BUILDING WEB LAYOUTS WITH CSS GRID AND FLEXBOX

TRILOGY
EDUCATION SERVICES

Presented by Alex Rosenkranz

TODAY'S AGENDA

- The History Of Web Layout
  - Normal Flow
  - Floats & Clearfix
  - Positioning
- Flexbox
- CSS Grid

NORMAL FLOW

- This is HTML without CSS
- Default flow is “left to right” (horizontal) where the X-Axis is “Inline” and the Y-Axis is “Block”
- If you switch writing direction to “top to bottom” (vertical), X-Axis becomes “Block” and Y-Axis becomes ”Inline”
- Everything reads in order and elements stay next in line from previous sibling element due to it’s default “block” dimension spacing
- This is good for us! In the event CSS does not load or if the user is using a screen reader the content of the site still remains readable in an order that makes sense.

FLOATS

- Used to shift a box left or right and allow sibling content to wrap around it
- This is ideal for inline imagery, NOT for entire site layout
- The CONTENT of the sibling box/container is wrapped around floated element, but the CONTAINER keeps it's full dimensions
- Elements following a floated item will wrap with the float until horizontal space is used up, so a "clear" needs to be applied for those elements to return to normal.
- These "clear" fixes can be applied as it's own element (hard for those using a CMS) or by applying a pseudo class

POSITION PROPERTY

- Used to remove or shift an element from normal flow. The value provided will determine whether that shift/offset originates from it's sibling element, parent element, or browser window
- This is good for creating overlapping content using the top, right, bottom, left properties
- **Relative:** Offset position based on where it originally starts in normal flow and keeps reserved space
- **Absolute:** Takes element out of flow and sets origin point to top/left values of parent element. Sibling elements take up the space originally reserved for it.

POSITION PROPERTY (con't)

- **Fixed:** Takes element out of flow and sets origin point to top/left of browser window / viewport. The position of this element remains whenever user scrolls on the page. Typically seen being used with persistent navigation
- **Sticky:** Element scrolls with page as it would in "normal flow" until it hits a certain, predetermined position on the page and then it remains "fixed" the rest of the page. This is also very popular with navigation bars and sub-navigation on longer pages

FLEXBOX (display: flex)

- Means “Flexible Box”
- Created in 2009 (originally display: box)
- Updated in 2011 (display: flexbox)
- Today (display: flex)
- It is a layout method meant for one-dimensional layouts. This means you control either X-axis (rows) OR Y-axis (columns) and flexbox will handle overflow gracefully
- If you’ve used Bootstrap 4’s Grid you’ve already used flexbox!

HOW TO USE FLEXBOX

- Every flexbox is comprised of a “Flex Container” / “Flex Item” relationship
- Flex container handles how all of its “child” items will behave like setting the direction, how to handle overflowing elements, and how everything should be spaced out
- Flex items are the children inside of the box. From here we can control the order of where the elements come into the box, how they can align themselves (if it needs to be different from what the container dictated), and how it should handle negative space around it when resizing

CSS GRID (display: grid)

- Seems like CSS4. It's not. (CSS has moved into module versioning instead of entire language, see: https://www.w3.org/TR/css-2015/)
- Became official CSS Specification as of March 2017. Incorporated in all major browsers by now.
- Adopted incredibly fast, unlike previous CSS advances
- No "row" markup, meaning we can specify our element positioning on BOTH axes (rows and columns). Keeps HTML really nice and clean
- For detailed documentation about CSS Grid and its intended use going forward, check out https://drafts.csswg.org/css-grid-1/

HOW TO USE CSS GRID

- Like Flexbox, Grid operates off of a container/item relationship as well
- We can apply *most* styles to our grid's container and its children will do a great job following those rules without much individual customization
- **Grid parent/container properties** include defining how many explicit columns/rows there can be and how much space they should take up (it is common to explicitly define columns and allow rows to be created as needed), how empty space should be treated, how much of a gap exists between items, and how elements are spaced in general.

HOW TO USE CSS GRID (con't)

- **Grid child/item properties** include how much space it should take up (span multiple rows and/or columns), how it should align itself, what grid-area it belongs to (if predefined)
- This all makes it a much more enjoyable experience when writing media queries and creating a responsive layout

WRAPPING UP

- Questions?
- All material and information from this presentation can be found here: https://github.com/arosenkranz/WebLayout