UCD DUBLIN PROFESSIONAL ACADEMY

Front-End Web Development

Unit 5: Making layouts using CSS

UCD DUBLIN PROFESSIONAL ACADEMY

Course Outline

1. Getting Started
2. HTML - Structuring the Web
3. CSS - Styling the Web
4. JavaScript - Dynamic client-side scripting
5. **CSS - Making Layouts**
6. Introduction to Websites/Web Applications
7. CSS - Advanced
8. JavaScript - Modifying the Document Object Model (DOM)
9. Dynamic HTML
10. Web Forms - Working with user data
11. JavaScript - Advanced
12. Building a Web Application with JavaScript
13. Introduction to CSS Frameworks – Bootstrap
14. Building a Web Application with Svelte
15. SEO, Web security, Performance
16. Walkthrough project

Course Learning Outcomes

- Competently write HTML and CSS code
- Create web page layouts according to requirements using styles
- Add interactivity to a web page with JavaScript
- Access and display third-party data on the web page
- Leverage Bootstrap and Static Site Generator

Assessment

- Final Project - 100% of the grade

  - Design and Build functioning Website using HTML5, CSS (including Bootstrap), JavaScript (browser only)

✔ Code will be managed in GitHub
✔ Website will be deployed to GitHub Pages
✔ All code to follow best practice and be documented

- Details and How-To-Guide are available on the course page under the section called Assessments

5. CSS - Making Layouts

In This Unit

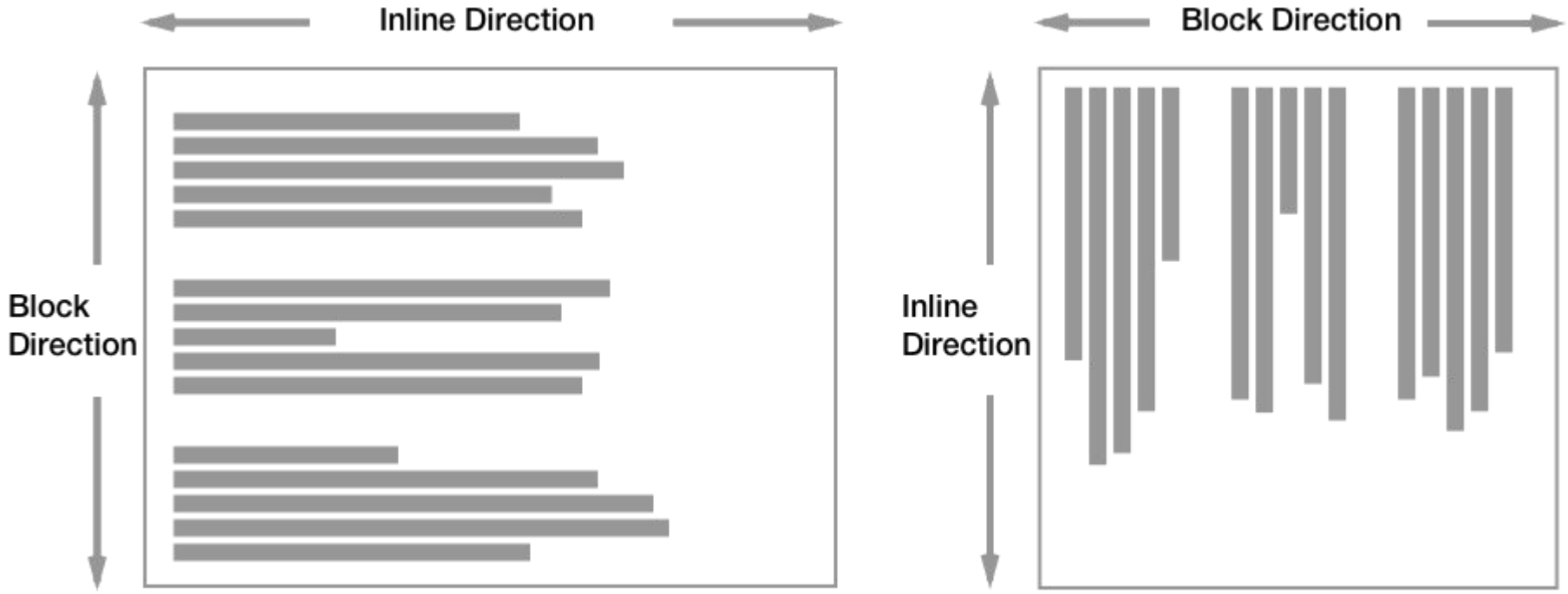
Title
Introduction to Layouts
Advanced Layouts
Multi - Device Layouts

Introduction to Layouts:

- display,
- Float,
- box-sizing,
- column

Normal Flow

- If you take an HTML webpage which has no CSS applied to change the layout, the elements will display in normal flow
- In normal flow, boxes are displayed one after another based on the Writing Mode of the document
- This means that if you have a horizontal writing mode, one in which sentences run left to right or right to left, normal flow will display the boxes of block level elements one after the other vertically down the page



no Layout

- By default, the content width will be 100% of the browser window
- very annoying to read for wide browser windows
- web pages typically have multiple components such as
  - header
  - navigation menu sidebar(s)
  - main content area
  - footer
- Number of ways to address this issue:
  - display, position, float, columns
  - CSS flexbox
  - CSS grid

Example - HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8"><title>Website Layout</title>
  </head>
  <body>
    <header><h1>My Website</h1><p>Resize the browser window to see the effect.</p></header>
    <nav><ul><li><a  href="#">Link</a></li>...</ul><ul  class="right"><li><a  ref="#">Link</a></li></ul></nav>
    <main>
      <section class="card">
        <h2>TITLE HEADING</h2>
        <h5>Title description, Dec 7, 2017</h5>
        <div class="fakeimg 200">Image</div>
        <p>Some text..</p>
      </section>
      <section class="card">...</section>
    </main>
    <aside>
      <section class="card">...</section>
      ...
    </aside>
    <footer><h2>Footer</h2></footer>
  </body>
</html>
```

Example - Browser, without CSS

My Website

Resize the browser window to see the effect.

- Link
- Link
- Link

- Link

TITLE HEADING

Title description, Dec 7, 2017

Image

Some text..

Sunt in culpa qui officia deserunt mollit anim id est laborum consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco.

TITLE HEADING

Title description, Sep 2, 2017

Image

Some text..

Sunt in culpa qui officia deserunt mollit anim id est laborum consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco.

About Me

Image

Some text about me in culpa qui officia deserunt mollit anim..

Popular Post

Layout - Header

Add CSS to the various parts of the page

```
* {
    box-sizing: border-box;
}
body {
    font-family: Arial;
    padding: 10px;
    background: #f1f1f1;
}
/* Header/Blog Title */
header {
    padding: 30px;
    text-align: center;
    background: white;
}
header h1 {
    font-size: 50px;
}
```

Layout - Top Navigation

```
nav {
    overflow: hidden;
    background-color: #333;
}
nav ul { margin: 0%; }
nav ul,
nav li {
    display: inline-block;
}
nav .right { float: right; }
nav a {
    display: block;
    color: #f2f2f2;
    text-align: center;
    padding: 14px 16px;
    text-decoration: none;
}
/* Change color on hover */
nav a:hover {
    background-color: #ddd;
    color: black;
}
/* Clear floats after the columns */
.row::after {
    content: "";
    display: table;
    clear: both;
}
```

Layout - Main, Aside, Fake Image

```
/* Create two unequal columns that floats next to each other */
main {
    display: inline-block;
    float: left;
    width: 75%;
}
aside {
    display: inline-block;
    float: left;
    width: 25%;
    background-color: #f1f1f1;
    padding-left: 20px;
}

.fakeimg {
    background-color: #aaa;
    padding: 20px;
    margin: 10px 0;
}
.fakeimg.h100 { height: 100px; }
.fakeimg.h200 { height: 200px; }
```

Layout - Card, Footer

```
/* Add a card effect for articles */
.card {
    background-color: white;
    padding: 20px;
    margin-top: 20px;
}

/* Footer */
footer {
    padding: 20px;
    text-align: center;
    background: #ddd;
    margin-top: 20px;
}
```

Layout - Columns

The CSS multi-column layout allows easy definition of multiple columns of text - just like in newspapers

Property	Description
column-count	Specifies the number of columns an element should be divided into
column-fill	Specifies how to fill columns
column-gap	Specifies the gap between the columns
column-rule	A shorthand property for setting all the column-rule-* properties
column-rule-color	Specifies the color of the rule between columns
column-rule-style	Specifies the style of the rule between columns
column-rule-width	Specifies the width of the rule between columns
column-span	Specifies how many columns an element should span across
column-width	Specifies a suggested, optimal width for the columns

Layout - Columns

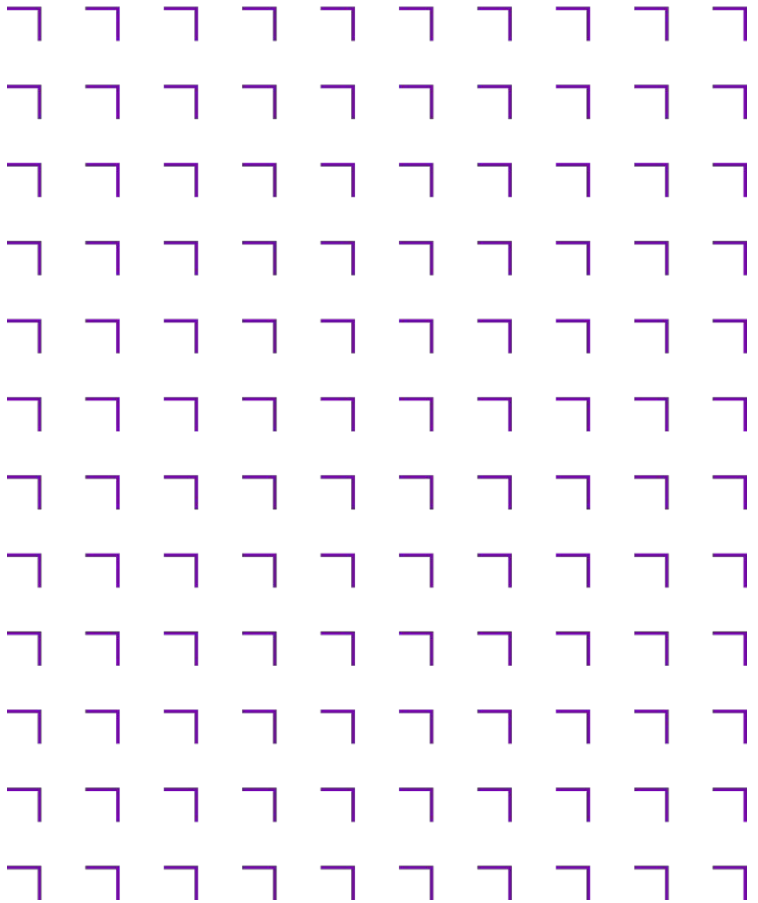
```
/* columns */ main
.card {
    columns: 2;
}
main .card h2, main
.card h5,
main .card .fakeimg {
    column-span: all;
}
```

Activity

- Join a breakout room
- Download the previous example code from Moodle
- Modify the CSS and text as you like
- You have 35 minutes
- Lecturer will visit each room in turn, etc...
- Will start next topic on the hour

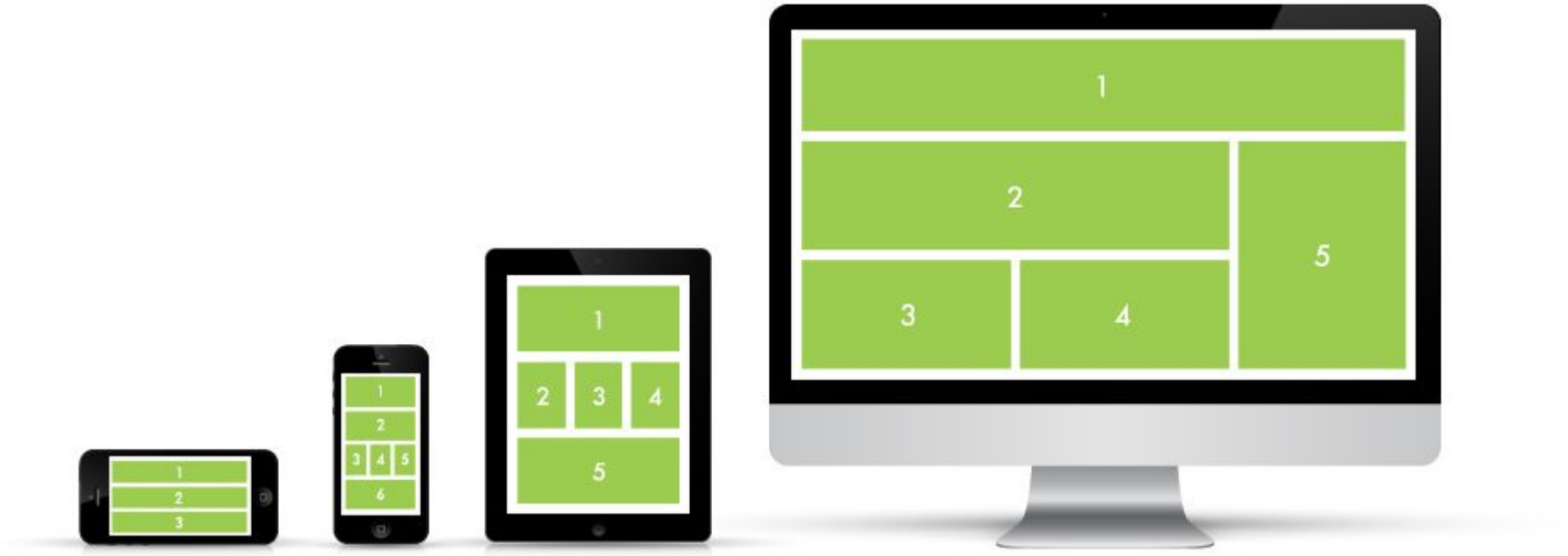
Multi-device layouts:

- Responsive design
- Media queries



Responsive web design (RWD)

Responsive web design (RWD) is a web design approach to make web pages render well on all screen sizes and resolutions while ensuring good usability. It is the way to design for a multi-device web



Responsive web design (RWD)

- HTML is fundamentally "responsive", or "fluid"
- A web page containing only HTML, with no CSS will automatically reflows the text to fit the viewport as the window is resized
- Will also be "responsive" with CSS as long as relative units are used, e.g. `%` or `em`, `rem`, etc...

However, there are issues:

- long lines of text displayed full screen on a wide monitor can be difficult to read
- similarly full-width text can be hard to read on narrow devices such as phones
- problems with multi-column layouts on tablets and phones

Key Components

- Media Queries
- Responsive layout technologies, e.g. Multi column, Flexbox, CSS Grid
- Responsive images/media
- Responsive typography
- The viewport meta tag

RWD - Media Queries

CSS Media Query gives you a way to apply CSS only when the browser and device environment matches a rule that you specify, e.g. "viewport is wider than 480 pixels"

- are a key part of responsive web design, as they allow you to create different layouts depending on the size of the viewport
- can also be used to detect other things about the environment your site is running on, e.g. whether the user is using a touchscreen rather than a mouse

Media Query - Basics

```
@media media-type and (media-feature-rule) {
   /* CSS rules go here */
}
```

It consists of:

- A media type, which tells the browser what kind of media this code is for (e.g. print, or screen)
- A media expression, which is a rule, or test that must be passed for the contained CSS to be applied
- A set of CSS rules that will be applied if the test passes and the media type is correct

Media types
The possible types of media you can specify are:

- all (default)
- print
- screen

Media Query - Example

```
@media print {
    body {
        font-size: 12pt;
    }
}
```

The `font-size` will only be set on the `body` tag when the page is being printed

Note: Media types are optional; if you do not indicate a media type in your media query, then the media query will default to being for all media types

Media Query - Media feature rules

Width and Height

Viewport width is the primary feature used to create responsive designs - `min-width` , `max-width` , and `width`

For example:

```
@media screen and (width: 600px) {
    body {
        color: red;
    }
}
```

Media Query - Media feature rules

The width (and height) media features can be used as ranges, and therefore be prefixed with min- or max- to indicate that the given value is a minimum, or a maximum. For example, to make the color blue if the viewport is 600 pixels or narrower, use max-width:

```
@media screen and (max-width: 600px) {
   body {
      color: blue;
   }
}
```

- Using minimum or maximum values is much more useful for responsive design so you will rarely see width or height used alone
- Another well-supported media feature is `orientation` which allows you to test for portrait or landscape mode
- See link below for the full list: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_media_queries/Using_media_queries#syntax

Media Query - Ranged syntax

- viewport width between two values:

```
@media (min-width: 30em) and (max-width: 50em) {
   /* ... */
}
/* "range" syntax version */ @media
(30em <= width <= 50em) {
   /* ... */
}
```

- "and" logic

```
@media screen and (min-width: 600px) and (orientation: landscape) {
   body {
      color: blue;
   }
}
```

Media Query - Ranged syntax

- "or" logic

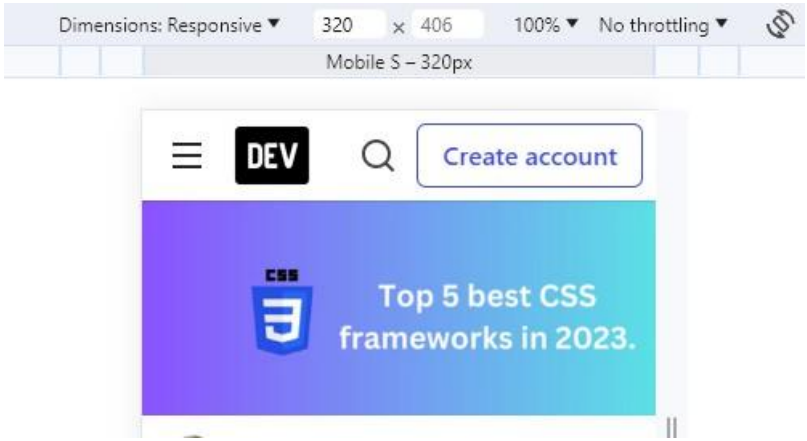
```
@media screen and (min-width: 600px), screen and (orientation: landscape) {
   body {
      color: blue;
   }
}
```

- "not" logic

```
@media not all and (orientation: landscape) {
   body {
      color: blue;
   }
}
```

Media Query - Choosing breakpoints

The points at which a media query is introduced are known as **breakpoints**



Mobile first responsive design

Two approaches:

- `desktop-first` start with desktop or widest view and add breakpoints as viewport becomes smaller or
- `mobile-first` start with smallest devices (small mobiles) and add breakpoints as viewport becomes bigger
  - e.g. common CSS frameworks such as Bootstrap, Tailwind.css, Foundation

Responsive Images

There are two distinct problems with making images behave responsibly:

- **Resolution switching:**

  You want to serve smaller image files to narrow-screen devices, as they don't need huge images like desktop displays do — and to serve different resolution images to high density/low density screens. You can solve this problem using vector graphics (SVG images) and the srcset with sizes attributes

- **Art direction:**

  You want to serve cropped images for different layouts — for example a landscape image showing a full scene for a desktop layout, and a portrait image showing the main subject zoomed in for a mobile layout

Responsive Images/media

To ensure media is never larger than its responsive container, the following approach can be used:

```
img,
picture,
video {
   max-width: 100%;
}
```

This scales media to ensure they never overflow their containers

However using a single large image and scaling it down to fit small devices wastes bandwidth by downloading images larger than what is needed

Responsive Images, using the `<picture>` element and the `<img>` `srcset` and `sizes` attributes enables serving images targeted to the user's viewport and the device's resolution

E.g. you can include a square image for mobile, but show the same scene as a landscape image on desktop

Responsive Images/media

The `<picture>` element enables providing multiple sizes along with "hints" (metadata that describes the screen size and resolution the image is best suited for), and the browser will choose the most appropriate image for each device, ensuring that a user will download an image size appropriate for the device they are using
Using `<picture>` along with `max-width` removes the need for sizing images with media queries It

enables targeting images with different aspect ratios to different viewport sizes

You can also art direct images used at different sizes, thus providing a different crop or completely different image to different screen sizes

```
<img srcset="elva-fairy-480w.jpg 480w, elva-fairy-800w.jpg 800w"
     sizes="(max-width: 600px) 480px, 800px"
     src="elva-fairy-800w.jpg"
   alt="Elva dressed as a fairy" >
```

Responsive Images - useful tips:

- Use an appropriate image format for your website images (such as PNG, JPG, AVIF, WebP), and make sure to optimize the file size using a graphics editor
- Make use of CSS features like gradients and shadows to implement visual effects without using images
- Use media queries inside the media attribute on <source> elements nested inside <video>/<audio> elements to serve video/audio files as appropriate for different devices (responsive video/audio)

Responsive Typography

Responsive typography describes changing font sizes within media queries or using viewport units to reflect lesser or greater amounts of screen real estate

```
html {
   font-size: 1em;
}
h1 {
   font-size: 2rem;
}
@media (min-width: 1200px) {
   h1 {
      font-size: 4rem;
   }
}
```

You do not need to restrict media queries to only changing the layout of the page

They can be used to tweak any element to make it more usable or attractive at alternate screen sizes

The viewport meta tag

The following `<meta>` tag is found in the `<head>` section of a html document

```
<meta name="viewport" content="width=device-width,initial-scale=1" />
```

This viewport meta tag tells mobile browsers that they should set the width of the viewport to the device width, and scale the document to 100% of its intended size, which shows the document at the size that you intended

This meta tag exists because when smartphones first arrived, most sites were not mobile optimized. The mobile browser would, therefore, set the viewport width to 980 pixels, render the page at that width, and show the result as a zoomed-out version of the desktop layout. Users could zoom in and pan around the website to view the bits they were interested in, but it looked bad

By setting `width=device-width` you are overriding a mobile device's default, like Apple's default `width=980px` , with the actual width of the device. Without it, your responsive design with breakpoints and media queries may not work as intended on mobile browsers. If you've got a narrow screen layout that kicks in at 480px viewport width or less, but the device is saying it is 980px wide, that user will not see your narrow screen layout

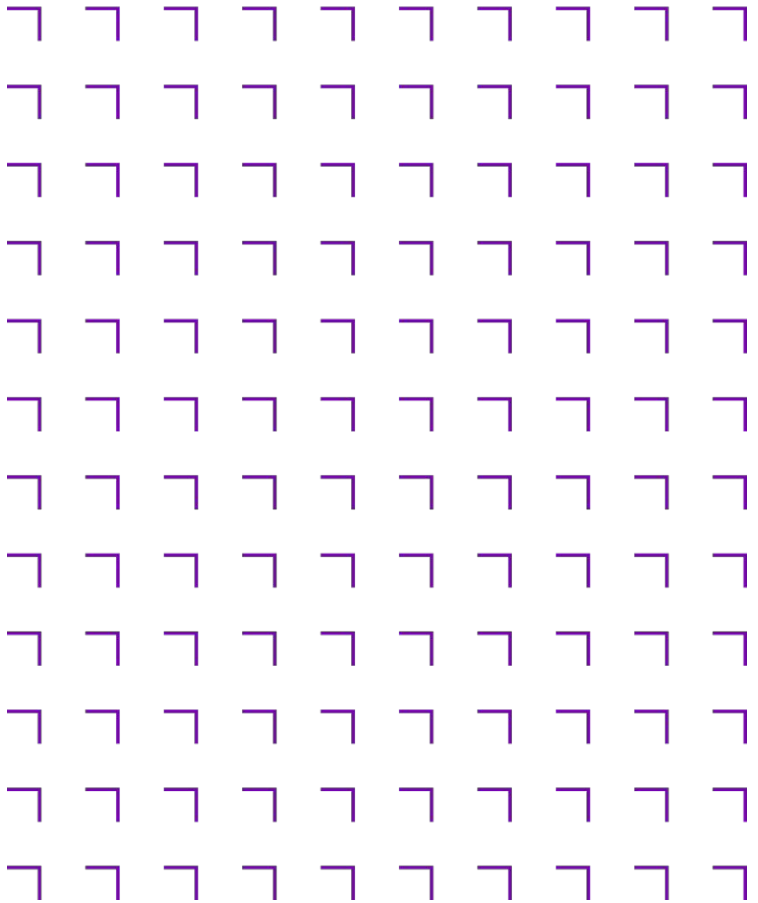
So you should `always` include the viewport meta tag in the head of your documents

Activity

- Join a breakout room
- Download the previous example code from Moodle
- Modify the CSS and text as you like
- You have 35 minutes
- Lecturer will visit each room in turn, etc...
- Will start next topic on the hour

Advanced Layouts:

- Flexbox
- CSS Grid



Flexbox

Before the Flexbox Layout module, introduced in 2009, there were four layout modes:

- Block, for sections in a webpage
- Inline, for text
- Table, for two-dimensional table data
- Positioned, for explicit position of an element

The Flexible Box Layout Module, makes it easier to design flexible responsive layout structure without using float or positioning

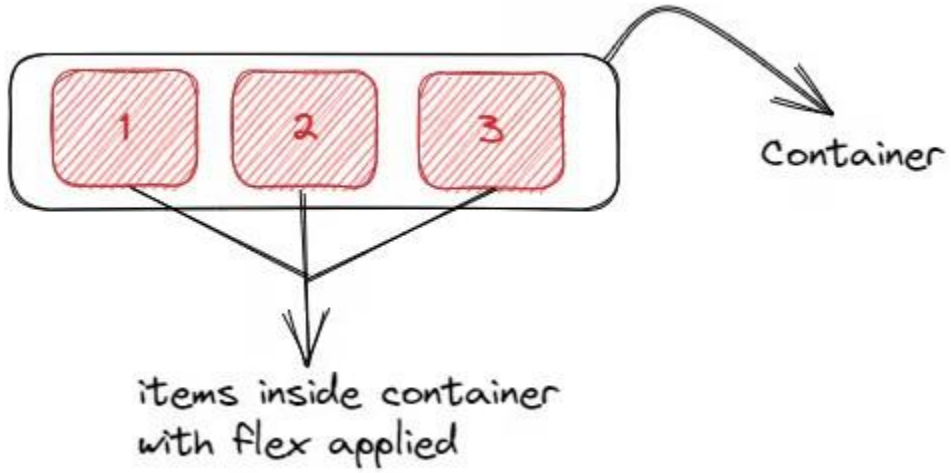
Flexbox is a one-dimensional layout system that we can use to create a row or a column axis layout

It makes it easier to design and build responsive layouts without having to use tricky hacks and a lot of `float` and `position` properties

Flexbox - example

```
<div class="container">
    <div id="one">1</div>
    <div id="two">2</div>
    <div id="three">3</div>
 </div>
```

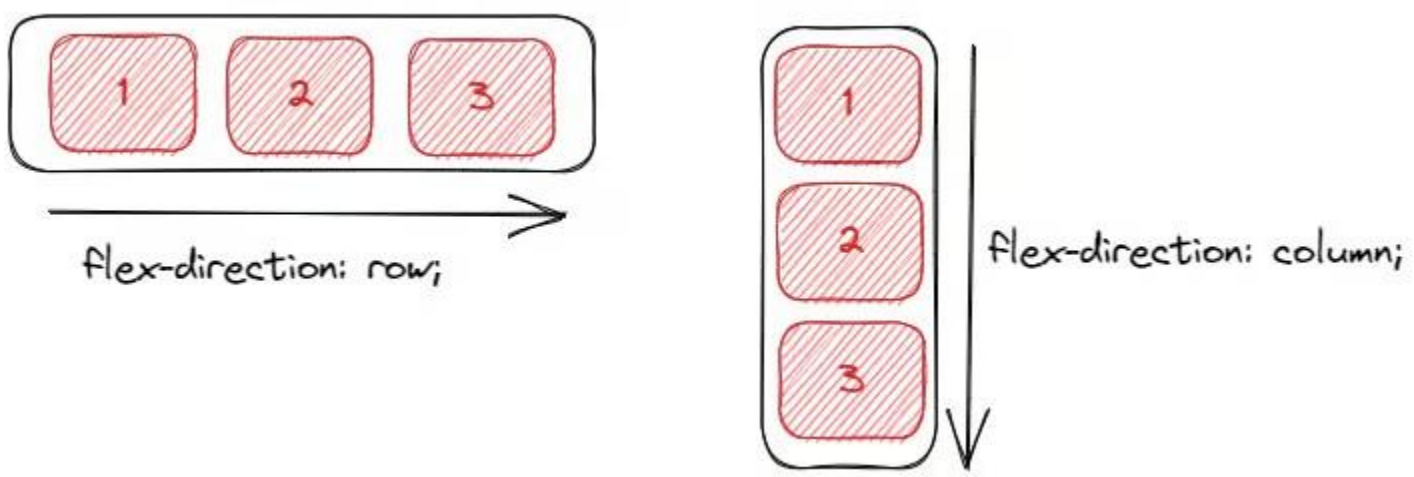
```
.container{
   display: flex;
}
```



Flexbox - `flex-direction`

Sets the direction of the items in the container. The most frequently used flex directions are row and column

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

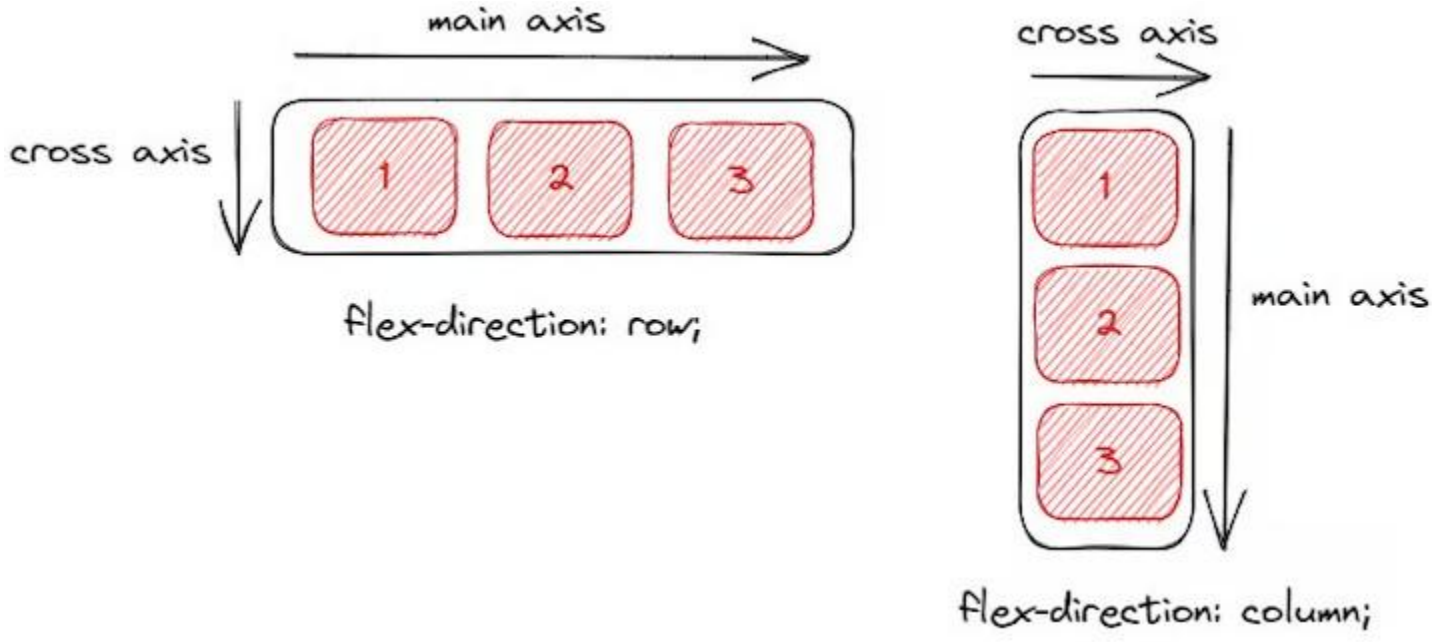


Flexbox - axes

`flex-direction` could be set to row-reverse or column-reverse

Depending on the flex direction, we can have a main axis and a cross axis
Where the `flex-direction` is row, the main axis is in the horizontal direction, and the cross axis is in the vertical

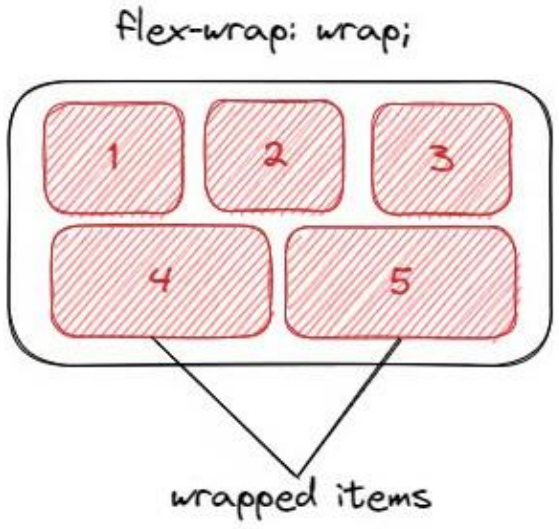
The opposite is of the case when the `flex-direction` is column. This will be useful when we look into aligning



Flexbox - flex-wrap

`flex-wrap` lets items in a flex container move on to the next line when there is no more room:

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

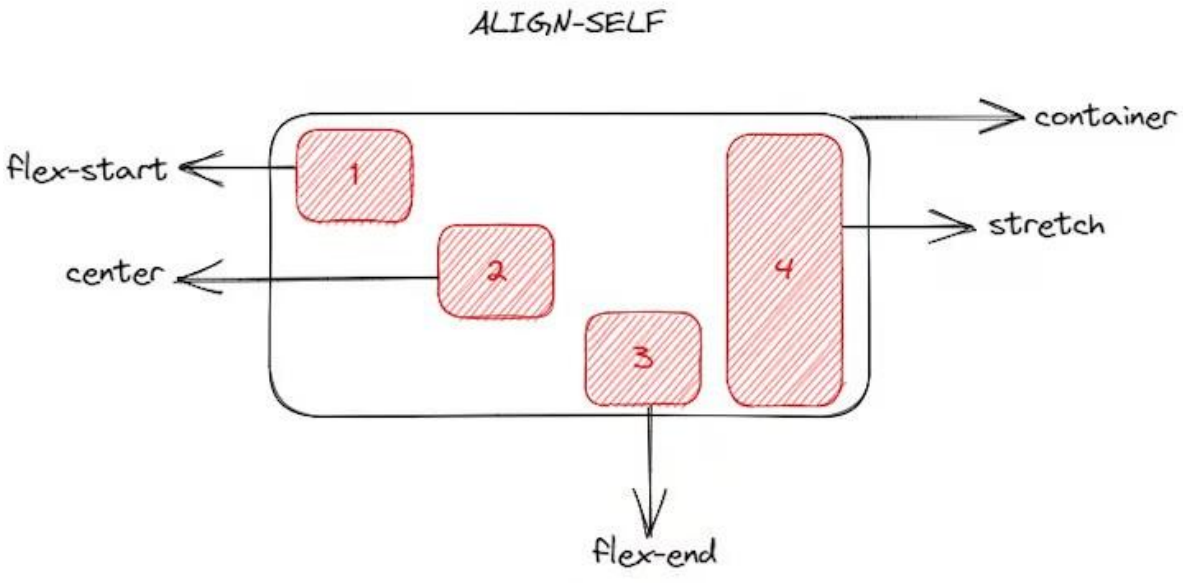


There is an issue in that when the items wrap, they form their own flex line below the ones above, and so are not perfectly aligned with the items above them, e.g. item4 and item5

Flexbox - align-self

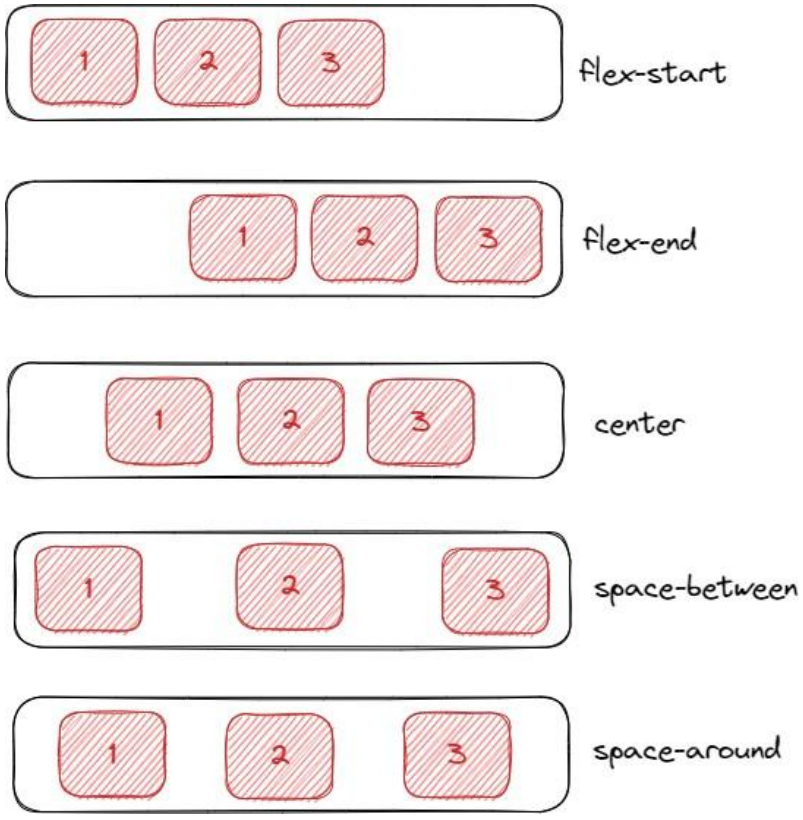
```
<div id="container">
   <div id="one">1</div>
   <div id="two">2</div>
   <div id="three">3</div>
   <div id="four">4</div>
</div>
```

```
#one{ align-self: flex-start | flex-end | center | stretch }
```



Flexbox - justify-content

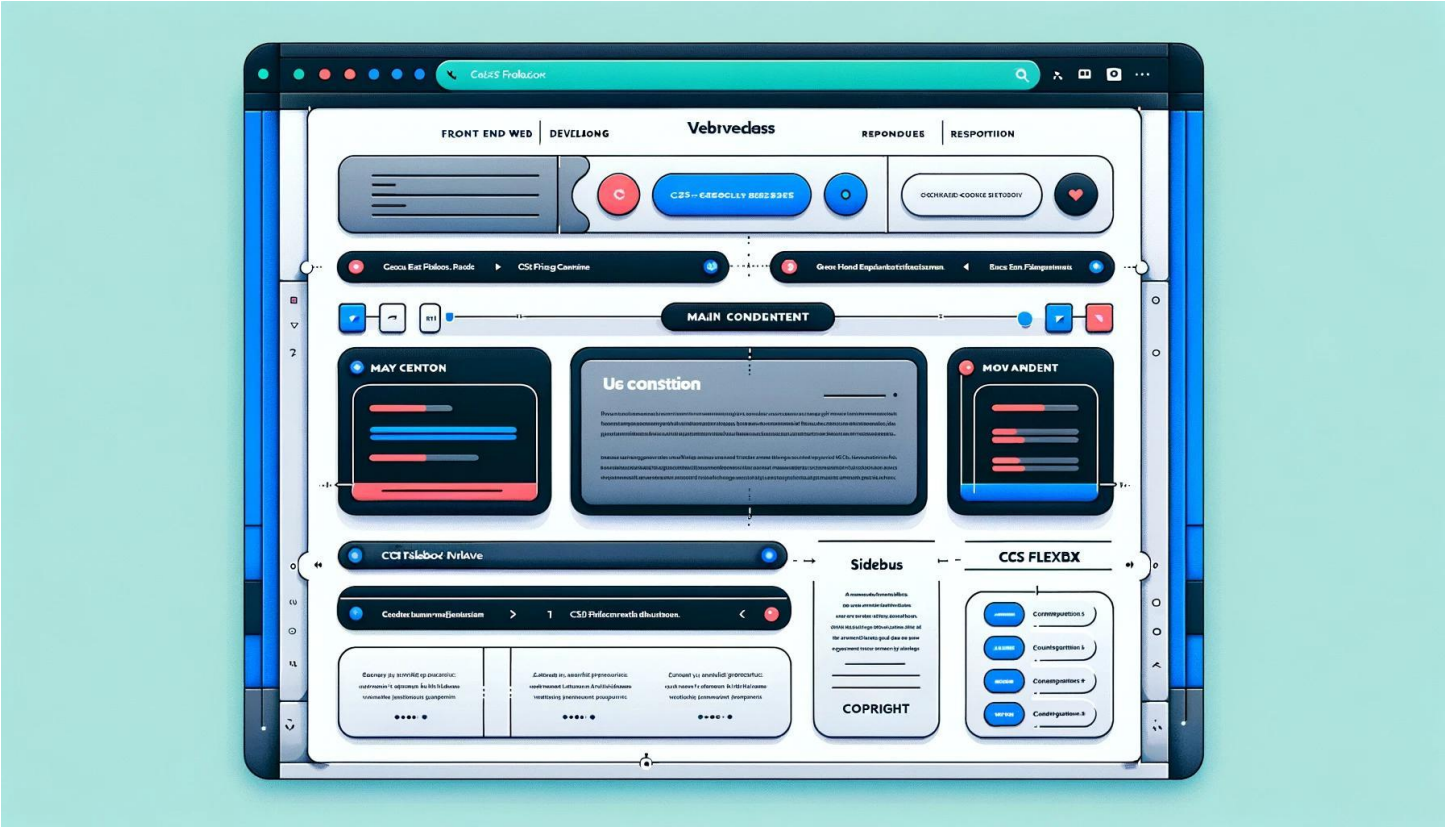
```
.container{
   justify-content: flex-start | flex-end | center | space-between | space-around
}
```



Flexbox - Items

Name	Description
`order`	specifies the order of the flex items. Must be a number, default is 0, e.g. `order: 3;`
`flex-grow`	specifies how much a flex item will grow relative to the rest of the flex items, e.g. `flex-grow: 8;`
`flex-shrink`	specifies how much a flex item will shrink relative to the rest of the flex items, default is 1, e.g. `flex-shrink: 0;`
`flex-basis`	specifies the initial length of a flex item, e.g. `flex-basis: 200px;`
`flex`	shorthand property for the `flex-grow` , `flex-shrink` , and `flex-basis` properties, e.g. `flex: 0 0 200px;`
`align-self`	specifies the alignment for the selected item inside the flexible container, e.g. `align-self: center ;`

Flexbox - Example



This is an AI generated visual representation of a web page layout using CSS Flexbox. This example illustrates a simple layout with a header, navigation bar, main content area, sidebar, and footer, all arranged using Flexbox to demonstrate its flexibility and responsiveness.

CSS Grid

CSS Grid is a two-dimensional layout system, (rows and columns together), which means that it opens a lot of different possibilities to build more complex and organized design systems

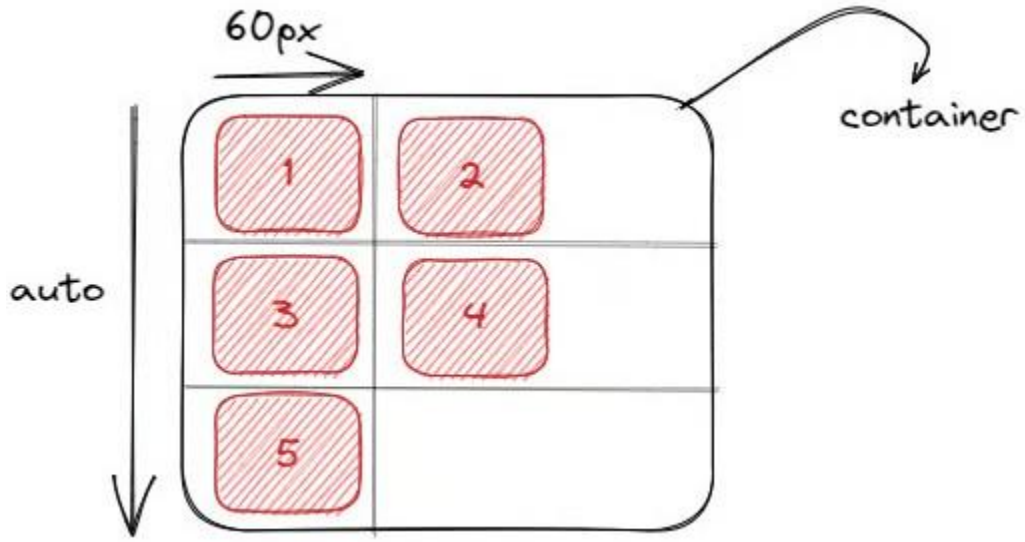
```
<div class="container">
   <div id="one">1</div>
   <div id="two">2</div>
   <div id="three">3</div>
   <div id="four">4</div>
   <div id="five">5</div>
</div>
```

```
display: grid;
grid-template-columns: 60px 60px;
grid-template-rows: auto;
```

To define a grid container, pass a `display: grid;` property to your block element

CSS Grid

The layout will be set to two columns, each occupying 60px of the container width (you can use any length unit or percentage), and since there are five elements, we'll have three rows due to the auto attribute:



CSS Grid - flexible grids with fr unit

The `fr` unit represents one fraction of the available space in the grid container to flexibly size grid rows and columns

Change your track listing to the following definition, creating two 1fr tracks:

```
    display: grid;
    grid-template-columns: 1fr 1fr;
    grid-template-rows: auto;
```

You can mix `fr` units with fixed length units. In this case, the space needed for the fixed tracks is used up first before the remaining space is distributed to the other tracks

Note: The `fr` unit distributes available space, not all space. Therefore, if one of your tracks has something large inside it, there will be less free space to share

CSS Grid - Gaps between tracks

To create gaps between tracks, we use the properties:

- `column-gap` for gaps between columns
- `row-gap` for gaps between rows
- `gap` as a shorthand for both

```
    display: grid;
    grid-template-columns: 2fr 1fr 1fr;
    gap: 20px;
```

CSS Grid - Repeating track listings

You can repeat all or merely a section of your track listing using the CSS `repeat()` function. Change your track listing to the following:

```
display: grid;
grid-template-columns: repeat(2, 1fr);
gap: 20px;
```

You'll now get two 1fr tracks just as before. The first value passed to the `repeat()` function specifies the number of times you want the listing to repeat, while the second value is a track listing, which may be one or more tracks that you want to repeat

CSS Grid - Implicit and explicit Grids

Up to this point, we've specified only column tracks, but rows are automatically created to hold the content. This concept highlights the distinction between explicit and implicit grids:

- Explicit grid is created using `grid-template-columns` or `grid-template-rows`
- Implicit grid extends the defined explicit grid when content is placed outside of that grid, such as into the rows by drawing additional grid lines

By default, tracks created in the implicit grid are auto sized, which in general means that they're large enough to contain their content. If you wish to give implicit grid tracks a size, you can use the `grid-auto-rows` and `grid-auto-columns` properties. If you add `grid-auto-rows` with a value of 100px to your CSS, you'll see that those created are now 100 pixels tall

```
    display: grid;
    grid-template-columns: repeat(2, 1fr);
    grid-auto-rows: 100px;
    gap: 20px;
```

CSS Grid - `minmax()` function

The `minmax()` function lets us set a minimum and maximum size for a track

```
display: grid;
grid-template-columns: repeat(2, 1fr);
grid-auto-rows: minmax(100px, auto);
gap: 20px;
```

If you add extra content, the track expands to allow it to fit

Note that the expansion happens right along the row

CSS Grid - as many columns as will fit

We can combine some of the lessons we've learned about track listing, repeat notation, and `minmax()` to create a useful pattern

It can be helpful to be able to ask grid to create as many columns as will fit into the container

We do this by setting the value of `grid-template-columns` using the `repeat()` function, but instead of passing in a number, pass in the keyword `auto-fit`

For the second parameter of the function we use `minmax()` with a minimum value equal to the minimum track size that we would like to have and a maximum of 1fr

```
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
    grid-auto-rows: minmax(100px, auto);
    gap: 20px;
```

This works because grid is creating as many 200px columns as will fit into the container, then sharing whatever space is leftover among all the columns. The maximum is 1fr which distributes space evenly between tracks

CSS Grid - to conclude

There are other topics not covered such as line-based placement and `grid-template-areas` which you can find covered in the resources below:

- https://www.smashingmagazine.com/2020/01/understanding-css-grid-lines/
- https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_grid_layout/Grid_layout_using_line-based_placement
- https://developer.mozilla.org/en-US/docs/Web/CSS/grid-template

Activity

Join a breakout room

Download the previous example code from Moodle

Modify the CSS and text as you like

You have 35 minutes
Lecturer will visit each room in turn, etc... Will start next topic

on the hour

Resources

Responsive Layouts

- https://alistapart.com/article/responsive-web-design/ (original Ethan Marcotte article from 2010)
- https://www.smashingmagazine.com/2011/01/guidelines-for-responsive-web-design/
- https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Responsive_Design
- https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Media_queries

CSS Flexbox

- https://blog.logrocket.com/css-flexbox-vs-css-grid/
- https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Flexbox

CSS Grid

- https://learncssgrid.com/
- https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Grids

Summary

Completed this Week

- Introduction to Layouts
- Multi-device Layouts
- Advanced Layouts
  - Flexbox
  - CSS Grid

For Next Week

- Complete the remaining exercises for unit 5 before next class
- Review the slides and examples for unit 6