Saturday, June 18, 2016

How well do you know CSS display?

The *display* property is one of the most important CSS properties we use for layout. Most of us would have used `block`, `inline` and `none`. `table` and `inline-block` are also quite common. The new darling is definitely `flex`, because it's a display property that was created specifically for layout. The upcoming `grid` (currently still being actively worked on) is another layout-specific property that we'll soon have in our arsenal as well.

This post grew much longer than I initially expected so feel free to skip to a subsection if you wish. Though I would really appreciate it if you took the time to read the whole post 😁.

Table of contents

Through my experience of building various responsive designs, I learnt a lot about the *display* and *position* properties, how they work and how they can be combined with media queries to achieve the desired layouts. I'll briefly cover each value, and also reminisce about a few responsive components I built previously that utilised *display* quite heavily.

We can't talk about *display* without mentioning something called a box tree. Basically the browser parses CSS and renders it by generating a box tree, which represents the formatting structure of the rendered document. The *display* property defines the box's display type.

The topic of how browsers render stuff on the screen is a really fascinating one and I highly suggest reading How Browsers Work: Behind the scenes of modern web browsers by Talia Garsiel. Another must-read is Evolution of CSS Layout: 1990s to the Future by Fantasai, who works on CSS specifications at W3C. It's actually a talk she gave at the Emerging Technologies for the Enterprise conference, but there's a full transcript if video is not your thing.

Those we know quite well already

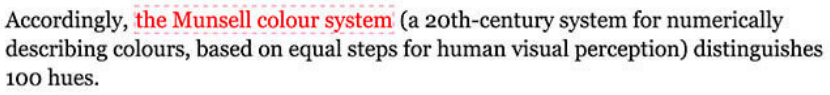
Fun fact: the *display* values we use all the time are actually short-hand. For example, `block` is actually short-hand for `block flow`. Refer to the specification for full list.

All elements have a default *display* value, but they can be overridden by explicitly setting the *display* value to something else.

display: none;

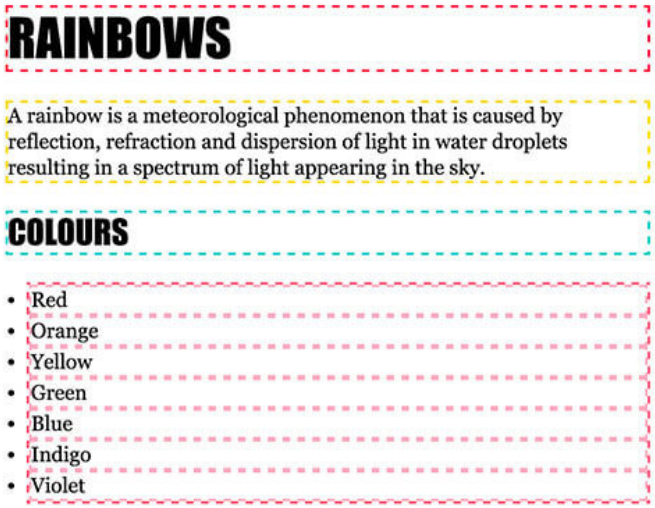
Removes the element and its children from the normal document flow. The document is rendered as if the element was never there to begin with, which means the space it occupied is collapsed. The content of the element is also ignored by screen readers.

display: inline;



The element generates one or more inline boxes. Inline-level elements take up, as the name suggests, as much space on the line as its tags define. Can be considered the complement to block-level elements.

display: block;



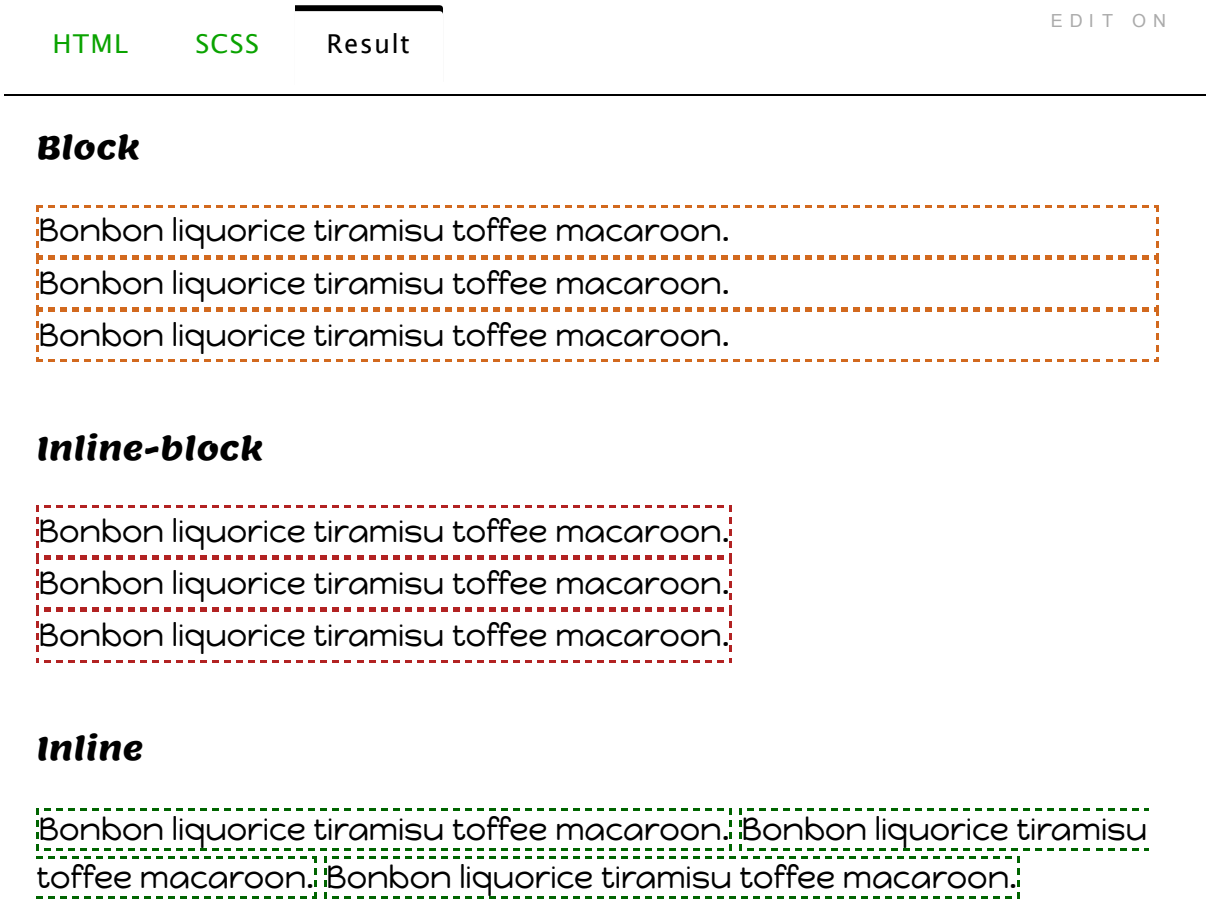
The element generates a block level box. All block-level elements start on a new line and,

unless otherwise specified, stretches to width of its container.

display: list-item;

An element rendered as a list-item behaves exactly like that of a block-level element, but it also generates a marker box, which can be styled by the *list-style* property. Only `<li>` elements have the default value of `list-item`. Usually used to reset `<li>` elements back to their default behaviour.

display: inline-block;



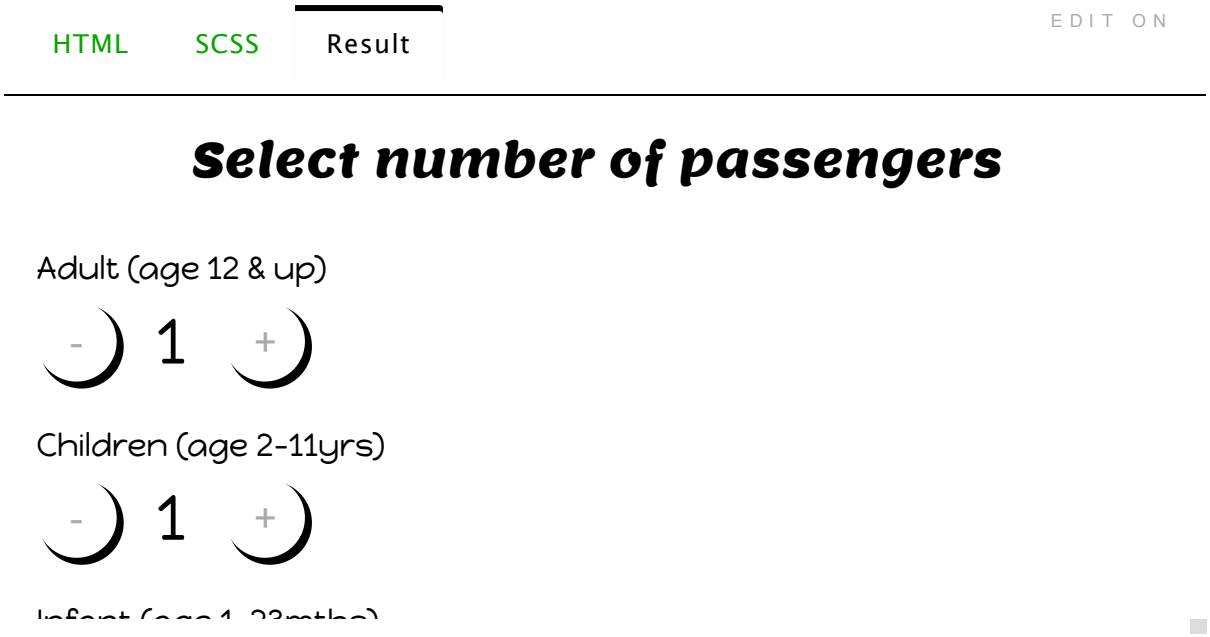
The element generates a block level box, but the entire box behaves like an inline element. Try it opening the above example on CodePen and adjusting your window width, it'll make more sense that way.

A responsive numeric stepper

One of the components I had to build was a numeric stepper for selecting different types

of passengers. I got a static photoshop file with 1 mobile layout and 1 desktop layout. But there were all the in-between widths that weren't accounted for which "broke" the layout.

It was mainly due to the text in parenthesis that didn't collapse nicely. So I had to toss in a bunch of media queries to adjust the width and display of the relevant elements at different widths. Check out full-sized Codepen to see how the component responds at different window widths.



Remember them table-based layouts?

There are a set of *display* values that allow your elements to behave just like HTML tables. My fellow Singapore-based developer Colin Toh wrote a great post on the display: table property, which you should really check out.

Although most of us no longer use table-based layouts, `display: table` is still pretty useful in certain cases. For example, if you wanted to have tables only on wider layouts, but retain a typical block layout on smaller widths. This can be achieved with a combination of media queries and *display* (with some pseudo-elements thrown in for good measure), just resize this window to see how it works.

display: table;

Corresponds to the `<table>` HTML element. It defines a block-level box.

display: table-header-group;

Corresponds to the `<thead>` HTML element.

display: table-row;

Corresponds to the `<tr>` HTML element.

display: table-cell;

Corresponds to the `<td>` HTML element.

display: table-row-group;

Corresponds to the `<tbody>` HTML element.

display: table-footer-group;

Corresponds to the `<tfoot>` HTML element.

display: table-column-group;

Corresponds to the `<colgroup>` HTML element.

display: table-column;

Corresponds to the `<col>` HTML element.

display: table-

caption;

Corresponds to the `<caption>` HTML element.

display: inline-table;

This is the only value that does not have a direct mapping to a HTML element. The element will behave like a table HTML element but as an inline-block rather than a block-level element.

```
@media screen and (min-width: 720px) {
  .table {
    display: table;
    width: 100%;
    border-collapse: collapse;
  }
}

.tr {
  margin-bottom: 1.6rem;
}

@media screen and (min-width: 720px) {
  .tr {
    display: table-row;
  }
}

@media screen and (min-width: 720px) {
  .td {
    display: table-cell;
    border: #f0f0f0 1px solid;
    padding: 0.4rem;
  }
  .td:first-child {
    width: 11em;
  }
}
```

```
.th {
  font-size: 1rem;
  line-height: 1.6rem;
  font-family: "Palo Alto";
}

@media screen and (min-width: 720px) {
  .th {
    font-size: 1.294rem;
    line-height: 1.6rem;
  }
}

@media screen and (min-width: 720px) {
  .th {
    font-size: 0.8rem;
    line-height: 1.6rem;
    font-family: "Roboto Slab", Rockwell, serif;
    font-weight: 700;
  }
}

@media screen and (min-width: 720px) and (min-width: 720px) {
  .th {
    font-size: 1rem;
    line-height: 1.6rem;
  }
}

.th::before {
  content: 'display: ';
}

@media screen and (min-width: 720px) {
  .th::before {
    content: '';
  }
}

.th::after {
```

```
    content: ';';
  }

  @media screen and (min-width: 720px) {
    .th::after {
      content: '';
    }
  }
```

New kids on the block

Tab Atkins Jr., the primary author of the Flexbox and Grid specifications, made a salient point about these new layout-specific display modes.

> *“ Flexbox is for one-dimensional layouts - anything that needs to be laid out in a straight line (or in a broken line, which would be a single straight line if they were joined back together).*
> *Grid is for two-dimensional layouts. It can be used as a low-powered flexbox substitute (we’re trying to make sure that a single-column/row grid acts very similar to a flexbox), but that’s not using its full power.*
> *- Tab Atkins Jr. to www-style ”*

Something to keep in mind as you adopt these new CSS layouts in your work, and are confused about when to use which.

display: flex;

The introduction of the flexbox layout mode, or CSS Flexible Box, marks the first time we have a specification that is really meant for laying out content in the browser. Laying out content on the web has evolved quite a bit since HTML was first introduced. When designers wanted to have some creative layout, the first technique used was nesting HTML tables, or what we refer to as table-based layouts.

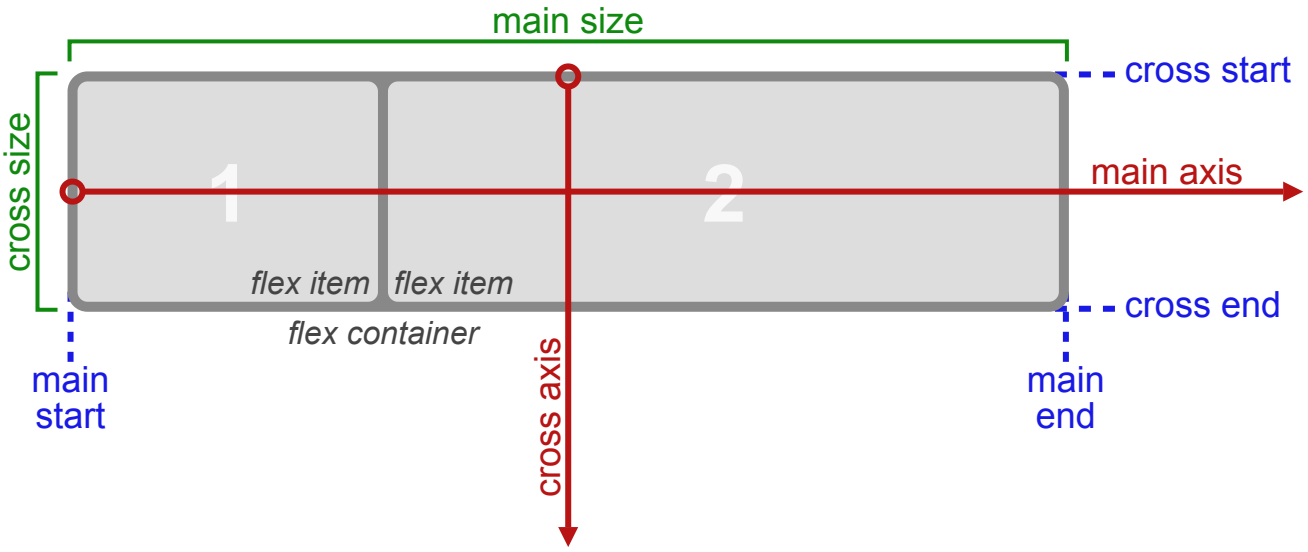
And when CSS started to pick up, we moved over to float-based layouts, by nesting our content in different divs to float them around to get the desired effect. Float-based layouts are still very common but with flexbox being fully supported by all current browsers as of time of writing, I think it won't be too long before flexbox and grid, which will be covered later, become the prevailing method for layout.

I'm going to reference Scott Vandehey's article What IS Flexbox?, where he asks Tab Atkins Jr. about the history of Flexbox. The earliest draft for the specification is dated July 23, 2009 but discussions started a few years before that.

However, nothing was formally structured and the various browser vendors sort of implemented flexbox but didn't really follow the specification. Which is why the flexbox syntax became pretty messy (and still is when it comes to backward compatibility on older browsers).

The flexbox model is very powerful, and because it can do a lot, some effort is required to fully understand how it works and how to use it. Both flexbox and grid require full-length articles to cover in depth, so I'll list my go-to resources for flexbox here:

- A Complete Guide to Flexbox by Chris Coyier
- Solved by Flexbox by Philip Walton
- Flexbox Froggy by Thomas Spark
- Using CSS flexible boxes by Mozilla Developer Network
- CSS Flexbox Specification (Editor's Draft)



By declaring `display: flex` on an element, it becomes a flex container, and its child elements become flex items. This does not cascade further, meaning the flex properties do not extend to the element's grandchildren. Both the flex container and flex items have their own respective flex properties.

Properties for flex container

display: flex-direction;

Defines the main axis and direction of the flex items. Full list of flex-direction values.

display: flex-wrap;

Specifies if the flex items adjust to fit in a single row or be allowed to wrapped onto multiple rows. Full list of flex-wrap values.

display: flex-flow;

Short-hand property for flex-direction and flex-wrap. Full list of flex-flow values

display: justify-content;

Defines how space between and around flex items are distributed along the main axis.

Full list of justify-content values

display: align-items;

Defines how space between and around flex items are distributed perpendicular to the main axis. Full list of align-items values

display: align-content;

Specifies how lines of flex items are distributed within the flex container. Does not apply if flex items are on a single line only. Full list of align-content values

Properties for flex items

display: order;

Specifies the order of how flex items are laid out, in ascending order of the order value. Flex items with the same order value are laid out according to source order. Full list of order values

display: flex-grow;

Defines the ability for an element to grow if there is available space, with the value determining the proportion of space the element can grow into. (Told you this was sort of complicated). Full list of flex-grow values

display: flex-shrink;

Defines the ability for an element to shrink if there isn't enough space, with the value determining the proportion of space the element can shrink to. Full list of flex-shrink values

display: flex-basis;

Defines the default size of an element before any available space is distributed among all the flex items. Full list of flex-basis values

display: flex;

Short-hand for flex-grow, flex-shrink and flex-basis, in that order. Full list of flex values

display: align-self;

Allows the alignment of a single flex-item to be overridden. Full list of align-self values

Again, I highly recommend you check out the list of flexbox resources above, which are chock full of examples that help with the understanding of to use flexbox in your code.

display: grid;

For anything related to the Grid layout, I always refer to Rachel Andrew, whom I regard as the guru of CSS grids. She has been spearheading the effort to increase awareness about this new display property through her talks, articles and tutorials.

CSS grid gives us a way to create grid systems and control the positioning of grid items purely through CSS, a clear separation of concerns from HTML. When used together with media queries, CSS grid becomes a powerful addition to your tool-belt when it comes to designing and building flexible layouts.

The current CSS Grid Layout Module Level 1 we have now started off as a working draft in 2011. Like flexbox, this specification came about from the growing need to have a proper method for laying out content on the web without compromising the semantics of HTML.

Note that CSS grid is not officially implemented in any browser, although Microsoft Edge and Internet Explorer support an older version of the specification behind the `-ms-` prefix. This is not surprising because a majority of the editors for the original grid specification were from Microsoft.

After the messy implementation of the flexbox specification, the development of CSS

grids is taking a different approach. Browser vendors make use of vendor prefixes to add experimental features to browsers for developers to test out. This helps with the process of refining the specification and work out any kinks before they become official.

Instead of doing that, CSS grid has developed behind a flag. It has to be manually enabled by developers. In Chrome and Opera, navigate to `chrome://flags` and `opera://flags` respectively, and enable "experimental web platform features." For Firefox, navigate to `about:config` and set `layout.css.grid.enabled` and `layout.css.grid-template-subgrid-value.enabled` to true.

Key CSS grid terminology

display: Grid Container;

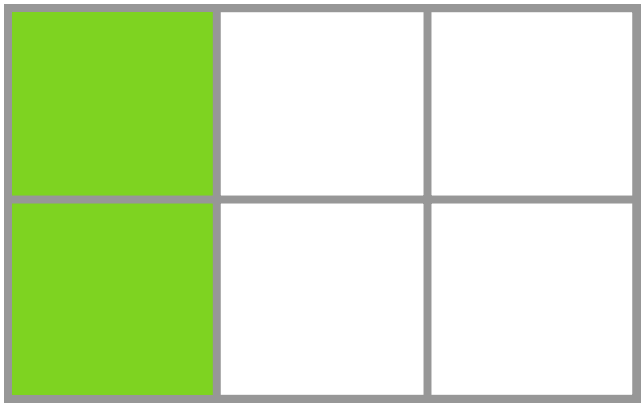
Similar to the flex container concept, where applying `display: grid;` to an element makes it's direct descendants (child elements) grid items..

display: Grid Item;

If an element's parent has `display: grid;` applied to it, then this element is considered a grid item. A grid item's child elements are NOT considered grid items..

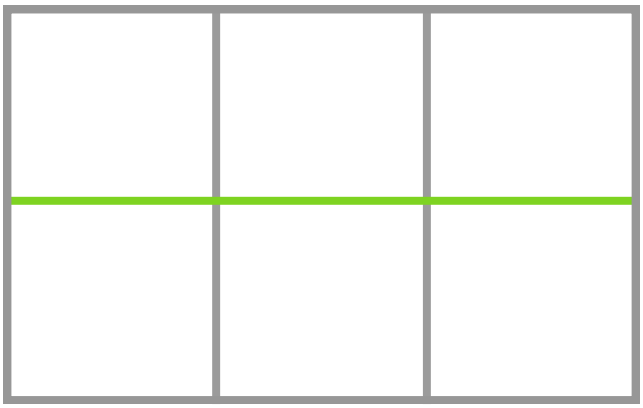
display: Grid Track;

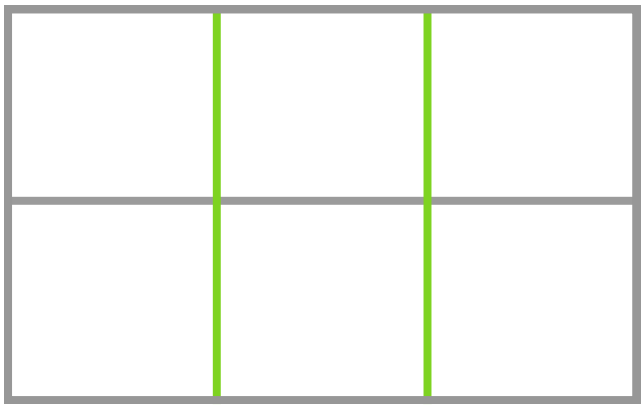
Can be either the column or row of the grid.



display: Grid Line;

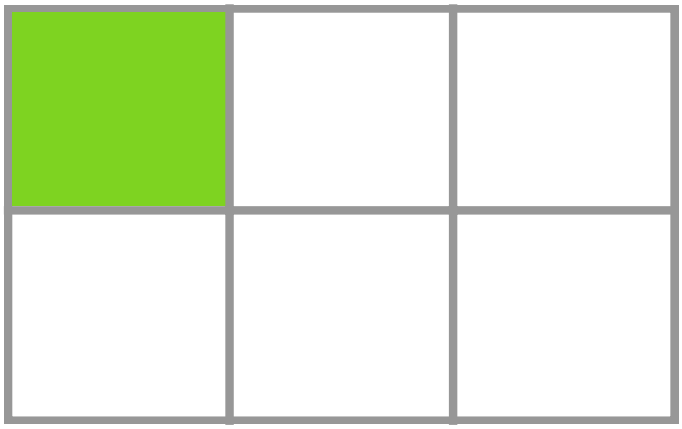
Lines that define the structure of the grid. Think of them as the lines between the grid tracks.





display: Grid Cell;

An individual grid unit, the space enclosed by adjacent horizontal and vertical grid lines.



display: Grid Area;

Now this is the cool part. Grid allows you to define an area made up of multiple grid cells.

To try to cover grid in this short subsection is really doing the specification a disservice because the totality of what it can do is huge. Please do read through the following

resources and experiment around with CSS grids. In fact, you can go to Grid by Example right now and access links to various CodePens that demonstrate how to use CSS grids for all kinds of use-cases.

- Deep Dive into Grid Layout Placement by Manuel Rego Casasnovas
- A Complete Guide to Grid by Chris House
- Grid by Example by Rachel Andrew

The relatively obscure and experimental

display: run-in;

Now this is a fun one I hadn't heard of until I started reading the CSS Display specification. And I also uncovered the 2010 article, CSS Run-in Display Value by Chris Coyier. Unfortunately, it seems that browser vendors are not fond of this specification at all and it has since been removed from all browsers, so you can think of this as an alternate reality specification ¯_(ツ)_/¯

Theoretically, if you set an element's *display* property to `run-in`, it renders as a **run-in box**. The use-case is to have a native method to create run-in headings, which in graphic design parlance is a heading positioned on the same line as the next line of body copy.

Sameen Shaw. Also known as **Indigo Five Alpha**, **Dr. Sameen Shaw** or simply **Shaw**, is a physician and a former operative for the U.S. Army Intelligence Support Activity. Prior to joining the team Shaw was part of an operation known as Catalyst Indigo, responsible for acting on relevant list intelligence delivered by the Machine, which she knew only as "Research".

You could use floats to achieve a similar effect, but it is sort of a hack-ish method. Lining up the baseline of the header with the body copy is quite challenging, as you have to tweak the font-size of the header and the line-height of the body copy until they match up. And there may be situations where the header just 'catches' more than a single line.

If you want to use `display: inline` on the header instead, it won't work unless you nest the header element in the paragraph element of body copy (because `p` is a block element), and that is semantically incorrect. So I personally would have liked to see this implemented, but I suppose the browser vendors have more high priority specifications to worry about at the moment.

display: ruby;

This particular property needs an introduction to the <ruby> element for it to make sense to you. In a nutshell, there is an element for displaying annotations alongside a base line of text, usually to help with pronunciation. They're a pretty common sight for East Asian languages, like Chinese or Japanese. Most of the articles I came across during my research were dated around 2010, so I wrote about the 2016 state of HTML <ruby>.

There are some parallels between `display: ruby;` and `display: table;`, but the specification strongly discourages applying ruby display values to non-ruby elements like `span` to display ruby text. Rather, we should markup our content using the HTML ruby elements so screen readers and non-CS renderers can interpret the ruby structures.

display: ruby;

Corresponds to the `<ruby>` HTML element. It generates a ruby container box, which establishes a ruby formatting context for child elements marked as internal ruby boxes.

display: ruby-base;

Corresponds to the `<rb>` HTML element. An internal ruby box in the ruby formatting context.

display: ruby-text;

Corresponds to the `<rt>` HTML element. An internal ruby box in the ruby formatting context.

display: ruby-base-container;

Corresponds to the `<rbc>` HTML element. An internal ruby box in the ruby formatting context.

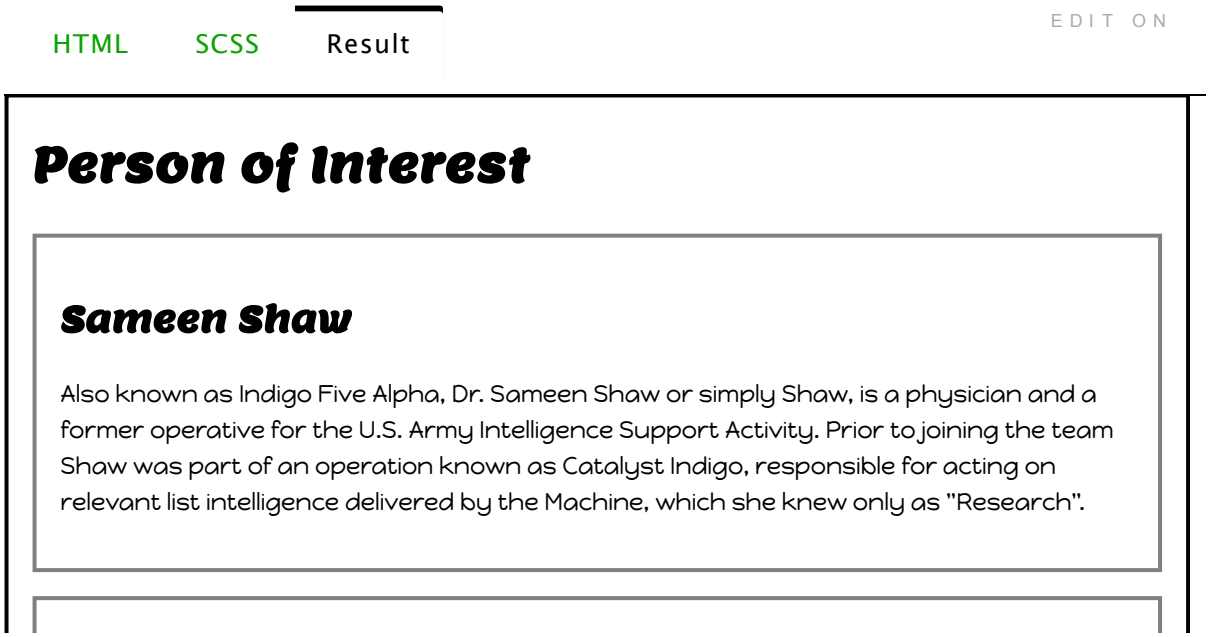
display: ruby-text-container;

Corresponds to the `<rtc>` HTML element. An internal ruby box in the ruby formatting context.

display: contents;

> *“The element itself does not generate any boxes, but its children and pseudo-elements still generate boxes as normal. For the purposes of box generation and layout, the element must be treated as if it had been replaced with its children and pseudo-elements in the document tree.*
> *- CSS Display Module Level 3* ”

What the specification is trying to say is that, when you set `display: contents` on an element, it will disappear from the DOM but all its children remain and take up the space it occupied. Unfortunately, this specification is only supported by Firefox for now. Resize the full size CodePen in Firefox to get a feel of how it works.



I've managed to uncover 2 articles that talk about this display property thus far, Firefox is releasing support for CSS display: contents by Sam Rueby and Vanishing boxes with display contents by Rachel Andrew. Rachel Andrew also presents a fantastic use-case for this property with flex-items. Do check out both articles.

Wrapping up

Whew, that ended up being way longer than I initially expected. So a big thank you if you actually read the whole thing. I'm really excited about the new options we'll have very soon to create unique layouts without having to resort to hacks, and I hope this post will encourage you to learn more about CSS layouts as well.

Further reading

- Evolution of CSS Layout: 1990s to the Future by Fantasai
- CSS Display Module Level 3
- MDN CSS display reference

Credits: OG:image from Design layout for a book by Olivier Reynaud

CSS

Hello!

I'm Hui Jing, a designer and front-end developer who believes HTML and CSS are the foundation of the web.

Connect with me

@ Email

GitHub

Twitter

Drupal.org

Hand-crafted with love by Chen Hui Jing.

© 2016 Chen Hui Jing. All rights reserved.

Resources

Style Guide