HTML & CSS CONTINUED, RESPONSIVENESS, MEDIA QUERIES

RESPONSIVE WEB DESIGN

What is that?

Responsive web design is an umbrella term we use to describe techniques and practices we use to ensure that our project functions and looks good on all devices.

When we design our project, we're really designing our project several times. Sometimes 3, sometimes 9, sometimes more, depending on our target audience and what standards we need to comply with.

Examples of devices that we must design and account for?

There's the obvious examples of smart phones, tablets, and desktop computers. But what if we break those down further?

There are old and new versions all of those devices. Every time a new phone comes out, the screen size changes. We also have various pixel densities now. A computer from 5 years ago has less pixel density than the latest 5k iMac.

We also have very small desktops and very large desktops now. A website that looks good on an 11 inch laptop will not look good on a 30 inch ultra wide display.

So our job becomes more difficult when trying to accomodate all of these variables.

The best way to deal with this is simply to plan ahead, make concrete decisions on how things will look, and stick to them.

Last week we briefly introduced **flexbox**.

This week we'll also introduce **CSS grids** to make creating responsive websites easier.

We may or may not get to **media queries** during class.

Before getting into that, let's talk about some pros and cons of **fixed width layouts** and **responsive (or sometimes refered to as liquid) layouts**

Fixed width layouts advantages:

- Accurate pixel values
- Designer has much better control over appearance
- Can consistently control widths
- Image sizes stay the same

Fixed width layouts disadvantages:

- You'll likely end up with gaps on the side of the site
- Users with extremely high resolution might see the site smaller with tiny text
- Often occupies more vertical space, which means more scrolling, and less likely the user will stay on the site

Responsive/liquid layouts advantages:

- Pages expand and contract according to the screen, less wasted space
- The designs are tolerant of user changes, like trying to increase font sizes
- Generally more flexible when rendering on devices that didn't exist when we built the site

Responsive/liquid layouts disadvantages:

- If you're not precise and don't do any quality assurance, the site can render in ways you did not intend or predict
- Super wide monitors may make the site too wide and hard to read/interact with
- If images are smaller/larger than the parent container, they may overflow or not occupy the intended space

It should be obvious that a cost/benefit analysis should be done to decide your approach.

At the very least all projects should be **responisve**, but that doesn't necessarily mean the layout is liquid

You can have a responsive fixed width layout by using media queries, which we'll cover later.

A FEW NOTES ON
MEDIA QUERIES...

When adding **media queries**, we should mentioned the **viewport meta tag**. Let's expand on that.

From MDN:

"The browser's viewport is the area of the window in which web content can be seen. This is often not the same size as the rendered page, in which case the browser provides scrollbars for the user to scroll around and access all the content."

"Narrow screen devices (e.g. mobiles) render pages in a virtual window or viewport, which is usually wider than the screen, and then shrink the rendered result down so it can all be seen at once. Users can then pan and zoom to see different areas of the page. For example, if a mobile screen has a width of 640px, pages might be rendered with a virtual viewport of 980px, and then it will be shrunk down to fit into the 640px space."

"This is done because many pages are not mobile optimized, and break (or at least look bad) when rendered at a small viewport width. This virtual viewport is a way to make non-mobile-optimized sites in general look better on narrow screen devices."

Here's where that content was pulled from:

https://developer.mozilla.org/en-US/docs/Mozilla/Mobile/Viewport_meta_tag

And here's an extremely useful resource that has done the tedious job of collecting pixel dimension sizes for many of the most common devices used today:

https://css-tricks.com/snippets/css/media-queries-for-standard-devices/

CSS GRIDS

What I haven't explicitly said is that we've kind of moved historically through the different ways web developers have layed out websites.

We've gone from **absolute and relative positioning**, mentioned **floats**, **inline-block display**, etc.

Each method attempted to address particular issues at the time, but CSS grids is the most common and contemporary these days. If you focus on one CSS technique, focus on grids.

We did a demo on **flexbox**, which is great, but **CSS grids** are more powerful and dynamic.

Flexbox definitely has its place, but I use it more frequently for sections of a website rather than laying out an entire project.

Think **grids** for the whole project and maybe **flexbox** for a navigation or thumbnail image gallery.

One of the major reasons grids are dynamic (and also slightly more complicated) is because it's two-dimensional.

CSS grids make use of both **rows** and **columns**.

Working with grid layouts is as simple as applying CSS rules, just like we've done the past two weeks.

The rules need to be applied to both the **parent** and **child elements**, because the rules work in conjunction with each other.

If you're missing one or the other, the layout likely won't work.

So, the basics.

To get started you must define a **container/parent element** as a grid:

```css
#parent {
  display: grid;
}
```

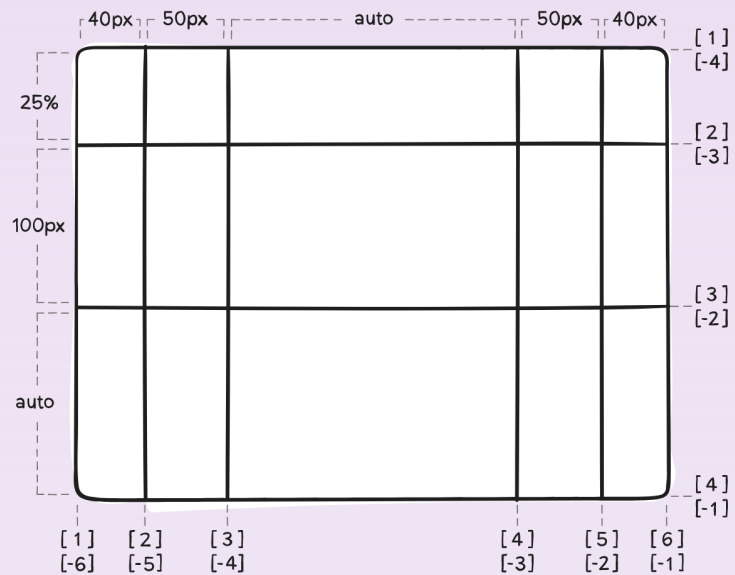You must also set column and row sizes:

```css
#parent {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr 1fr;
}
```

```css
#parent {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr 1fr;
  grid-template-rows: 25% 100px auto;
}
```

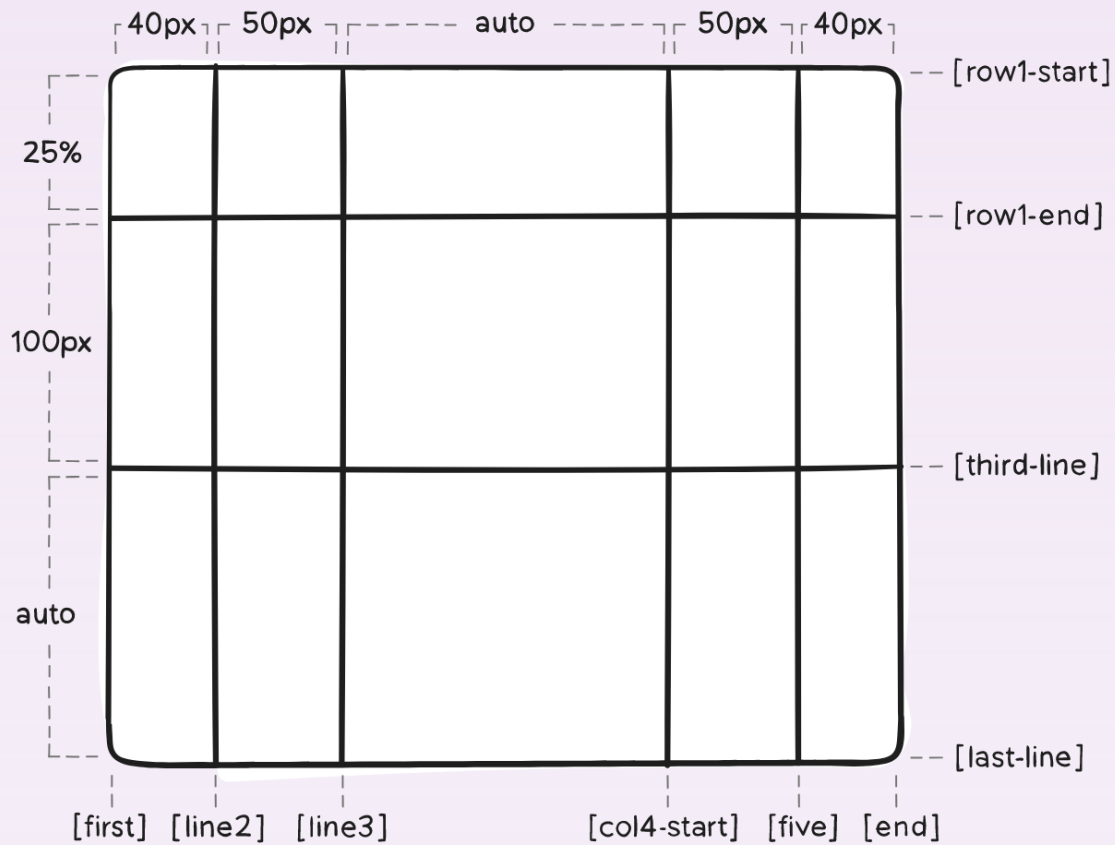**fr** is a unit of measurement, meaning fraction of the free space.

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

You can see in the example that the lines are automatically assigned positive and negative values. But you can actually name the lines yourself.

```css
#parent {
  display: grid;
  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];
}
```

Top column labels (left to right): 40px, 50px, auto, 50px, 40px

Left row labels (top to bottom): 25%, 100px, auto

Right row line labels (top to bottom): [row1-start], [row1-end], [third-line], [last-line]

Bottom column line labels (left to right): [first], [line2], [line3], [col4-start], [five], [end]

Why would we care about that? It may make it easier for us to place our **child elements**.

When placing **child elements**, we have several options:

```css
.child {
  grid-column-start: <number> | <name> |
    span <number> | span <name> | auto;

  grid-column-end: <number> | <name> |
    span <number> | span <name> | auto;

  grid-row-start: <number> | <name> |
    span <number> | span <name> | auto;

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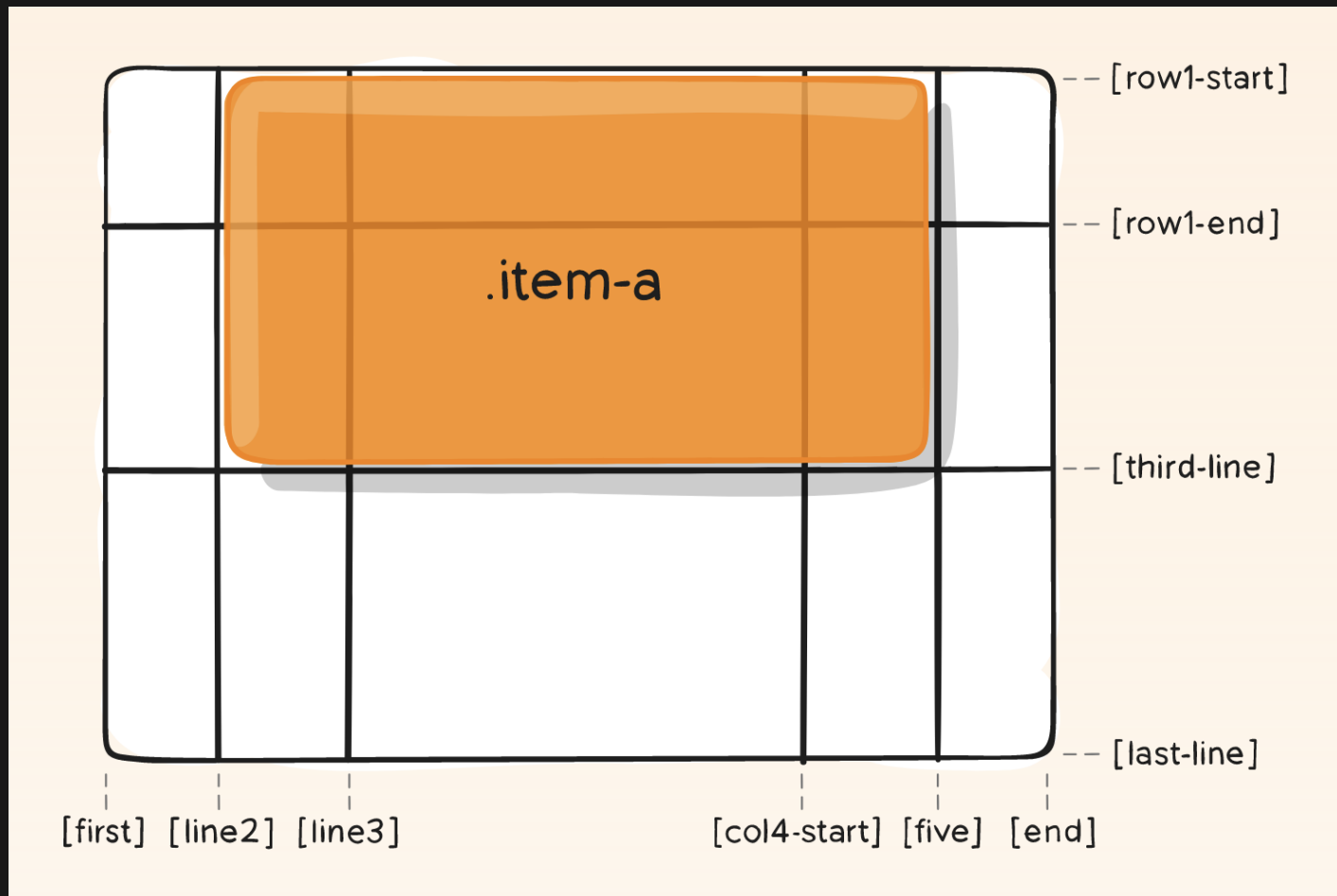
- grid-column-start
- grid-column-end
- grid-row-start
- grid-row-end

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
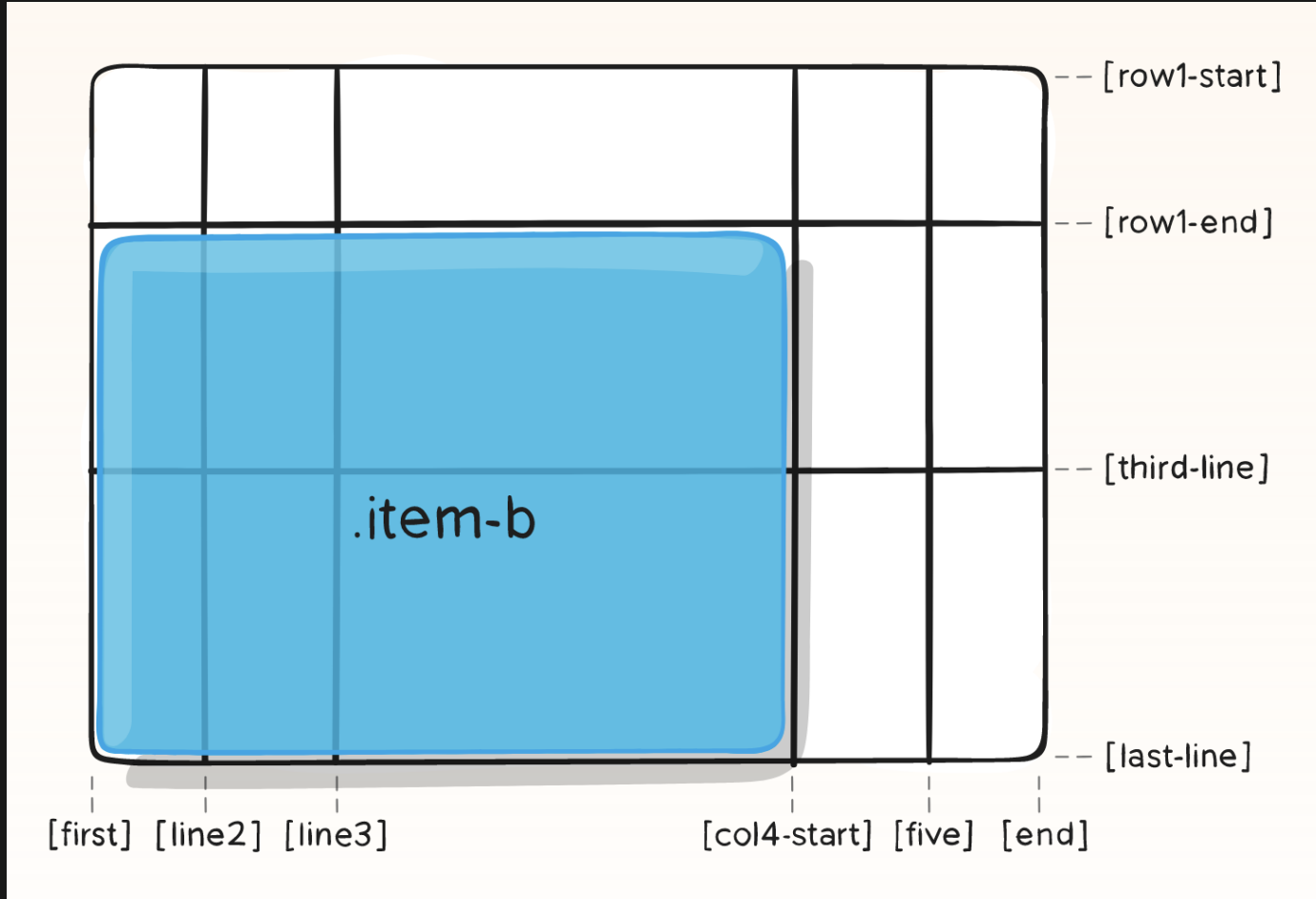
So if we have a child element with a class of item-a:

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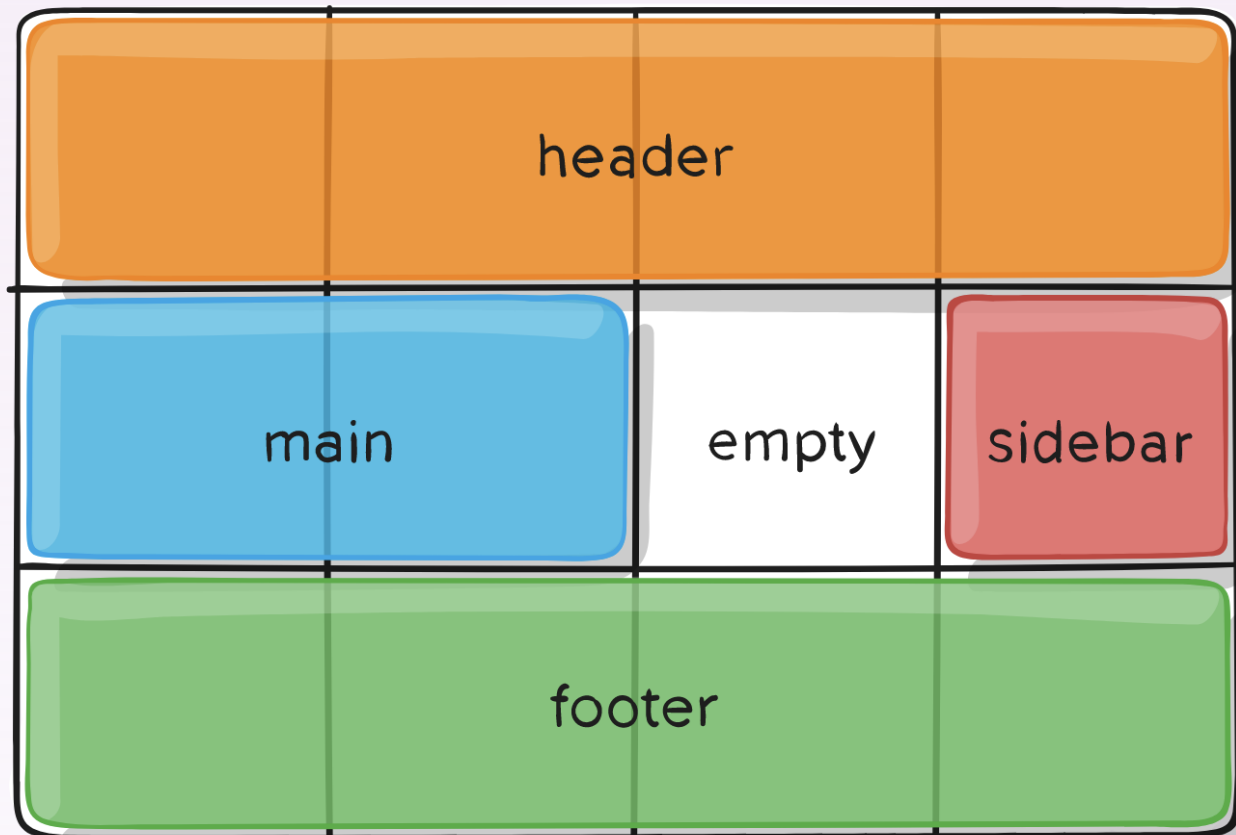
One more example:

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

[row1-start]

[row1-end]

[third-line]

.item-b

[last-line]

[first] [line2] [line3] [col4-start] [five] [end]

With this system, it should be apparent that you can quickly build out sections and elements of site and have them reliably placed:

So, we've only scratched the surface of all of the options, but as usual we just don't have time to go in depth with every single variation.

Some things you consider looking into:

- **grid-template-areas**
- **grid-template**
- **grid-column-gap** and **grid-row-gap**
- **grid-gap**
- **justify-items**
- **justify-self**
- **align-self**
- **place-self**

Note, the images linked in the presenation are from https://css-tricks.com/snippets/css/complete-guide-grid/

It's a phenomenal resource, check it out over the week.

With that, we'll do in-class demos on a single page project using **CSS grids**.

Next week we start **JavaScript** in earnest.