Making Layouts

An important concept to understand first is that every single element on a web page is a block. Literally a rectangle of pixels. This is easy to understand when you set the element to *display: block;* or if that element is block-level by default like a <*div*>. This means you can set a width and a height and that element will respect that. But elements that are *display: inline;* like a <*span*> by default, are also rectangles, they just flow onto the page differently, lining up horizontally as they can.

Now that you are picturing every single page element as a block of pixels, we can talk about how positioning is used to get the blocks of pixels exactly where you want them to go.

Position: Static

This is the default for every single page element. Different elements don't have different default values for positioning, they all start out as static. Static doesn't mean much; it just means that the element will flow into the page as it normally would. The only reason you would ever set an element to *position: static*; is to forcefully remove some positioning that got applied to an element outside of your control. This is fairly rare, as positioning doesn't cascade.

Position: Relative

This type of positioning is probably the most confusing and misused. What it really means is "relative to itself". If you set *position: relative*; on an element but no other positioning attributes (top, left, bottom or right), it will have no effect on it's positioning at all, it will be exactly as it would be if you left it as *position: static*; But if you do give it some other positioning attribute, say, *top: 10px;*, it will shift its position 10 pixels down from where it would normally be. I'm sure you can imagine, the ability to shift an element around based on its regular position is pretty useful. I find myself using this to line up form elements many times that have a tendency to not want to line up how I want them to.

There are two other things that happen when you set *position: relative;* on an element that you should be aware of. One is that it introduces the ability to use z-index on that element, which doesn't work with statically positioned elements. Even if you don't set a

z-index value, this element will now appear on top of any other statically positioned element. You can't fight it by setting a higher z-index value on a statically positioned element.

The other thing that happens is it **limits the scope of absolutely positioned child elements**. Any element that is a child of the relatively positioned element can be absolutely positioned within that block. This brings up some powerful opportunities

Position: Absolute

This is a very powerful type of positioning that allows you to literally place any page element exactly where you want it. You use the positioning attributes top, left, bottom, and right to set the location. Remember that these values will be relative to the next parent element with relative (or absolute) positioning. If there is no such parent, it will default all the way back up to the <html> element itself meaning it will be placed relative to the page itself.

The trade-off (and most important thing to remember) about absolute positioning is that these elements are **removed from the flow** of elements on the page. An element with this type of positioning is not affected by other elements and it doesn't affect other elements. This is a serious thing to consider every time you use absolute positioning. Its overuse or improper use can limit the flexibility of your site.

Position: Fixed

A fixed position element is positioned relative to the viewport, or the browser window itself. The viewport doesn't change when the window is scrolled, so a fixed positioned element will stay right where it is when the page is scrolled.

This might be used for something like a navigation bar that you want to remain visible at all times regardless of the pages scroll position. The concern with fixed positioning is that it can cause situations where the fixed element overlaps content such that it is inaccessible.

Position: Sticky

Sticky positioning is really unique! A sticky element will just sit there like a static element, but as you scroll past it, if it's parent element has room (usually: extra height) the sticky element will behave as if it's fixed until that parent element is out of room. It sounds weird in words like that, but it's easy to see what's happening in a demo.

The Box Model

Everything in CSS has a box around it, and understanding these boxes is key to being able to create more complex layouts with CSS, or to align items with other items.

Block and inline boxes

In CSS we have several types of boxes that generally fit into the categories of block boxes and inline boxes. The type refers to how the box behaves in terms of page flow and in relation to other boxes on the page. Boxes have an inner display type and an outer display type.

In general, you can set various values for the display type using the display property, which can have various values.

Outer display type

If a box has an outer display type of block, then:

- The box will break onto a new lie;
- The width and height properties are respected;
- Padding, margin and border will cause other elements to be pushed away from the box;
- If width is not specified, the box will extend in the inline direction to fill the space available in its container. In most cases, the box will become as wide as its container, filling up 100% of the space available;

Some HTML elements, such as *<h1>, <p>,*use block as their outer display type by default.

If a box has an outer display type of inline, then:

- The box will not break onto a new line;
- The width and height properties will not apply;
- Top and bottom padding, margins, and borders will apply but will not cause other inline boxes to move away from the box;

- Left and right padding, margins, and borders will apply and will cause other inline boxes to move away from the box;

  Some HTML elements, such as *<a>, , * and ** use inline as their outer display type by default.

Inner display type

Boxes also have an inner display type, which dictates how elements inside that box are laid out.

Block and inline layout is the default way things behave on the web. By default and without any other instruction, the elements inside a box are also laid out in normal flow and behave as block or inline boxes.

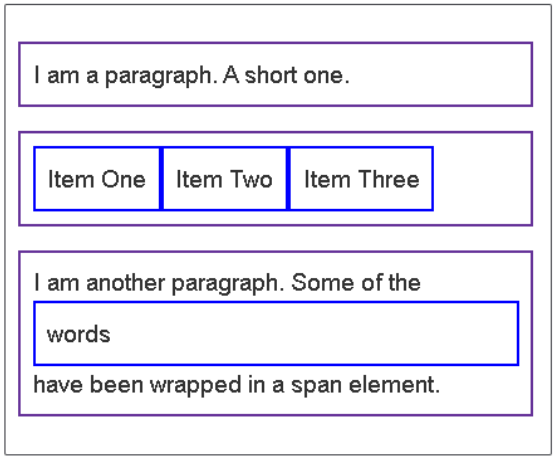
You can change the inner display type for example by setting display: flex;. The element will still use the outer display type block but this changes the inner display type to flex. Any direct children of this box will become flex items and behave according to the Flexbox specification.

When you move on to learn about CSS Layout in more detail, you will encounter flex, and various other inner values that your boxes can have, for example grid.

Examples of different display types

The example below has three different HTML elements, all of which have an outer display type of block.

- A paragraph with a border added in CSS. The browser renders this as a block box. The paragraph starts on a new line and extends the entire available width.
- A list, which is laid out using display: flex. This establishes flex layout for the children of the container, which are flex items. The list itself is a block box and — like the paragraph — expands to the full container width and breaks onto a new line.
- A block-level paragraph, inside which are two elements. These elements would normally be inline, however, one of the elements has a class of "block" which gets set to display: block.



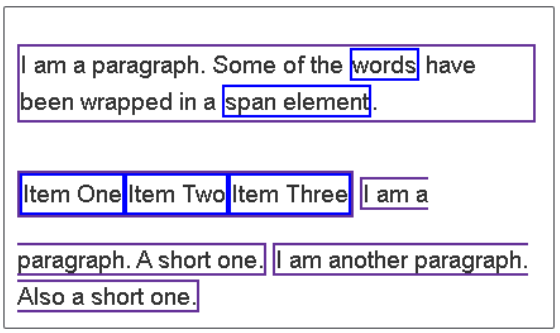

```
p,
ul {
  border: 2px solid rebeccapurple;
  padding: .5em;
}

.block,
li {
  border: 2px solid blue;
  padding: .5em;
}

ul {
  display: flex;
  list-style: none;
}

.block {
  display: block;
}
```

```
<p>I am a paragraph. A short one.</p>
<ul>
  <li>Item One</li>
  <li>Item Two</li>
  <li>Item Three</li>
</ul>
<p>I am another paragraph. Some of the <span
class="block">words</span> have been wrapped in a
<span>span element</span>.</p>
```

In the next example, we can see how inline elements behave.

- The elements in the first paragraph are inline by default and so do not force line breaks.
- The <ul> element that is set to display: inline-flex creates an inline box containing some flex items.
- The two paragraphs are both set to display: inline. The inline flex container and paragraphs all run together on one line rather than breaking onto new lines (as they would do if they were displaying as block-level elements).

To toggle between the display modes, you can change display: inline to display: block or display: inline-flex to display: flex.




```
p,
ul {
  border: 2px solid rebeccapurple;
}

span,
li {
  border: 2px solid blue;
}

ul {
  display: inline-flex;
  list-style: none;
  padding: 0;
}

.inline {
  display: inline;
}
```

```
<p>
  I am a paragraph. Some of the
  <span>words</span> have been wrapped in a
  <span>span element</span>.
</p>
<ul>
  <li>Item One</li>
  <li>Item Two</li>
  <li>Item Three</li>
</ul>
<p class="inline">I am a paragraph. A short one.
</p>
<p class="inline">I am another paragraph. Also a
short one.</p>
```

The key thing to remember for now is: Changing the value of the display property can change whether the outer display type of a box is block or inline. This changes the way it displays alongside other elements in the layout.

What is the CSS box model?

The CSS box model as a whole applies to block boxes and defines how the different parts of a box — margin, border, padding, and content — work together to create a box that you can see on a page. Inline boxes use just some of the behavior defined in the box model.

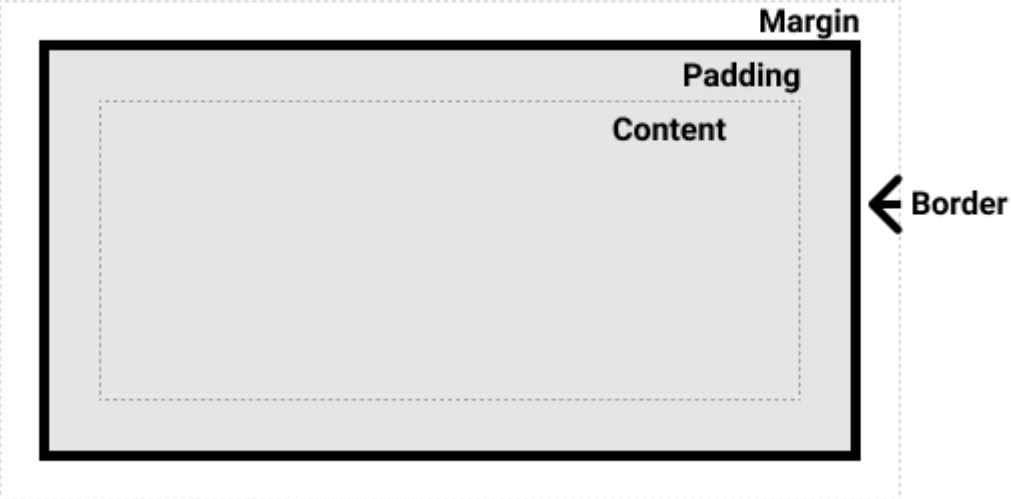
To add complexity, there is a standard and an alternate box model. By default, browsers use the standard box model.

Parts of a box

Making up a block box in CSS we have the:

- **Content box:** The area where your content is displayed; size it using properties like inline-size and block-size or width and height.
- **Padding box:** The padding sits around the content as white space; size it using padding and related properties.
- **Border box:** The border box wraps the content and any padding; size it using border and related properties.
- **Margin box:** The margin is the outermost layer, wrapping the content, padding, and border as whitespace between this box and other elements; size it using margin and related properties.

  The below diagram shows these layers:



The standard CSS box model

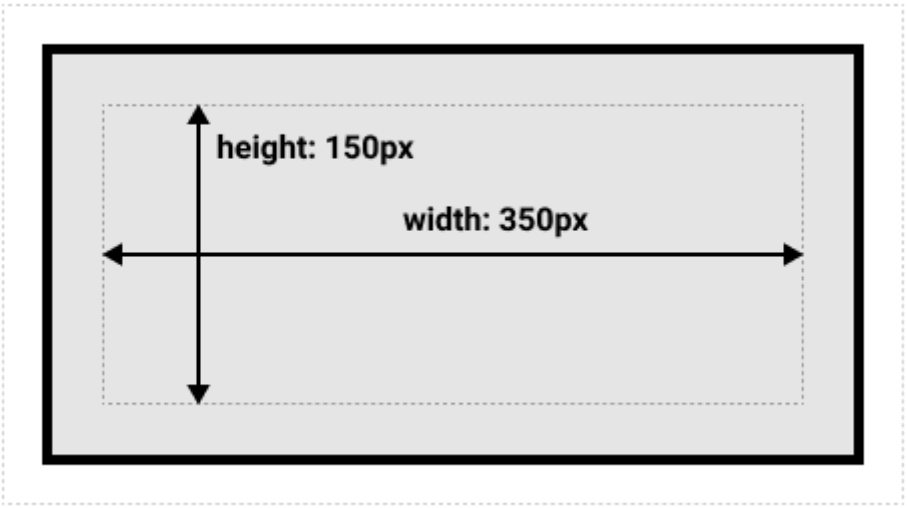
In the standard box model, if you give a box an inline-size and a block-size (or width and a height) attributes, this defines the inline-size and block-size (width and height in horizontal languages) of the content box. Any padding and border is then added to those dimensions to get the total size taken up by the box (see image below).

If we assume that a box has the following CSS:

```
.box {
  width: 350px;
  height: 150px;
  margin: 10px;
```

```
  padding: 25px;
  border: 5px solid black;
}
```

The actual space taken up by the box will be 410px wide (350 + 25 + 25 + 5 + 5) and 210px high (150 + 25 + 25 + 5 + 5).



The alternative CSS box model

In the alternative box model, any width is the width of the visible box on the page. The content area width is that width minus the width for the padding and border (see image below). No need to add up the border and padding to get the real size of the box.

To turn on the alternative model for an element, set box-sizing: border-box on it:

```
.box {
  box-sizing: border-box;
  width: 350px;
  inline-size: 350px;
  height: 150px;
  block-size: 150px;
  margin: 10px;
  padding: 25px;
  border: 5px solid black;

}
```

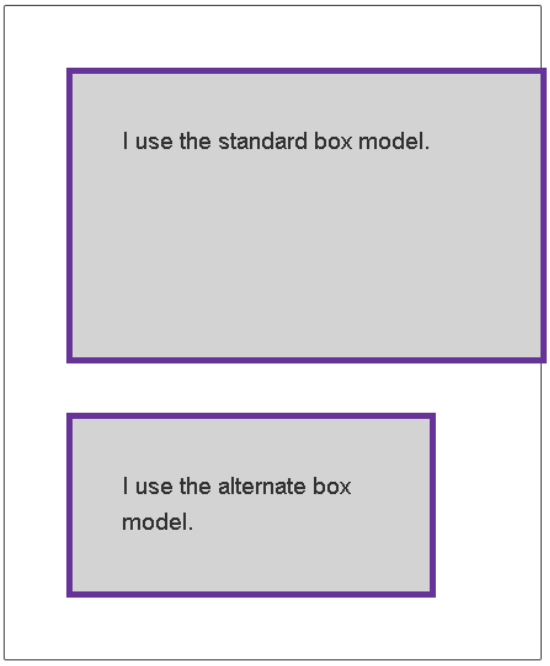
Now, the actual space taken up by the box will be 350px in the inline direction and 150px in the block direction.

To use the alternative box model for all of your elements (which is a common choice among developers), set the box-sizing property on the <html> element and set all other elements to inherit that value:

```
html {
  box-sizing: border-box;
}
*,
*::before,
*::after {
  box-sizing: inherit;
}
```

Playing with box models

In the example below, you can see two boxes. Both have a class of .box, which gives them the same width, height, margin, border, and padding. The only difference is that the second box has been set to use the alternative box model.

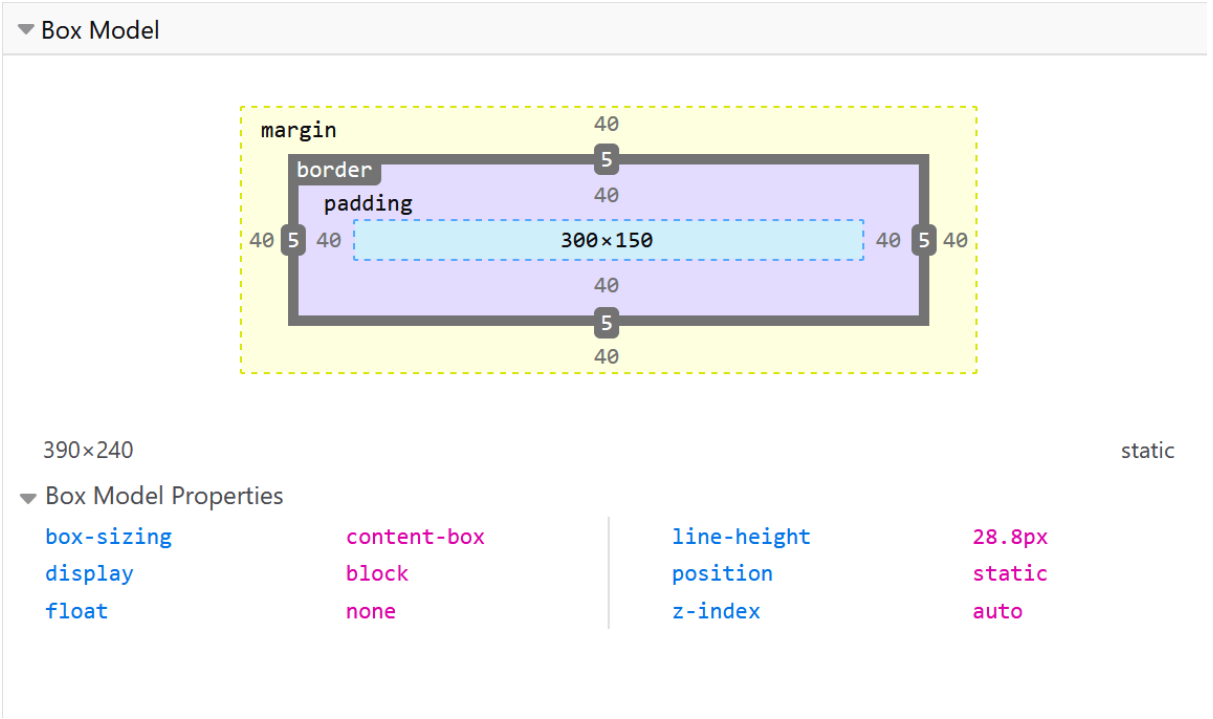



```
.box {
  border: 5px solid rebeccapurple;
  background-color: lightgray;
  padding: 40px;
  margin: 40px;
  width: 300px;
  height: 150px;
}

.alternate {
  box-sizing: border-box;
}
```

```
<div class="box">I use the standard box model.
</div>
<div class="box alternate">I use the alternate box
model.</div>
```

Use browser DevTools to view the box model

Your browser developer tools can make understanding the box model far easier. If you inspect an element in Firefox's DevTools, you can see the size of the element plus its margin, padding, and border. Inspecting an element in this way is a great way to find out if your box is really the size you think it is!



Margins, padding, and borders

You've already seen the margin, padding, and border properties at work in the example above. The properties used in that example are shorthands and allow us to set all four sides of the box at once. These shorthands also have equivalent longhand properties, which allow control over the different sides of the box individually.

Let's explore these properties in more detail.

Margin

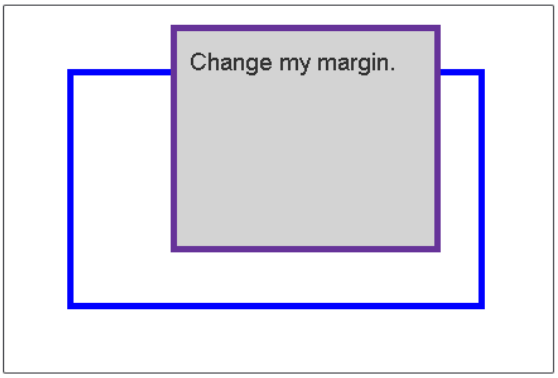
The margin is an invisible space around your box. It pushes other elements away from the box. Margins can have positive or negative values. Setting a negative margin on one side of your box can cause it to overlap other things on the page. Whether you are using the standard or alternative box model, the margin is always added after the size of the visible box has been calculated.

We can control all margins of an element at once using the margin property, or each side individually using the equivalent longhand properties:

- margin-top
- margin-right
- margin-bottom
- margin-left

In the example below, try changing the margin values to see how the box is pushed around

due to the margin creating or removing space (if it is a negative margin) between this element and the containing element.




```
.box {
  margin-top: -40px;
  margin-right: 30px;
  margin-bottom: 40px;
  margin-left: 4em;
}
```

```
<div class="container">
  <div class="box">Change my margin.</div>
</div>
```

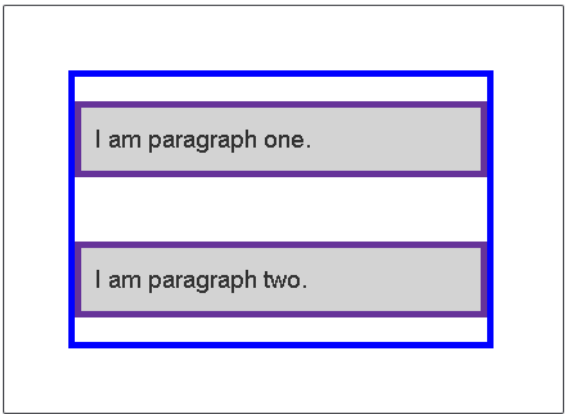
Margin collapsing

Depending on whether two elements whose margins touch have positive or negative margins, the results will be different:

- Two positive margins will combine to become one margin. Its size will be equal to the largest individual margin.
- Two negative margins will collapse and the smallest (furthest from zero) value will be used.
- If one margin is negative, its value will be subtracted from the total.

In the example below, we have two paragraphs. The top paragraph has a margin-bottom of 50 pixels, the other has a margin-top of 30 pixels. The margins have collapsed together so the actual margin between the boxes is 50 pixels and not the total of the two margins.

You can test this by setting the margin-top of paragraph two to 0. The visible margin between the two paragraphs will not change — it retains the 50 pixels set in the margin-bottom of paragraph one. If you set it to -10px, you'll see that the overall margin becomes 40px — it subtracts from the 50px.




```
.one {
  margin-bottom: 50px;
}

.two {
  margin-top: 30px;
}
```

```
<div class="container">
  <p class="one">I am paragraph one.</p>
  <p class="two">I am paragraph two.</p>
</div>
```

Borders

The border is drawn between the margin and the padding of a box. If you are using the standard box model, the size of the border is added to the width and height of the content box. If you are using the alternative box model then the size of the border makes the content box smaller as it takes up some of that available width and height of the element box.

For styling borders, there are a large number of properties — there are four borders, and each border has a style, width, and color that we might want to manipulate.

You can set the width, style, or color of all four borders at once using the border property.

To set the properties of each side individually, use:

- border-top
- border-right
- border-bottom
- border-left

To set the width, style, or color of all sides, use:

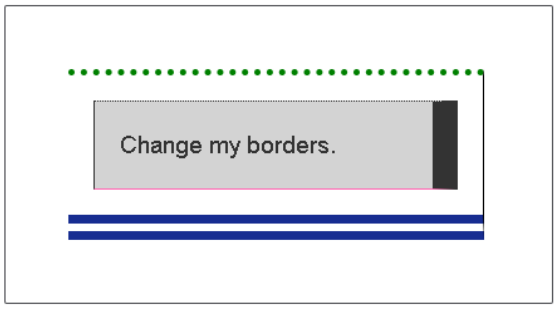
- border-width
- border-style
- border-color

To set the width, style, or color of a single side, use one of the more granular longhand properties:

- border-top-width

- border-top-style
- border-top-color
- border-right-width
- border-right-style
- border-right-color
- border-bottom-width
- border-bottom-style
- border-bottom-color
- border-left-width
- border-left-style
- border-left-color

In the example below, we have used various shorthands and longhands to create borders. Play around with the different properties to check that you understand how they work. The MDN pages for the border properties give you information about the different available border styles.




```
.container {
  border-top: 5px dotted green;
  border-right: 1px solid black;
  border-bottom: 20px double rgb(23,45,145);
}

.box {
  border: 1px solid #333333;
  border-top-style: dotted;
  border-right-width: 20px;
  border-bottom-color: hotpink;
}
```

```
<div class="container">
  <div class="box">Change my borders.</div>
</div>
```

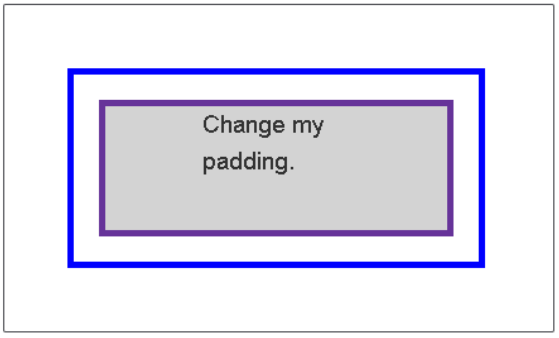
Padding

The padding sits between the border and the content area and is used to push the content away from the border. Unlike margins, you cannot have a negative padding. Any background applied to your element will display behind the padding.

The padding property controls the padding on all sides of an element. To control each side individually, use these longhand properties:

- padding-top
- padding-right
- padding-bottom
- padding-left

In the example below, you can change the values for padding on the class .box to see that this changes where the text begins in relation to the box. You can also change the padding on the class .container to create space between the container and the box. You can change the padding on any element to create space between its border and whatever is inside the element.



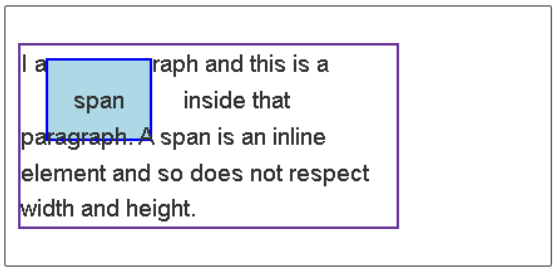

```
.box {
  padding-top: 0;
  padding-right: 30px;
  padding-bottom: 40px;
  padding-left: 4em;
}

.container {
  padding: 20px;
}
```

```
<div class="container">
  <div class="box">Change my padding.</div>
</div>
```

The box model and inline boxes

All of the above fully applies to block boxes. Some of the properties can apply to inline boxes too, such as those created by a <*span*> element.

In the example below, we have a <*span*> inside a paragraph. We have applied a width, height, margin, border, and padding to it. You can see that the width and height are ignored. The top and bottom margin, padding, and border are respected but don't change the relationship of other content to our inline box. The padding and border overlap other words in the paragraph. The left and right padding, margins, and borders move other content away from the box.




```
span {
  margin: 20px;
  padding: 20px;
  width: 80px;
  height: 50px;
  background-color: lightblue;
  border: 2px solid blue;
}
```

```
<p>
  I am a paragraph and this is a <span>span</span>
inside that paragraph. A span is an inline element
and so does not respect width and height.
</p>
```

Using display: inline-block

---

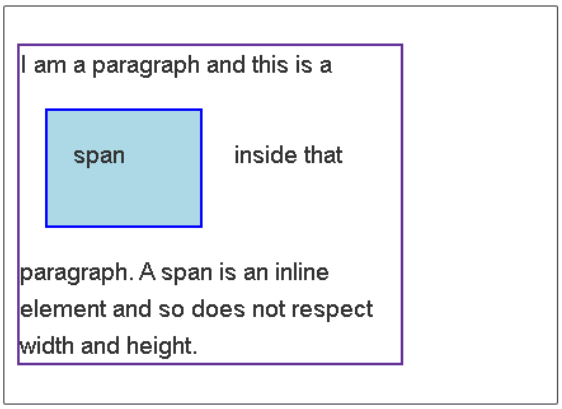
display: inline-block is a special value of display, which provides a middle ground between inline and block. Use it if you do not want an item to break onto a new line, but do want it to respect width and height and avoid the overlapping seen above.

An element with display: inline-block does a subset of the block things we already know about:

- The width and height properties are respected.
- padding, margin, and border will cause other elements to be pushed away from the box.

It does not, however, break onto a new line, and will only become larger than its content if you explicitly add width and height properties.

In this next example, we have added display: inline-block to our element. Try changing this to display: block or removing the line completely to see the difference in display models.



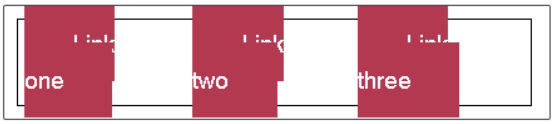

```
span {
  margin: 20px;
  padding: 20px;
  width: 80px;
  height: 50px;
  background-color: lightblue;
  border: 2px solid blue;
  display: inline-block;
}
```

```
<p>
  I am a paragraph and this is a <span>span</span>
inside that paragraph. A span is an inline element
and so does not respect width and height.
</p>
```

Where this can be useful is when you want to give a link a larger hit area by adding padding. <a> is an inline element like ; you can use display: inline-block to allow padding to be set on it, making it easier for a user to click the link.

You see this fairly frequently in navigation bars. The navigation below is displayed in a row using flexbox and we have added padding to the <a> element as we want to be able to change the background-color when the <a> is hovered. The padding appears to overlap the border on the <ul> element. This is because the <a> is an inline element.

Add display: inline-block to the rule with the .links-list a selector, and you will see how it fixes this issue by causing the padding to be respected by other elements.




```
.links-list a {
  background-color: rgb(179,57,81);
  color: #fff;
  text-decoration: none;
  padding: 1em 2em;
}

.links-list a:hover {
  background-color: rgb(66, 28, 40);
  color: #fff;
}
```

```
<nav>
  <ul class="links-list">
    <li><a href="">Link one</a></li>
    <li><a href="">Link two</a></li>
    <li><a href="">Link three</a></li>
  </ul>
</nav>
```

Flexbox

The Flexbox Layout (Flexible Box) module aims at providing a more efficient way to lay out, align and distribute space among items in a container, even when their size is unknown and/or dynamic (thus the word “flex”).

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 (mostly to accommodate all kinds of display devices and screen sizes). A flex container expands items to fill available free space or shrinks them to prevent overflow.

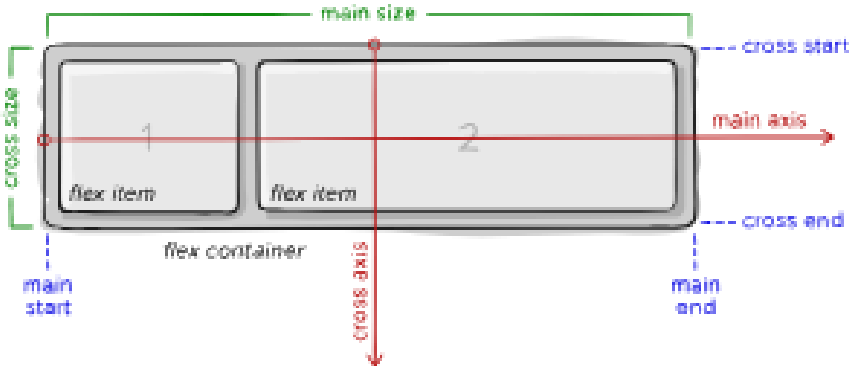
Most importantly, the flexbox layout is direction-agnostic as opposed to the regular layouts (block which is vertically-based and inline which is horizontally-based). While those work well for pages, they lack flexibility (no pun intended) to support large or complex applications (especially when it comes to orientation changing, resizing, stretching, shrinking, etc.).

Note: Flexbox layout is most appropriate to the components of an application, and small-scale layouts, while the Grid layout is intended for larger scale layouts.

Basics and terminology

Since flexbox is a whole module and not a single property, it involves a lot of things including its whole set of properties. Some of them are meant to be set on the container (parent element, known as “flex container”) whereas the others are meant to be set on the children (said “flex items”).

If “regular” layout is based on both block and inline flow directions, the flex layout is based on “flex-flow directions”. Please have a look at this figure from the specification, explaining the main idea behind the flex layout.



Items will be laid out following either the main axis (from main-start to main-end) or the cross axis (from cross-start to cross-end).

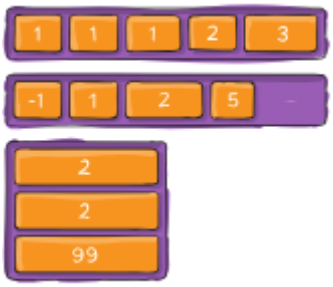
- **main axis –** The main axis of a flex container is the primary axis along which flex items are laid out. Beware, it is not necessarily horizontal; it depends on the flex-direction property (see below).
- **main-start | main-end –** The flex items are placed within the container starting from main-start and going to main-end.
- **main size –** A flex item's width or height, whichever is in the main dimension, is the item's main size. The flex item's main size property is either the 'width' or 'height' property, whichever is in the main dimension.
- **cross axis –** The axis perpendicular to the main axis is called the cross axis. Its direction depends on the main axis direction.
- **cross-start | cross-end –** Flex lines are filled with items and placed into the container starting on the cross-start side of the flex container and going toward the cross-end side.
- **cross size –** The width or height of a flex item, whichever is in the cross dimension, is the item's cross size. The cross size property is whichever of 'width' or 'height' that is in the cross dimension.

Flexbox properties

- **display -** This defines a flex container; inline or block depending on the given value. It enables a flex context for all its direct children;

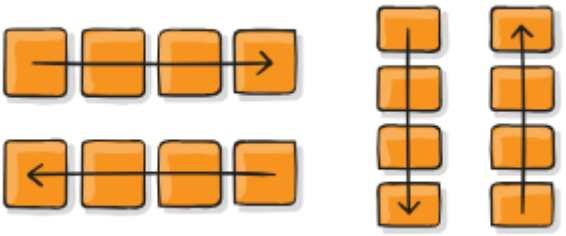
```
.container {
  display: flex; /* or inline-flex */
}
```

- **order -** By default, flex items are laid out in the source order. However, the order property controls the order in which they appear in the flex container;




```
.item {
  order: 5; /* default is 0 */
}
```

- **flex-direction -** This establishes the main-axis, thus defining the direction flex items are placed in the flex container. Flexbox is (aside from optional wrapping) a single-direction layout concept. Think of flex items as primarily laying out either in horizontal rows or vertical columns;
    - **row (default) -** left to right in ltr; right to left in rtl;
    - **row-reverse -** right to left in ltr; left to right in rtl;
    - **column -** same as row but top to bottom;
    - **column-reverse -** same as row-reverse but bottom to top;



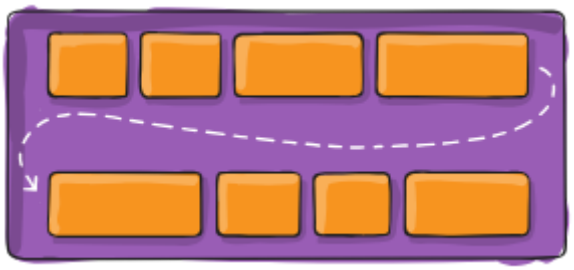
```
.container {
  flex-direction: row | row-reverse | column | column-reverse;
}
```

- **flex-grow -** This defines the ability for a flex item to grow if necessary. It accepts a unitless value that serves as a proportion. It dictates what amount of the available space inside the flex container the item should take up;

  If all items have flex-grow set to 1, the remaining space in the container will be distributed equally to all children. If one of the children has a value of 2, that child would take up twice as much of the space either one of the others (or it will try, at least)


```
.item {
  flex-grow: 4; /* default 0 */
}
```

- **flex-wrap -** By default, flex items will all try to fit onto one line. You can change that and allow the items to wrap as needed with this property.
  - **nowrap (default) -** all flex items will be on one line;
  - **wrap -** flex items will wrap onto multiple lines, from top to bottom;
  - **wrap-reverse -** flex items will wrap onto multiple lines from bottom to top;



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

- **flex-shrink -** This defines the ability for a flex item to shrink if necessary;

```
.item {
  flex-shrink: 3; /* default 1 */
}
```

- **flex-flow -** This is a shorthand for the flex-direction and flex-wrap properties, which together define the flex container's main and cross axes. The default value is row nowrap

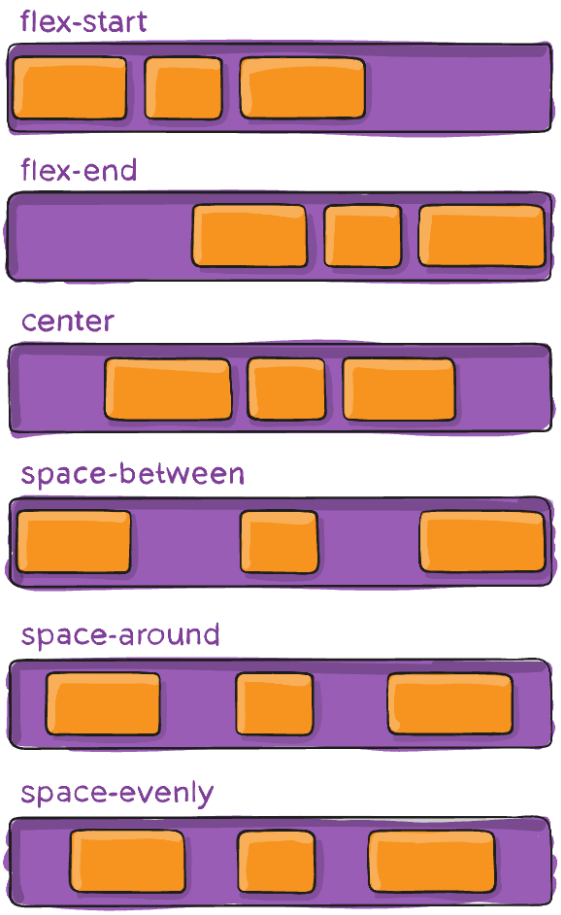
```
container {
  flex-flow: column wrap;
}
```

- **flex-basis -** This defines the default size of an element before the remaining space is distributed. It can be a length or a keyword. The auto keyword means "look at my width or height property". The content keyword means "size it based on the item's content" - this keyword isn't well supported yet, so it's hard to test and harder to know what its brethren max-content, min-content, and fit-content do;

  If set to 0, the extra space around content isn't factored in. If set to auto, the extra space is distributed based on its flex-grow value;

```
.item {
  flex-basis:  | auto; /* default auto */
}
```

- **justify-content -** This defines the alignment along the main axis. It helps distribute extra free space leftover when either all the flex items on a line are inflexible, or are flexible but have reached their maximum size. It also exerts some control over the alignment of items when they overflow the line;
    - **flex-start (default) -** items are packed toward the start of the flex-direction.
    - **flex-end -** items are packed toward the end of the flex-direction.
    - **start -** items are packed toward the start of the writing-mode direction.
    - **end -** items are packed toward the end of the writing-mode direction.
    - **left -** items are packed toward left edge of the container, unless that doesn't make sense with the flex-direction, then it behaves like start.
    - **right -** items are packed toward right edge of the container, unless that doesn't make sense with the flex-direction, then it behaves like end.
    - **center -** items are centered along the line
    - **space-between -** items are evenly distributed in the line; first item is on the start line, last item on the end line
    - **space-around -** items are evenly distributed in the line with equal space around them. Note that visually the spaces aren't equal, since all the items have equal space on both sides. The first item will have one unit of space against the container edge, but two units of space between the next item because that next item has its own spacing that applies.
    - **space-evenly:** items are distributed so that the spacing between any two items (and the space to the edges) is equal.




```
.container {
  justify-content: flex-start | flex-end | center | space-between
| space-around | space-evenly | start | end | left | right ... +
safe | unsafe;
}
```

- **flex -** This is the shorthand for flex-grow, flex-shrink and flex-basis combined. The second and third parameters (flex-shrink and flex-basis) are optional. The default is 0 1 auto, but if you set it with a single number value, like flex: 5;, that changes the flex-basis to 0%, so it's like setting flex-grow: 5; flex-shrink: 1; flex-basis: 0%;

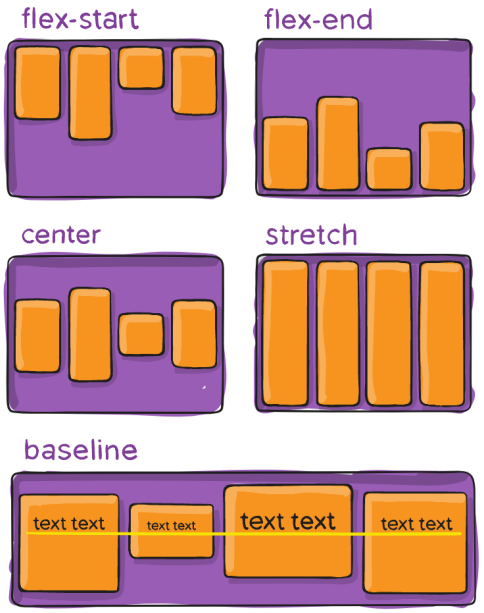
  It is recommended that you use this shorthand property rather than set the individual properties. The shorthand sets the other values intelligently.

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

- **align-self -** This allows the default alignment (or the one specified by align-items) to be overridden for individual flex items.

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

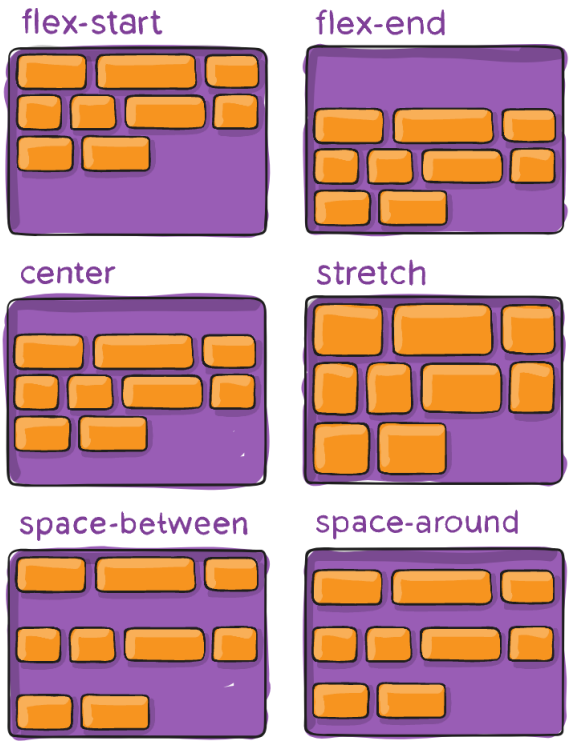
- **align-items -** This defines the default behavior for how flex items are laid out along the cross axis on the current line. Think of it as the justify-content version for the cross-axis;
    - **stretch (default) -** stretch to fill the container (still respect min-width/max-width)
    - **flex-start / start / self-start -** items are placed at the start of the cross axis. The difference between these is subtle, and is about respecting the flex-direction rules or the writing-mode rules.
    - **flex-end / end / self-end -** items are placed at the end of the cross axis. The difference again is subtle and is about respecting flex-direction rules vs. writing-mode rules.
    - **center -** items are centered in the cross-axis
    - **baseline -** items are aligned such as their baselines align




```
.container {
  align-items: stretch | flex-start | flex-end | center | baseline
```

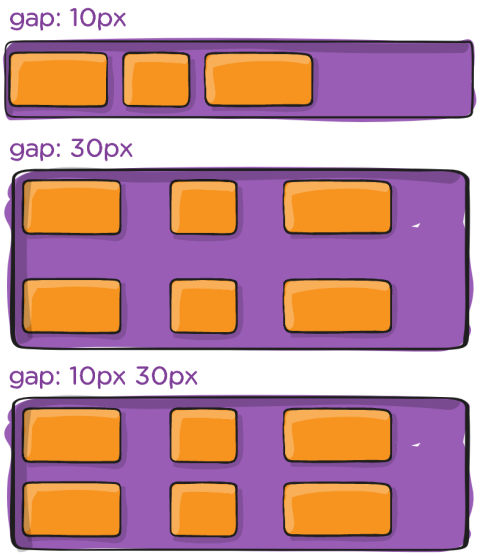
```
| first baseline | last baseline | start | end | self-start |
self-end + ... safe | unsafe;
}
```

- **align-content -** This aligns a flex container's lines within when there is extra space in the cross-axis, similar to how justify-content aligns individual items within the main-axis;
    - ***Note -*** *This property only takes effect on multi-line flexible containers, where flex-wrap is set to either wrap or wrap-reverse). A single-line flexible container (i.e. where flex-wrap is set to its default value, no-wrap) will not reflect align-content.*
    - **normal (default) -** items are packed in their default position as if no value was set.
    - **flex-start / start -** items packed to the start of the container. The (more supported) flex-start honors the flex-direction while start honors the writing-mode direction.
    - **flex-end / end -** items packed to the end of the container. The (more support) flex-end honors the flex-direction while end honors the writing-mode direction.
    - **center -** items centered in the container
    - **space-between -** items evenly distributed; the first line is at the start of the container while the last one is at the end
    - **space-around -** items evenly distributed with equal space around each line
    - **space-evenly -** items are evenly distributed with equal space around them
    - **stretch -** lines stretch to take up the remaining space




```
.container {
  align-content: flex-start | flex-end | center | space-between |
space-around | space-evenly | stretch | start | end | baseline |
first baseline | last baseline + ... safe | unsafe;
}
```

- **gap, row-gap, column-gap -** The gap property explicitly controls the space between flex items. It applies that spacing only between items not on the outer edges




```
.container {
```

```
  display: flex;
  ...
  gap: 10px;
  gap: 10px 20px; /* row-gap column gap */
  row-gap: 10px;
  column-gap: 20px;
}
```

Prefixing Flexbox

Flexbox requires some vendor prefixing to support the most browsers possible. It doesn't just include prepending properties with the vendor prefix, but there are actually entirely different property and value names. This is because the Flexbox spec has changed over time, creating an "old", "tweener", and "new" versions.

Perhaps the best way to handle this is to write in the new (and final) syntax and run your CSS through Autoprefixer, which handles the fallbacks very well.

Alternatively, here's a Sass @mixin to help with some of the prefixing, which also gives you an idea of what kind of things need to be done:

```
@mixin flexbox() {
  display: -webkit-box;
  display: -moz-box;
  display: -ms-flexbox;
  display: -webkit-flex;
  display: flex;
}

@mixin flex($values) {
  -webkit-box-flex: $values;
  -moz-box-flex:  $values;
  -webkit-flex:  $values;
  -ms-flex:  $values;
  flex:  $values;
}

@mixin order($val) {
  -webkit-box-ordinal-group: $val;
```

```
  -moz-box-ordinal-group: $val;
  -ms-flex-order: $val;
  -webkit-order: $val;
  order: $val;
}

.wrapper {
  @include flexbox();
}

.item {
  @include flex(1 200px);
  @include order(2);
}
```

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.

CSS Grid Basics

To get started you have to define a container element as a grid with display: grid, set the column and row sizes with grid-template-columns and grid-template-rows, and then place its child elements into the grid with grid-column and grid-row. Similarly to flexbox, the source order of the grid items doesn’t matter. Your CSS can place them in any order, which makes it super easy to rearrange your grid with media queries. Imagine defining the layout of your entire page, and then completely rearranging it to accommodate a different screen width

all with only a couple lines of CSS. Grid is one of the most powerful CSS modules ever introduced.

Important CSS Grid Terminology

Before diving into the concepts of Grid it's important to understand the terminology. Since the terms involved here are all kinda conceptually similar, it's easy to confuse them with one another if you don't first memorize their meanings defined by the Grid specification. But don't worry, there aren't many of them.

- 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 gridcontainer;

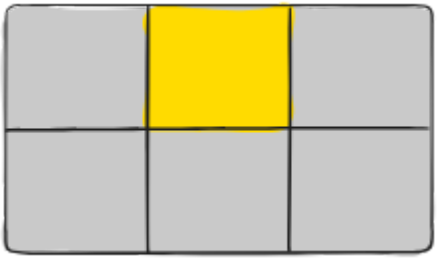
```
<div class="container">
  <div class="item item-1"> </div>
  <div class="item item-2"> </div>
  <div class="item item-3"> </div>
</div>
```

- Grid Item - The children (i.e. direct descendants) of the grid container. Here the item elements are grid items, but sub-item isn't.

```
<div class="container">
  <div class="item"> </div>
  <div class="item">
    <p class="sub-item"> </p>
  </div>
  <div class="item"> </div>
</div>
```

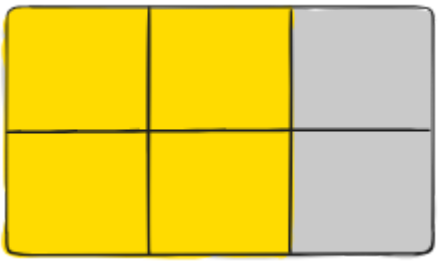
- Grid Line - The dividing lines that make up the structure of the grid. They can be either vertical ("column grid lines") or horizontal ("row grid lines") and reside on either side of a row or column. Here the yellow line is an example of a column grid line.

- 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.



- 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.



CSS Grid Properties

- **display -** Defines the element as a grid container and establishes a new grid formatting context for its contents.
    - grid – generates a block-level grid
    - inline-grid – generates an inline-level grid

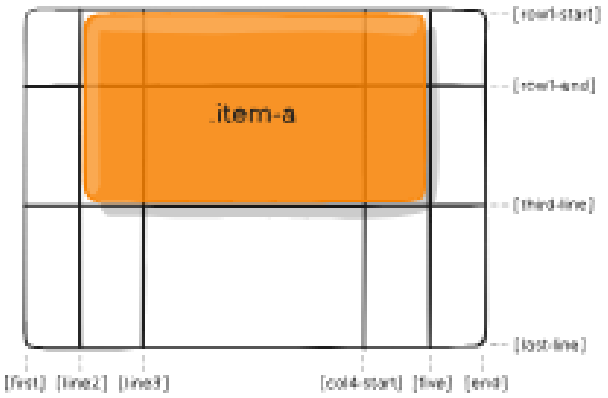
```
.container {
  display: grid | inline-grid;
}
```

- **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.
    - **<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

```
.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;
}
```

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



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

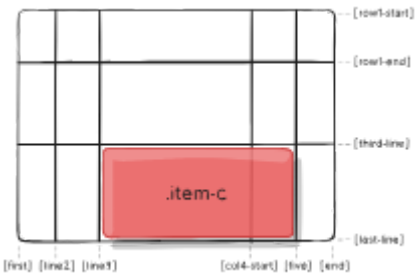
```
.container {
  grid-template-columns: ...  ...;
  /* e.g.
      1fr 1fr
      minmax(10px, 1fr) 3fr
      repeat(5, 1fr)
      50px auto 100px 1fr
```

```
  */
  grid-template-rows: ... ...;
  /* e.g.
      min-content 1fr min-content
      100px 1fr max-content
  */
}
```

- **grid-column | grid-row -** Shorthand for grid-column-start + grid-column-end, and grid-row-start + grid-row-end, respectively.
    - **<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>;
}
```

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

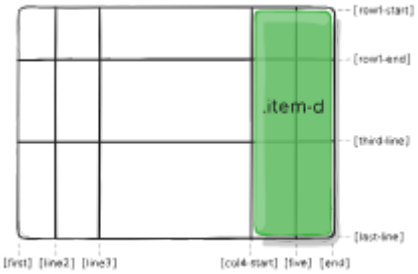


- **grid-area -** Gives an item a name so that it can be referenced by a template created with the grid-template-areas property. Alternatively, this property can be used as an even shorter shorthand for grid-row-start + grid-column-start + grid-row-end + grid-column-end.
    - **<name>** – a name of 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>;
}
```

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

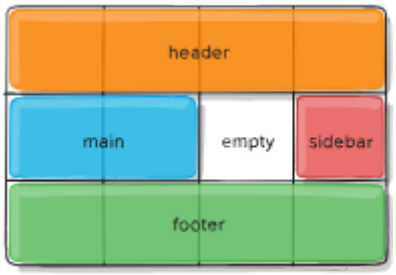


- **grid-template-areas -** Defines a grid template by referencing the names of the grid areas which are specified with the grid-area property. Repeating the name of a grid area causes the content to span those cells. A period signifies an empty cell. The syntax itself provides a visualization of the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

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

```
.item-a {
```

```
  grid-area: header;
}
.item-b {
  grid-area: main;
}
.item-c {
  grid-area: sidebar;
}
.item-d {
  grid-area: footer;
}

.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";
}
```

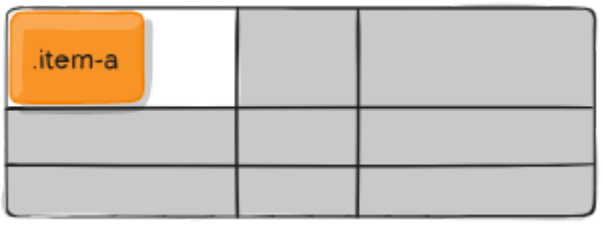


- Each row in your declaration needs to have the same number of cells.
  You can use any number of adjacent periods to declare a single empty cell. As long as the periods have no spaces between them they represent a single cell.
  Notice that you're not naming lines with this syntax, just areas. When you use this syntax the lines on either end of the areas are actually getting named automatically. If the name of your grid area is foo, the name of the area's starting row line and starting column line will be foo-start, and the name of its last row line and last column line will be foo-end. This means that some lines might have multiple names, such as the far left line in the above example, which will have three names: header-start, main-start, and footer-start.

- **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.
    - **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;
}
```

```
.item-a {
  justify-self: start;
}
```



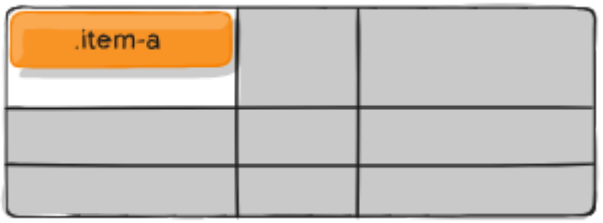
- **grid-template -** A shorthand for setting grid-template-rows, grid-template-columns, and grid-template-areas in a single declaration.
    - **none –** sets all three properties to their initial values
    - **<grid-template-rows> / <grid-template-columns> –** sets grid-template-columns and grid-template-rows to the specified values, respectively, and sets grid-template-areas to none

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

- **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.
    - **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;
}
```

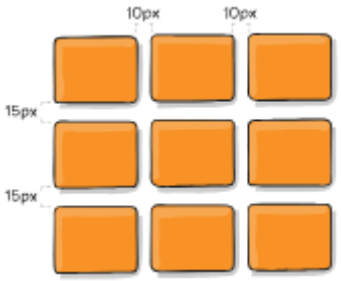
```
.item-a {
  align-self: start;
}
```



- **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.
    - **<line-size> –** a length value

```
.container {
  /* standard */
  column-gap: <line-size>;
  row-gap: <line-size>;

  /* old */
  grid-column-gap: <line-size>;
  grid-row-gap: <line-size>;
}
```

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



- **place-self -** place-self sets both the align-self and justify-self properties in a single declaration.
    - **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;
}
```

- **gap | grid-gap -** A shorthand for row-gap and column-gap
    - **<grid-row-gap> <grid-column-gap> –** length values

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).

- **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.
    - **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;
}
```

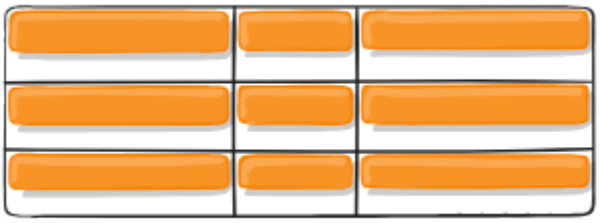
```
.container {
  justify-items: start;
}
```

- **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.
    - **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;
}
```

```
.container {
  align-items: start;
}
```



- **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 “E pluribus unum”, the min-content is likely the width of the word “pluribus”.
  - **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.
  - **Fractional units -** see above

- Sizing Functions - The fit-content() function uses the space available, but never less than min-content and never more than max-content.

  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;
```

- 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 and one of the all-time great CSS tricks:

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

- **Masonry -** An experimental feature of CSS grid is masonry layout. Note that there are lots of approaches to CSS masonry, but mostly of them are trickery and either have major downsides or aren't what you quite expect. The spec has an official way now, and this is behind a flag in Firefox:

```
.container {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  grid-template-rows: masonry;
}
```

- **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.

```
.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;
}
```

- **place-items -** place-items sets both the align-items and justify-items properties in a single declaration.
    - **<align-items> / <justify-items> –** The first value sets align-items, the second value justify-items. If the second value is omitted, the first value is assigned to both properties.

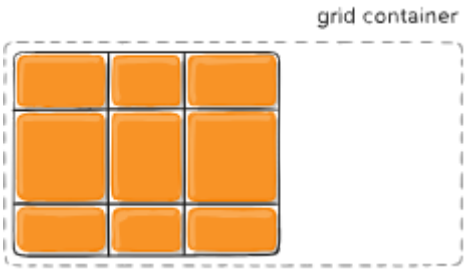
```
.center {
  display: grid;
  place-items: center;
}
```

- **justify-content -** Sometimes the total size of your grid might be less than the size of its grid container. This could happen if all of your grid items are sized with non-flexible units like px. In this case you can set the alignment of the grid within the grid container. This property aligns the grid along the inline (row) axis (as opposed to align-content which aligns the grid along the block (column) axis).
    - **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;
}
```

```
.container {
  justify-content: start;
}
```

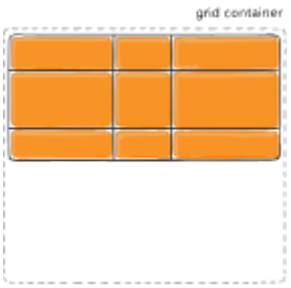


- **align-content -** Sometimes the total size of your grid might be less than the size of its grid container. This could happen if all of your grid items are sized with non-flexible units like px. In this case you can set the alignment of the grid within the grid container. This property aligns the grid along the block (column) axis (as opposed to justify-content which aligns the grid along the inline (row) axis).
  - **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;
}
```

```
.container {
  align-content: start;
}
```



- **place-content** - place-content sets both the align-content and justify-content properties in a single declaration.
  - **<align-content> / <justify-content>** – The first value sets align-content, the second value justify-content. If the second value is omitted, the first value is assigned to both properties.

- **grid-auto-columns | grid-auto-rows -** Specifies the size of any auto-generated grid tracks (aka implicit grid tracks). Implicit tracks get created when there are more grid items than cells in the grid or when a grid item is placed outside of the explicit grid.
  - **<track-size>** – can be a length, a percentage, or a fraction of the free space in the grid (using the fr unit)

```
.container {
  grid-auto-columns: <track-size> ...;
```

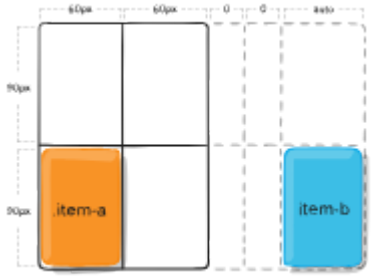
```
  grid-auto-rows: <track-size> ...;
}
```

```
.container {
  grid-template-columns: 60px 60px;
  grid-template-rows: 90px 90px;
}
```

This creates a 2 x 2 grid.

But now imagine you use grid-column and grid-row to position your grid items like this:

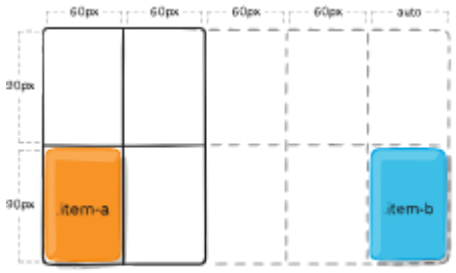
```
.item-a {
  grid-column: 1 / 2;
  grid-row: 2 / 3;
}
.item-b {
  grid-column: 5 / 6;
  grid-row: 2 / 3;
}
```



We told .item-b to start on column line 5 and end at column line 6, but we never defined a column line 5 or 6. Because we referenced lines that don't exist, implicit tracks with widths of 0 are created to fill in the gaps. We can use grid-auto-columns and grid-auto-rows to specify the widths of these implicit tracks:

```
.container {
  grid-auto-columns: 60px;
```

```
}
```



- **grid-auto-flow -** If you have grid items that you don't explicitly place on the grid, the auto-placement algorithm kicks in to automatically place the items. This property controls how the auto-placement algorithm works.
    - **row** – tells the auto-placement algorithm to fill in each row in turn, adding new rows as necessary (default)
    - **column** – tells the auto-placement algorithm to fill in each column in turn, adding new columns as necessary
    - **dense** – tells the auto-placement algorithm to attempt to fill in holes earlier in the grid if smaller items come up later

```
.container {
  grid-auto-flow: row | column | row dense | column dense;
}
```

```
<section class="container">
  <div class="item-a">item-a</div>
  <div class="item-b">item-b</div>
  <div class="item-c">item-c</div>
  <div class="item-d">item-d</div>
  <div class="item-e">item-e</div>
</section>
```

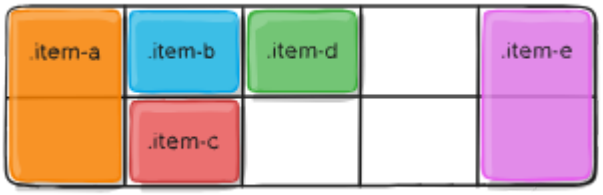
```
.container {
  display: grid;
  grid-template-columns: 60px 60px 60px 60px 60px;
  grid-template-rows: 30px 30px;
  grid-auto-flow: row;
}
```

```
.item-a {
  grid-column: 1;
  grid-row: 1 / 3;
}
.item-e {
  grid-column: 5;
  grid-row: 1 / 3;
}
```

Because we set grid-auto-flow to row, our grid will look like this. Notice how the three items we didn't place (item-b, item-c and item-d) flow across the available rows:

If we instead set grid-auto-flow to column, item-b, item-c and item-d flow down the columns:

```
.container {
  display: grid;
  grid-template-columns: 60px 60px 60px 60px 60px;
  grid-template-rows: 30px 30px;
  grid-auto-flow: column;
}
```



References

[Absolute, Relative, Fixed Positioning: How Do They Differ? | CSS-Tricks - CSS-Tricks]

[The box model - Learn web development | MDN (mozilla.org)]

[A Complete Guide to Flexbox | CSS-Tricks - CSS-Tricks]

[A Complete Guide to CSS Grid | CSS-Tricks - CSS-Tricks]