CSS IV
(Responsive Design)

Programming with
Web Technologies

THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Auckland
ICT Graduate School

Last time

- Filters
- Transforms
- Transitions
- Animations

Today

- Responsive design
- Media queries
- Modern page layouts
    - CSS Grid

Responsive Design

The story so far

So far the web pages you have developed have been fairly simple. Elements have been placed on the page and mostly allowed to lay themselves out, and no consideration has been made for the size or type of display being used for viewing the document

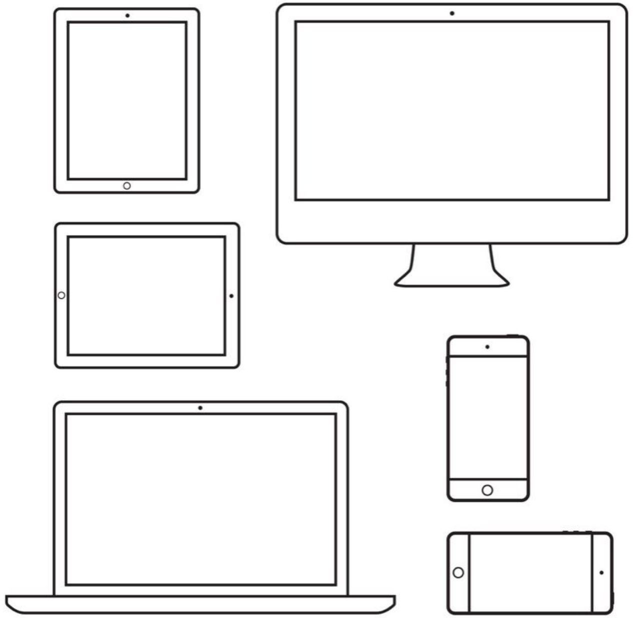
This style of web pages might have been fine in the 90's and early 00's, but not now. Modern computing devices include the cellphone, tablet, and smartwatch, and there are often more users on these than on 'traditional' desktops devices

We need to ensure that our pages work well on these devices

Device considerations

The types of devices that we need to consider can be broadly split into 5 groups:

- Regular computer displays
  - Desktops, Laptops, Projectors
- Small displays
  - Phones, Tablets, Smartwatches
- Large displays
  - Digital signage, Televisions
- Accessibility tools
  - Screen readers, Braille
- Print



Device considerations

Each of these device categories have their own layout requirements if we want to make our page usable and attractive, and these are often complex and contradictory

Regular displays favor a more open, spread out layout with plenty of space; whereas on a small screen device you want to make the best use of limited space, so the layout is compact and not wasteful

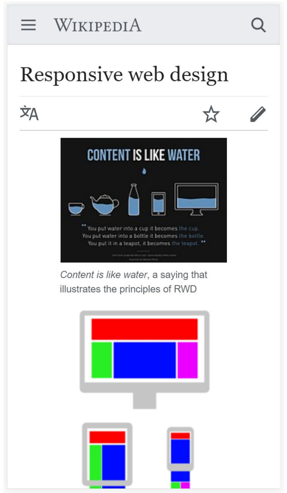
When printing pages you want dense, simple, black text with minimal wasteful images; but on large format screens you want bright colors that stand out with big clear text

Device considerations

If we have to support these different devices, how do we write our web pages?

One approach that was popular throughout the 2000's and early-mid 2010's was to have seperate pages for each of the supported devices. This leads to different URLs for each device, with layouts that need to be maintained separately - difficult and not scalable with the variety of devices

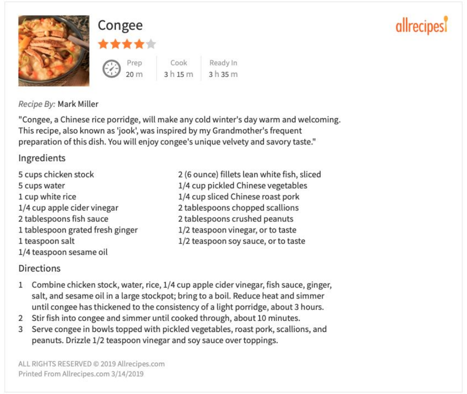
m.wikipedia.org



Wikipedia
Responsive web design
CONTENT IS LIKE WATER
" You put water into a cup it becomes the cup.
You put water into a bottle it becomes the bottle.
You put it in a teapot, it becomes the teapot. "

Content is like water, a saying that illustrates the principles of RWD

allrecipes.com/.../print/...



Congee

allrecipes

Prep	Cook	Ready In
20 m	3 h 15 m	3 h 35 m

Recipe By: Mark Miller

"Congee, a Chinese rice porridge, will make any cold winter's day warm and welcoming. This recipe, also known as 'jook', was inspired by my Grandmother's frequent preparation of this dish. You will enjoy congee's unique velvety and savory taste."

Ingredients

5 cups chicken stock
5 cups water
1 cup white rice
1/4 cup apple cider vinegar
2 tablespoons fish sauce
1 tablespoon grated fresh ginger
1 teaspoon salt
1/4 teaspoon sesame oil
2 (6 ounce) fillets lean white fish, sliced
1/4 cup pickled Chinese vegetables
1/4 cup sliced Chinese roast pork
2 tablespoons chopped scallions
2 tablespoons crushed peanuts
1/2 teaspoon vinegar, or to taste
1/2 teaspoon soy sauce, or to taste

Directions

1 Combine chicken stock, water, rice, 1/4 cup apple cider vinegar, fish sauce, ginger, salt, and sesame oil in a large stockpot; bring to a boil. Reduce heat and simmer until congee has thickened to the consistency of a light porridge, about 3 hours.
2 Stir fish into congee and simmer until cooked through, about 10 minutes.
3 Serve congee in bowls topped with pickled vegetables, roast pork, scallions, and peanuts. Drizzle 1/2 teaspoon vinegar and soy sauce over toppings.

ALL RIGHTS RESERVED © 2019 Allrecipes.com
Printed From Allrecipes.com 3/14/2019

Media Queries

Media queries - Types

- A much better way to deal with supporting differing devices
- Supported on most devices beginning from 2012
- Allow us to write blocks of CSS that are conditionally applied, depending on what type of device the page is being viewed on

```css
@media screen { /* Apply rules when viewed on a screen */
    p { color: red; font-family: Impact; }
}
@media print { /* Apply rules when printing */
    p { color: black; font-family: serif; }
}
```

Media queries - Types

- A media query's **type** filters for different types of devices on which we might render a page
- The following types are supported

  `all`, `braille`, `embossed`, `handheld`, `print`, `projection`, `screen`, `speech`, `tty`, TV

- Useful to start - it allows us to detect accessibility devices, printing and screen categories
- But it doesn't give any granular control when we want to deal with different screen sizes (e.g. desktop monitors vs mobile device displays)

Media queries - Features

- **Features** extend media queries, adding the ability to filter based on the capabilities of the device, which can be combined with the type query.
- There are a variety of features you can add into these queries to filter on:
  - width, height, aspect ratio, device orientation, color support, and a further growing list of features.
- Lets us get much more granular with our queries to set different **breakpoints** where the layout can change in the document

```css
@media screen and (max-width: 768px) {} /* small display */
@media screen and (min-width: 768px) and (max-width: 1024px) {}
@media screen and (min-width: 1024px) {} /* regular display */
```

Media Queries - Combinatorial logic

- Media queries can combine **type** and **feature** queries with the `and` operator
    - We also have the `not` operator that can be used to negate the result of a query
    - soon we will have the `or` operator fully supported
    - Full complement of boolean operators we are used to so we can create complex queries
- We can currently combine two queries so that if the device matches either the block will be applied using the same syntax as we do in css, with a comma:

```
@media (max-width: 768px), print {}
```

Using media queries

With what we know now, we can write one CSS stylesheet that can style our page for whatever device we like - better than maintaining multiple versions of the site!

But there are still some concerns:

- If we need to write selectors inside the media query blocks, won't there be a lot of duplication of selectors?
- If we have CSS for all the device types in one file, won't it get hard to maintain and won't it be slower to load?

Eliminating duplication

- We do not need to rewrite the rules inside each media query block
  - We can write rules at the top-level of the stylesheet (outside of any media query blocks) that will apply to every device
  - We can nest media queries so that we can specialize the CSS for devices
- Rules will be applied as they are encountered in the file, reading top to bottom applying selectors at the top level of the file and within any matching media query blocks.
- We can overwrite earlier general rules inside media queries where necessary

Eliminating duplication

```css
/* Applies to all devices */
p { font-family: serif; }

@media print { /* Applies only to print */ }

@media screen {
    /* Applies to all screens */
    p { color: goldenrod; }

    @media (max-width: 768px) { /* small displays only */ }
    @media (min-width: 768px) { /* bigger displays only */ }
}
```

Breaking up the CSS

- Avoiding repetition is great, but it can still leave us with a large, complex, less maintainable CSS file
- We can break this large CSS file up into smaller CSS files then include each one separately in our HTML

```
<head>
    <link rel="stylesheet" href="./css/common.css">
    <link rel="stylesheet" href="./css/screen-small.css">
    <link rel="stylesheet" href="./css/screen-large.css">
    <link rel="stylesheet" href="./css/print.css">
    <!-- etc -->
</head>
```

Breaking up the CSS

Or alternatively include the smaller CSS files into one master css file which is imported to the HTML

```
<head><link rel="stylesheet" href="master.css"></head>

/* in master.css */
@import url('./css/common.css');
@import url('./css/screen-small.css');
@import url('./css/screen-large.css');
@import url('./css/print.css');
/* etc */
```

Problem solved?

- This has made our CSS files easier to deal with, as each deal with one device. However there is still an issue, we are downloading data that we don't need.
- When browsers hit a `link` or `@import` statement, they download the referenced file and then parse it
  - Waste of time and bandwidth if the data turned out not to be necessary
  - Not a big deal on high-speed non-data-capped internet connections, but for people stuck on dial-up or satellite it is not ideal
- Wouldn't it be nice if we could use media queries to prevent downloading the unneeded files?

Selective Downloading

- When `link`ing a file in HTML, we can supply a media attribute which contains a media query as a value
    - If the device matches the query, the file is fetched

```
<link rel="stylesheet" href="small.css"
      media="screen and (max-width: 768px)">
<link rel="stylesheet" href="print.css" media="print">
```

- For `@import` in CSS, we simply append the query to the line to achieve the same result

```
@import url('large.css') screen and (min-width: 768px);
```

Modern Page Layouts

THE UNIVERSITY OF AUCKLAND Te Whare Wānanga o Tāmaki Makaurau NEW ZEALAND

THE UNIVERSITY OF WAIKATO Te Whare Wānanga o Waikato

Auckland ICT Graduate School

Layouts - A history lesson

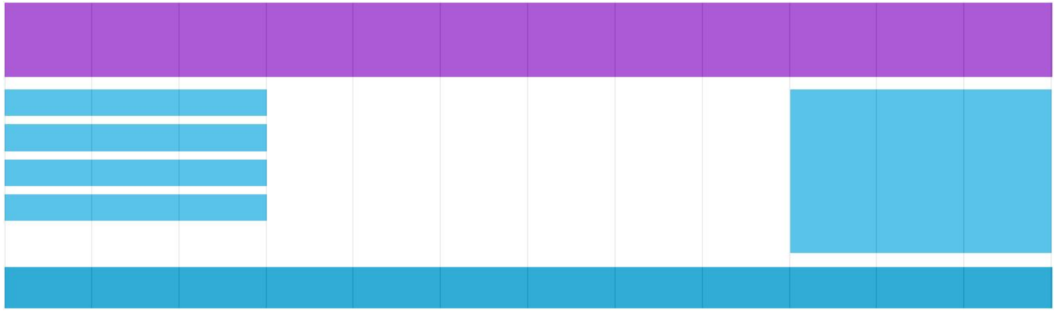
Modern websites are more complex and visually pleasing than what we have been working on so far. How do they accomplish this?

In the bad old days, complex layouts were accomplished using `table`s. These suited well as people understood them and could rapidly prototype the layout on paper or in Excel.

Doing this is not good practice however as the browser loses the ability to differentiate between data and layout - not good for accessibility or discoverability through search engines. They are also complex to maintain. Getting started is easy, but maintaining the mess of `tr`s and `td`s that make up your layout quickly becomes unpleasant

Layouts - More history

Following tables, the use of floats become the choice of champions. With clever CSS, floats could be used to accomplish complex reactive web pages through the use of fixed-width 'grid' CSS classes. Starting from a sketch like so



You could sketch column guides and align elements to each of these. You could work out that each column is 100% / col-num wide (~8.3% in this case)

Float-based grid systems

From that sketch, you could create some CSS classes like this:

```
.col-1 { width: 8.3%; float: left; }
.col-2 { width: 16.7%; float: left; }
.col-3 { width: 25%; float: left; }
…
.col-11 { width: 91.7%; float: left; }
.col-12 { width: 100%; float: left; }
```

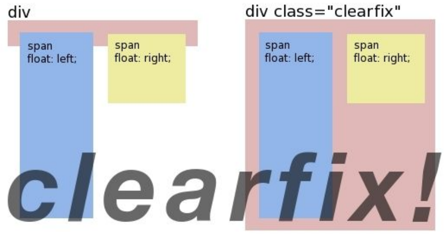
And then apply these to elements in our layout to make them the correct width. Float will cause elements with these classes to 'stack' left-to-right. If you create 'row' containers to hold these elements you get a usable layout

Float-based layouts

Layouts with floats are still fairly prevalent, not in small part to them being used by popular frameworks like Bootstrap - and they work pretty well! However, they aren't perfect.

Floats were never designed with the intention of being used for complex layouts - they were designed for moving blocks to the left or right of another block. There is no centering option, so you have to pad to the left/right with other elements to move a block to the center - not ideal.

Blocks that contain only floating elements are unable to calculate their height, requiring a 'clearfix' workaround to get proper layout.

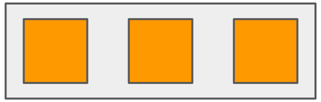


Modern layouts

- The current approach for layouts is to use two new CSS features, which were designed from the ground up to facilitate layouts. Challenging layouts are now quite simple!
- **Flexbox** is designed for 1-dimensional layouts, allowing for arranging and aligning items in a row or a column
- **Grid** addresses 2-dimensional layouts, presenting something that gives the same ease-of-thought of using a table, but with none of the extraneous tags

Flexbox

- **Flexbox**, as mentioned, is concerned primarily with laying out, aligning and distributing elements in a single row or column.
- Can be used for much more complex layouts than we will discuss, however it is intended to be used in conjunction with CSS Grid
    - A combination of the two technologies should be used for those more complex layouts
- Flexbox layouts utilize two components, the **container** in which the layout is described and where elements appear; and **items** which are laid out according to rules set by the container



Flexbox containers - `flex-direction`

- Containers are any element into which we can place other elements that contain one key CSS property: `display: flex`;.
  - With this property set, we gain access to the other flex-related properties that can control the layout for any items nested within the container
- The `flex-direction` property controls the direction child components are laid out, and can have the following values:
  - `row`: Horizontal alignment, left-to-right
  - `row-reverse`: Horizontal alignment, right-to-left
  - `column`: Vertical alignment, top-to-bottom
  - `column-reverse`: Vertical alignment, bottom-to-top

Flexbox containers - `align-items`

- The `align-items` property determines how child elements are aligned with respect to each other. It can have the following values:
    - `center`: Aligns items by their centerlines
    - `flex-start`: Aligns items by their top / left margins
    - `flex-end`: Aligns items by their bottom/right margins
    - `stretch`: Causes items to stretch to become equal height
    - `baseline`: Aligns items by the baselines of text content (where an underline would appear)

Flexbox containers - `justify-content`

- The `justify-content` property determines how children are distributed within the container, and can have the following values:
    - `flex-start` or `flex-end`: Arranges items much like floating left or right would do, with the items being stacked to either side of the container.
    - `center`: Places elements with their edges against each other with an equal amount of space to the left and right sides.
    - `space-between`: Places the first and last items to the start and end of the container, with the rest of the elements spaced with equal spaces between these.
    - `space-evenly`: Extends this so that there is an equal share of space between the edge of the container and the first/last items.
    - `space-around`: Divides the space available into `2n` pieces, then places one of these spaces on either side of each element

More flexbox properties

Flexbox containers also have controls for what happens if there is not enough space in the row/column for all items - `flex-wrap`, and how to manage items aligned with `align-items` if wrapping does occur - `align-content`.

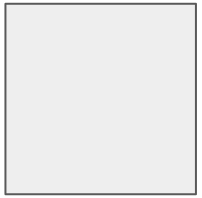
Items that are placed inside flexbox containers also have a number of properties they can set to change how they are laid out. The first is `order`; this determines where the element it is applied to will be placed in the layout. By default they will be placed as they are defined in the HTML, but `order` allows this to be changed.

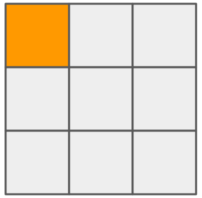
The other of particular interest is `align-self`, which lets an item override the `align-items` property set in the container

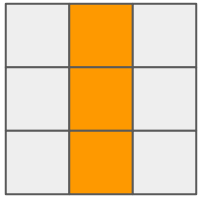
Grid

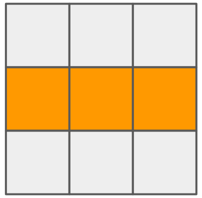
Grid expands on the ideas given by flexbox into 2-dimensions, making it perfectly suited to layouts of a whole page rather than just a small part.

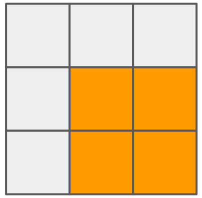
As with flexbox, Grid deals with **containers** and **items** for its layouts, but it also introduces **tracks** (rows and columns within the layout), **lines** (the 'borders' between tracks), and **areas** (rectangular groups of items within the layout)


Container


Item (cell)


Track (col)


Track (row)


Area

Grid container

- To set a container as a grid, use `display: grid;`
- This is not useful on its own - two extra properties are required to control the number and sizes of columns and rows in the grid

```
grid-template-columns: 1fr 100px auto 100px 2fr;
grid-template-rows: 1fr 2fr 3fr;
```

- This setup creates a 5 column, 3 row grid. The number of measurements provided determine the number of rows or columns, while each value determines the width or height of the column or row

Grid container - Units

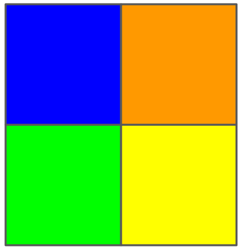
In the `grid-template-column` property from before, you will have noticed two new unit types that you haven't encountered yet:

- `auto`: indicates that the row or column should have its size determined automatically by the content contained within it
  - E.g. if the content is `100px` x `100px`, then that will be the width and height of that row/column
- `fr`: represents a `fr`action of available space. Used to create proportional layouts.
  - A `grid-template-column` definition of `1fr 1fr 1fr` will create 3 equally sized columns, while `5fr 1fr 1fr` will create a setup where the left-most column occupies 5/7ths of the page and the remainder is shared by the other 2 columns

Grid container - Layout

- Once a grid container has been defined, items can be laid out inside it.
- In Grid, as we are dealing with two dimensions, we need to layout with that in mind:
    - Elements are laid out as they are encountered, left to right, top to bottom, with each item occupying a cell in the grid

```html
<div class="container">
  <div id="blue">
  <div id="orange">
  <div id="green">
  <div id="yellow">
</div>
```



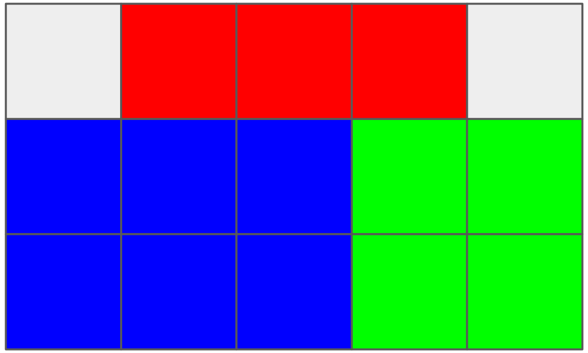
```css
.container {
  display: grid;
  grid-template-columns: 1fr 1fr;
  grid-template-rows: 1fr 1fr;
}
```

Grid container - Named areas

To make children span multiple grid cells, we can use `grid-template-areas`

- This property allows us to name areas within our grid
- Children can occupy specific named areas

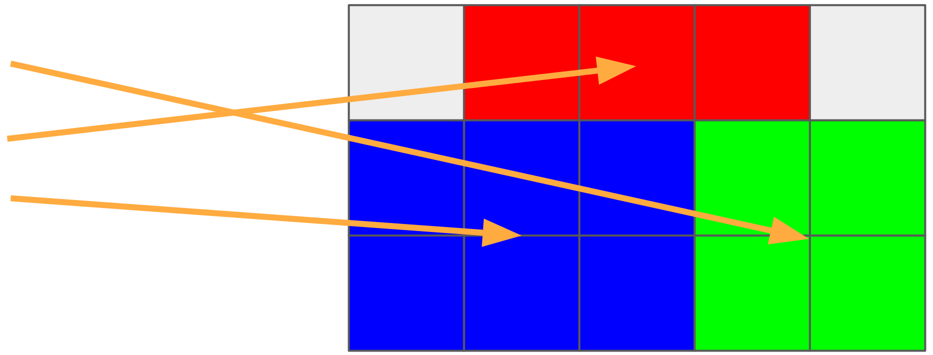
```
grid-template-columns: repeat(5, 1fr);
grid-template-rows: repeat(3, 1fr);
grid-template-areas:
    ". r r r ."
    "b b b g g"
    "b b b g g";
```



Grid Items - Named Areas

With named areas defined, items can place themselves into the appropriate area using the `grid-area` property. As they get to choose their position, the order in which they are placed in the HTML no longer matters

```
<div class="container">
  <div style="grid-area: g">
  <div style="grid-area: r">
  <div style="grid-area: b">
</div>
```



More grid

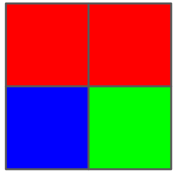
- Grid is a very complex topic, and there are a great many properties and values that we have not discussed. For example:
  - Lines can be adjusted to space between tracks
  - Lines can be named so that their areas can be defined with respect to lines rather than laid out as a visual grid
  - Items in the grid cells can be aligned and justified as with flexbox.
- There is an excellent Grid guide created by css-tricks that explains each of these properties in detail, and gives visual aids to understanding each
- However, we can use what's covered in this lecture to begin designing complex layouts with ease!

Grid and media queries

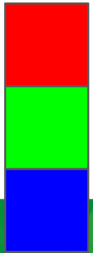
Using named areas for Grid layouts makes adapting our layouts for multiple devices very straightforward:

- We can simply modify the container `grid-template-*` properties inside the media query blocks

```css
@media (min-width: 768px) {
  .container {
    grid-template-columns: 1fr 1fr;
    grid-template-rows: 1fr auto;
    grid-template-areas: "r r" "b g";
  }
}
```



```css
@media (max-width: 768px) {
  .container {
    grid-template-columns: 1fr;
    grid-template-rows: auto auto auto;
    grid-template-areas: "r" "g" "b";
  }
}
```



Recommended Reading

- w3schools Responsive Web Design
- MDN Media Queries

- CSS Grid Garden
- Grid By Example
- MDN Grid Layout
- MDN Grid Tutorial
- CSS Flexbox Explained
- CSS Grid Explained