Week 03: CSS, layouting, BEM

Agenda

- CSS properties
- Selector recap
- Flexbox vs grid
- BEM revisited
- Normalization
- Modern CSS checklist
- **Hands on: Live coding**

Let's start!

Main CSS properties: divided by purpose

- Display: Whether the element is treated as a block/inline block and the layout used for its children
- Background: solid colors, gradient, images, positioning, repetition
- Box model: width and height, padding and margin, border color, style, and width
- Positioning: left, right, top, and bottom, z-index
- Typography: color, font-size, -family, -weight, line-height, text-align, -transform
- Transitions
- Animations
- **Flex parents: flex-direction, -wrap, (-flow), align-items, justify-content**
- **Flex children: flex-basis, -grow, -shrink, order**
- **Grid parents: grid-template-rows, -template-columns, -template-areas, -column/row-gap, ...**
- **Grid children: -column-start and -end (-column shorthand), ditto for column, ...**

Property reference

- CSSreference.io
- MDN

Selector recap - CSS selector game

https://flukeout.github.io/

Can you reach level 17?

Flexbox and grid

- Which one to use? It depends
- Flexbox is useful for one-dimensional layouts
  - Can change orientation based on viewport width
  - Order of children can change as well
  - Easy to distribute and align space between elements
- Grid is better suited for two-dimensional layouts
  - Essentially behaves like a table

Grid can be used (and it is preferred to do so) for more complex one dimensional layouts. Great example is the complete mobile page layout where you need to position a header, navbar, main content, footer... The transition between the mobile and desktop layouts is easier when you start with the 1D grid layout on the mobile, and expand it into 2D on the desktop.

Understanding flex properties

- **For parent**: flex-direction, flex-wrap, align-items, justify-content, align-content
- **For items**: align-self, flex-grow, flex-shrink, flex, *order*
- Interactive examples
- Flexbox froggy game

Understanding grid: part 1

Get started by defining a container:

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

Understanding grid: part 2

Lay out the layout:

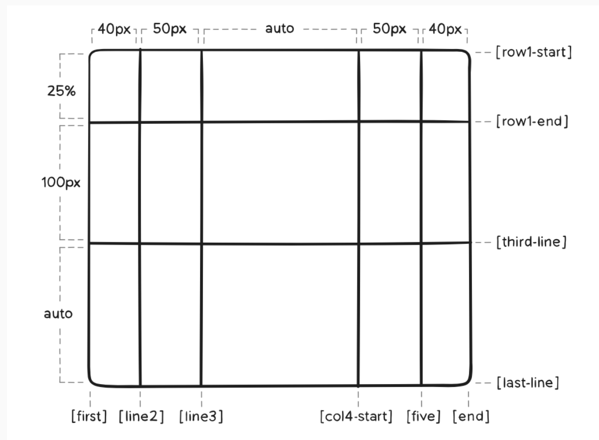
- `grid-template-columns` or `grid-template-rows` takes
  - Track-size (length, percentage, free space portion `fr`, or `auto`)
  - Arbitrary name to label this section (optional)

```
.container {
  grid-template-columns: 1fr 50px 1fr 1fr;
  /* One 50px column with the rest of the space distributed evenly
     between the rest of the columns */
}
```

Understanding grid: part 2.5

Lines between rows and columns can be explicitly named (square bracket notation):

```
.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];
}
```



Tip: repeating parts in column/row definition can be streamlined with `repeat(n, ...)`

Understanding grid: part 3

- Define where slots start/end by referring to line numbers or names
- Slots can span across multiple tracks (`span <number>`) or until they hit a specific line (`span <name>`)

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

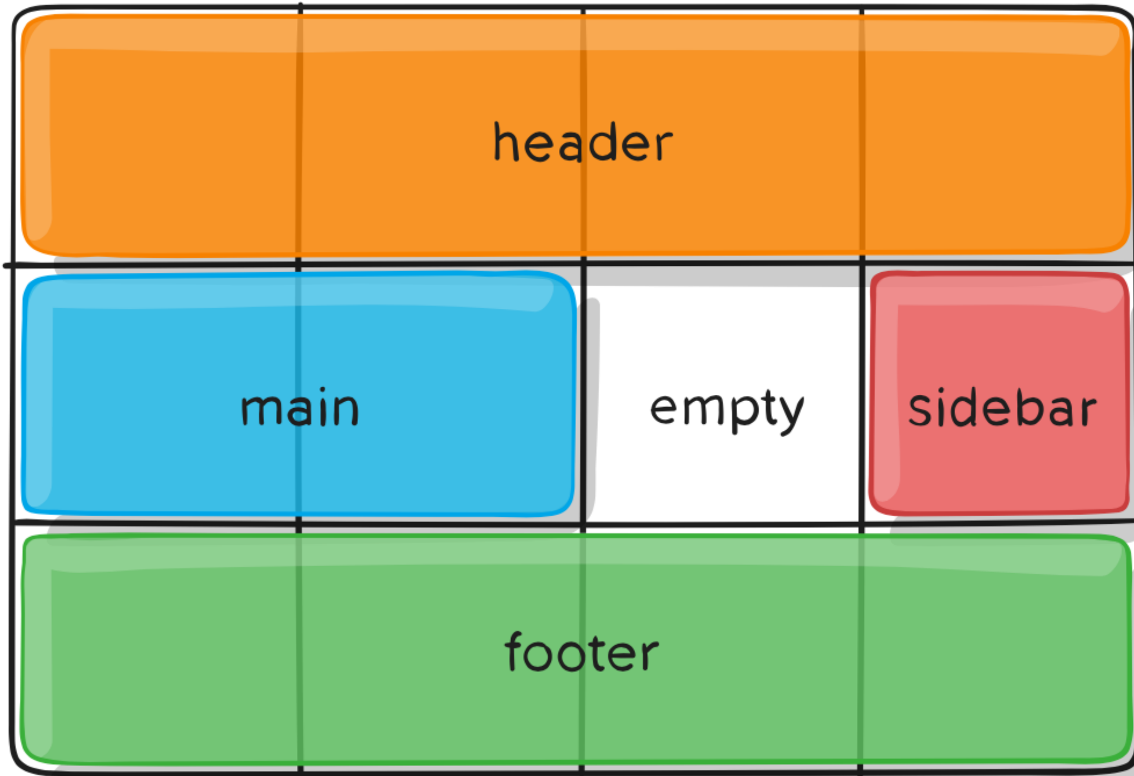
- `grid-column: a b` = shorthand for `grid-column-start: a` and `grid-column-end: b`
- ditto for rows

Understanding grid: part 4

- Assign "grid areas" to items
- Define layout on grid element
- Dots signify empty cells

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

.container {
  display: grid;
  grid-template-columns: 50px 50px 50px 50px;
  grid-template-rows: auto;
  grid-template-areas:
    "header header header header"
    "main main . sidebar"
    "footer footer footer footer";
}
```


header
main
empty
sidebar
footer

Congratulations on understanding CSS Grid!

For more thorough explanations, refer to the Complete Grid Guide.

For an interactive game, visit Grid garden.

Let's talk BEM

Block

- An independent page **component** that can and should be reused
- Its **name describes its purpose** (button), not its appearance (not red, not big)
- Blocks can be nested in each other

```html
<!-- form - BEM `search-form` *block*, also uses a generic `form` *block* -->
<!-- which can be reused for all HTML forms  -->
<form class="search-form form">
    <!-- input - BEM *element* of the `search-form` block: `search-form__input`  -->
    <!--  also uses a generic `input` *block*, which can be reused for all HTML inputs -->
    <input class="search-form__input input">

    <!-- button - BEM *element* in the `search-form` block: `search-form__button` -->
    <!-- also uses a generic `button` *block* which can be reused for all HTML buttons -->
    <button class="search-form__button button">Search</button>
</form>
```

Element

- A semantic part of a block, **unable** to stand on its own
- Separated from the block name with a double underscore (`block-name__element-name`)
- Can be nested, but only the outermost block is projected into element name (so **never** `block__elem1__elem2`)

When to use a block and when an element?

- If a section of code might be reused & it doesn't depend on other page components being implemented => **block**
- If a section of code can't be used separately without the parent entity => **element**
- When a reusable part of code is also used within a context of another block => it can be both **element** and a **block**
    - example: code on the previous slide; the button can have generic reset styles within the `button` class (BEM block), and specific styles such as positioning and specific size in the `search-form` block. These are specific only for that context and are specified in the `search-form__button` class (BEM element)

Modifier

- Defines the appearance, state or behavior of its parent (block or element)
- Separated with a double hyphen (`block-name--modifier`)
- Can never be used alone (is semantically tied)

```html
<!-- The `search-form` block has the `focused` Boolean modifier -->
<form class="search-form search-form--focused form">
    <input class="search-form__input input">

    <!-- The `button` element has the `disabled` Boolean modifier -->
    <button
     class="search-form__button search-form__button--disabled button"
     disabled>
        Search
    </button>
</form>
```

Normalization

- Sometimes, each browser has some default styles, which can lead to UI inconsistencies
- We can load `normalize.css` first, followed by our own styles
- Do not attempt to create your own normalization. Use the well-developed versions available on GitHub: for example (51.6k stars): github.com/necolas/normalize.css
- The benefit of normalization is reduced CSS "debugging"
- Some libraries, like TailwindCSS, already normalize CSS under the hood

Modern CSS checklist

- I normalize the default styles for my page with some kind of a normalizer
- I've designed the layout and styles with the mobile-first approach
- I am using CSS variables defined in the root for colours, border-radius padding...
- I am using simple selectors (HTML element selectors such as `h1`, id selectors (`#desired-id`), class selectors (`.classname`)) whenever possible
- I order my CSS selectors in the order: **reset selectors / classes**, **generic reusable classes**, **layout / style specific classes**
- I try to avoid using `px` and `%` for margins and paddings
- I use `em` or `rem` for sizes and variables most of the time
- I use computation in CSS with `calc`, `min`, `max` and other value functions
- I use media queries for the layout changes depending on the viewport changes
- I have split my CSS into mobile and desktop files
- I conditionally load my desktop CSS
- I write my CSS simple, stupid
- I am reusing components (in BEM terminology "blocks") whenever it is possible
- *OPTIONAL: I've defined a light and dark mode for my page*

Live coding: let's make a layout and styles according to a design together!

[The Figma design]

Questions?

Question: When will we finally get the first iteration?

Answer: It will be assigned on March 10

The iteration will require you to know **HTML semantics**, **layouting & styling**, a little bit of **TypeScript** and some **Storybook** knowledge. The Storybook lecture will be held during week four, so you will get an instructions video to be able to start working on the iteration prior to the Storybook lecture. We felt we could not release the iteration prior to the TypeScript lecture, so that's the reason the first iteration is to be released third week into the semester.