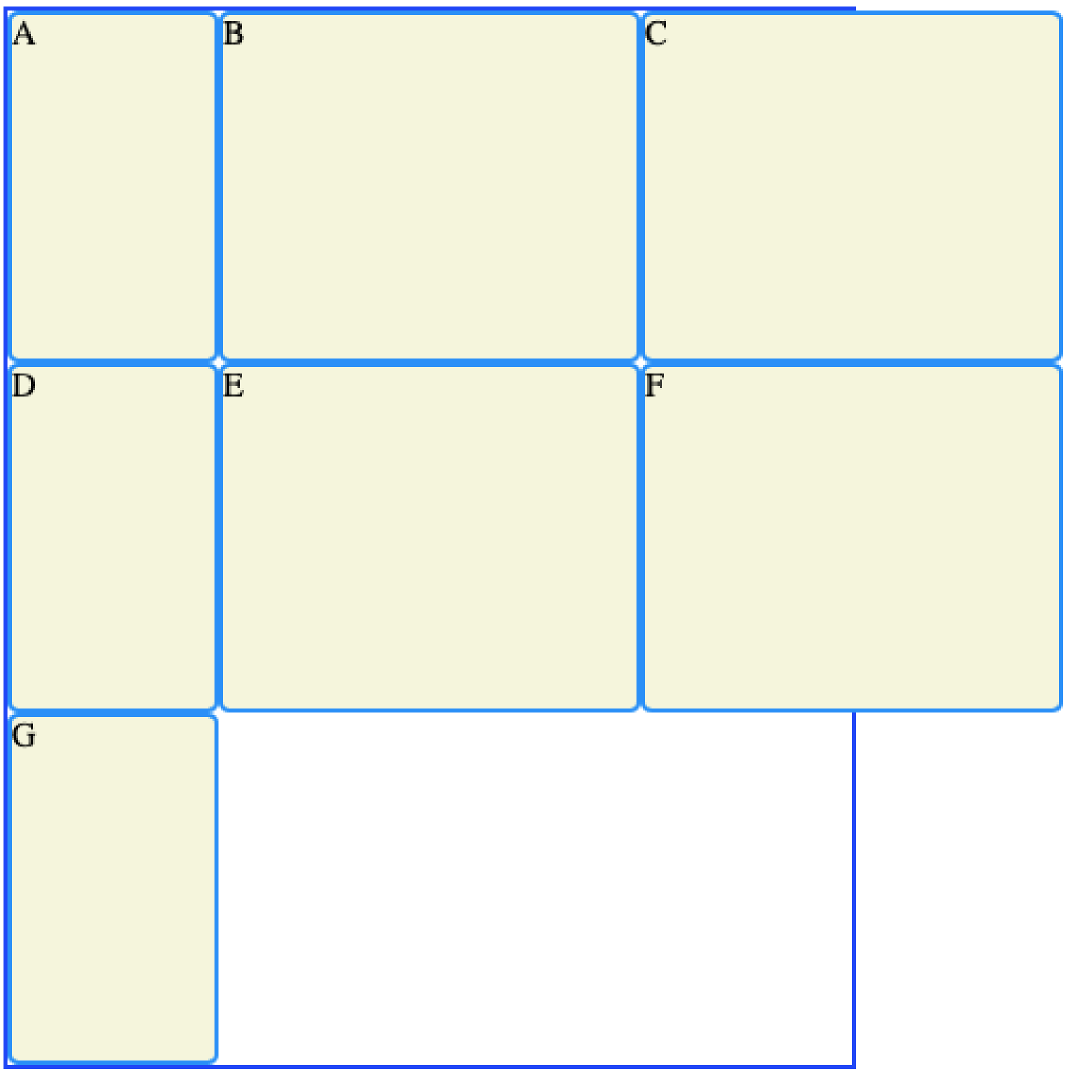
This property is almost identical to `grid-template-columns` .

Note: We can define size in both of these properties in '%' and pixels both.

Consider the example :

```
<!DOCTYPE html>
<html>
<head>
  grid {
  display: grid;
  border: 2px blue solid;
  width: 400px;
  height: 500px;
  grid-template-columns: 100px 50% 200px;
}

.box {
  background-color: beige;
  color: black;
  border-radius: 5px;
  border: 2px dodgerblue solid;
}
</head>

<body>
  <div class="grid">
    <div class="box a">A</div>
    <div class="box b">B</div>
    <div class="box c">C</div>
    <div class="box d">D</div>
    <div class="box e">E</div>
    <div class="box f">F</div>
    <div class="box f">G</div>
  </div>
</body>

</html>
```

Output:



Grid Template:

The shorthand property, `grid-template` , can replace the previous two CSS properties. Both `grid-template-rows` and `grid-template` `columns` .

When using `grid-template` , the values before the slash will determine the size of each row. The values after the slash determine the size of eac column.

Example:

```
.grid {
  display: grid;
  width: 1000px;
  height: 500px;
  grid-template: 200px 300px / 20% 10% 70%;
}
```

In this example, we've made two rows and three columns of varying sizes.

Fraction unit in css Grid:

By using the `fr` unit, we can define the size of columns and rows as a fraction of the grid's length and width. This unit was specifically created for us in CSS Grid. Using `fr` makes it easier to prevent grid items from overflowing the boundaries of the grid.

Consider the example:

```
.grid {
  display: grid;
  width: 1000px;
  height: 400px;
  grid-template: 2fr 1fr 1fr / 1fr 3fr 1fr;
}
```

In this example, the grid will have three rows and three columns. The rows are splitting up the available 400 pixels of height into four parts. The first ro gets two of those parts, the second row gets one, and the third row gets one. Therefore the first row is 200 pixels tall, and the second and third rows a 100 pixels tall.

Repeat property:

The properties that define the number of rows and columns in a grid can take a function as a value. `repeat()` is one of these function The `repeat()` function was created specifically for CSS Grid.

The repeat function will duplicate the specifications for rows or columns a given number of times.

Consider the example:

```
.grid {
  display: grid;
  width: 300px;
  grid-template-columns: repeat(3, 100px);
}
```

In the example above, using the repeat function will make the grid have three columns that are each 100 pixels wide.

Minmax property:

So far, all of the grids that we have worked with have been a fixed size. The grid in our example has been 400 pixels wide and 500 pixels tall. B sometimes you might want a grid to resize based on the size of your web browser.

In these situations, you might want to prevent a row or column from getting too big or too small. For example, if you have a 100-pixel wide image in yo grid, you probably don't want its column to get thinner than 100 pixels! The `minmax()` function can help us solve this problem.

Consider the example:

```
.grid {
  display: grid;
  grid-template-columns: 100px minmax(100px, 500px) 100px;
}
```

In this example, the first and third columns will always be 100 pixels wide, no matter the size of the grid. The second column, however, will vary in size a the overall grid resizes. The second column will always be between 100 and 500 pixels wide.

Grid Gap:

The CSS properties `row-gap` and `column-gap` will put blank space between every row and column in the grid. It is important to note that grid ga properties does not add space at the beginning or end of the grid.``

Consider the example:

```
.grid {
  display: grid;
  width: 320px;
  grid-template-columns: repeat(3, 1fr);
  column-gap: 10px;
}
```

In the example above, our grid will have three columns with two ten-pixel gaps between them.The grid is 320 pixels wide and 20 of those pixels are take up by the two grid gaps. Therefore each column takes a piece of the 300 available pixels. Each column gets `1fr` , so the columns are evenly divide into thirds (or 100 pixels each).

Shorthand for Grid Gap:

There is a shorthand CSS property, `gap` , that can set the row and column gap at the same time.

Consider the example:

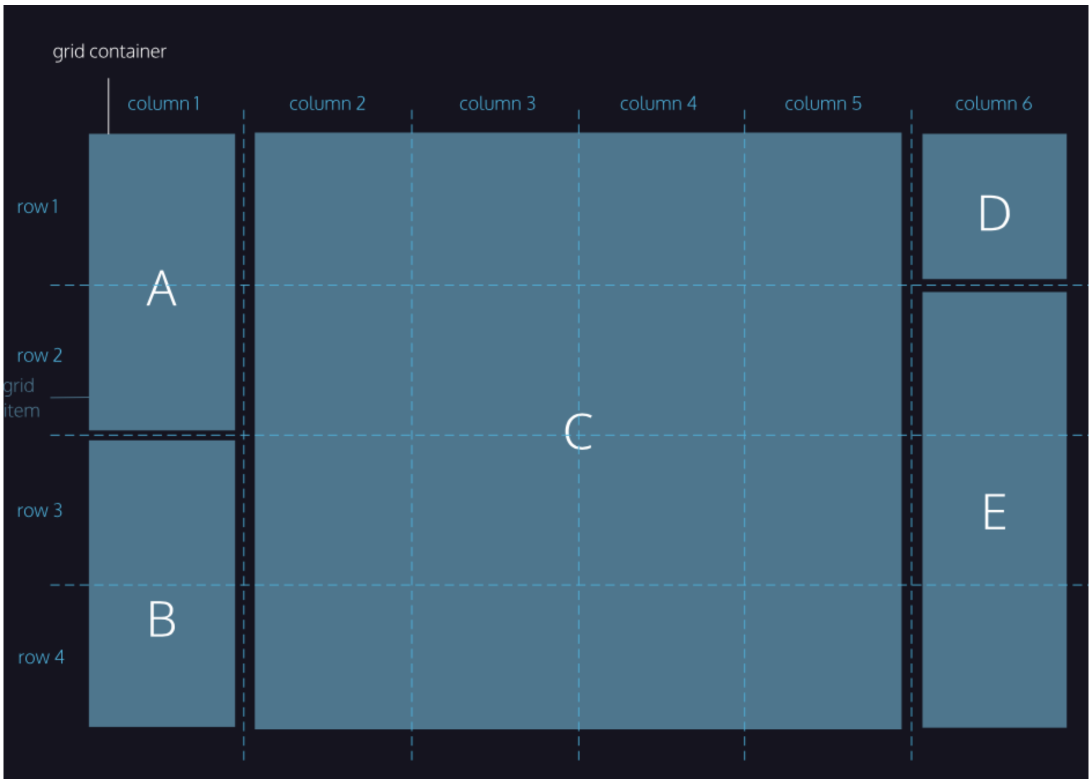
```
.grid {
  display: grid;
  width: 320px;
  grid-template-columns: repeat(3, 1fr);
  gap: 20px 10px;
}
```

Let’s now start understanding the gird items and their properties:

Grid Items:

Now, coming so far we can understand that the items within the grid container are grid items.

But till now we have seen that In all of our examples, the items placed in the grid have always taken up exactly one square. This does not always need be the case; we can drastically change the look of our grid by making grid items take up more than one row and one column. You can see this in th diagram below. Items A, B, C, and E span more than one row!



Let’s now start understanding properties that can be applied on the grid items itself:

Multiple Row Items:

Using the CSS properties `grid-row-start` and `grid-row-end` , we can make single grid items take up multiple rows.

Row grid lines and column grid lines start at 1 and end at a value that is 1 greater than the number of rows or columns the grid has. For example, if a gr has 5 rows, the grid row lines range from 1 to 6.

The value for `grid-row-start` should be the row at which you want the grid item to begin. The value for `grid-row-end` should be one greater tha the row at which you want the grid item to end.

Consider the example:

```
.item {
  grid-row-start: 1;
  grid-row-end: 3;
}
```

Shorthand: grid row

We can use the property `grid-row` as shorthand for `grid-row-start` and `grid-row-end` .

Consider the example:

```
.item {
  grid-row: 4 / 6;
}
```

Grid Column:

`grid-column-start` , `grid-column-end` and `grid-column` work identically to the row properties as first three properties were working for th columns.

Consider the example:

```
.item {
  grid-column: 4 / 6;
}
```

```
.item {
  grid-column-start: 4;
  grid-column-end: 6;
}
```

Note: When using these properties, we can use the keyword `span` to start or end a column or row, relative to its other end. Look at how `span` is use in the code below:

```
.item {
  grid-column: 4 / span 2;
}
```

This is telling the `item` element to begin in column four and take up two columns of space.

Grid Area:

We've already been able to use `grid-row` and `grid-column` as shorthand for properties like `grid-row-start` and `grid-row-end` . We ca refactor even more using the property `grid-area` . This property will set the starting and ending positions for both the rows and columns of an item.

`grid-area` takes four values separated by slashes. The order is important! This is how `grid-area` will interpret those values.

1. `grid-row-start`
2. `grid-column-start`
3. `grid-row-end`
4. `grid-column-end`

Consider the example:

```
.item {
  grid-area: 2 / 3 / 4 / span 5;
}
```

In the above example, the item will occupy rows two and three and columns three through eight.

Using `grid-area` is an easy way to place items exactly where you want them in a grid.

So, till we have learnt all the topics in css and now we can style or layout our web page as per our requirements however we want to.

Let’s now understand how can we debug our css code easily:

Debugging in css:

Sometimes when writing CSS you will encounter an issue where your CSS doesn't seem to be doing what you expect. Perhaps you believe that a certa selector should match an element, but nothing happens, or a box is a different size than you expected.

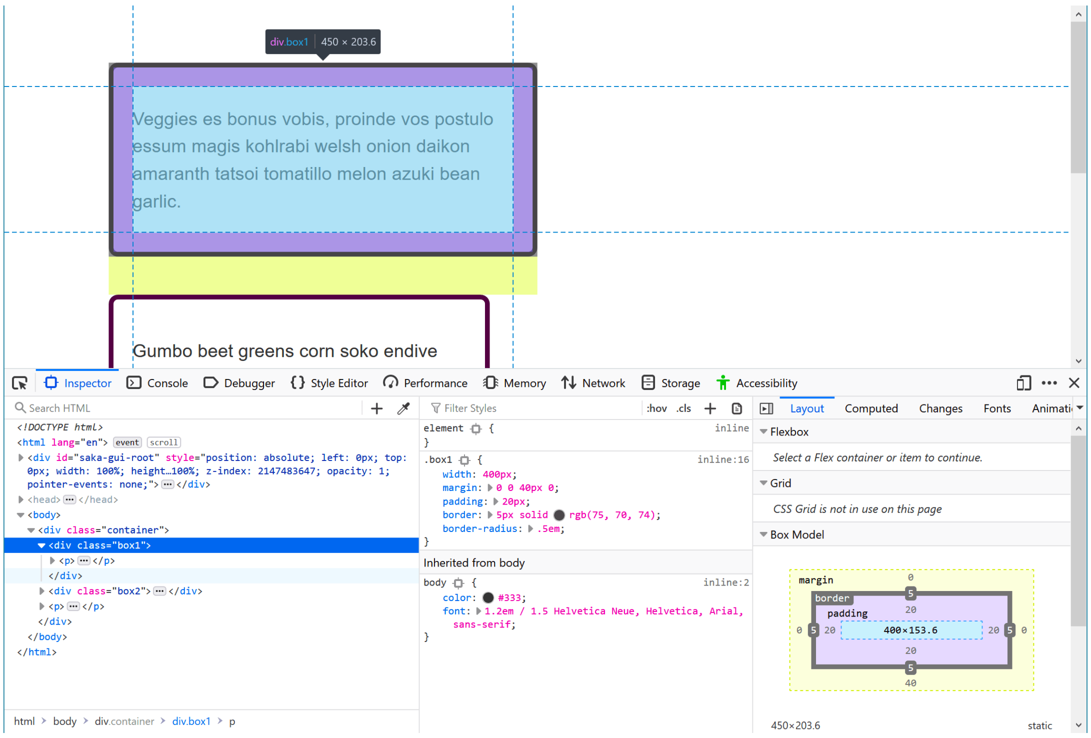
This is when our debugging knowledge helps us a lot to debug our code.

We can access the browser’s DevTools in order to start debugging our code, with the help of these devTools we are going to debug our code.

Inspecting the applied css in browser’s devTools:

Select an element on your page, either by right/ctrl-clicking on it and selecting *Inspect* , or selecting it from the HTML tree on the left of the DevToo display.

In the image below we have selected a element with the class box1.


div.box1 | 450 × 203.6
Veggies es bonus vobis, proinde vos postulo essum magis kohlrabi welsh onion daikon amaranth tatsoi tomatillo melon azuki bean garlic.
Gumbo beet greens corn soko endive
Inspector
Console
Debugger
Style Editor
Performance
Memory
Network
Storage
Accessibility
Search HTML
Filter Styles
:hov
.cls
Layout
Computed
Changes
Fonts
Animati
Flexbox
Select a Flex container or item to continue.
Grid
CSS Grid is not in use on this page
Box Model
margin
border
padding
400×153.6
450×203.6
static
html > body > div.container > div.box1 > p

```
<!DOCTYPE html>
<html lang="en"> event scroll
  <div id="saka-gui-root" style="position: absolute; left: 0px; top:
  0px; width: 100%; height…100%; z-index: 2147483647; opacity: 1;
  pointer-events: none;"> ••• </div>
  <head> ••• </head>
  <body>
    <div class="container">
      <div class="box1">
        <p> ••• </p>
      </div>
      <div class="box2"> ••• </div>
      <p> ••• </p>
    </div>
  </body>
</html>
```

```
element {                                   inline
}
.box1 {                                     inline:16
    width: 400px;
    margin: 0 0 40px 0;
    padding: 20px;
    border: 5px solid ● rgb(75, 70, 74);
    border-radius: .5em;
}
Inherited from body
body {                                      inline:2
    color: ● #333;
    font: 1.2em / 1.5 Helvetica Neue, Helvetica, Arial,
      sans-serif;
}
```

If you look at the Rules Views to the right of your HTML, you should be able to see the CSS properties and values applied to that element. You will se the rules directly applied to class `box1` and also the CSS that is being inherited by the box from its ancestors, in this case from `<body>` . This is usef if you are seeing some CSS being applied that you didn't expect. Perhaps it is being inherited from a parent element and you need to add a rule overwrite it in the context of this element.

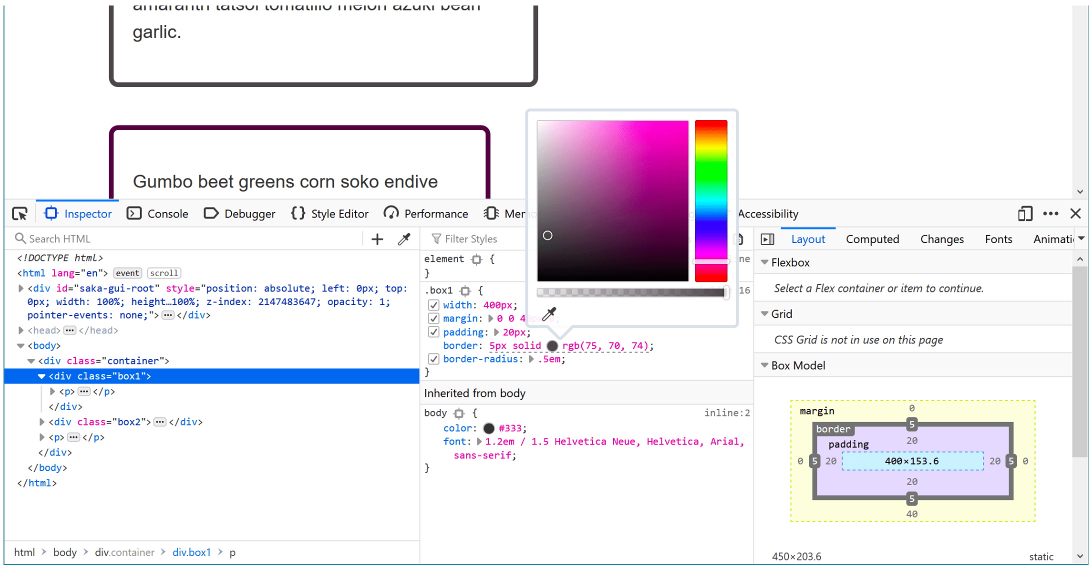
Click on the little arrow to expand the view, showing the different longhand properties and their values.

You can toggle values in the Rules view on and off when that panel is active — if you hold your mouse over it, checkboxes will appea Uncheck a rule's checkbox, for example `border-radius` , and the CSS will stop applying.

Editing the values:

In addition to turning properties on and off, you can edit their values. Perhaps you want to see if another color looks better, or wish to tweak the size something? DevTools can save you a lot of time editing a stylesheet and reloading the page.

With `box1` selected, click on the swatch (the small colored circle) that shows the color applied to the border. A color picker will open up ar you can try out some different colors; these will update in real time on the page. In a similar fashion, you could change the width or style the border.



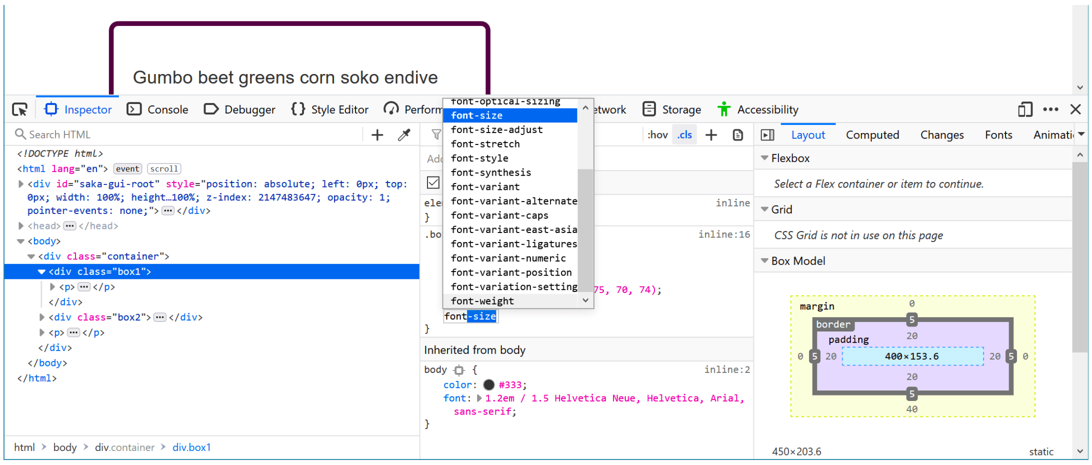
```
<!DOCTYPE html>
<html lang="en"> event scroll
  <div id="saka-gui-root" style="position: absolute; left: 0px; top:
  0px; width: 100%; height…100%; z-index: 2147483647; opacity: 1;
  pointer-events: none;"> … </div>
  <head> … </head>
  <body>
    <div class="container">
      <div class="box1">
        <p> … </p>
      </div>
      <div class="box2"> … </div>
      <p> … </p>
    </div>
  </body>
</html>
```

```
element {
}
.box1 {
  width: 400px;
  margin: 0 0 4
  padding: 20px;
  border: 5px solid rgb(75, 70, 74);
  border-radius: .5em;
}
Inherited from body
body {                                              inline:2
    color: #333;
    font: 1.2em / 1.5 Helvetica Neue, Helvetica, Arial,
      sans-serif;
}
```

Adding a new property:

You can add properties using the DevTools. Perhaps you have realized that you don't want your box to inherit the `<body>` element's font size, and wa to set its own specific size? You can try this out in DevTools before adding it to your CSS file.

You can click the closing curly brace in the rule to start entering a new declaration into it, at which point you can start typing the new proper and DevTools will show you an autocomplete list of matching properties. After selecting `font-size` , enter the value you want to try. You ca also click the + button to add an additional rule with the same selector, and add your new rules there.



```
font-optical-sizing
font-size
font-size-adjust
font-stretch
font-style
font-synthesis
font-variant
font-variant-alternate
font-variant-caps
font-variant-east-asia
font-variant-ligatures
font-variant-numeric
font-variant-position
font-variation-setting
font-weight
```

```
    font-size
}
Inherited from body
body {                                              inline:2
    color: #333;
    font: 1.2em / 1.5 Helvetica Neue, Helvetica, Arial,
      sans-serif;
}
```

By going through all of these steps in your devTools you will be able to understand the abnormal behavior of your code and also be able to find most the bugs in your code.

But even after these are some other problems that can cause a bug in our css code that can be as:

Do we have a valid HTML or CSS syntax ?

Browsers expect your CSS and HTML to be correctly written, however browsers are also very forgiving and will try their best to display your webpage even if you have errors in the markup or stylesheet. If you have mistakes in your code the browser needs to make a guess at what you meant, and might make a different decision to what you had in mind. In addition, two different browsers might cope with the problem in two different ways. A goo first step, therefore, is to run your HTML and CSS through a validator, to pick up and fix any errors.

Do the browser supports the properties and values you are using ?

Browsers ignore CSS they don't understand. If the property or value you are using is not supported by the browser you are testing in then nothing w break, but that CSS won't be applied. DevTools will generally highlight unsupported properties and values in some way.

Is there something else overriding your css ?

This is where the information you have learned about specificity will come into much use. If you have something more specific overriding what you a trying to do, you can enter into a very frustrating game of trying to work out what. However, as described above, DevTools will show you what CSS applying and you can work out how to make the new selector specific enough to override it.

So, till now we have learned HTML and CSS and hence now we are good enough to create any web page we want and style it to the way we want style.

In this and last two css modules we have learned about:

- Basics of css, how it works, how it behaves
- Adding css to our HTML pages
- Selecting elements for css in HTML page
- We have learned about css colors, typography, box modelling, display and positioning in css
- We have learned about the layouts flexbox, grid
- And at last we have seen that how we can debug our css code.

So, now from next module we will be starting to make our page interactive with the help of Javascript.

> Difference between CSS grid vs flexbox?

1. CSS Grid Layout is a two-dimensional system along with rows and columns. It is used for large-sized layouts.
2. Flexbox is a Grid layout with a one-dimensional system either within a row or a column. It is used for the components of an application.

> How does absolute positioning work?

Absolute positioning is used to place the element which is then removed from the HTML document from the normal workflow without creating any spac for the element in the HTML page layout. The element can be positioned respectively to the closest positioned ancestor; otherwise, if the ancestor is n found, the element is placed with respect to the initial container box. The values provided to the top, right, left and bottom determine the final position the element.

> What are the properties of flexbox?

The properties of flexbox are flex-direction, wrap, flow, content, and align-items, and content.

Thank You !

Thank You !