CSS Grid

November 2022

Agenda

1 WHAT IS CSS GRID

2 PROPERTIES FOR THE PARENT

3 PROPERTIES FOR THE CHILDREN

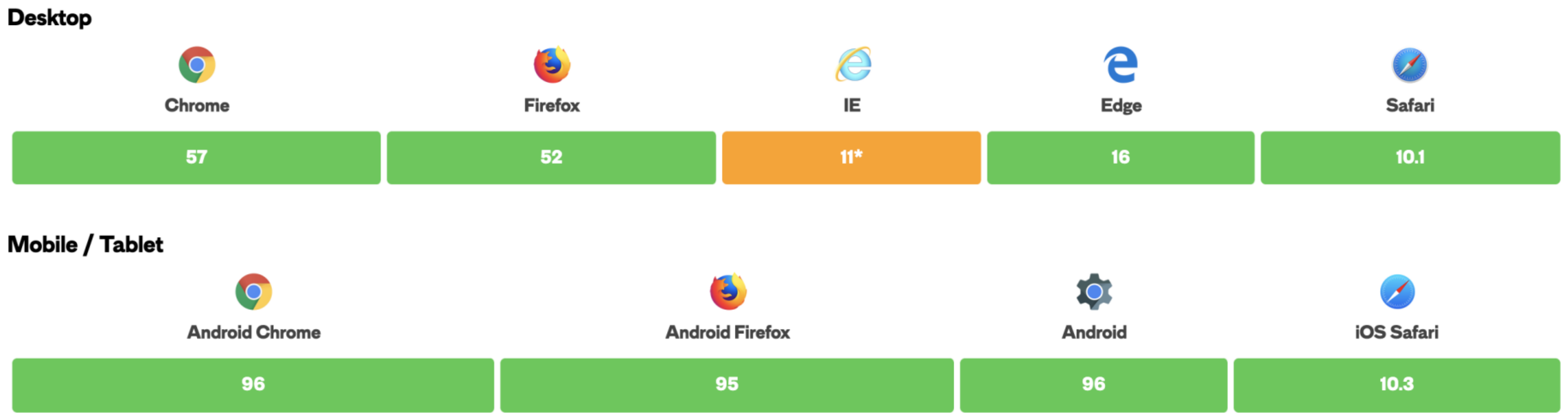
4 SPECIAL UNITS & FUNCTIONS

What is CSS Grid?

CSS Grid Layout (aka “Grid” or “CSS Grid”), is a two-dimensional grid-based layout system that, compared to any web layout system of the past, completely changes the way we design user interfaces. CSS has always been used to layout our web pages, but it’s never done a very good job of it. First, we used tables, then floats, positioning and inline-block, but all of these methods were essentially hacks and left out a lot of important functionality (vertical centering, for instance). Flexbox is also a very great layout tool, but its one-directional flow has different use cases — and they actually work together quite well! Grid is the very first CSS module created specifically to solve the layout problems we’ve all been hacking our way around for as long as we’ve been making websites.

WHAT IS CSS GRID?

Browser support



Desktop

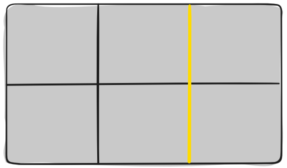
Chrome	Firefox	IE	Edge	Safari
57	52	11*	16	10.1

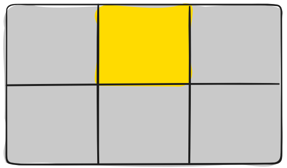
Mobile / Tablet

Android Chrome	Android Firefox	Android	iOS Safari
96	95	96	10.3

Important Terminology

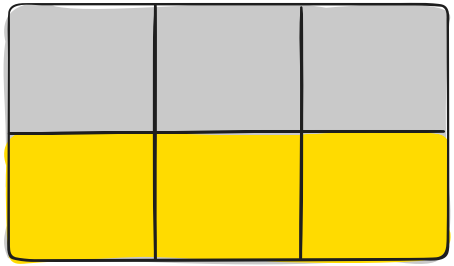
- Grid Container
  - The element on which display: grid is applied. It’s the direct parent of all the grid items. In this example container is the grid container.
- Grid Item
  - The children (i.e. *direct* descendants) of the grid container. Here the item elements are grid items, but sub-item isn’t.
- Grid Line
  - The dividing lines that make up the structure of the grid. They can be either vertical (“column grid lines”) or horizontal (“row grid lines”) and reside on either side of a row or column.
- Grid Cell
  - The space between two adjacent row and two adjacent column grid lines. It’s a single “unit” of the grid. Here’s the grid cell between row grid lines 1 and 2, and column grid lines 2 and 3.



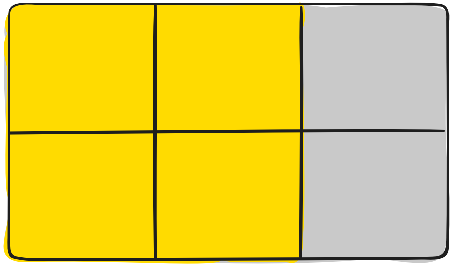


Important Terminology

- Grid Track
    - The space between two adjacent grid lines. You can think of them as the columns or rows of the grid. Here’s the grid track between the second and third-row grid lines.



- Grid Area
    - The total space surrounded by four grid lines. A grid area may be composed of any number of grid cells. Here’s the grid area between row grid lines 1 and 3, and column grid lines 1 and 3.



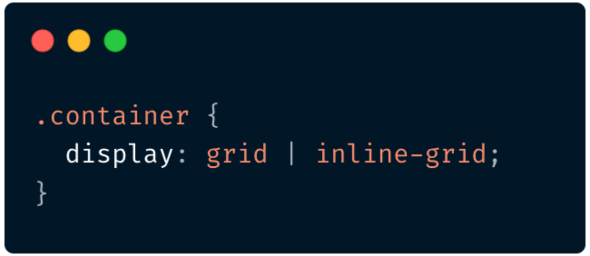
PROPERTIES FOR THE PARENT (GRID CONTAINER)

display

Defines the element as a grid container and establishes a new grid formatting context for its contents.

Values:

- **grid** – generates a block-level grid
- **inline-grid** – generates an inline-level grid



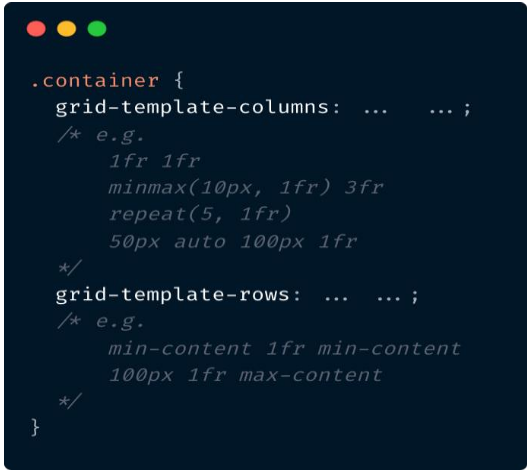
```css
.container {
  display: grid | inline-grid;
}
```

grid-template-columns, grid-template-rows

Defines the columns and rows of the grid with a space-separated list of values. The values represent the track size, and the space between them represents the grid line.

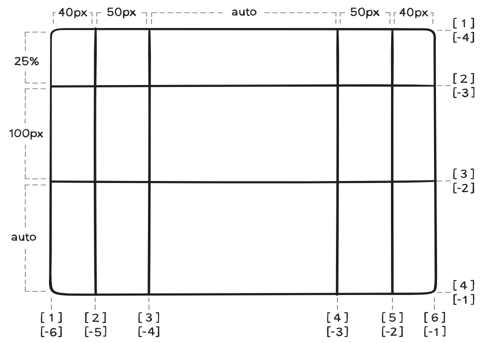
Values:

- **<track-size>** – can be a length, a percentage, or a fraction of the free space in the grid (using the fr unit)
- **<line-name>** – an arbitrary name of your choosing



```css
.container {
  grid-template-columns: ...   ... ;
  /* e.g.
      1fr 1fr
      minmax(10px, 1fr) 3fr
      repeat(5, 1fr)
      50px auto 100px 1fr
  */
  grid-template-rows: ...  ... ;
  /* e.g.
      min-content 1fr min-content
      100px 1fr max-content
  */
}
```

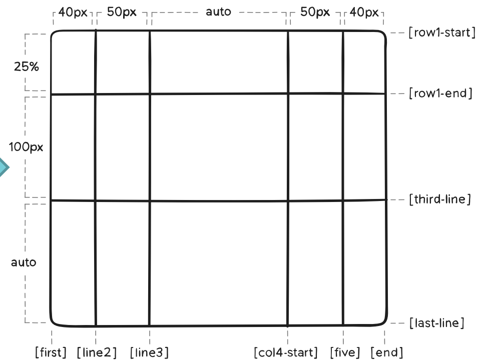
Grid lines are automatically assigned positive numbers from these assignments (-1 being an alternate for the very last row).



grid-template-columns, grid-template-rows

You can choose to explicitly name the lines. Note the bracket syntax for the line names:

```css
.container {
  grid-template-columns: [first] 40px [line2] 50px [line3] auto [col4-start] 50px [five] 40px [end];
  grid-template-rows: [row1-start] 25% [row1-end] 100px [third-line] auto [last-line];
}
```



If your definition contains repeating parts, you can use the repeat() notation to streamline things:

```css
.container {
  grid-template-columns: repeat(3, 20px [col-start]);
}
```

The fr unit allows you to set the size of a track as a fraction of the free space of the grid container. For example, this will set each item to one third the width of the grid container:

```css
.container {
  grid-template-columns: 1fr 1fr 1fr;
}
```

grid-template-areas

Defines a grid template by referencing the names of the grid areas which are specified with the grid-area property. Repeating the name of a grid area causes the content to span those cells. A period signifies an empty cell. The syntax itself provides a visualization of the structure of the grid.

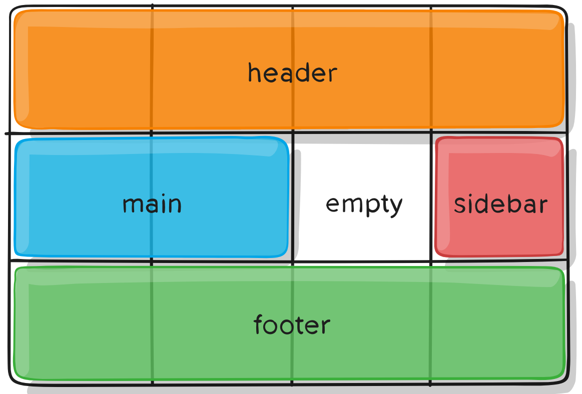
```
.container {
  grid-template-areas:
    "<grid-area-name> | . | none | ... "
    " ... ";
}
```

Values:

- **<grid-area-name>** – the name of a grid area specified with grid-area
- **.** – a period signifies an empty grid cell
- **none** – no grid areas are defined

Example:

```
.container {
  display: grid;
  grid-template-columns: 50px 50px 50px 50px;
  grid-template-rows: auto;
  grid-template-areas:
    "header header header header"
    "main main . sidebar"
    "footer footer footer footer";
}
```

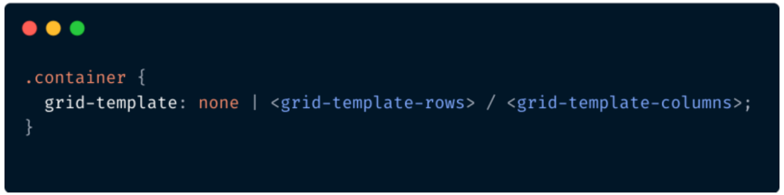


grid-template

A shorthand for setting grid-template-rows, grid-template-columns, and grid-template-areas in a single declaration.

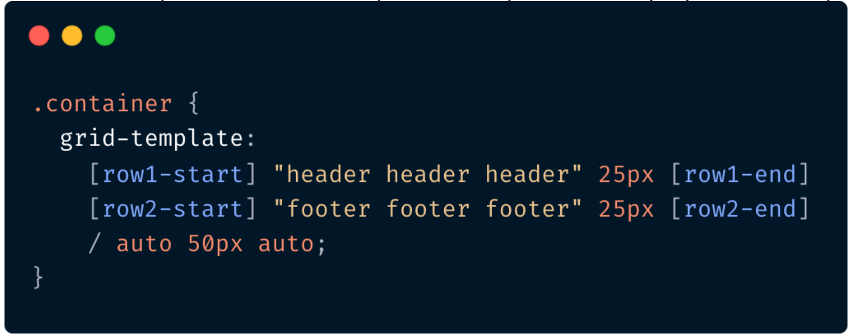
Values:

- **none** – sets all three properties to their initial values
- **<grid-template-rows> / <grid-template-columns>** – sets grid-template-columns and grid-template-rows to the specified values, respectively, and sets grid-template-areas to none



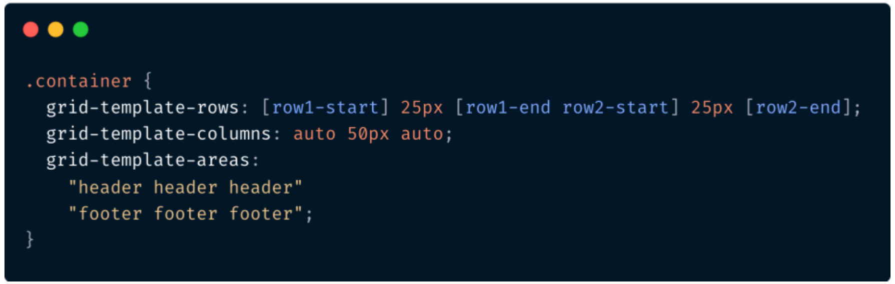
```css
.container {
  grid-template: none | <grid-template-rows> / <grid-template-columns>;
}
```

It also accepts a more complex but quite handy syntax for specifying all three. Here's an example:



```css
.container {
  grid-template:
    [row1-start] "header header header" 25px [row1-end]
    [row2-start] "footer footer footer" 25px [row2-end]
    / auto 50px auto;
}
```

=



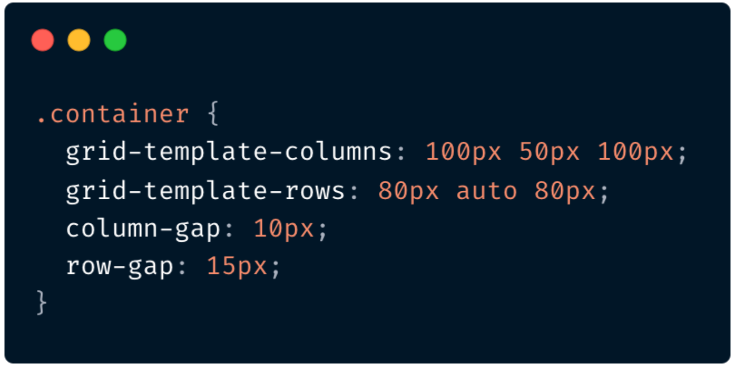
```css
.container {
  grid-template-rows: [row1-start] 25px [row1-end row2-start] 25px [row2-end];
  grid-template-columns: auto 50px auto;
  grid-template-areas:
    "header header header"
    "footer footer footer";
}
```

column-gap, row-gap, grid-column-gap, grid-row-gap

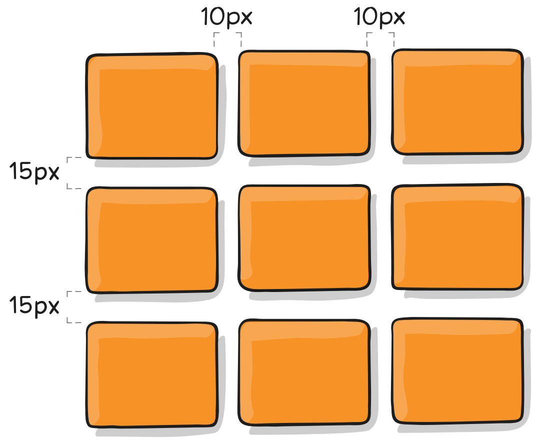
Specifies the size of the grid lines. You can think of it like setting the width of the gutters between the columns/rows.

Values:

- **<line-size>** – a length value



```
.container {
  grid-template-columns: 100px 50px 100px;
  grid-template-rows: 80px auto 80px;
  column-gap: 10px;
  row-gap: 15px;
}
```

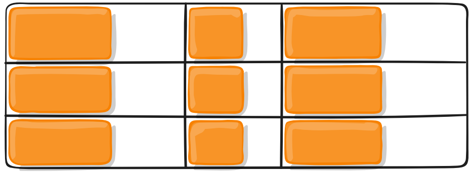


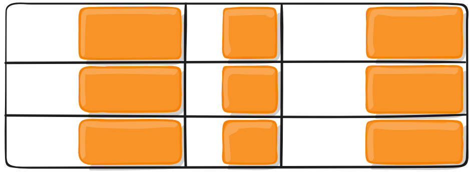
justify-items

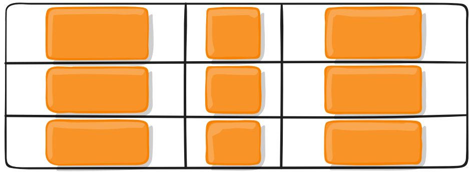
Aligns grid items along the *inline (row)* axis (as opposed to align-items which aligns along the *block (column)* axis). This value applies to all grid items inside the container.

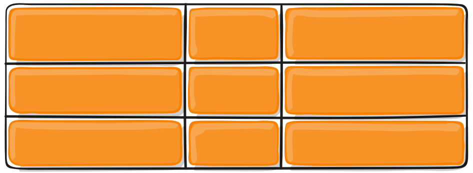
Values:

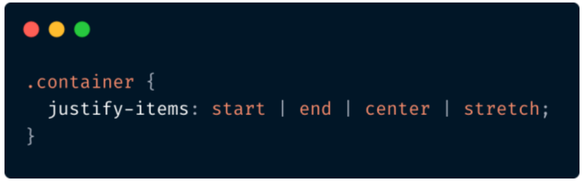
- **start** – aligns items to be flush with the start edge of their cell
- **end** – aligns items to be flush with the end edge of their cell
- **center** – aligns items in the center of their cell
- **stretch** – fills the whole width of the cell (this is the default)











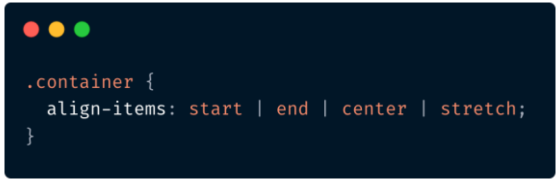
```
.container {
  justify-items: start | end | center | stretch;
}
```

align-items

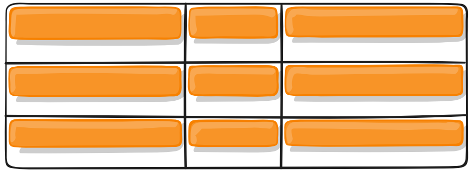
Aligns grid items along the *block (column)* axis (as opposed to justify-items which aligns along the *inline (row)* axis). This value applies to all grid items inside the container.

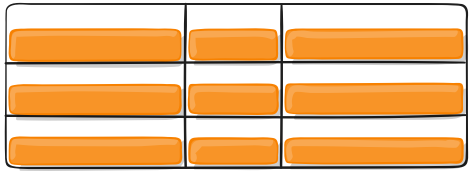
Values:

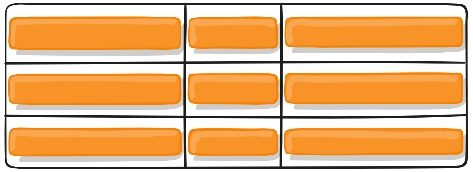
- **stretch** – fills the whole height of the cell (this is the default)
- **start** – aligns items to be flush with the start edge of their cell
- **end** – aligns items to be flush with the end edge of their cell
- **center** – aligns items in the center of their cell
- **baseline** – align items along text baseline. There are modifiers to baseline — first baseline and last baseline which will use the baseline from the first or last line in the case of multi-line text.

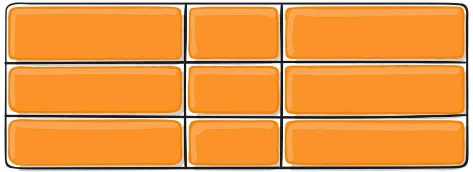


```
.container {
  align-items: start | end | center | stretch;
}
```







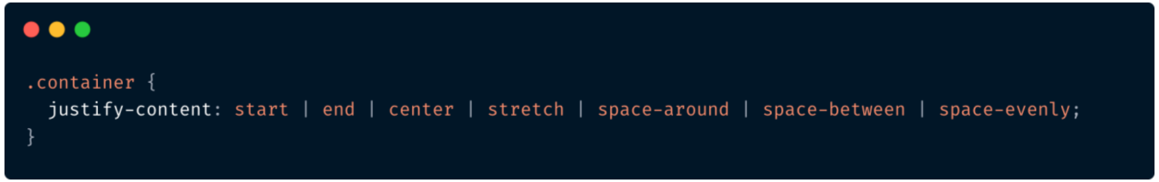


justify-content

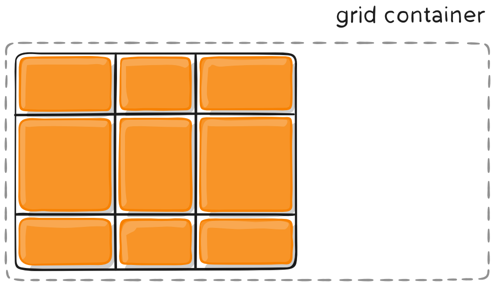
This property aligns the grid along the *inline (row)* axis (as opposed to align-content which aligns the grid along the *block (column)* axis).

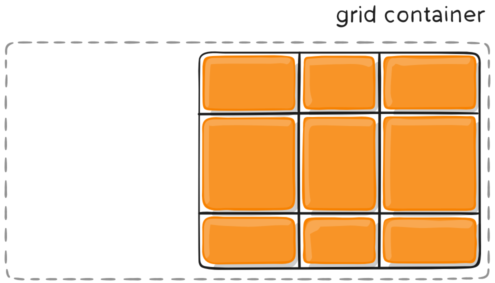
Values:

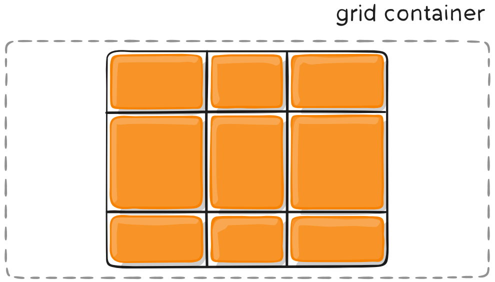
- start – aligns the grid to be flush with the start edge of the grid container
- end – aligns the grid to be flush with the end edge of the grid container
- center – aligns the grid in the center of the grid container
- stretch – resizes the grid items to allow the grid to fill the full width of the grid container
- space-around – places an even amount of space between each grid item, with half-sized spaces on the far ends
- space-between – places an even amount of space between each grid item, with no space at the far ends
- space-evenly – places an even amount of space between each grid item, including the far ends



```
.container {
  justify-content: start | end | center | stretch | space-around | space-between | space-evenly;
}
```





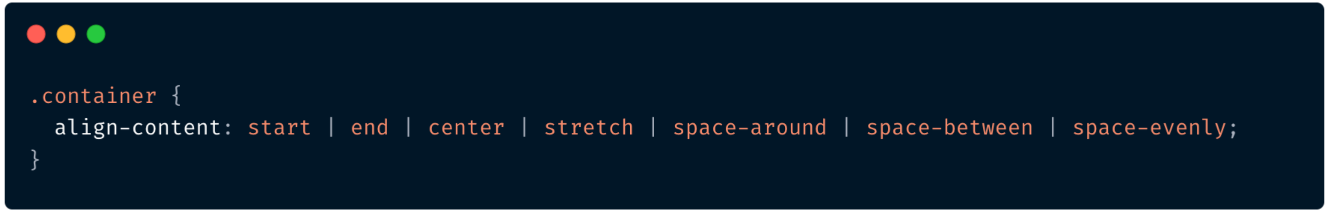


align-content

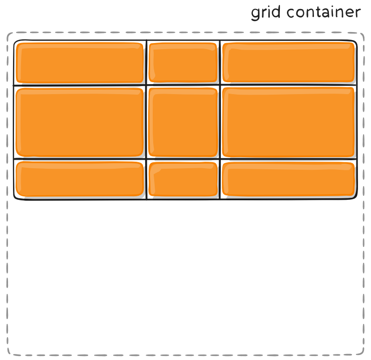
This property aligns the grid along the *block (column)* axis (as opposed to justify-content which aligns the grid along the *inline (row)* axis).

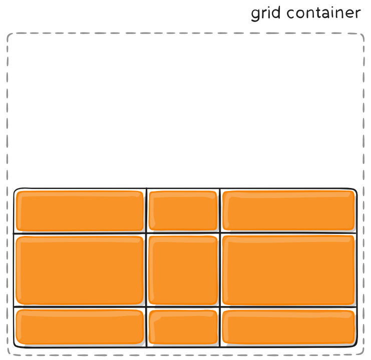
Values:

- **start** – aligns the grid to be flush with the start edge of the grid container
- **end** – aligns the grid to be flush with the end edge of the grid container
- **center** – aligns the grid in the center of the grid container
- **stretch** – resizes the grid items to allow the grid to fill the full height of the grid container
- **space-around** – places an even amount of space between each grid item, with half-sized spaces on the far ends
- **space-between** – places an even amount of space between each grid item, with no space at the far ends
- **space-evenly** – places an even amount of space between each grid item, including the far ends



```
.container {
  align-content: start | end | center | stretch | space-around | space-between | space-evenly;
}
```





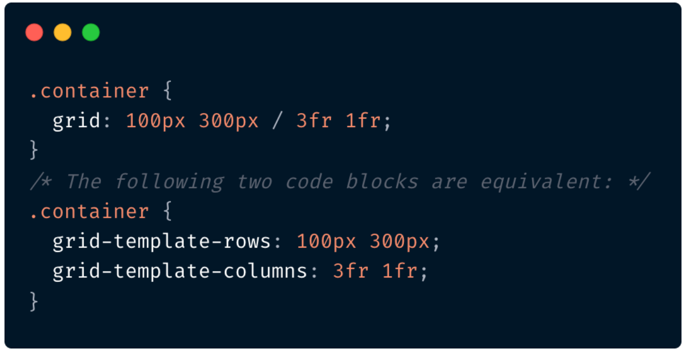
grid

A shorthand for setting all of the following properties in a single declaration: grid-template-rows, grid-template-columns, grid-template-areas, grid-auto-rows, grid-auto-columns, and grid-auto-flow (Note: You can only specify the explicit or the implicit grid properties in a single grid declaration).

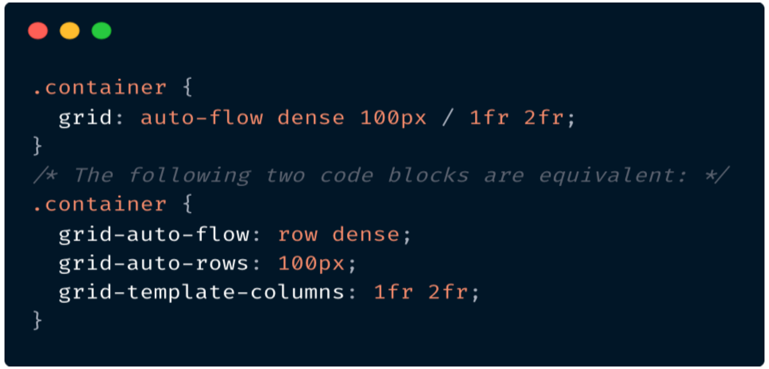
Values:

- **none** – sets all sub-properties to their initial values.
- **<grid-template>** – works the same as the grid-template shorthand.
- **<grid-template-rows> / [auto-flow && dense?] <grid-auto-columns>?** – sets grid-template-rows to the specified value. If the auto-flow keyword is to the right of the slash, it sets grid-auto-flow to column. If the dense keyword is specified additionally, the auto-placement algorithm uses a “dense” packing algorithm. If grid-auto-columns is omitted, it is set to auto.
- **[auto-flow && dense?] <grid-auto-rows>? / <grid-template-columns>** – sets grid-template-columns to the specified value. If the auto-flow keyword is to the left of the slash, it sets grid-auto-flow to row. If the dense keyword is specified additionally, the auto-placement algorithm uses a “dense” packing algorithm. If grid-auto-rows is omitted, it is set to auto.

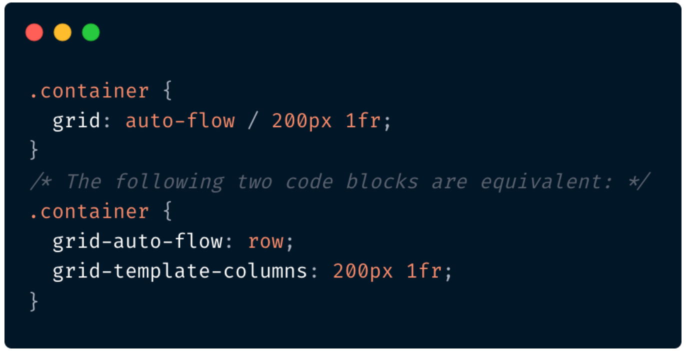
grid property expamples



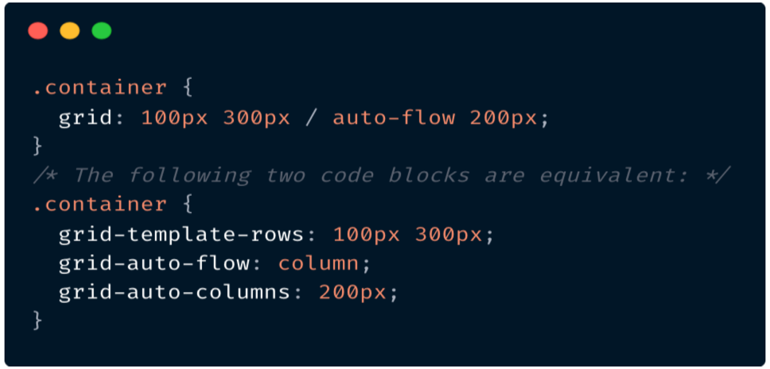
```css
.container {
  grid: 100px 300px / 3fr 1fr;
}
/* The following two code blocks are equivalent: */
.container {
  grid-template-rows: 100px 300px;
  grid-template-columns: 3fr 1fr;
}
```



```css
.container {
  grid: auto-flow dense 100px / 1fr 2fr;
}
/* The following two code blocks are equivalent: */
.container {
  grid-auto-flow: row dense;
  grid-auto-rows: 100px;
  grid-template-columns: 1fr 2fr;
}
```



```css
.container {
  grid: auto-flow / 200px 1fr;
}
/* The following two code blocks are equivalent: */
.container {
  grid-auto-flow: row;
  grid-template-columns: 200px 1fr;
}
```



```css
.container {
  grid: 100px 300px / auto-flow 200px;
}
/* The following two code blocks are equivalent: */
.container {
  grid-template-rows: 100px 300px;
  grid-auto-flow: column;
  grid-auto-columns: 200px;
}
```

PROPERTIES FOR THE CHILDREN (GRID ITEMS)

grid-column-start, grid-column-end, grid-row-start, grid-row-end

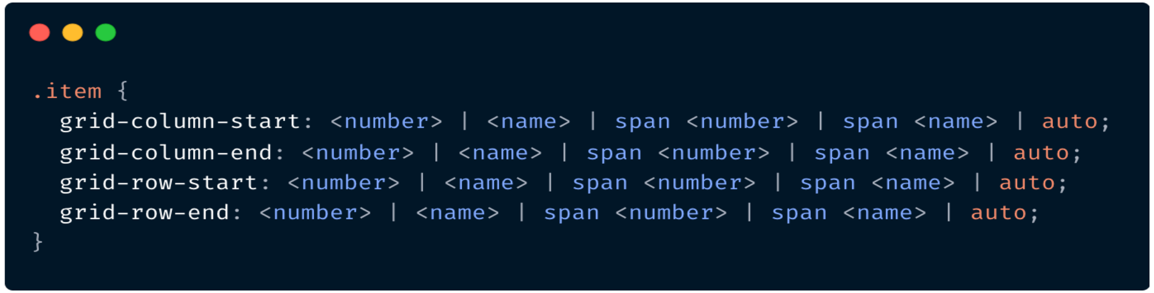
Determines a grid item’s location within the grid by referring to specific grid lines. grid-column-start/grid-row-start is the line where the item begins, and grid-column-end/grid-row-end is the line where the item ends.

Values:

- **<line>** – can be a number to refer to a numbered grid line, or a name to refer to a named grid line
- **span <number>** – the item will span across the provided number of grid tracks
- **span <name>** – the item will span across until it hits the next line with the provided name
- **auto** – indicates auto-placement, an automatic span, or a default span of one

If no grid-column-end/grid-row-end is declared, the item will span 1 track by default.

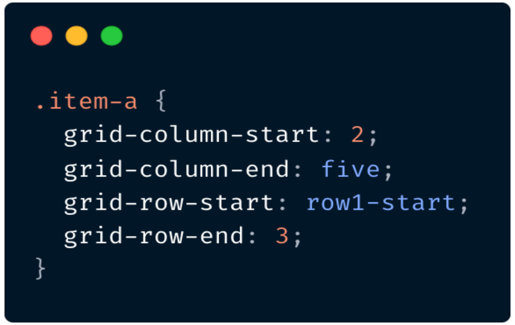
Items can overlap each other. You can use z-index to control their stacking order.



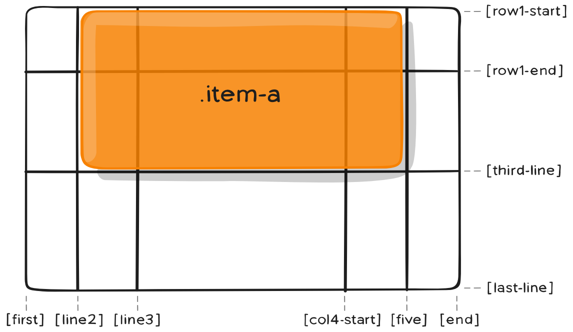
```css
.item {
  grid-column-start: <number> | <name> | span <number> | span <name> | auto;
  grid-column-end: <number> | <name> | span <number> | span <name> | auto;
  grid-row-start: <number> | <name> | span <number> | span <name> | auto;
  grid-row-end: <number> | <name> | span <number> | span <name> | auto;
}
```

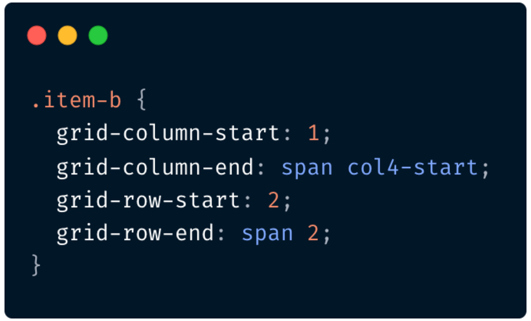
grid-column-start, grid-column-end, grid-row-start, grid-row-end

Examples:

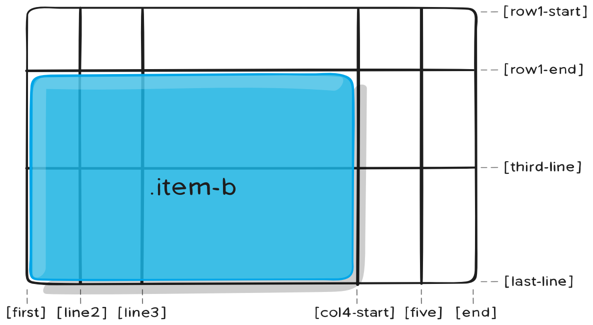


```css
.item-a {
  grid-column-start: 2;
  grid-column-end: five;
  grid-row-start: row1-start;
  grid-row-end: 3;
}
```





```css
.item-b {
  grid-column-start: 1;
  grid-column-end: span col4-start;
  grid-row-start: 2;
  grid-row-end: span 2;
}
```



grid-column, grid-row

Shorthand for grid-column-start + grid-column-end, and grid-row-start + grid-row-end, respectively.

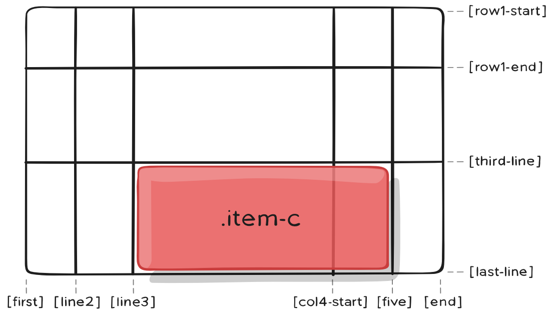
Values:

- **<start-line> / <end-line>** – each one accepts all the same values as the longhand version, including span

```
.item {
  grid-column: <start-line> / <end-line> | <start-line> / span <value>;
  grid-row: <start-line> / <end-line> | <start-line> / span <value>;
}
```

Example:

```
.item-c {
  grid-column: 3 / span 2;
  grid-row: third-line / 4;
}
```



grid-area

Gives an item a name so that it can be referenced by a template created with the grid-template-areas property. Alternatively, this property can be used as an even shorter shorthand for grid-row-start + grid-column-start + grid-row-end + grid-column-end.

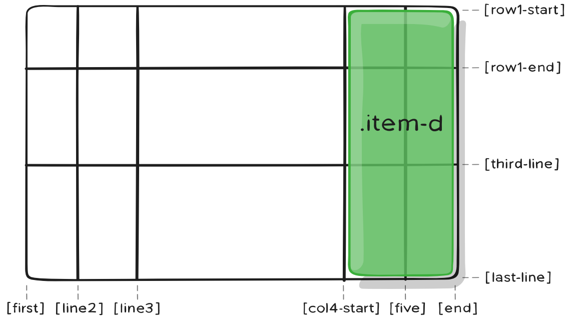
Values:

- **<name>** – a name of your choosing
- **<row-start> / <column-start> / <row-end> / <column-end>** – can be numbers or named lines

```
.item {
  grid-area: <name> | <row-start> / <column-start> / <row-end> / <column-end>;
}
```

Example:

```
.item-d {
  grid-area: 1 / col4-start / last-line / 6;
}
```



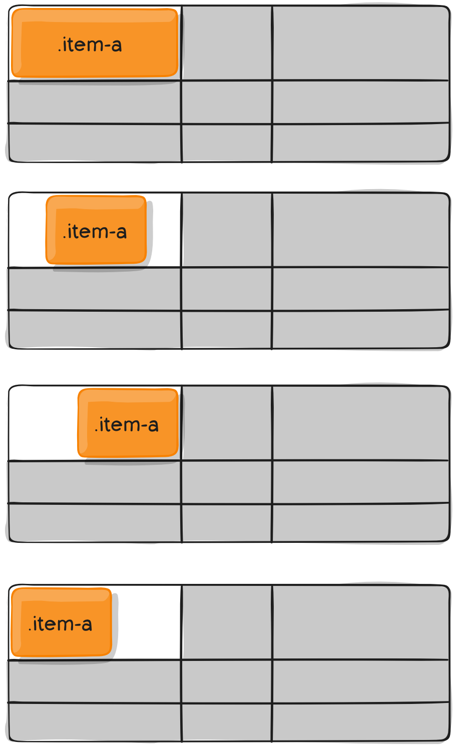
justify-self

Aligns a grid item inside a cell along the *inline (row)* axis (as opposed to align-self which aligns along the *block (column)* axis). This value applies to a grid item inside a single cell.

Values:

- **start** – aligns the grid item to be flush with the start edge of the cell
- **end** – aligns the grid item to be flush with the end edge of the cell
- **center** – aligns the grid item in the center of the cell
- **stretch** – fills the whole width of the cell (this is the default)

```
.item {
  justify-self: start | end | center | stretch;
}
```

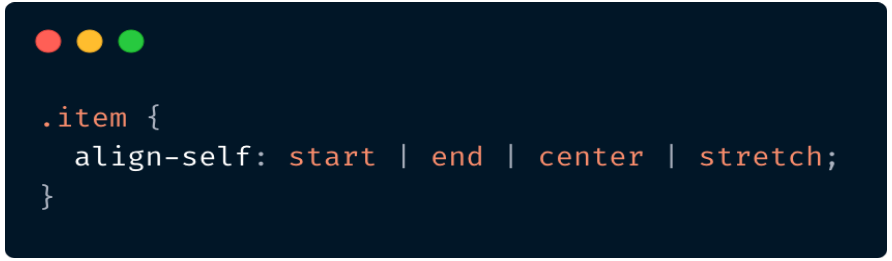


align-self

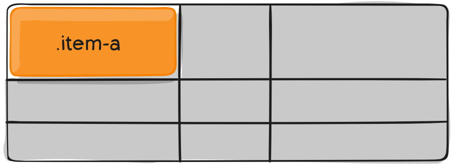
Aligns a grid item inside a cell along the *block (column)* axis (as opposed to justify-self which aligns along the *inline (row)* axis). This value applies to the content inside a single grid item.

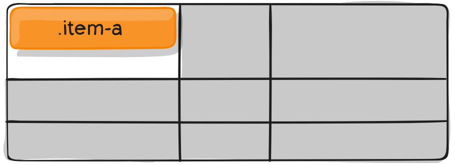
Values:

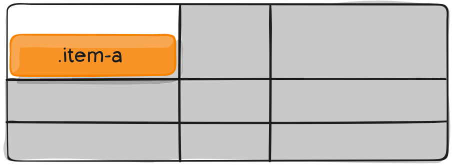
- **start** – aligns the grid item to be flush with the start edge of the cell
- **end** – aligns the grid item to be flush with the end edge of the cell
- **center** – aligns the grid item in the center of the cell
- **stretch** – fills the whole height of the cell (this is the default)

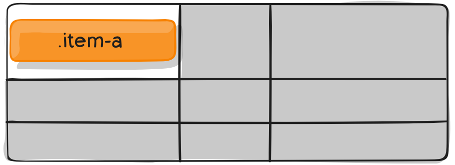


```
.item {
  align-self: start | end | center | stretch;
}
```







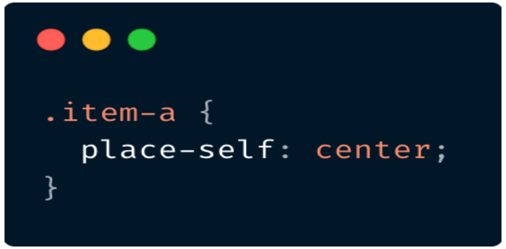


place-self

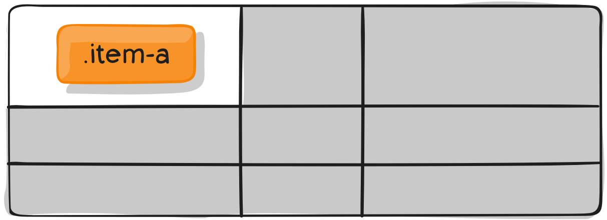
place-self sets both the align-self and justify-self properties in a single declaration.

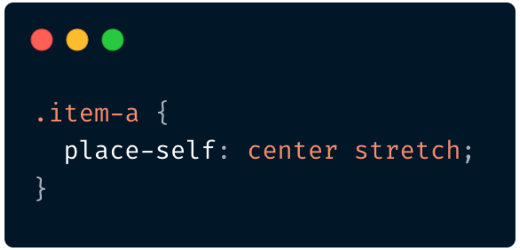
Values:

- **auto** – The “default” alignment for the layout mode.
- **<align-self> / <justify-self>** – The first value sets align-self, the second value justify-self. If the second value is omitted, the first value is assigned to both properties.

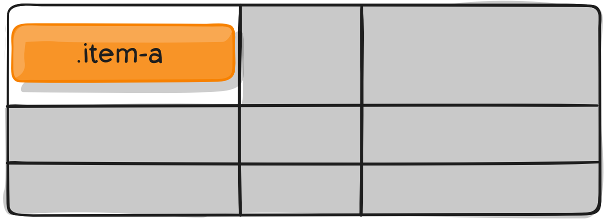


```
.item-a {
  place-self: center;
}
```





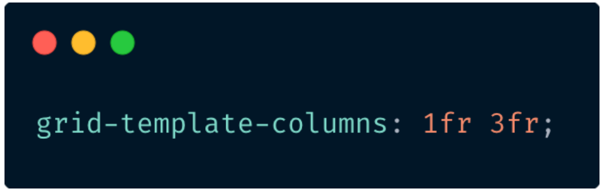
```
.item-a {
  place-self: center stretch;
}
```



SPECIAL UNITS & FUNCTIONS

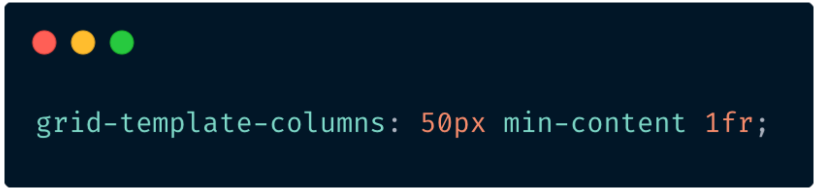
fr units

You'll likely end up using a lot of fractional units in CSS Grid, like 1fr. They essentially mean "portion of the remaining space". So a declaration like:



```
grid-template-columns: 1fr 3fr;
```

Means, loosely, 25% 75%. Except that those percentage values are much more firm than fractional units are. For example, if you added padding to those percentage-based columns, now you've broken 100% width (assuming a content-box box model). Fractional units also much more friendly in combination with other units, as you can imagine:



```
grid-template-columns: 50px min-content 1fr;
```

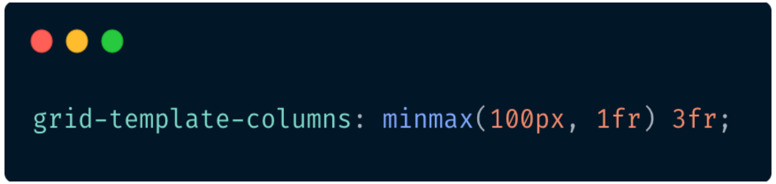
Sizing Keywords

When sizing rows and columns, you can use all the lengths you are used to, like px, rem, %, etc, but you also have keywords:

- min-content: the minimum size of the content. Imagine a line of text like “The very long hotdog.”, the min-content is likely the width of the world “The”.
- max-content: the maximum size of the content. Imagine the sentence above, the max-content is the length of the whole sentence.
- auto: this keyword is a lot like fr units, except that they “lose” the fight in sizing against fr units when allocating the remaining space.
- fit-content: use the space available, but never less than min-content and never more than max-content.
- fractional units: see above

Sizing Functions

- The minmax() function does exactly what it seems like: it sets a minimum and maximum value for what the length is able to be. This is useful for in combination with relative units. Like you may want a column to be only able to shrink so far. This is extremely useful and probably what you want:

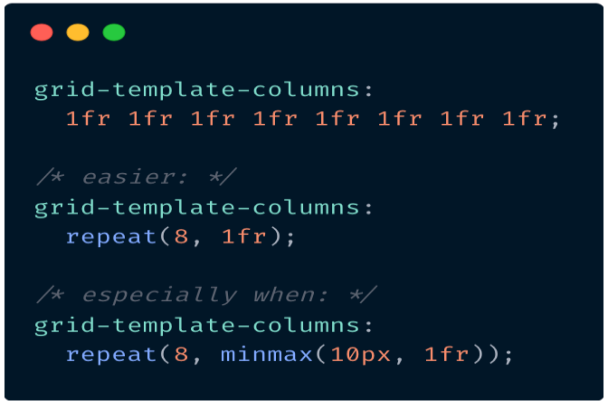


```
grid-template-columns: minmax(100px, 1fr) 3fr;
```

- The min() function.
- The max() function.

The repeat() Function and Keywords

The repeat() function can save some typing:

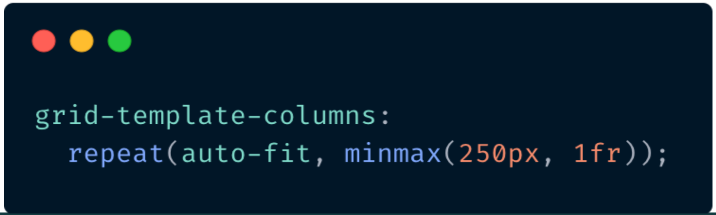


```
grid-template-columns:
  1fr 1fr 1fr 1fr 1fr 1fr 1fr 1fr;

/* easier: */
grid-template-columns:
  repeat(8, 1fr);

/* especially when: */
grid-template-columns:
  repeat(8, minmax(10px, 1fr));
```

But repeat() can get extra fancy when combined with keywords:

- auto-fill: Fit as many possible columns as possible on a row, even if they are empty.
- auto-fit: Fit whatever columns there are into the space. Prefer expanding columns to fill space rather than empty columns.

This bears the most famous snippet in all of CSS Grid:



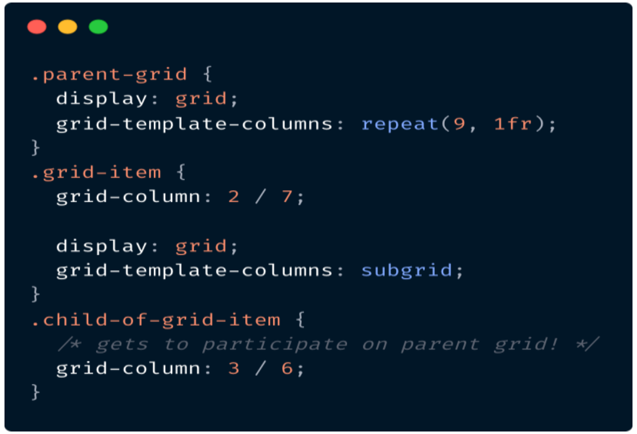
```
grid-template-columns:
  repeat(auto-fit, minmax(250px, 1fr));
```

Subgrid

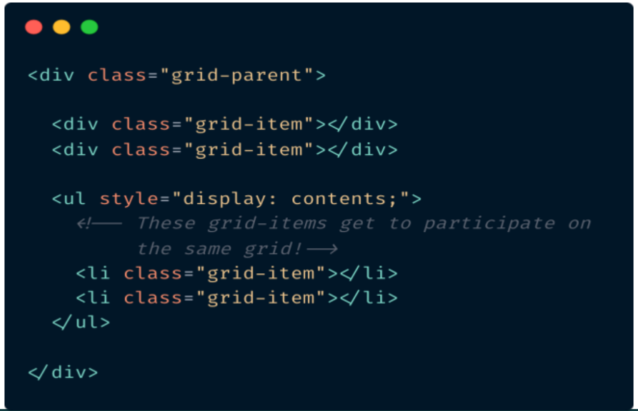
Subgrid is an extremely useful feature of grids that allows grid items to have a grid of their own that inherits grid lines from the parent grid.

This is only supported in Firefox right now, but it really needs to get everywhere.

It's also useful to know about display: contents;. This is *not* the same as subgrid, but it can be a useful tool sometimes in a similar fashion.



```
.parent-grid {
  display: grid;
  grid-template-columns: repeat(9, 1fr);
}
.grid-item {
  grid-column: 2 / 7;

  display: grid;
  grid-template-columns: subgrid;
}
.child-of-grid-item {
  /* gets to participate on parent grid! */
  grid-column: 3 / 6;
}
```



```
<div class="grid-parent">

  <div class="grid-item"></div>
  <div class="grid-item"></div>

  <ul style="display: contents;">
    <!-- These grid-items get to participate on
         the same grid!-->
    <li class="grid-item"></li>
    <li class="grid-item"></li>
  </ul>

</div>
```

Flexbox vs Grid

One Vs Two Dimension:

- Grid is made for two-dimensional layout while Flexbox is for one. This means Flexbox can work on either row or columns at a time, but Grids can work on both.
- Flexbox, gives you more flexibility while working on either element (row or column). HTML markup and CSS will be easy to manage in this type of scenario.
- GRID gives you more flexibility to move around the blocks irrespective of your HTML markup.

Content-First vs Layout-First:

- Major Uniqueness between Flexbox and Grids is that the former works on content while the latter is based on the layout.
- The Flexbox layout is best suited to application components and small-scale layouts, while the Grid layout is designed for larger-scale layouts that are not linear in design.

Dimensionality and Flexibility:

- Flexbox offers greater control over alignment and space distribution between items. Being one-dimensional, Flexbox only deals with either columns or rows.
- Grid has two-dimension layout capabilities which allow flexible widths as a unit of length. This compensates for the limitations in Flex.

Alignment:

Flex Direction allows developers to align elements vertically or horizontally, which is used when developers create and reverse rows or columns.

CSS Grid deploys fractional measure units for grid fluidity and auto-keyword functionality to automatically adjust columns or rows.

Flexbox vs Grid

Item Management

- Flex Container is the parent element while Flex Item represents the children. The Flex Container can ensure balanced representation by adjusting item dimensions. This allows developers to design for fluctuating screen sizes.
- Grid supports both implicit and explicit content placement. Its inbuilt automation allows it to automatically extend line items and copy values into the new creation from the

Conclusion

- CSS Grids helps you create the outer layout of the webpage. You can build complex as well responsive design with this. Therefore, it is called ‘layout first’.
- Flexbox mostly helps align content & move blocks.
- CSS grids are for 2D layouts. It works with both rows and columns.
- Flexbox works better in one dimension only (either rows OR columns).
- It will be more time saving and helpful if you use both at the same time.

THANK YOU!