**Task 1: -**

**Make these buttons using HTML CSS**

A close-up of a sign

Description automatically generated with medium confidence

Basic:-

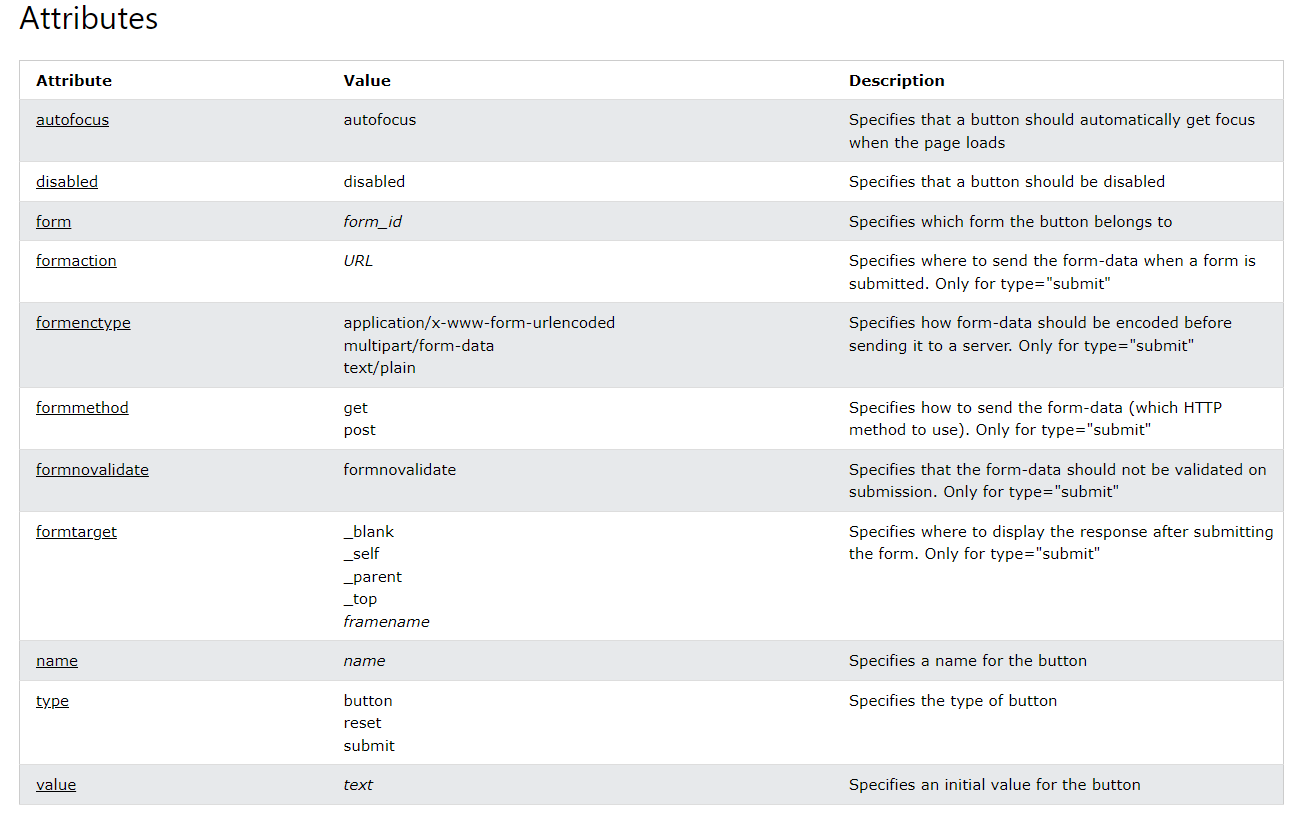
Definition and Usage

The <button> tag defines a clickable button.

Inside a <button> element you can put text (and tags like <i>, <b>, <strong>, <br>, <img>, etc.). That is not possible with a button created with the [<input>](https://www.w3schools.com/tags/tag_input.asp) element!

**Tip:** Always specify the type attribute for a <button> element, to tell browsers what type of button it is.

**Tip:** You can easily style buttons with CSS! Look at the examples below or visit our [CSS Buttons](https://www.w3schools.com/css/css3_buttons.asp) tutorial.

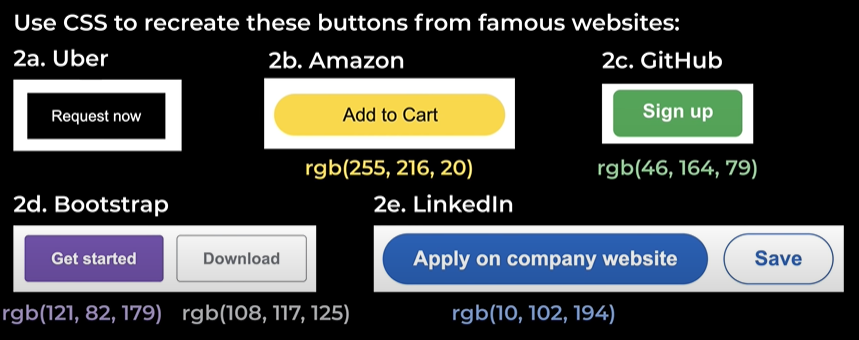


Result:

A picture containing text, font, logo, screenshot

Description automatically generated

Task 2:



Result:-

A picture containing text, screenshot, font, logo

Description automatically generated

**Hover – Transitions – Shadows**

.tweet{

    background-color: rgb(24, 157, 235) ;

    color: white;

    border: none;

    height: 35px;

    width: 65px;

    border-radius: 32px;

    margin: 5px;

Transition: property\_name duration,

Another\_property\_name duration;

    font-weight: bold;

    cursor: pointer;

    transition: box-shadow 0.15s;

}

For adding shadow

.tweet:hover{

    box-shadow: 5px 5px 10px rgba(0,0,0, 0.15);

}

.tweet:active{

When Button is clicked!

    background-color: rgb(10, 115, 177) ;

}

**Void Element:-** A void element is an element whose content model never allows it to have contents under any circumstances. Void elements can have attributes.

Image, link, meta, doctype

Images:

.prac-img{

    width: 300px;

    height: 300px;

    border-width: 3px;

    border-color: red;

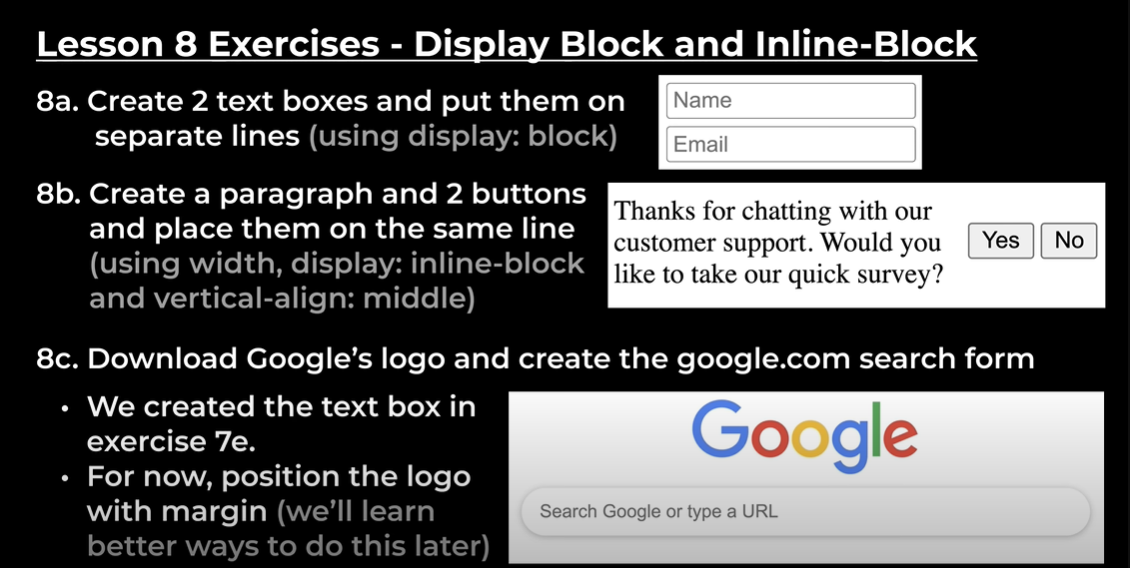
    border-style: solid;

    object-fit: contain;

    object-position: bottom;

}

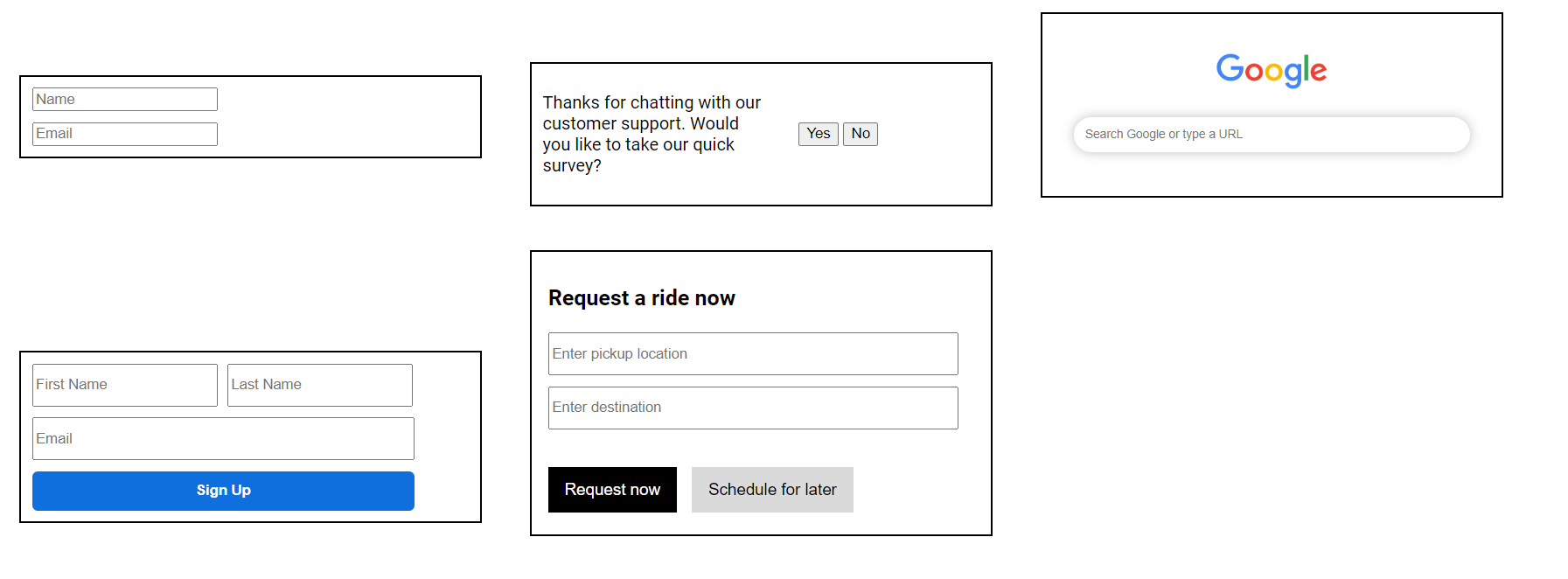
Exercise 8:



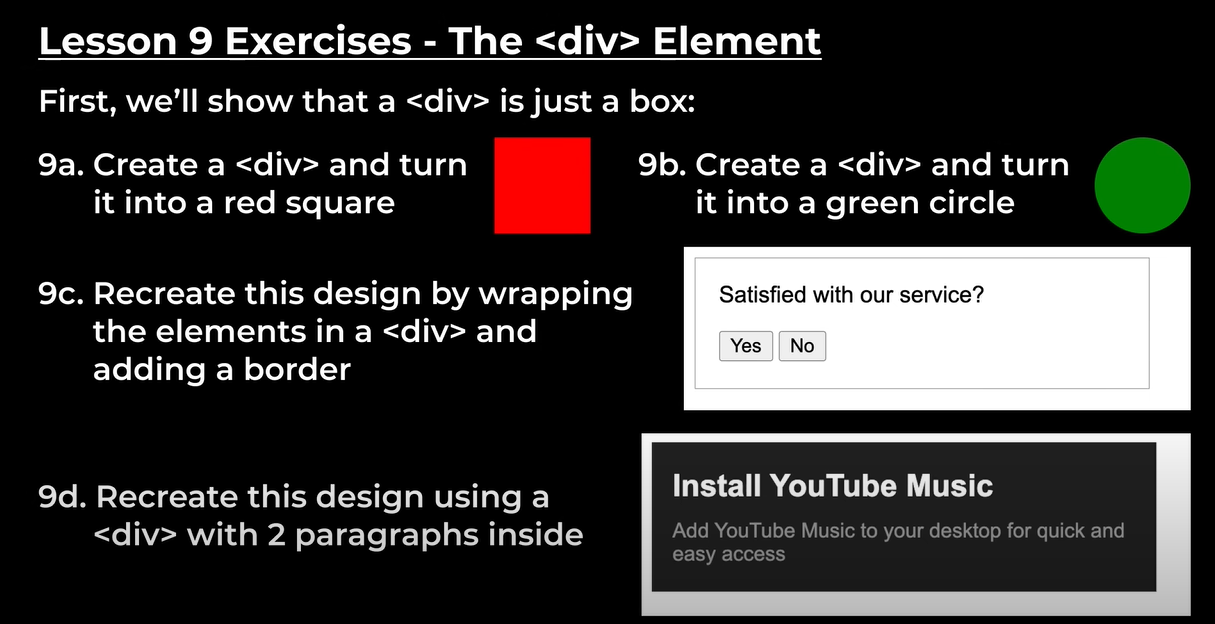
Screens screenshot of a computer screen

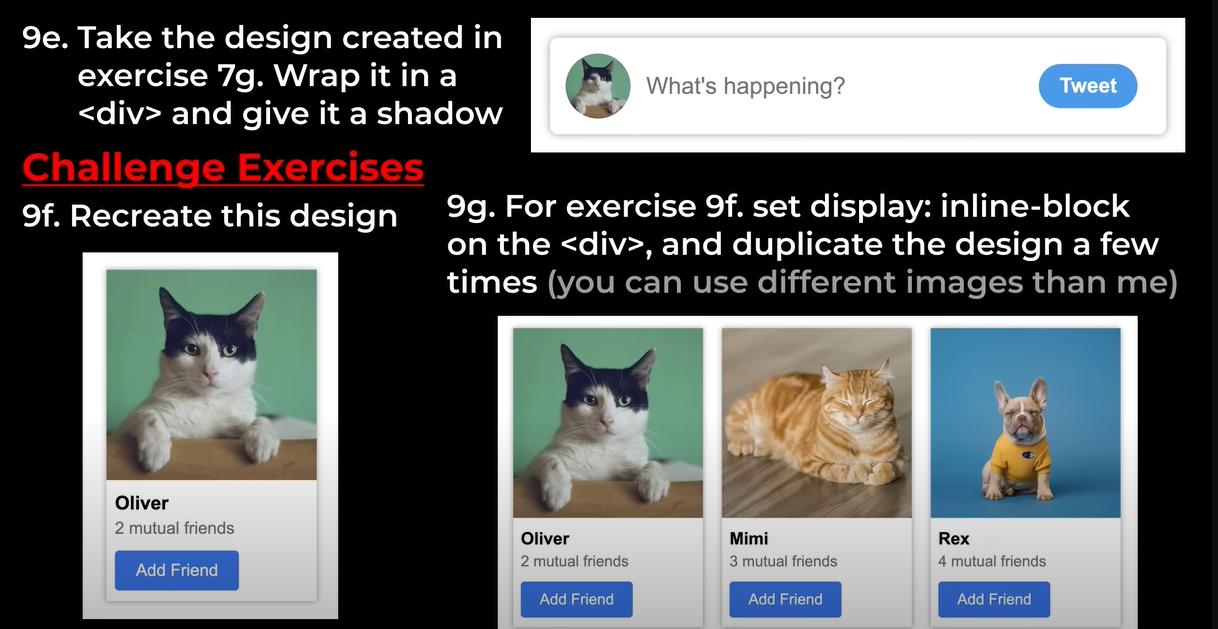
Description automatically generated with low confidence

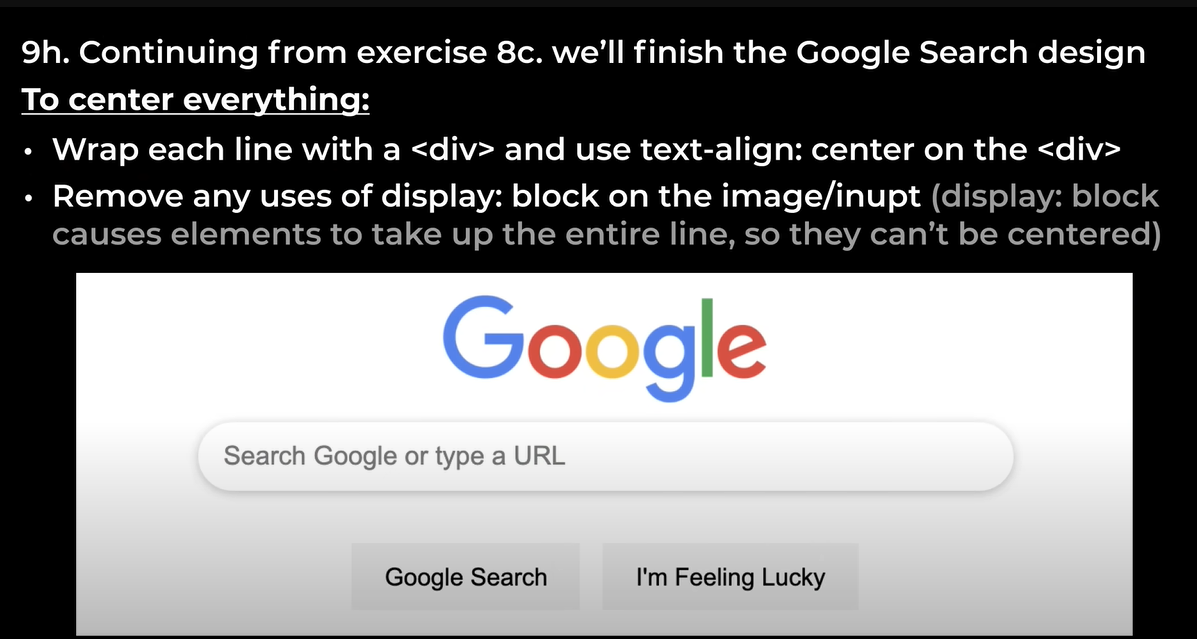
Solution:



Exercise 9:-







Answer:

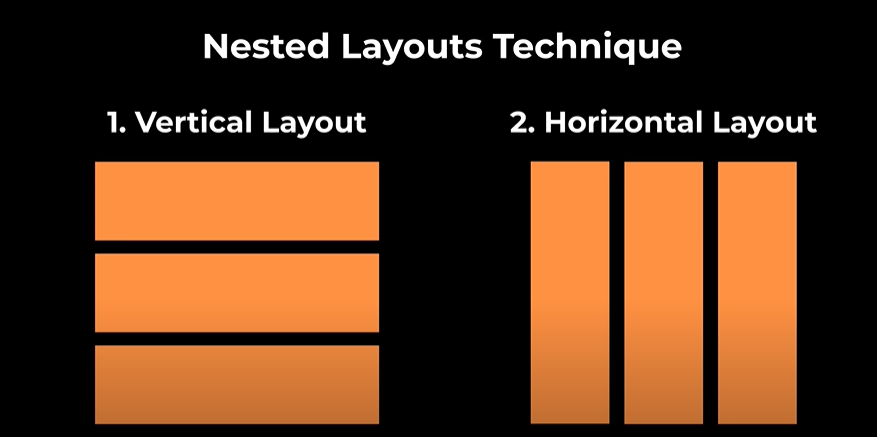
A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

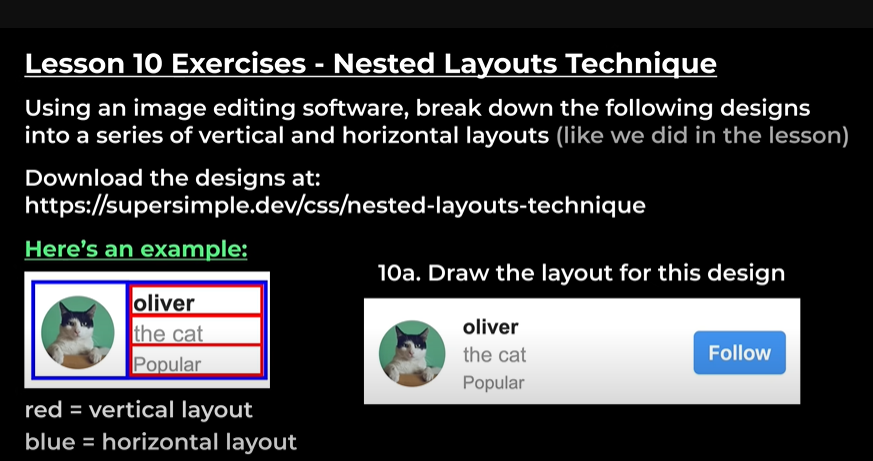
Description automatically generated with low confidence

Nested Layout Technique





Exercise 10:



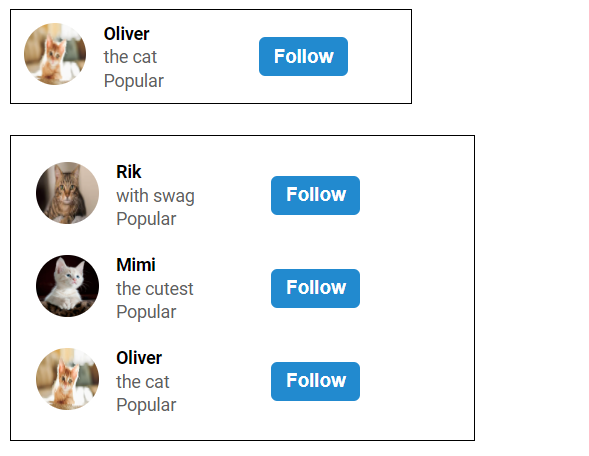
A screenshot of a video game

Description automatically generated

A screenshot of a cat and dog

Description automatically generated

Answer:-



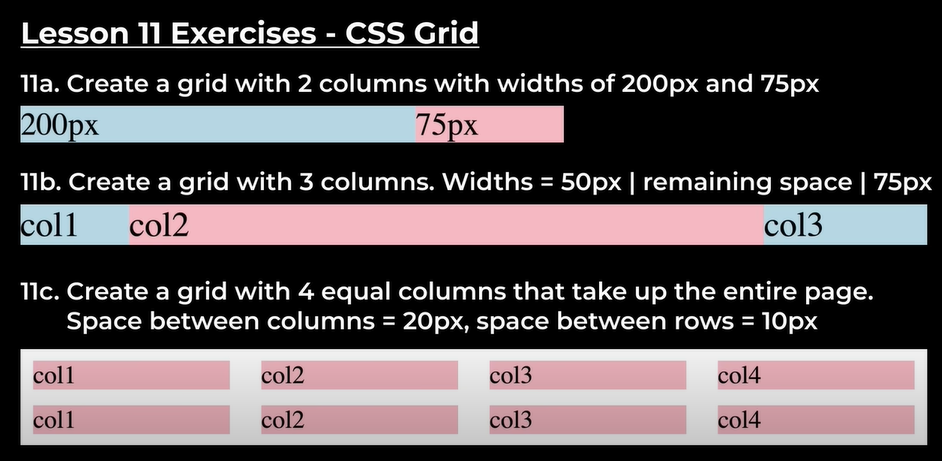
A screenshot of a social media post

Description automatically generated

A screenshot of a cat

Description automatically generated

Lesson 11:-



A screenshot of a computer

Description automatically generated

Answer:

A screenshot of a computer

Description automatically generated

A cat with blue eyes

Description automatically generated

 display: grid;

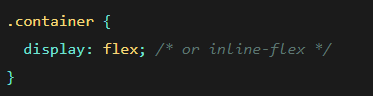
    grid-template-columns: 1fr 1fr 1fr;

    margin-top: 100px;

    column-gap: 80px;

    align-items: center;

Flexbox:



A black background with yellow and white text

Description automatically generated

**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.**

A black background with white text

Description automatically generated

**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.**

**A black screen with white text

Description automatically generated**

**A screenshot of a game

Description automatically generated**

.container {

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

}

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.

Note that that browser support for these values is nuanced. For example, space-between never got support from some versions of Edge, and start/end/left/right aren’t in Chrome yet. MDN has detailed charts. The safest values are flex-start, flex-end, and center.

There are also two additional keywords you can pair with these values: safe and unsafe. Using safe ensures that however you do this type of positioning, you can’t push an element such that it renders off-screen (e.g. off the top) in such a way the content can’t be scrolled too (called “data loss”).



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

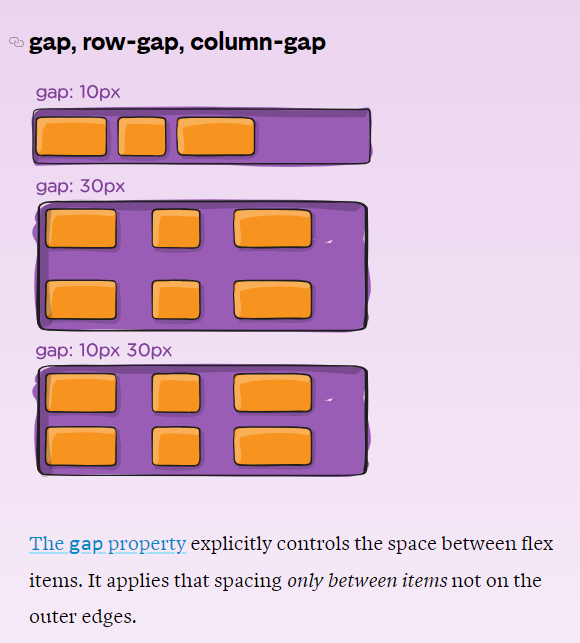
The safe and unsafe modifier keywords can be used in conjunction with all the rest of these keywords (although note browser support), and deal with helping you prevent aligning elements such that the content becomes inaccessible.

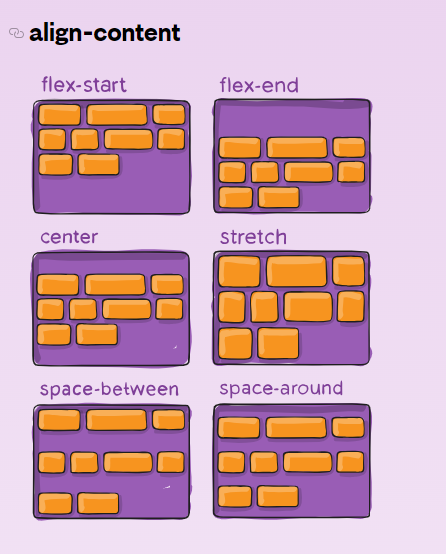
.container {

align-items**: stretch | flex-start | flex-end | center | baseline | first baseline | last baseline | start | end | self-start | self-end + ... safe | unsafe;**

}

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 (perpendicular to the main-axis).

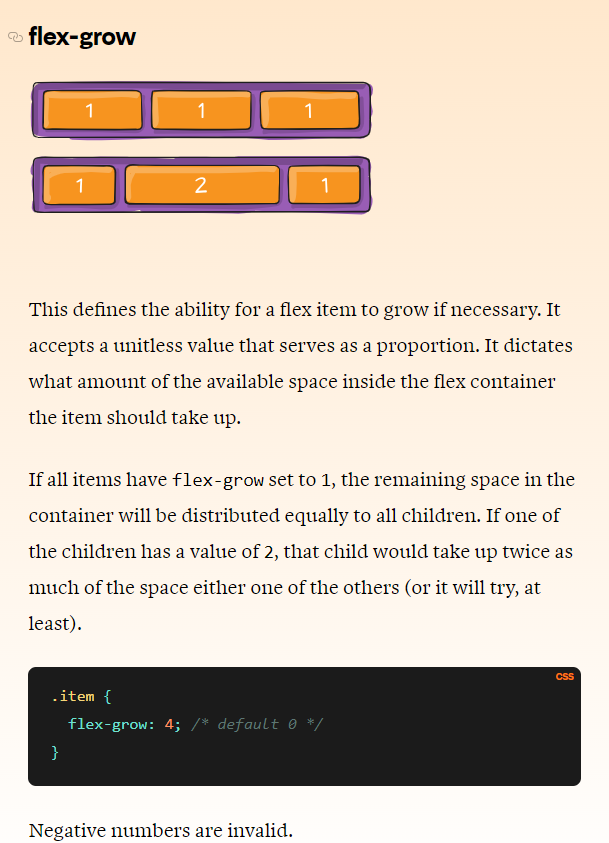
A white background with black text

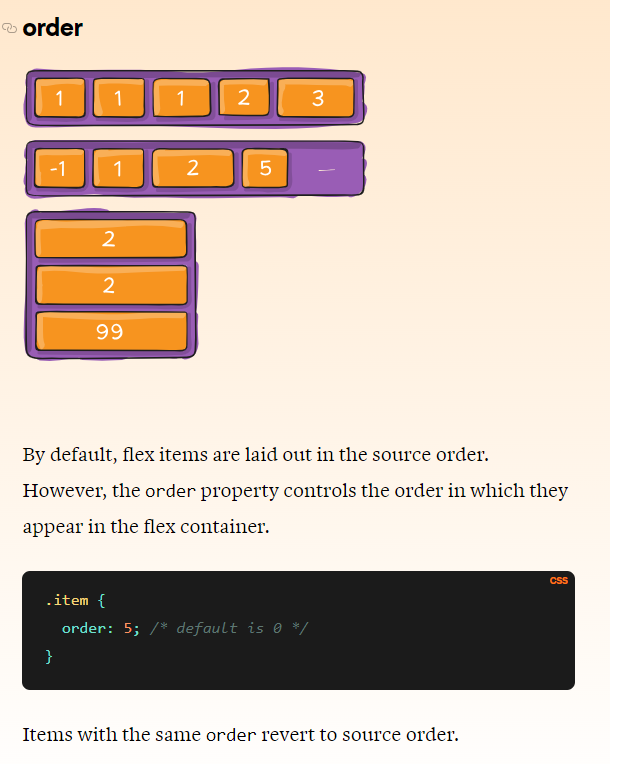
Description automatically generated

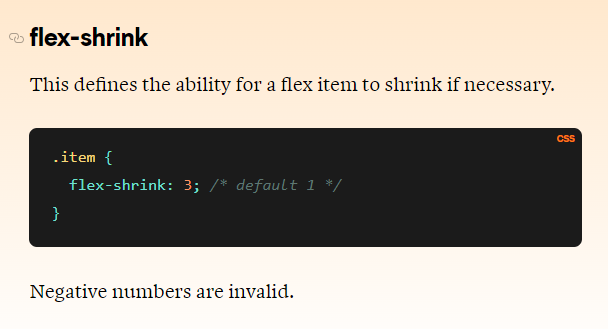
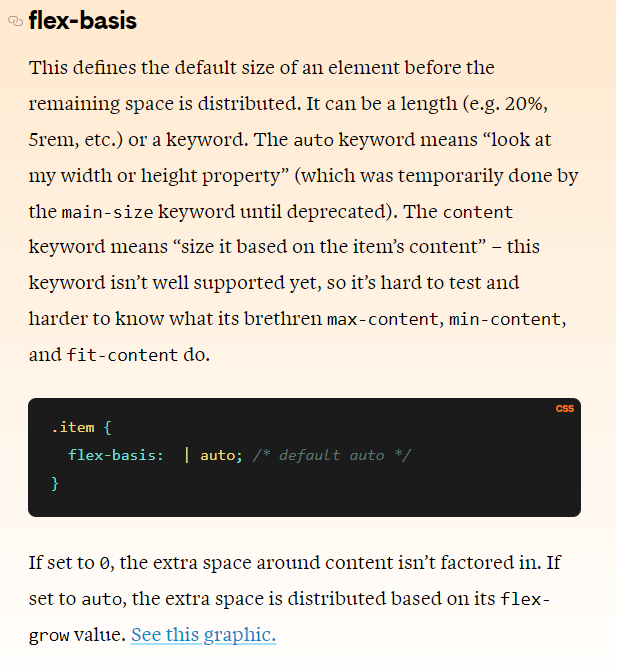
The behavior could be thought of as a minimum gutter, as if the gutter is bigger somehow (because of something like justify-content: space-between;) then the gap will only take effect if that space would end up smaller.

It is not exclusively for flexbox, gap works in grid and multi-column layout as well.

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.

Flex Items Properties:

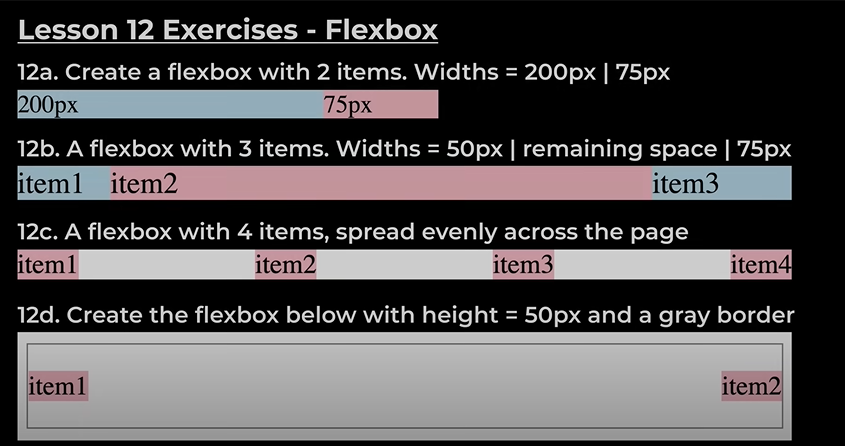




A screenshot of a computer

Description automatically generated

Exercise 12:



A screenshot of a video game

Description automatically generated

Result: -

