



CS472 Web Programming

Lecture 3: Layout



Except where otherwise noted, the contents of this document are Copyright 2012 Marty Stepp, Jessica Miller, Victoria Kirst and Roy McElmurry IV. All rights reserved. Any redistribution, reproduction, transmission, or storage of part or all of the contents in any form is prohibited without the author's expressed written permission. Slides have been modified for Maharishi University of Management Computer Science course CS472 in accordance with instructors agreement with authors.

Reference:

<https://learnlayout.com/inline-block.html>

<https://css-tricks.com/snippets/css/a-guide-to-flexbox/>

<https://css-tricks.com/snippets/css/complete-guide-grid/>

Maharishi University of Management -Fairfield, Iowa

© 2019



All rights reserved. No part of this slide presentation may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying or recording, or by any information storage and retrieval system, without permission in writing from Maharishi University of Management.

Wholeness Statement

CSS provides different tools for creating a layout. There are a variety of ways to position an element; most of them are based on taking a block level element and placing it in relation to some other block. It is this relationship that becomes the tricky part. The question is always: “This positioned is relative to what?” This illustrates the general principle that individual parts must often be understood in terms of a larger context.

The whole is greater than the sum of its parts.

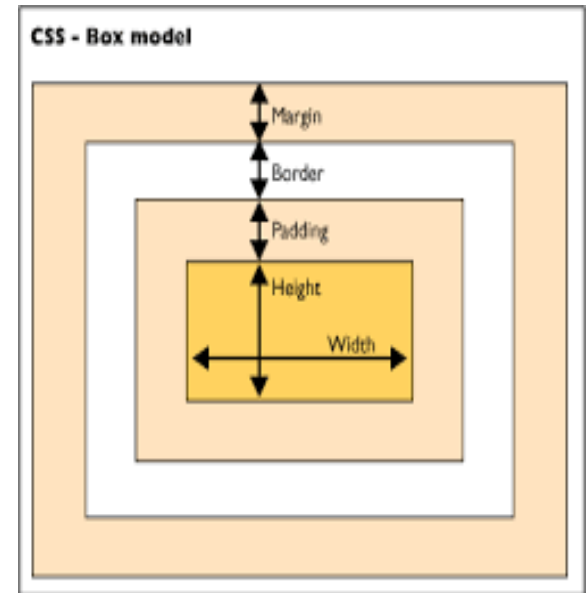
Main Point Preview

The box model is a description of how every element has a basic width and height, outside of which it has padding, a border, and margin. For inline elements only the left and right margin and padding affect surrounding elements.

The box model is another encapsulation mechanism that allows layout style to be separate from the page content. Life is found in layers.

The CSS Box Model

- ▶ For layout purposes, every element is composed of:
 - The actual element's **content**
 - A **border** around the element
 - **padding** between the content and the border (inside)
 - A **margin** between the border and other content (outside)
- ▶ $\text{width} = \text{content width} + \text{L/R padding} + \text{L/R border} + \text{L/R margin}$
- ▶ $\text{height} = \text{content height} + \text{T/B padding} + \text{T/B border} + \text{T/B margin}$
- ▶ The standard `width` and `height` properties refer ONLY to the content's width and height.





CSS properties for borders

```
h2 { border: 5px solid red; }
```

This is the best WAP course!

Property	Description
border	thickness/style/size of border on all 4 sides

- ▶ **thickness** (specified in px, pt, em, or thin, medium, thick)
- ▶ **style** (none, hidden, dotted , dashed , double , groove , inset , outset , ridge , solid)
- ▶ **color** (specified as seen previously for text and background colors)

More border properties

Property	Description
border-color, border-width, border-style	specific properties of border on all 4 sides
border-bottom, border-left, border-right, border-top	all properties of border on a particular side
border-bottom-color, border-bottom-style, border-bottom-width, border-left-color, border-left-style, border-left-width, border-right-color, border-right-style, border-right-width, border-top-color, border-top-style, border-top-width	properties of border on a particular side
Complete list of border properties	



Border example 2

```
p {  
  border-left: thick dotted #CC0088;  
  border-bottom-color: rgb(0, 128, 128);  
  border-bottom-style: double;  
}
```

This is the best WAP course!

- ▶ each side's border properties can be set individually
- ▶ if you omit some properties, they receive default values (e.g. border-bottom-width above)



Dimensions

- ▶ For Block elements and `img` element only, set how wide or tall this element, or set the max/min size of this element in given dimension.
- ▶ Using `max-width` instead of `width` in this situation will improve the browser's handling of small windows.
- ▶ `Max-width` will have a smaller width box on smaller viewports versus fixed width will result in extending off screen

`width`, `height`, `max-width`, `max-height`, `min-width`, `min-height`

```
p { width: 350px; background-color: yellow; }  
h2 { width: 50%; background-color: aqua; }
```

This paragraph uses the first style above.

An h2 heading

Or See demo: <Lesson03/max-width.html>

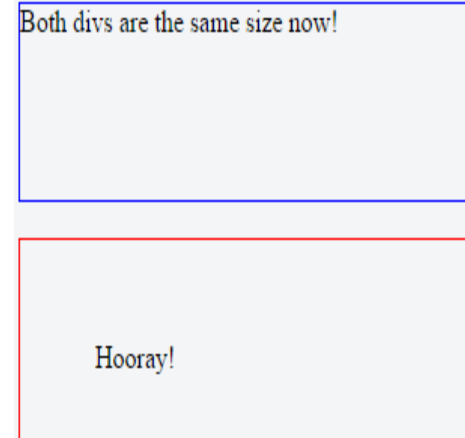




box-sizing

- ▶ `box-sizing: content-box;` - initial and default value. The `width` and `height` properties are measured including only the content, but not the padding, border or margin.
- ▶ `box-sizing: border-box;` The `width` and `height` properties include the content, the padding and border, but not the margin. Note that padding and border will be inside of the box.
- ▶ Q: Why is Hooray indented?
- ▶ [responsive design advantage demo](#)


```
.div3 {  
  width: 300px;  
  height: 100px;  
  border: 1px solid blue;  
  box-sizing: border-box;  
}  
.div4 {  
  width: 300px;  
  height: 100px;  
  padding: 50px;  
  border: 1px solid red;  
  box-sizing: border-box;  
}
```



Rounded corners `border-radius`



```
p {  
  border: 3px solid blue;  
  border-radius: 12px;  
}
```



Text Paragraph

- ▶ Each side's border radius can be set individually, separated by spaces
 - ▶ **Four values:** top-left, top-right, bottom-right, bottom-left
 - ▶ **Three values:** top-left, top-right and bottom-left, bottom-right
 - ▶ **Two values:** top-left and bottom-right, top-right and bottom-left
 - ▶ **One value:** all four corners are rounded equally

Padding

- ▶ The padding shorthand property sets all the padding properties in one declaration. Padding shares the background color of the element. This property can have from one to four values:

```
padding: 10px 5px 15px 20px; /* Top, right, bottom, left */  
padding: 10px 5px 15px; /* Top, right and left, bottom */  
padding: 10px 5px; /* Top and bottom, right and left */  
padding: 10px; /* All four paddings are 10px */
```

- ▶ padding-bottom, padding-left, padding-right, padding-top

```
h1 { padding: 20px; }  
h2 {  
    padding-left: 200px;  
    padding-top: 30px;  
}
```



Margin

- ▶ Margins are always transparent. This property can have from one to four values:

```
margin: 10px 5px 15px 20px; /* Top, right, bottom, left */
```

```
margin: 10px 5px 15px; /* Top, right and left, bottom */
```

```
margin: 10px 5px; /* Top and bottom, right and left */
```

```
margin: 10px; /* All four margins are 10px */
```

- ▶ margin-bottom, margin-left, margin-right, margin-top

```
h1 {margin: 20px; }
```

```
h2 {  
    margin-left:    200px;  
    margin-top:     30px;  
}
```

- ▶ See example: lesson3_examples/paddingmargin.html
 - ▶ Inspect element in console



Margin Collapse

- ▶ Vertical margins on different elements that touch each other (thus have no content, padding, or borders separating them) will collapse, forming a single margin that is equal to the greater of the adjoining margins.
 1. Collapsing Margins Between Adjacent Elements
 2. Collapsing Margins Between Parent and Child Elements

See demo: [Lesson03/margincollapse*.html](#)

Main Point

The box model is a description of how every element has a basic width and height, outside of which it has padding, a border, and margin. For inline elements only the left and right margin and padding affect surrounding elements.

The box model is another encapsulation mechanism that allows layout style to be separate from the page content. Life is found in layers.

Main Point Preview: Block vs inline elements

A block-level element always starts on a new line and takes up the full width available (stretches out to the left and right as far as it can). An inline element does not start on a new line and only takes up as much width as necessary. These are the fundamental display types for almost all HTML elements. Understanding these fundamental types is critical to creating effective layouts.

Familiarity with fundamental levels of awareness is critical to successful action.

Details about **block** boxes

- ▶ By default **block** elements take the entire width space of the page unless we specify.
- ▶ To align a **block** element at the **center** of a horizontal space you must set a **width** first, and **margin: auto;**
- ▶ **text-align** does not align **block** elements within the page.

Centering a block element: auto margins



```
<p> Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do  
    eiusmod tempor incididunt ut la-bore et dolore magna aliqua.  
</p>
```

```
p {  
  border: 2px solid black;  
  margin-left: auto;  
  margin-right: auto;  
  width: 33%;  
}
```

- ▶ Set the width of a block-level element to prevent it from stretching out to the edges of its container.
 - ▶ Set left and right margins to auto to horizontally center that element.
 - ▶ Remaining space will be split evenly between the two margins.
- ▶ to center inline elements within a block element, use `text-align: center;`



Details about **inline** boxes

- ▶ size properties (width, height, min-width, etc.) are ignored for inline boxes
 - ▶ margin-top and margin-bottom are ignored, but margin-left and margin-right are not
 - ▶ Padding top and bottom ignored
 - ▶ each inline box's vertical-align property aligns it vertically within its block box
 - ▶ **text-align** describes how inline content is aligned in its parent block element.
 - ▶ does not control the alignment of block elements, only their inline content
-
- ▶ 2▶ See lesson3_examples/textalign.html

The `vertical-align` property

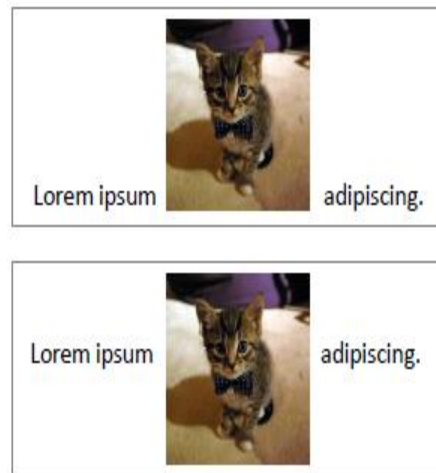


- ▶ Specifies where an inline element should be aligned vertically, with respect to other content on the same line within its block element's box
- ▶ Can be top, middle, bottom, baseline (default), sub, super, text-top, text-bottom, or a length value or %. baseline means aligned with bottom of non-hanging letters
- ▶ image is vertically aligned to the baseline of the paragraph, which isn't the same as the bottom



```
img {  
  vertical-align: baseline;  
}  
img {  
  vertical-align: middle;  
}
```

- ▶ See common error example:
 lesson3_examples/verticalalign.html



display property: block vs inline

- ▶ The default value of display **property** for most elements is usually **block** or **inline**.
- ▶ **Block:** **div** is the standard block-level element. A block-level element starts on a new line and stretches out to the left and right as far as it can. Other common block-level elements are **p** and **form**, and new in HTML5 are **header**, **footer**, **section**, and more.
- ▶ **Inline:** **span** is the standard inline element. An inline element can wrap some text inside a paragraph **** like this **** without disrupting the flow of that paragraph. The **a** element is the most common inline element, since you use them for links.
- ▶ **None:** Another common display value is **none**. Some specialized elements such as **script** use this as their default. It is commonly used with JavaScript to hide and show elements without really deleting and recreating them.
- ▶ **inline-block** elements are like **inline** elements but they can have a width and height.
- ▶ Flex and grid: new options for responsive layouts

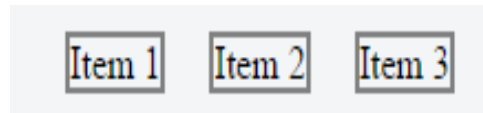
Displaying **block** elements as **inline**



- ▶ Lists and other **block** elements can be displayed **inline**, flow left-to-right on same line.

Note: Width will be determined by content (while block elements are 100% of page width)

```
<ul id="topmenu">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```



```
#topmenu li {
  display: inline;
  border: 2px solid gray;
  margin-right: 1em;
  list-style-type: none;
}
```

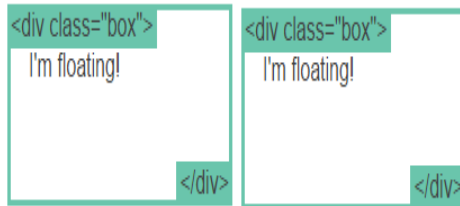

Boxes (Photo Gallery)



The Hard Way (using float)

```
.box {  
  float: left;  
  width: 200px;  
  height: 100px;  
  margin: 1em;  
}
```

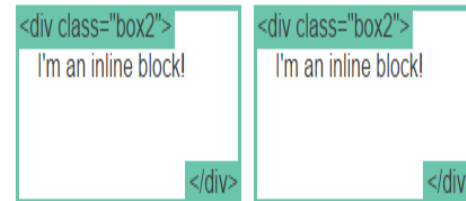
```
.after-box {  
  clear: left;  
}
```



```
<div class="after-box">  
  I'm using clear so I don't float next to the above boxes.  
</div>
```

The Easy Way (using inline-block)

```
.box2 {  
  display: inline-block;  
  width: 200px;  
  height: 100px;  
  margin: 1em;  
}
```

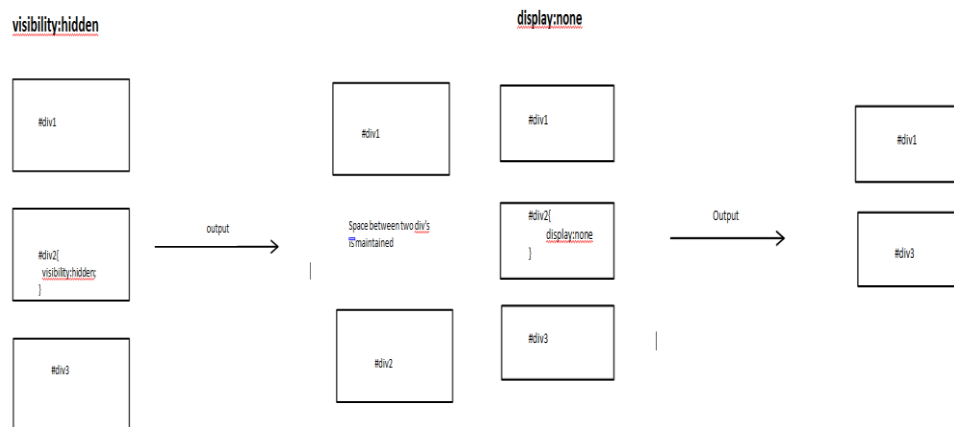


```
<div>  
  I don't have to use clear in this case. Nice!  
</div>
```

nav links another possible use case for inline-block display

Visibility vs Display

- ▶ Setting **display** to **none** will render the page as though the element does not exist.
- ▶ **visibility: hidden;** will hide the element, but the element will still take up the space it would if it was fully visible.



[display: none vs visibility: hidden](#)



Opacity

- ▶ The **opacity** property sets the opacity level for an element.
- ▶ Value ranges from 1.0 (opaque) to 0.0 (transparent)
- ▶ Example usage with :hover

```
div {  
  opacity: 0.5;  
}
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Phasellus imperdiet, nulla et dictum interdum, nisi lorem
egestas odio, vitae scelerisque enim ligula venenatis dolor.

Main Point: Block vs inline elements

A block-level element always starts on a new line and takes up the full width available (stretches out to the left and right as far as it can). An inline element does not start on a new line and only takes up as much width as necessary. These are the fundamental display types for almost all HTML elements. Understanding these fundamental types is critical to creating effective layouts.

Familiarity with fundamental levels of awareness is critical to successful action.

Main Point preview

Static position flows box elements from top to bottom, and inline elements from left to right. **Relative** position keeps the space in the original flow but displays the element at an offset. **Absolute** position takes the element out of the flow and places it relative to the "containing element". **Fixed** position takes the element out of the flow and places it relative to the viewport.

*Layouts require understanding how parts fit into a larger whole.
The whole is greater than the sum of its parts.*



The position property

Property	Meaning	Values
position	Location of element on page	static : default position relative : offset from its normal static position absolute : at a particular offset within its containing element fixed : at a fixed location within the browser window
top, bottom, left, right	Offsets of element's edges	A size in px, pt, em or %

`position: static;`

- ▶ **static** is the default position value for all elements.
- ▶ An element with **position: static;** is not positioned in any special way.
- ▶ A static element is said to be not positioned and an element with its position set to anything else is said to be positioned.

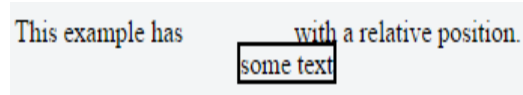


position: relative;

- ▶ Set the location of an element to an offset from its normal static position.
- ▶ **relative** behaves the same as **static** unless you add some extra properties. Setting the **top**, **right**, **bottom**, and **left** properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element. **relative** element stays in its place!

```
<p> This example has <span id="lifted">some text</span> with  
a relative position. </p>
```

```
#lifted {  
  position: relative;  
  left: 2em;  
  top: 1em;  
  border: 2px solid black;  
}
```



This example has some text with a relative position.

See example: lesson3_examples/position-relative.html



position: absolute;

- ▶ Elements that are positioned relatively are still considered to be in the normal flow of elements in the document.
- ▶ In contrast, an element that is positioned absolutely is taken out of the flow and thus takes up no space when placing other elements.
- ▶ The absolutely positioned element is positioned relative to *nearest **positioned** ancestor (non-static)*. If a positioned ancestor doesn't exist, the initial container is used.

```
#menubar{  
  width: 100px;  
  height: 50px;  
  position: absolute;  
  top: 20px;  
  left: 50px;  
}
```

See example:

[lesson3_examples/position-absolute.html](#)

[lesson3_examples/position-absoluteblock.html](#)



position: fixed;

- ▶ Fixed positioning is similar to absolute positioning, with the exception that the element's containing block is the viewport. This is often used to create a floating element that stays in the same position even after scrolling the page.

```
#one {  
  position: fixed;  
  top: 80px;  
  left: 10px;  
}
```

See example:
[lesson3_examples/position-fixed.html](#)

- ▶ A fixed element does not leave a gap in the page where it would normally have been located.
- ▶ A fixed element loses its space in the flow
- ▶ A fixed element does not move when you scroll (stays in place)



Overlapping Elements

- ▶ When elements are positioned, they can overlap other elements.
- ▶ The z-index property specifies the stack order of an element.
- ▶ An element can have a positive or negative stack order.

Main Point

Static position flows box elements from top to bottom, and inline elements from left to right. **Relative** position keeps the space in the original flow but displays the element at an offset. **Absolute** position takes the element out of the flow and places it relative to the "containing element". **Fixed** position takes the element out of the flow and places it relative to the viewport.

*Layouts require understanding how parts fit into a larger whole.
The whole is greater than the sum of its parts.*

Main Point preview

- ▶ The CSS float property makes its element move to right or left side of the containing box. The clear property moves its element downwards if there is a floating element on the specified side. Float is a convenient way to have text wrap around an element or make something appear on the right or left side
- ▶ *Do less and accomplish more.*

float



Float has several important uses for layout

- ▶ One is for wrapping text around images.
- ▶ Another is to position elements on the left or right or center, and possibly columns (being replaced by Flexbox and grid)

```
img {  
  float: right;  
  margin: 0 0 1em 1em;  
}
```

- When we float a `div` element it will comply to `width` and `height` properties rather than behaving like a block element (does not take whole width)
- A floating element is removed from normal document flow.
- Underlying text wraps around it as necessary.
- See example:
lesson3_examples/floatrightimage.html

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Phasellus imperdiet, nulla et dictum interdum, nisi lorem egestas odio, vitae scelerisque enim ligula venenatis dolor. Maecenas nisl est, ultrices nec congue eget, auctor vitae massa. Fusce luctus vestibulum augue ut aliquet. Mauris ante ligula, facilisis sed ornare eu, lobortis in odio. Praesent convallis urna a lacus



Common float bug: missing width



- ▶ often floating block elements must have a width property value
 - ▶ if no width is specified, the floating element will size to fit its content. Large content may occupy 100% of the page width, so no content can wrap around it
 - ▶ See example: lesson3_examples/floatingwithoutwidth.html

The clear property

(Stop the float I want to get off!)



```
img.hoveringicon { float: left; margin-right: 1em; }  
h2 { clear: left; background-color: yellow; }  
p { background-color: fuchsia; }
```



Starhome Sprinter is a Flash animated Internet cartoon. It mixes surreal humour with references to 1980s and 1990s pop culture, notably video games, classic television and popular music.

My Starhome Sprinter Fan Site

See example: `lesson3_examples/clear.html`

Property	Meaning	Values
clear	Whether to move this element below any prior floating elements in the document	left, right, both, none(default)

Common error: container too short



- ▶ If you place a tall *floating element* inside a block element without much other content, the floating element may hang down past the bottom edge of the block element that contains it.



Starhome Sprinter is a Flash animated Internet cartoon. It mixes surreal humour with references to 1980s and 1990s pop culture, notably video games, classic television and popular music.

- ▶ See example:
lesson3_examples/commonerrorcontenttooshort.html

The overflow property

overflow: hidden

2 scenarios for use of overflow

- ▶ *Floating elements* that are too tall for containing block
 - ▶ Previous slide
 - ▶ `lesson3_examples/overflow.html`
- ▶ Blocks of fixed height that have more content than their space allows
 - ▶ https://www.w3schools.com/css/tryit.asp?filename=trycss_overflow_visible

Property	Meaning	Values
overflow	Action to take if element's content is larger than the element itself	visible(default), hidden, scroll, auto



Multi-column layouts

- ▶ When more than one element floats in the same direction, they stack horizontally

```
div, p { border: 2px solid black; }  
.column { float: right; width: 25%; }
```

```
<div class="column"> Lorem ipsum dolor sit amet,  
consectetuer adipiscing elit. Integer pretium dui  
sit amet felis. </div>
```

```
<div class="column"> Integer sit amet diam.  
Phasellus ultrices viverra velit. </div>
```

```
<div class="column"> Beware the Jabberwock, my son!  
The jaws that bite, the claws that catch! </div>
```

See example: lesson3_examples/multicolumn-float.html



Multi-column

```
#columns {  
  column-count: 3;  
  column-gap: 40px;  
  column-rule: 2px dotted gray;}
```

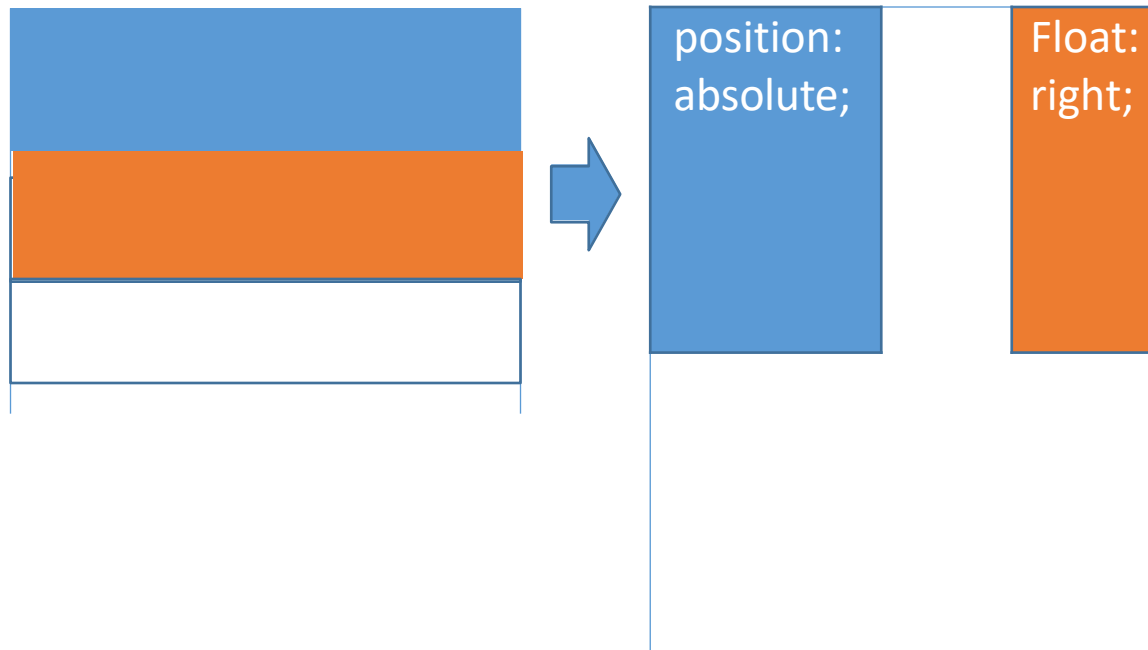
lesson3_examples/multicolumn-count.html

browser splits columns rather than split into smaller divs ourselves.

Property	Description	Values
column-count	Number of columns to use	an integer
column-fill	How to choose columns' size	balance(default), auto
column-gap	Space between columns	a size (px, pt, %, em)
column-rule	Vertical line between columns	a width, style and color
column-span	Lets element span many columns	1 (default) or all
column-width	width of each column	a size (px, pt, %, em)

Block elements behavior with **position/float**

- ▶ One thing to remember, that once a block element is positioned as **fixed** or **absolute**, it will ONLY occupy the space of its content rather than taking the whole width space.
- ▶ Same thing applies for block elements with **float**.



Alignment vs. float vs. position

1. if possible, lay out an element by aligning its content
 - ▶ horizontal alignment: text-align
 - ▶ set this on a block element; it aligns the content (text and inline elements) within it (not the block element itself)
 - ▶ E.g., see lesson3_examples/textalign.html example
 - ▶ Might need to make block elements have display: inline
 - ▶ vertical alignment: vertical-align
 - ▶ set this on an inline element, and it aligns it vertically within its containing element
2. if alignment won't work, try floating the element
3. if floating won't work, try positioning the element
 - ▶ absolute/fixed positioning are a last resort and should not be overused
4. OR, use flexbox or grid instead of float and positioning

Main Point

- ▶ The CSS float property makes its element move to right or left side of the containing box. The clear property moves its element downwards if there is a floating element on the specified side. Float is a convenient way to have text wrap around an element or make something appear on the right or left side
- ▶ *Do less and accomplish more.*

Main Point Preview Responsive Design

- ▶ **Responsive design** is the strategy of making a site that responds to the browser and device width. Responsive design utilizes media queries to determine the available display area and new CSS techniques such as flexbox and grid to make designs more flexible.
- ▶ *The unified field is the source of all possibilities and when we think from this level our actions are spontaneously responsive to whatever situation we encounter.*

Responsiveness guidelines

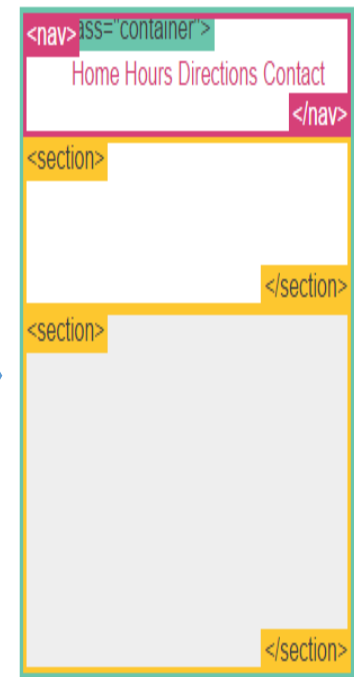
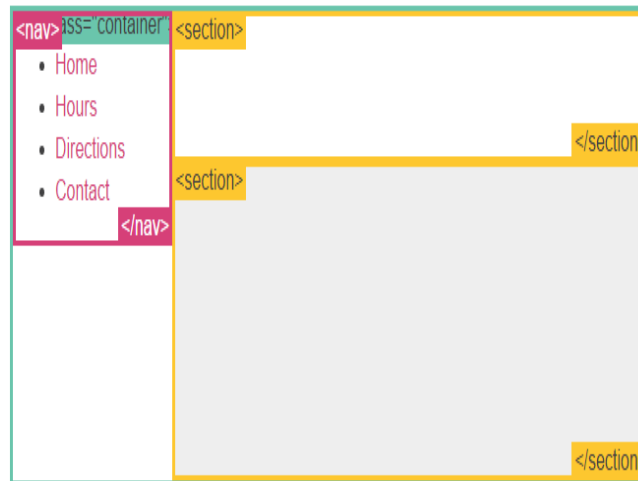
- ▶ (Mobile) users scroll websites vertically not horizontally!
- ▶ if forced to scroll horizontally, or zoom out it results in a poor user experience.
- ▶ Some additional rules to follow:
 - ▶ 1. Do NOT use large fixed width elements
 - ▶ 2. Do NOT let the content rely on a particular viewport width to render well
 - ▶ 3. Use CSS media queries to apply different styling for small and large screens
 - ▶ https://www.w3schools.com/css/css_rwd_viewport.asp

Media Queries

- Media queries specify a screen size for which a set of CSS rules apply

```
@media screen and (min-width:600px) {
  nav { float: left; width: 25%; }
  section { margin-left: 25%; }
}

@media screen and (max-width:599px) {
  nav li { display: inline; }
}
```



- See [lesson3_examples\mediaquery.html](#)

Design for Mobile First

- ▶ Mobile First means designing for mobile before designing for desktop or any other device
- ▶ Instead of changing styles when the width gets smaller than 768px, we should change the design when the width gets larger than 768px.

`/* applies to any screen 600px or smaller -- avoid this */
@media only screen and (max-width: 600px)`

`/* applies to any screen 600px or larger -- favor this */
@media only screen and (min-width: 600px)`

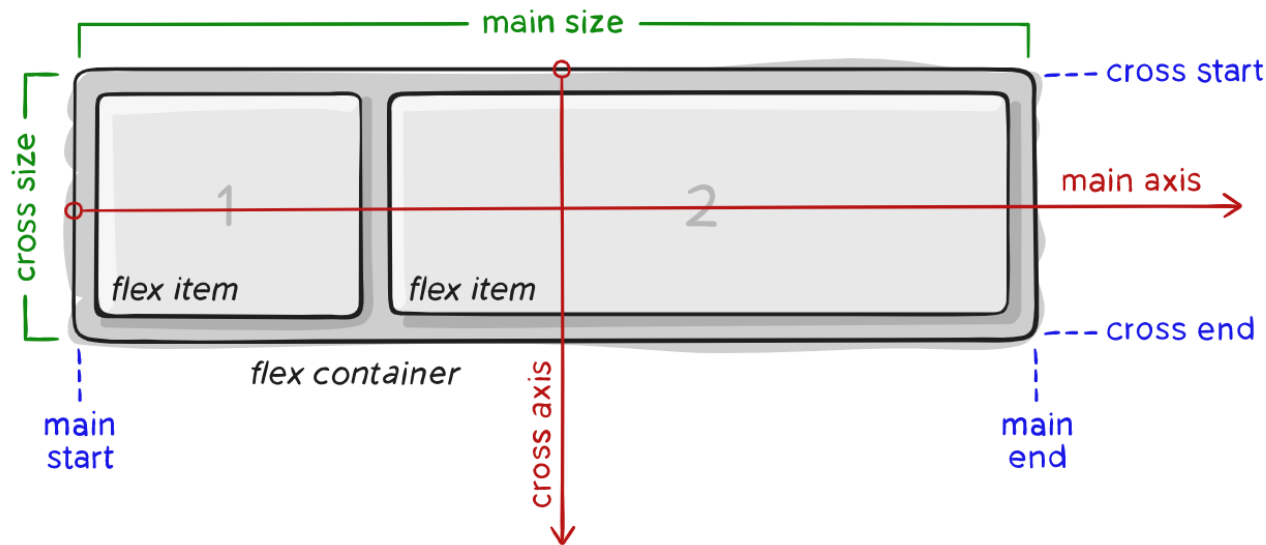


meta viewport

- ▶ Pages optimized to display well on mobile devices should include [meta viewport](#) in head
 - ▶ `<meta name=viewport content="width=device-width, initial-scale=1">`
 - ▶ gives browser instructions to control the page's dimensions and scaling
- ▶ Narrow screen devices render pages in a virtual window or viewport
 - ▶ usually wider than the screen,
 - ▶ mobile screen width of 640px, virtual viewport of 980px
 - ▶ Users pan and zoom to see different areas of the page
 - ▶ Or, browser might shrink the rendered result to fit into the 640px space
 - ▶ not good for pages using [media queries](#)
 - ▶ if the virtual viewport is 980px, queries at 640px or 480px never used
- ▶ viewport meta tag lets web developers control the viewport's size and scale.
 - ▶ width property controls the size of the viewport.
 - ▶ specific number of pixels like width=600
 - ▶ device-width, which is the width of the screen
 - ▶ [View from tablet or phone](#)
 - ▶ [Without viewport tag](#)
 - ▶ [With viewport tag](#)

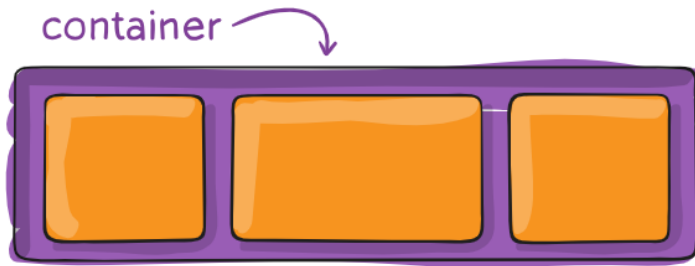
Flexbox Layout

- ▶ 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.
- ▶ Flexbox is for one dimensional layout (row or column).
- ▶ **set display property to flex** on the containing element
 - ▶ `display: flex;`
- ▶ direct child elements are automatically flexible items



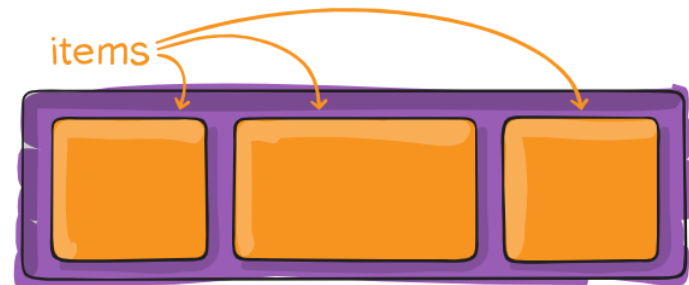
Flexbox properties

Properties for the Parent (flex container)



- ▶ `display: flex;`
- ▶ `flex-direction: row | row-reverse | column | column-reverse;`
- ▶ `flex-wrap: nowrap | wrap | wrap-reverse;`
- ▶ `flex-flow: <'flex-direction'> || <'flex-wrap'>`
- ▶ `justify-content: flex-start | flex-end | center | space-between | space-around | space-evenly;`
- ▶ `align-items: stretch | flex-start | flex-end | center | baseline;`
- ▶ `align-content: flex-start | flex-end | center | space-between | space-around | stretch;`

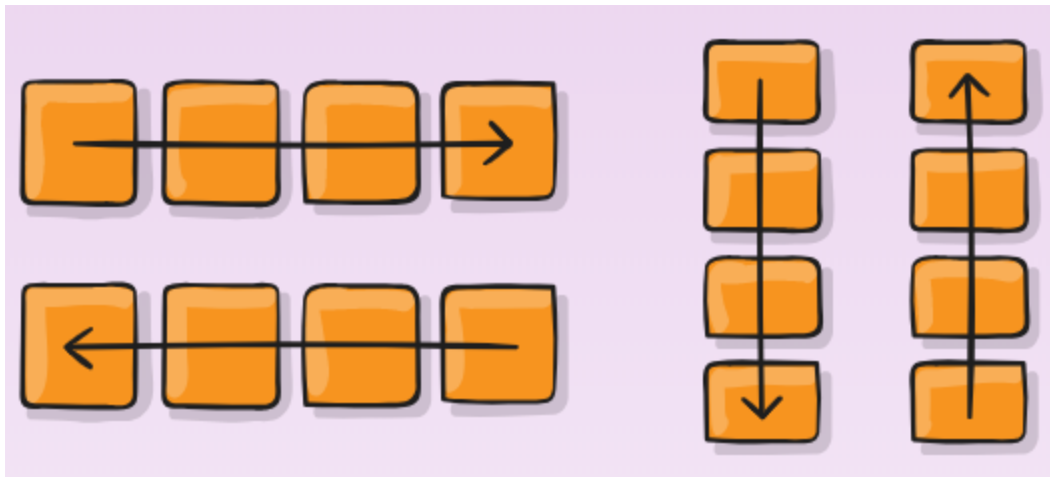
Properties for the Children (flex items)



- ▶ `order: <integer>; /* default is 0 */`
- ▶ `flex-grow: <number>; /* default 0 */`
- ▶ `flex-shrink: <number>; /* default 1 */`
- ▶ `flex-basis: <length> | auto; /* default auto */`
- ▶ `flex: none | [<'flex-grow'> <'flex-shrink'>? || <'flex-basis'>]`
- ▶ `align-self: auto | flex-start | flex-end | center | baseline | stretch;`

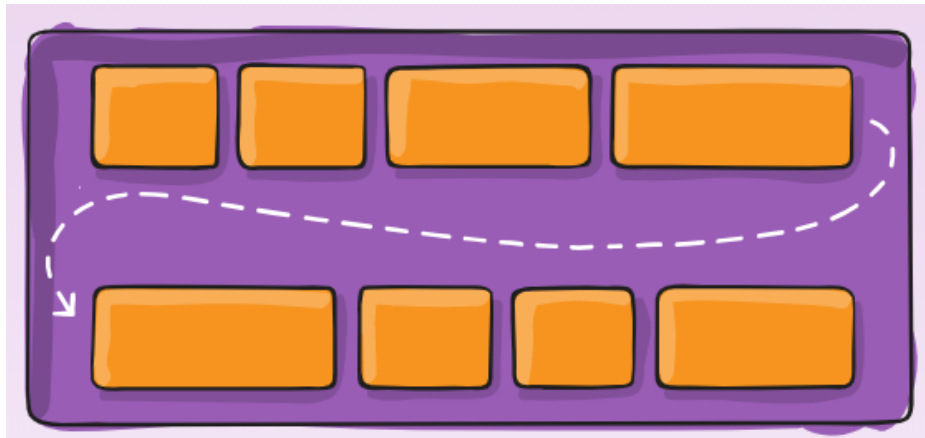
Flex container: **flex-direction**

- ▶ Specifies the direction of the flexible items inside a flex container
 - ▶ `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



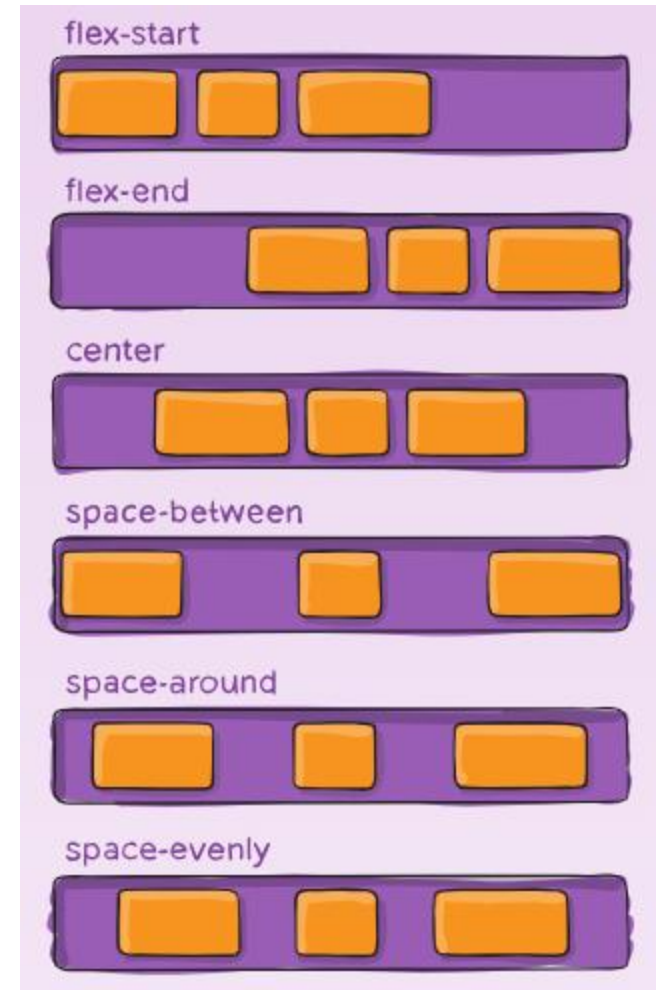
Flex container: **flex-wrap**

- ▶ Specifies whether the flex items should wrap or not, if there is not enough room for them on one flex line
- ▶ By default, flex items will all try to fit onto one line.
 - ▶ `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.



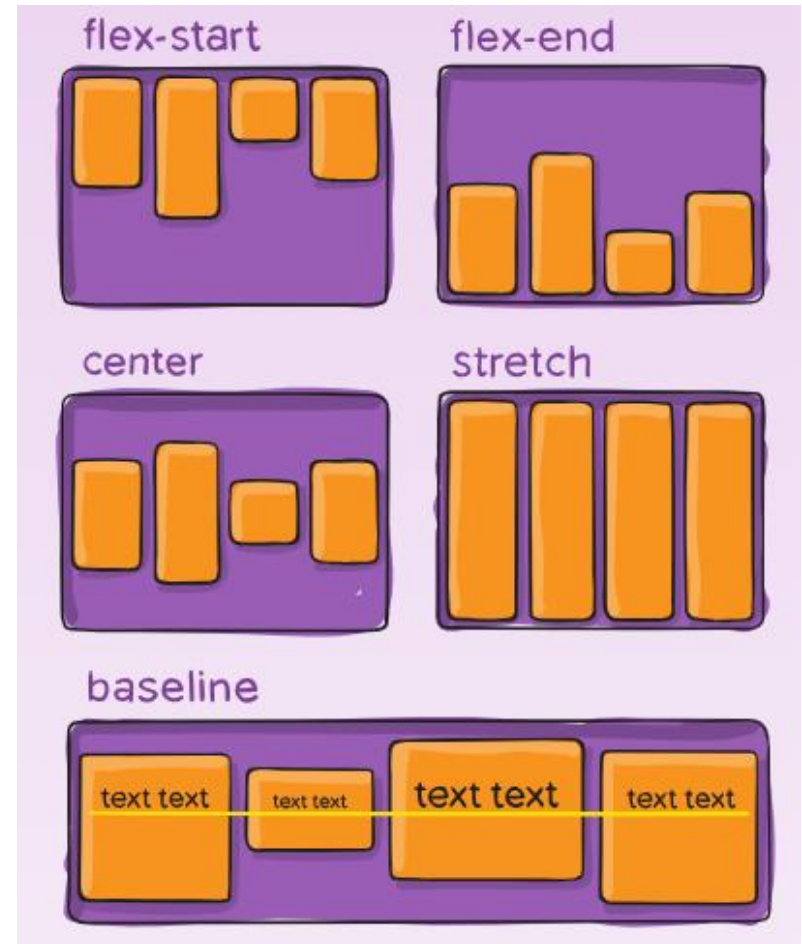
Flex container: **justify-content**

- ▶ Horizontally aligns the flex items when the items do not use all available space on the main-axis
 - ▶ `flex-start` (default): items are packed toward the start line
 - ▶ `flex-end`: items are packed toward the end line
 - ▶ `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.



Flex container: **align-items**

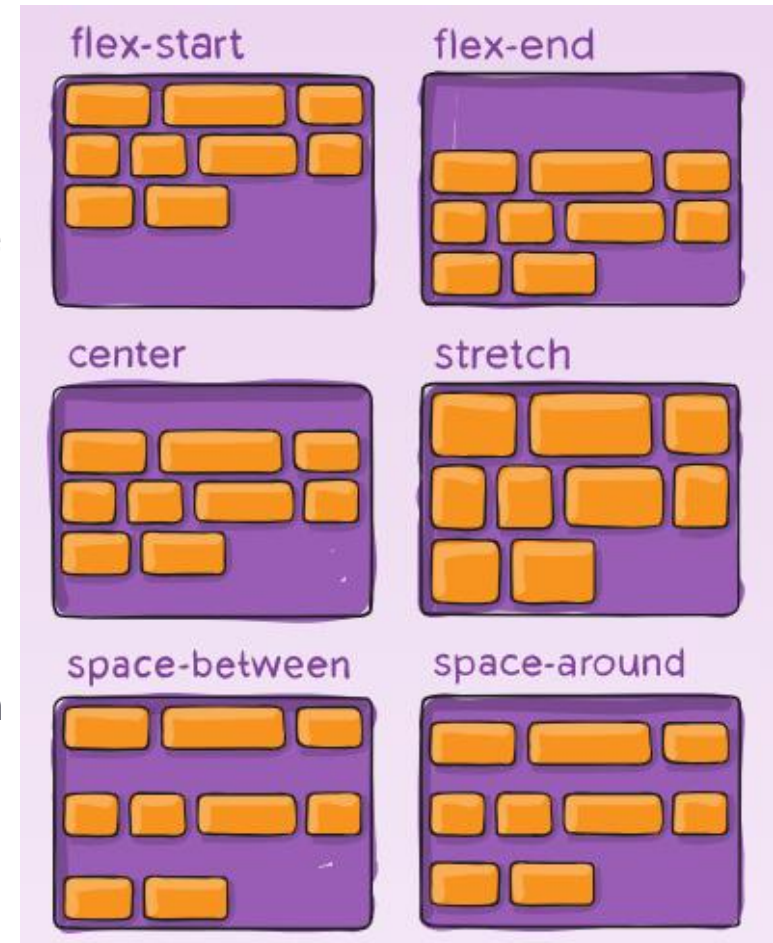
- ▶ Vertically aligns the flex items when the items do not use all available space on the cross-axis
 - ▶ `stretch` (default): stretch to fill the container (still respect min-width/max-width)
 - ▶ `flex-start`: cross-start margin edge of the items is placed on the cross-start line
 - ▶ `flex-end`: cross-end margin edge of the items is placed on the cross-end line
 - ▶ `center`: items are centered in the cross-axis
 - ▶ `baseline`: items are aligned such as their baselines align



Flex container: **align-content**



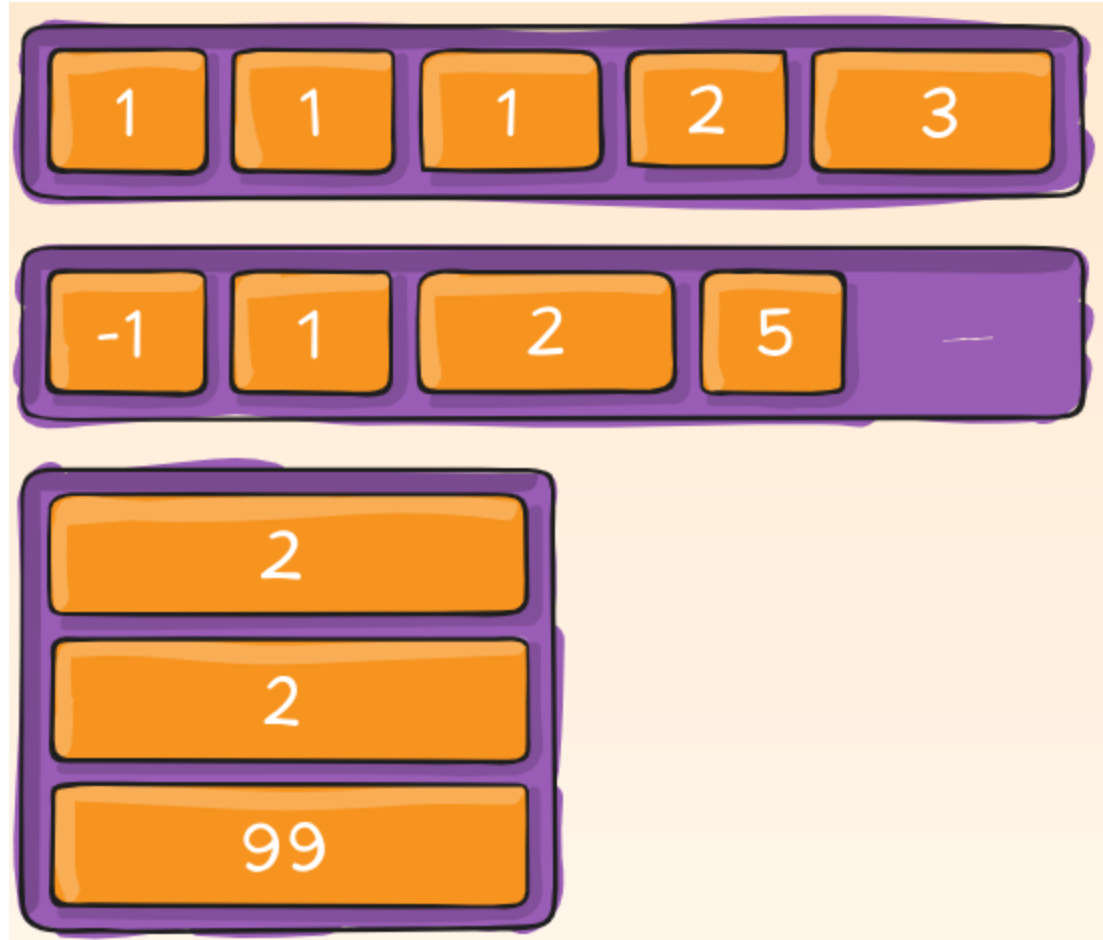
- ▶ Modifies the behavior of the flex-wrap property. It is similar to align-items, but instead of aligning flex items, it aligns flex lines
 - ▶ `flex-start`: lines packed to the start of the container
 - ▶ `flex-end`: lines packed to the end of the container
 - ▶ `center`: lines packed to the center of the container
 - ▶ `space-between`: lines evenly distributed; the first line is at the start of the container while the last one is at the end
 - ▶ `space-around`: lines evenly distributed with equal space around each line
 - ▶ `stretch` (default): lines stretch to take up the remaining space



Flex items: **order**

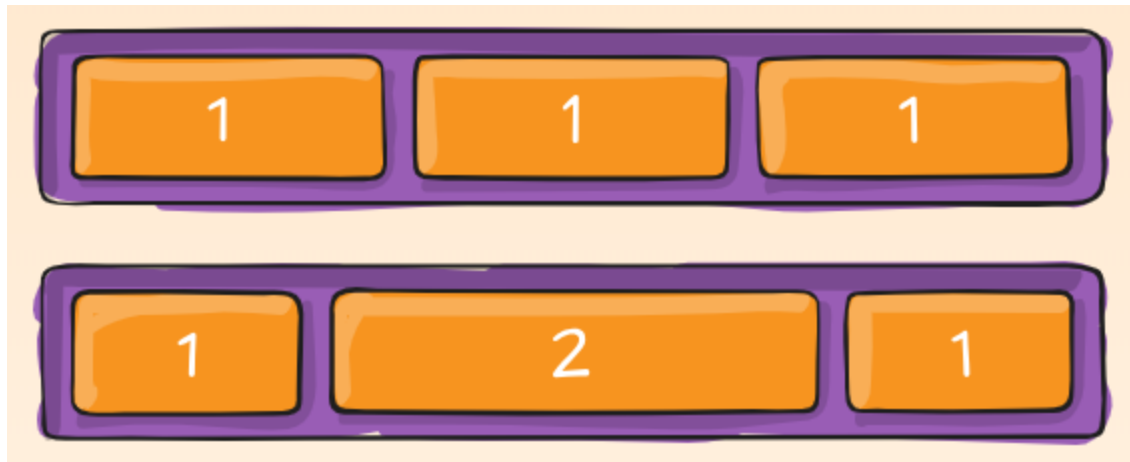
- ▶ By default, flex items are laid out in the source order. However, the `order` property controls the order in which they appear in the flex container.

- ▶ `order: <integer>;`
/ default is 0 */*



Flex items: **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.
- ▶ Negative numbers are invalid.
- ▶ `.item { flex-grow: <number>; /* default 0 */ }`



Flex items: **flex-shrink**, **flex-basis**, **flex**

- ▶ **flex-shrink**: This defines the ability for a flex item to shrink if necessary.

- ▶ `.item { flex-shrink: <number>; /* default 1 */ }`

- ▶ **flex-basis**: This defines the default size of an element before the remaining space is distributed.

- ▶ `.item { flex-basis: <length> | auto; /* default auto */ }`

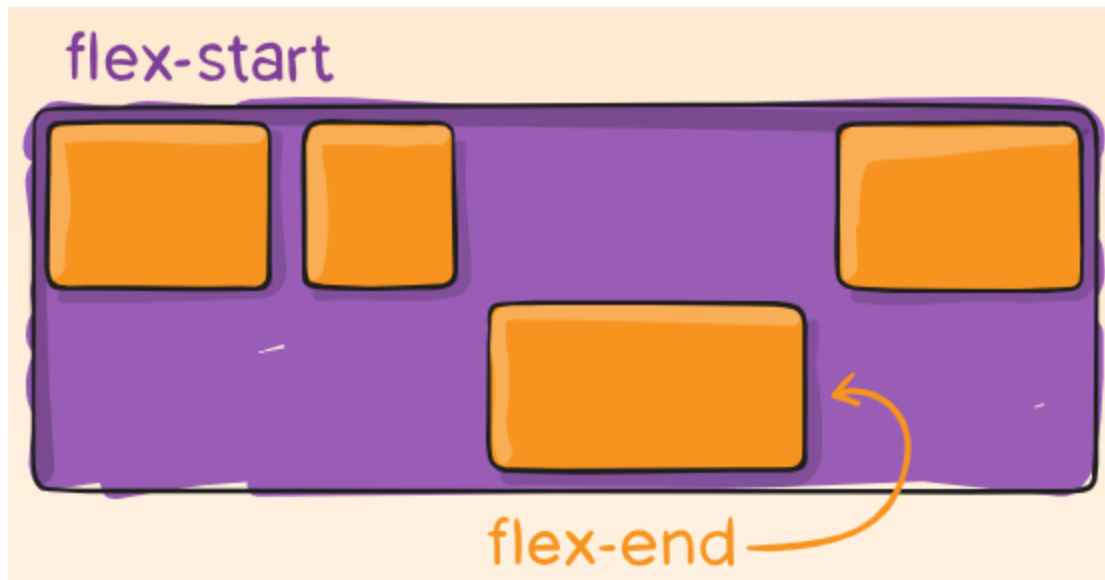
- ▶ **flex**: This is the shorthand for flex-grow, flex-shrink and flex-basis combined.

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

Flex items: **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; }`



Grid Layout

- ▶ CSS Grid offers a grid-based layout system, with rows and columns, making it easier to design web pages without having to use floats and positioning.
- ▶ A Grid Layout must have a parent element with the display property set to `grid`
- ▶ Direct child element(s) of the grid container automatically becomes grid items.
- ▶ CSS grid is for two dimensional page layout.
- ▶ **Grid is designed to be used with flexbox, not instead of it**

Terminology

- ▶ **Grid Container:** The element on which display: grid is applied. It's the direct parent of all the grid items.
- ▶ **Grid Item:** The direct children (i.e. *direct* descendants) of the grid container.
- ▶ **Grid Line:** The dividing lines that make up the structure of the grid.
- ▶ **Grid Track:** The space between two adjacent grid lines.
- ▶ **Grid Cell:** The space between two adjacent row and two adjacent column grid lines.
- ▶ **Grid Area:** The total space surrounded by four grid lines.

Grid Properties

Properties for the Parent (Grid Container)

- ▶ display
- ▶ grid-template-columns
- ▶ grid-template-rows
- ▶ grid-template-areas
- ▶ grid-template
- ▶ grid-column-gap
- ▶ grid-row-gap
- ▶ grid-gap
- ▶ justify-items
- ▶ align-items
- ▶ place-items
- ▶ justify-content
- ▶ align-content
- ▶ place-content
- ▶ grid-auto-columns
- ▶ grid-auto-rows
- ▶ grid-auto-flow
- ▶ grid

Properties for the Children (Grid Items)

- ▶ grid-column-start
- ▶ grid-column-end
- ▶ grid-row-start
- ▶ grid-row-end
- ▶ grid-column
- ▶ grid-row
- ▶ grid-area
- ▶ justify-self
- ▶ align-self
- ▶ place-self

Grid Container: 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 Container:

grid-template-columns

grid-template-rows

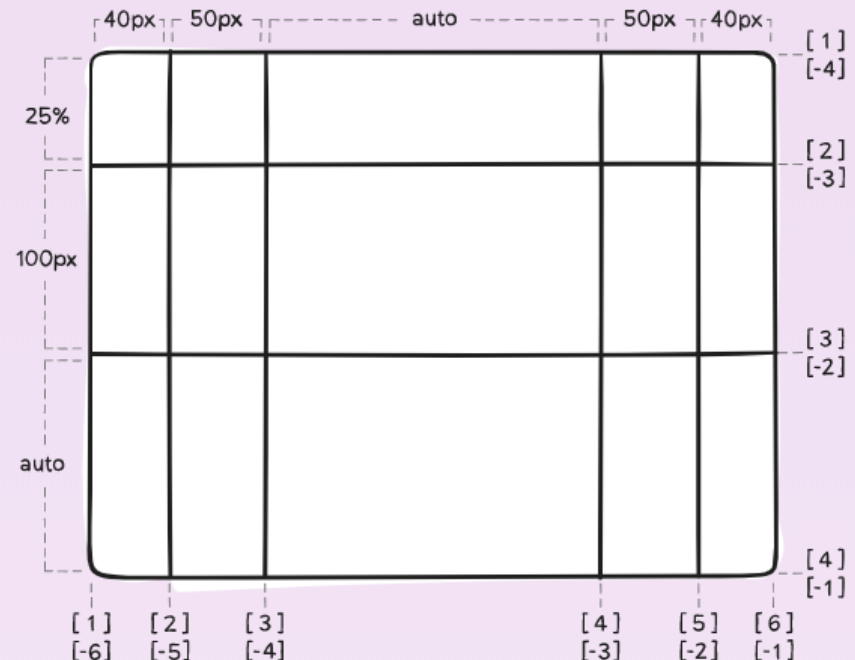


- Defines the columns and rows of the grid with a space-separated list of values. The values represent the track size, and the space between them represents the grid line.

```
.container {  
  grid-template-columns: <track-size>  
    ... | <line-name> <track-size> ...;  
  grid-template-rows: <track-size> ... |  
    <line-name> <track-size> ...;  
}  
  
.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];  
}
```

CSS

```
.container {  
  grid-template-columns: 40px 50px auto 50px 40px;  
  grid-template-rows: 25% 100px auto;  
}
```



Grid Container: grid-template-areas



- Defines a grid template by referencing the names of the grid areas which are specified with the grid-area property.

<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

CSS

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



The Fr Unit

▶ With CSS Grid Layout, we get a new flexible unit: the Fr unit. Fr is a fractional unit and 1fr is for 1 part of the available space.

```
.container {  
  display: grid;  
  
  grid-template-columns: 1fr 1fr 1fr 1fr;  
  grid-template-rows: 100px 200px 100px;  
  
  grid-template-areas:  
    "head head2 . side"  
    "main main2 . side"  
    "footer footer footer footer";  
}
```

The 4 columns each take up the same amount of space.



Grid Items:

`grid-column` `grid-row`

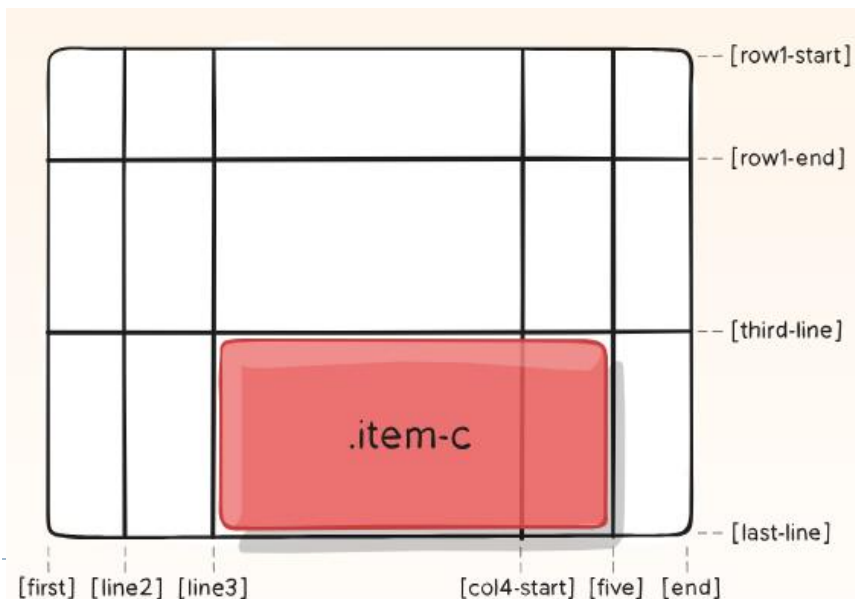


- ▶ `grid-column`: which columns an item covers
- ▶ `grid-row`: which rows an item covers
 - ▶ **Values:** `<start-line> / <end-line>` - each one accepts all the same values as the longhand version, including span

```
.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;  
}
```



CSS Frameworks

Because CSS layout is so tricky, there are CSS frameworks out there to help make it easier. Here are a few if you want to check them out. Using a framework is only a good idea if the framework really does what you need your site to do. They're no replacement for knowing how CSS works.

- [Blueprint](#)
- [Bootstrap](#)
- [Foundation](#)
- [SemanticUI](#)



Bootstrap grid system has responsive breakpoints

- ▶ create a row (`<div class="row">`).
 - ▶ add columns (tags with appropriate `.col-*-*` classes)
 - ▶ first star (*) represents the responsiveness: sm, md, lg or xl
 - ▶ second star represents a number, which should add up to 12 for each row

```
<div class="row">  
  <div class="col-sm-4">.col-sm-4</div>  
  <div class="col-sm-8">.col-sm-8</div>  
</div>
```

- ▶ col-lg, stack when the width is < 1200px.
- ▶ col-md, stack when the width is < 992px.
- ▶ col-sm, stack when the width is < 768px.
- ▶ col-xs, then the columns will never stack.

Bootstrap Examples

▶ Bootstrap table

- ▶ app specific modification to adjust the color of the striped rows
- ▶ important to understand css and not just blindly use bootstrap

▶ carousel

▶ basic bootstrap4 website

- ▶ header, nav, 2 col with links vs content..., footer

Main Point Responsive Design

- ▶ **Responsive Design** is the strategy of making a site that responds to the browser and device width. Responsive design utilizes media queries to determine the available display area and new CSS techniques such as flexbox and grid to make designs more flexible.
- ▶ *The unified field is the source of all possibilities and when we think from this level our actions are spontaneously responsive to whatever situation we encounter.*

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

CSS Layout: Whole Is Greater than the Sum of the Parts

1. You can use floats and positioning to change where elements are displayed.
 2. Modern web apps use responsive design principles including media queries, viewport, Flexbox, grid, and frameworks such as Bootstrap
-
3. **Transcendental consciousness** is the experience of pure wholeness.
 4. **Impulses within the Transcendental field:** At quiet levels of awareness thoughts are fully supported by the wholeness of pure consciousness.
 5. **Wholeness moving within itself:** In Unity Consciousness, one appreciates all parts in terms of their ultimate reality in wholeness.

