CSS - Summary

Basics

- 3 ways to include CSS
  - inline
    - using the `style` *attribute*

      ```
      <p style="color: blue;">lorem ipsum</p>
      ```

  - in the HTML document
    - using the `style` *tag*
    - usually added in *head* element to prevent Flash Of Unstyled Content (**FOUC**)

      ```
      <style>
        p {
            color: blue;
        }
      </style>
      ```

  - external CSS file
    - included using the link tag
    - relative or absolute path to file can be given
    - URL to a stylesheet on the internet can also be given

      ```
      <link rel="stylesheet" href="styles.css">
      ```

- Structure of a *CSS rule*
  - A CSS rule consists of a *selector* followed by a *declaration block*
  - A declaration block consists of multiple CSS delcarations, each being a css property and value. They css property and value are separated by a colon (:). The property-value pairs are separated by a semi-colon (;).

    ```
    selector {
      css_prop_1: value_1;
      css_prop_2: value_2;
      ...
    }
    ```

- Basic CSS selectors
  - *id* selector - `#id`
  - *class* selector - `.class`
  - *type* or selector - `type` (eg. `p`)
  - *attribute* and *attribute-value* selectors - `[attr]` , `[attr="value"]`
  - In terms of specificity (decreasing order of importance) - id, class = attribute, type
- Pseudo-class selectors
  - Based on **pseudo classes** that are applied to elements under certain conditions
  - Eg. `:hover` is applied when mouse hovers over (eg. a:hover { ... } styles are applied when mouse hovers over a link)
  - Specificity same as for a normal class
  - `:hover` (on mouse over), `:active` (when mouse clicked and held down)
  - `:visited` - link-specific pseudo class. applied when linked document in browser history
  - *Structural pseudo classes*

        - Pseudo classes applied based on relationship of the element with other elements (usually the parent and siblings)
        - `:first-child` - Selects elements that are the first children of their respective parents
        - `:last-child` - Selects elements that are the last children of their respective parents
        - `:nth-child( an + b )` - Selects elements that are the *an + b children* of their respective parents where n = 0, 1, 2, ...
            - For example, :nth-child(3n + 1) selects the 1st, 4th, 7th, ... children of their respective parents
        - `:first-of-type` , `:last-of-type` , `:nth-of-type( an + b )` - Selects elements that are the first, last and nth of their type within their respective parents (eg. p:last-of-type) would select the second paragraph in this structure

```
<div>
  <p>Para 1</p>
  <a href="#">Link 1</a>
  <p>Para 2</p>
  <a href="#">Link 2</a>
</div>
```

- *Combinators*
    - Operators used to combine basic selectors to form more complex selectors
    - They are used to form arbitrarily complex selectors by combining with basic selectors, pseudo selectors etc.
    - ` ` (space - ancestor-descendant combinator)
    - `>` (parent-child combinator)
    - No space - self combinator (selects elements that satisfy each individual selector)
    - `+` (adjacent sibling combinator)
    - `,` (comma - separating selectors so that same CSS rule can be applied to each of them)
- *Specificity of selectors*
    - Importance of selectors (in decreasing order) - id selector, class (including pseudo class and attribute selectors), type
    - *Specificity* is defined as a list of 4 values - *(a, b, c, d)*
    - Inline style specificity is *(1, 0, 0, 0)*
    - In-document and external style rules have a specificity for the selector.
        - In case of conflict of a CSS property (two or more selectors target the same element), the property value from the rule with higher specificity selector is chosen
        - Specificity for selectors is *(0, number_of_times_ids_appear, number_of_times_classes_appear, number_of_times_types_appear)*

```
/* (0, 2, 1, 2) */
#section-1 ol.X#ol-1 a {
  color: lightcoral;
}

/* (0, 1, 1, 2) */
ol.X#ol-1 a {
  color: lightblue;
  font-size: 1.5em;
}
```

        - The values are compared starting from left to right, till one of the values is different. The selector with the higher value at that point wins, and the conflicting property gets value from that rule.
        - *Example*: In the above example, suppose there are anchor elements that are selected by BOTH the selectors. a = 0 in both cases, b = 2 for the first rule, hence color applied to anchor elements that match BOTH selectors is lightcoral. font-size has no conflict, hence 1.5em is applied.
- Inheritance
    - Children inherit CSS properties from parents
    - Some CSS properties are inherited - eg. color, `font-size`
    - Some CSS properties are not inherited - eg. `border` (all shorthands derived from it), `padding` etc.
    - Which ones are inherited, and which are not is rather intuitive. In case of doubts, you must consult the documentation for the CSS property.

- Use `inherit` keyword as a property value, to explicitly inherit the value applied to parent. Here, the paragraph inherits padding from its parent (maybe a div, the document body etc.)

```
p {
  padding: inherit;
}
```

- Use `initial` keyword as a property value, to explicitly set the value applied to its default (and prevent inheritance in turn)

```
p {
  color: initial;
}
```

- General features of CSS properties and values
  - All *directional* CSS properties follow a clockwise order starting from top (top, right, bottom, left)
  - "Longhand properties" are generally preferred to shorthand properties, as shorthand properties end up assigning initial value for the CSS properties that remain unassigned in the shorthand representation. This can lead to confusion, and hence best avoided in complex cases.
  - CSS property values can have units. Units can be relative (eg. %) or absolute (eg. px). In general, relative units are preferred.
- Box model
  - Every element is allocated a *box*, i.e. a rectangular space on the page (boxes allocated top-down in source order, i.e. in the order they appear in HTML)
  - This is called *normal flow of rendering*. Note that there are situations where rendering does not follow the normal flow (eg. when you set the CSS `position` property - this is covered later)
  - A box has upto 4 parts
    - *Content area* - where content within the element appears
    - *Padding area* - space between border and content
    - *Border area* - borders for an element
    - *Margin area* - space (both vertical and horizontal) between 2 consecutive elements
  - An element box fills the content area of its parent's box.
  - Block-level elements get full width of parent (100%) by default. They respect all box model properties (explained below)
  - Inline elements get only as much width as required. They ignore `width`, `height`, `margin-top`, ```margin-bottom`` properties.
  - Various CSS properties set up the dimensions of the parts of a box
    - `width`, `height` - content area dimensions. When `box-sizing: border-box;` is used, this changes to mean the border box area dimensions (a box that includes border, padding and content areas).
    - `max-`, and `min-` modifiers help set up upper and lower limits on width and height

```
div {
  width: 50%; /* % of parent element content width */
  max-width: 400px; /* upper limit on width */
  min-width: 300px; /* lower limit on width */
}
```

    - `padding` sets up padding space
    - `padding` is a shorthand property consisting of `padding-top`, `padding-right` etc.
    - `border` is shorthand for `border-top` etc.
    - `border-top` is again shorthand for `border-top-width`, `border-top-style` and `border-top-color`
    - There are also shorthand properties `border-width`, `border-style`, and `border-color`.
    - `border-width` CANNOT be specified in relative units like %
    - `box-shadow` sets up shadow with x, y offsets, spread and blur distances, and color. You can apply multiple shadows (separate by commas)

```
div {
  box-shadow: 0px -20px 20px 40px rgba( 128, 128, 128, 0.5 ), 20px 20px 10px 10px lightblue;
}
```

        - `border-radius` applies rounded border (can be given in % or pixels)

          ```
          img {
            border-radius: 50%;
          }
          ```

        - `margin` is shorthand with corresponding "longhand" versions
- Background
    - `background-color` sets background color

      ```
      div {
        background-color: crimson;
      }
      ```

    - `background-image` sets background image. Prefer giving a relative path. Image is set up with natural dimensions, and repeats in both directions by default.

      ```
      div {
        background-image: url( "images/abstract_bg_image.jpg" );
      }
      ```

    - `background-repeat` causes image to repeat (`repeat`), not repeat in both directions (`no-repeat`), reeat horizontally (`repeat-x`), or vertically (`repeat-y`).

      ```
      div {
        background-image: url( "images/abstract_bg_image.jpg" );
        background-repeat: no-repeat;
      }
      ```

    - `background-size` can be used to set the image size to other than natural dimensions.
        - Value `cover` causes the image to expand / shrink to fit the entire element box

          ```
          div {
            background-image: url( "images/abstract_bg_image.jpg" );
            background-size: cover;
          }
          ```

        - Value `contain` causes the image to expand / shrink till an opposite pait of edges is hit (either height of box is filled or width is). Then the default repeat behavior fills the other dimension (unless it is overriden)

          ```
          div {
            background-image: url( "images/abstract_bg_image.jpg" );
            background-size: contain;
          }
          ```

        - size can also be set using relative and absolute units. Relative units set image dimensions with respect to element box dimensions. For example, to fill up entire space with the image resized according to element box dimensions, we can set

          ```
          div {
            background-image: url( "images/abstract_bg_image.jpg" );
            background-size: 100% 100%;
          }
          ```

    - `background-position` is used to move the image from default position (the default is (0, 0) which is the top-left corner of the element's padding box - which is the default value for another property - the `background-origin`)
        - percentage and pixel values can be used. percentages values are not with respect to the top-left corner however (check https://css-tricks.com/almanac/properties/b/background-position/)
        - named values like `top` , `right` , `bottom` , `left` and `center` can also be used to set position
    - Multiple backgrounds can be set by separating with commas. The related background properties can have respective values set at corresponding position within a comma-separated list too.
- Handling overflow
    - Content can overflow the element's horizontal and vertical space (especially when width or height is fixed)

- We use `overflow` property to handle overflowing content
  - Hide overflowing content (in both directions)

    ```
    div {
      overflow: hidden;
    }
    ```

  - Introduce scrollbars (in old browsers they are introduced even when there is no overflow)

    ```
    div {
      overflow: scroll;
    }
    ```

  - Or introduce scrollbars automatically, i.e. only when required

    ```
    div {
      overflow: auto;
    }
    ```

  - You can also set overflow behavior in each direction individually (`overflow` is shorthand for `overflow-x` and `overflow-y`)

    ```
    div {
      overflow-x: auto;
      overflow-y: hidden;
    }
    ```

  - You can also set up handle for resizing using `resize` - this is not generaly preferred, as when user resizes a box, the layout can get messy. You need to hide overflowing content for this to work.

    ```
    div {
      overflow: hidden;
      resize: both;
    }
    ```

- The `display` properties controls whether an element is `inline` , `block` , `inline-block` , `none` etc.
  - It has other possible values - `table` , `table-row` , `table-cell` , `flex` , `grid` etc.
  - A div is block-level by default - you can change it to another type like inline. However, once it is inline, some CSS box model properties do not have effect. An anchor is inline by default - you can make it block-level. Once it is block-level, it respects the CSS box model properties it ignores by default.

    ```
    div {
      display: inline;
      width: 50%; /* since this is inline, width is ignored! */
    }

    a {
      display: block;
      width: 50%; /* this takes effect */
    }
    ```

  - `inline-block` display type has characteristics of both inline and block-level elements.
    - Like block-elevel elements, they respect width setting etc.
    - Like inline elements, it will start on that line instead of a new line, if there is sufficient space left on a line. Default width is only as much content requires - again similar to inline elements.

      ```
      section {
        display: inline-block;
        width: 50%;
      }
      ```

    - **Note**: The inline-block display type is quite useful for layout purposes. Example, laying out list items (say navigation menu list items, each with links) in a line (with equal width for each list item).
  - The value `none` for display, takes the element out of the normal flow of rendering. It does not appear on the page! This is useful for hiding elements (eg. a dialog box which appear only on certain conditions - the display

will be changed accordingly in JavaScript to have it appear).

- `visibility` controls the visibility of an element. For example, the value `hidden` makes the element invisible.

```
div {
    visibility: hidden;
}
```

  - The element remains in the normal flow of rendering. The element is thus allotted space but does not show up there.
  - This is different from how `display: none` works (which is as good as element not being present on the page, at least for UI purposes).
- The *root* element (the CSS3 `:root` selector)
  - `:root` selects the html element
  - Being a pseudo-class selector, it has higher specificity than the `html` type selector
  - It is often used to create custom CSS properties (CSS variables) that can be used inside of any element (since they are inherited from parent to child)
  - The font-size set on this element forms the unit for `rem`

```
:root {
  font-size: 16px;
}
div {
  font-size: 2rem; /* 2 * 16px = 32px */
}
```

  - Tweaking the root element font-size for example helps change font-size for all elements whose font-size is set using the rem unit

```
:root {
  font-size: 16px;
}

/* change root element font-size for low width devices */
@media all and (max-width: 480px) {
  :root {
      font-size: 10px;
  }
}
```

- CSS variables (custom CSS properties)
  - CSS3 allows developers to create *custom CSS properties* (they begin with double hyphens)
  - These are also called *CSS variables*
  - These are usually set up in the :root element (although not necessarily set there).

```
:root {
  --green: green;
  --padding-y: 10px;
  --padding-x: 20px;
}
```

  - Use var() to use a custom CSS property. When using a custom CSS property you can give a fallback value that will be applied if the CSS property is missing.

```
.btn {
  padding: var( --padding-y, 10px ) var( --padding-x, 20px );
  border: 1px solid var( --green, green );
  color: var( --green, green );
}
```

- *Text and font properties*
  - `font-family` is used to specify a *font stack*. A font stack consists of fonts that will be tried in order till one is available (and it is then applied).
    - It is recommended to use fonts from the same family in a stack

        - It is recommneded to provide the font family as a fallback at the end.

          ```
          p {
            font-family: Cambria, Cochin, Georgia, Times, 'Times New Roman', serif;
          }
          ```

        - Some font families are `serif`, `sans-serif`, `monospace`, `cursive`
    - `font-size` specifies size of font
        - A commonly used absolute unit is `px`
        - Some common elative units are `em` (relative to inherited font-size), and `rem` (relative to root element font-size)

          ```
          p {
            font-size: 24px;
          }

          div {
            font-size: 2em;
          }

          section {
            font-size: 2rem;
          }
          ```

    - `line-height` sets height of line
        - It can be specified using absolute or relative units. The unitless value (relative value) is preferred. The line-height in this case is set with respect to font-size.

          ```
          p {
            line-height: 1.5;
          }
          ```

    - `text-align` aligns text. Some values are `left`, `right`, `center` and `justify`
    - `text-decoration` decorates text. Some values are `underline`, `overline` and `line-through`
    - The value `none` is used to remove underline - this is useful in case of links whose default underline may be removed like so

      ```
      a {
        text-decoration: none;
      }
      ```

    - `font-style` can be set to `italic`
    - `font-weight` can be set to `bold` or `normal`. Numbers 100, 200, 300, ..., 900 may also be used. This is usually useful for custom fonts which have more variants than just bold and normal.
    - `text-transform` can be used for case transformations. Some values are `uppercase`, `lowercase` and `capitalize`
- Colors
    - Colors (color, background-color, border-color etc.) can be specified in many ways
    - You can specify *named colors*, colors as a combination of *red*, *green*, and *blue* (primary colors), or as a combination of *hue* , *saturation*, and *lightness*.
    - Some named colors are `red`, `green`, `blue`, `yellow`, `cyan`, `magenta`, `white`, and `black`. There are many such named colors (some standard, and some available on certain browsers).
    - Using `rgb( red, green, blue )` where each color is value between 0 - 255. 0 means the color is not added at all, and 255 means the color is added with full hue. Gray shades have equal amount of 3 primary colors

      ```
      div {
        color: rgb( 64, 128, 255 );
      }

      section {
        /* shade of gray */
        color: rgb( 64, 64, 64 );
      }
      ```

- Using `rgba( red, green, blue, alpha )`, we can additionally supply an `alpha` value between 0 - 1. 0 makes the color ```transparent``, and 1 makes it opaque. Values in between make it semi-transparent to varying degrees.

```
div {
  color: rgba( 64, 128, 255, 0.5 );
}
```

- Using `#rrggbb` abd `#rgb` syntaxes - here rr, gg, and bb are hexadecimal values between 00 - ff (i.e 0 - 255 in decimal), and r, g, b are values between 0 - f (i.e. 0 - 15 in decimal). In #rgb, the color expands to #rrggbb (i.e. the r, g, and b hex digits are repeated).

```
div {
  color: #4080ff;
}

section {
  color: #48f; /* same as #4488ff */
}
```

- Using `hsl( hue, saturation, lightness )` and `hsla( hue, saturation, lightness, alpha )`. Hue is the base color from a color wheel, and is specified as an angle (degree units) between 0 - 360. Saturation and lightness are values between 0 - 100%. Saturation adds an amount of the chosen hue, and lightness is the degree of "whiteness" (100%), or "darkness" (0%).

```
p {
  background-color: hsla( 200, 100%, 50%, 0.5 );
}
```

- Opacity
  - `opacity` acts like an alpha value for an entire element (takes value between 0 - 1). It makes the element on which it is set, semi-transparent to varying degrees.

```
div {
  opacity: 0.5;
}
```

- Floats
  - The `float` property
    - It can be set to `left`, `right`, or `none`.
    - It floats the element as far as possible towards the left/right edge of the parent. The rest of the content within the parent element flows around the floated element.

```
img {
  float: right;
}

aside {
  float: left;
}
```

  - The problem of height for elements with floated children
    - The children that are floated within an element, do not (by default) contribute to the calculation of height of the element.
    - This can cause elements that appear after the one with floated children to appear next to the floated children.
    - We prevent this in various ways
      - Using `clear` with the value `left`, `right`, or `both` on the element that appears after the one with floated children. This makes sure there is nothing floated to the left/right/both directions of the element.

```
#element-after-one-with-floated-children {
  clear: both;
}
```

        - Techniques for containing floated children exists - these make sure the floated children indeed contribute to the height of the parent element.
            - Using `overflow` with value `auto`, or `hidden` ensures inclusion of floated children in calculation of height of element.

```
#element-with-floated-children {
  overflow: auto;
}
```

            - Using the *clearfix* class on element with floated children - this is a standard technique (discussion of how it works is out of scope of this course)

```
<head>
  <style>
      .clearfix:after {
          content: ".";
          display: block;
          height: 0;
          clear: both;
          visibility: hidden;
      }
  <style>
</head>
<body>
  <div class="clearfix">
      Assume this divison has floated children...
  </div>
</body>
```

    - Column layouts using float
- The `position` property takes the following values
    - `static` - this is the default value for all elements. The element is positioned according to the normal flow of rendering. The `top`, `right`, `bottom`, and left `left` properties (if set), are ignored.

```
div {
  position: static; /* not required - it is the default value */
  top: 100px; /* ignored */
}
```

    - `relative` - The element is positioned according to the normal flow of rendering. The `top`, `right`, `bottom`, and left `left` properties (if set), are respected. The element is moved from its normal flow rendering position by the amount specified by these properties.

```
div {
  position: relative;
  top: 100px; /* moves 100px downwards from its default position in normal flow of rendering */
}
```

    - `absolute` - The element is taken out of the normal flow of rendering. It is instead positioned with respect to some ancestor element.
        - The default is the document element (`html` element).

```
div {
  position: absolute;
  bottom: 0px; /* puts the element at the bottom of the document */
}
```

        - It is actually positioned with respect to the closest non-static positioned ancestor (i.e. closest ancestor whose position is `relative`, `absolute`, `fixed`). Such an ancestor is called the element's *offset parent*. In the example below, the element with class dialog is the offset parent of the element with class close (the close button is thus absolutely positioned with respect to the dialog, rather than the document).

```
<head>
  <style>
      .dialog {
          position: relative;
      }
```

```
            .dialog .close {
                position: absolute;
                top: 6px;
                right: 20px;
                font-size: 1.5em;
            }
        </style>
      </head>
      <body>
        <div class="dialog">
            <span class="close">&times;</span>
            Lorem ipsum dolor, sit amet consectetur adipisicing elit. Dolorem itaque excepturi ea, f
        </div>
      </body>
```

- `fixed` - The element is taken out of the normal flow of rendering. It is instead positioned with respect to the browser viewport. This is useful for position modal dialogs etc. that do not scroll when the document scrolls.

```
<head>
  <style>
      .dialog-overlay {
           /* positioned with respect to the viewport */
          position: fixed;
          background-color: rgba( 128, 128, 128, 0.5 );
      }

      .dialog {
          position: relative;
          top: 50vh;
          width: 50%;
          margin: 0px auto;
          margin-top: -75px; /* dialog is 150px height (-150px/2 = -75px) */
      }

      .dialog .close {
          position: absolute;
          top: 6px;
          right: 20px;
          font-size: 1.5em;
      }
  </style>
</head>
<body>
  <div class="dialog-overlay">
      <div class="dialog">
          <span class="close">&times;</span>
          Dialog box contents go in here...
      </div>
  </div>
</body>
```

- Scoping CSS selectors
  - When your selectors may inadvertently affect someone else's styles, make sure to *scope* them (make them more specific), so that they do not end up affecting other elements that were not intended to be styled by you. This is usually done by adding classes of the ancestors of the element you would like to select.

```
/* do not use a class like close as a standalone selector - someone else on your development team
.close {
  ...
}

/* prefer this instead (this may require further scoping in some cases however) */
.dialog .close {
  ...
}
```

- Using a third-party library (eg. *Font Awesome*)
  - A *Content Delivery Network* (**CDN**) is a global network of servers for serving web assets (like JS, CSS, images, font files, other documents). The service detects user location and serves from a nearby server, thus reducing

network latency, and making the web page load faster.
    - cdnjs.com is a popular CDN that hosts JavaScript and CSS libraries.
    - We can include a library like Font Awesome which is hosted in cdnjs.com

```
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.13.0/css/all.mi
```

    - We can use an icon like so

```
<i class="fas fa-globe"></i>
```

    - For more information, refer to the Font Awesome documentation - https://fontawesome.com/
- Responsive Web Design (RWD) concepts
    - *Responsive Web Design* (**RWD**) is a set of techniques to make sure the page layout works fine in all devices, irrespective of the device dimensions (Especially device/viewport width)
    - Remember, there are a range of devices with widely varying widths - mobile phones, tablets, desktops, Smart TVs etc.
    - *Mobile-first* is a design philosophy where primary importance is attached to layout for mobile devices. Only once that is set, is the layout for higher width devices considered.
    - This reflects the fact that browsing on mobile devices has become far more common than browsing using desktops etc.
    - Some important RWD techniques are *fluid layouts*, *fluid media* (images, videos etc.), *media queries* with *breakpoints* for various screen widths etc.
        - When designing fluid layouts, we make sure that horizontal dimensions (width, padding-left, padding-right, border-left, border-right, margin-left, margin-right) of all elements are set using relative units, i.e. percentages.
        - This ensures that as screen width varies, the width of each element in turn (starting from the body, down to the most deeply nested elements) varies proportionately, thus avoiding fixed widths and scrollbars that appear as a result.
        - `box-sizing: border-box` is usually set on all elements on the page to overcome problem that borders can be set only using absolute units. ```
        - { box-sizing: border-box; } ```
        - When adding media like images, videos etc., we make sure these media scale according to the device size. Media occupy their natural dimensions by default (eg. original width and height for images). This often causes them to overflow their parent element box (when parent box dimensions are usually themselves fluid). For this reason, we set the media widths as a percentage of parent element width, usually 100% of the parent. Thus overflow is prevented.

```
<head>
  <style>
      .container {
          width: 200px;
      }

      .fluid-img {
          width: 100%;
      }

      .fluid-media {
          width: 100%;
      }
  </style>
</head>
<body>
<div class="container">
      <img src="images/abstract_bg_image.jpg" class="fluid-img" />
  </div>
  <div class="container">
      <video src="videos/movie.mp4" controls class="fluid-media"></video>
  </div>
</body>
```

        - *Media query*

- It is a CSS3 feature (hence not supported in very old browsers).
- Media queries let us apply CSS conditionally. The condition tests a *media type* (like screen for all devices with a screen - mobile, desktop, Smart TVs etc. , print for printer) and *media features* (like screen width, resolution, orientation - landscape/portrait etc.), and apply CSS only when certain conditions are met.
- *Breakpoints* are logically arrived at values, usually for viewport widths of devices, in order to classify devices and apply CSS according to the breakpoint range it falls under. EFor example, the base set of classes in a mobile-first approach may be applied for all devices. Then devices that fall in *480px - 992px* may be classified as similar to *tablets*, and another set of styles applied to them that likely override the base set of styles.
- Media queries are defined using `@media` rule
- The most common media type are `screen`, `all`.
- The most common media feature that is tested is width of the viewport. `min-width` is used to test if width of the viewport exceeds a minimum width. It is useful when following a mobile-first approach and styles for mobile devices (i.e. small width) are specified first. Then media queries with breakpoints for higher width devices are then set with *min-width* queries. An alternative approach considers devices of large width first and then applies CSS conditionally for lower width devices - the media queries in this approach will make use of *max-width* queries.
- Some other media features are `orientation`, `max-resolution` etc.
- This is an example of a responsive 3-column layout using mobile-first approach. The common practice is to start by stacking up the columns one below another on mobile devices, i.e. 1 column per row, and keep increasing the number of columns progressively greater viewport widths.

```
<head>
  <style>
      .row {
          overflow: auto; /* to contain floated children */
      }

      /* "mobile first" */
      .row > .col {
          float: left;
          width: 100%;
          margin: 1em 0;
          margin-right: 0;
      }

      /* "tablet breakpoint" */
      @media all and ( min-width: 481px ) { /* width >= 481px */
          .row > .col {
              width: 47.5%;
              margin-right: 5%;
          }

          .col:nth-child(2n) {
              margin-right: 0;
          }
      }

      /* "desktop breakpoint" */
      @media all and ( min-width: 769px ) { /* width >= 769px */
          .row > .col {
              width: 30%;
              margin-right: 5%;
          }

          .col:nth-child(3n) {
              margin-right: 0;
          }
      }
  </style>
</head>
<body>
  <div class="row">
      <div class="col">Lorem ipsum dolor sit amet consectetur adipisicing elit.</div>
      <div class="col">Lorem ipsum dolor sit amet consectetur adipisicing elit.</div>
      <div class="col">Lorem ipsum dolor sit amet consectetur adipisicing elit.</div>
```

```
    </div>
  </body>
```

- Apart from this RWD has many patterns that have evolved over time, that are used to design widgets etc. A sample list and implementaton of these can be found here - https://bradfrost.github.io/this-is-responsive/patterns.html, and here - https://responsivedesign.is/patterns/

Theming

- Base CSS class + modifier CSS class pattern - The base CSS class (like `btn` below) has a set of styles shared between variants of a widget. The modifier classes (like `btn-primary` , and `bth-danger`) have styles specific to a variant of the widget.
- Creating custom CSS properties helps avoid duplication of colors, padding etc. when creating a themed set of widgets (dialogs, alerts. buttons etc. with same primary color, danger color etc.)

```
:root {
  --blue: navy;
  --red: crimson;
  --padding-y: 10px;
  --padding-x: 20px;
}
```

/* base CSS class */ .btn { padding: var(--padding-y, 10px) var(--padding-x, 20px); border: 1px solid black; border-radius: 0.2em; font-size: 1.1em; background-color: transparent; color: black; cursor: pointer; }

/* modifier CSS classes */ .btn-primary { border: 1px solid var(--blue, navy); color: var(--blue, navy); }

.btn-danger { border: 1px solid var(--red, crimson); color: var(--red, crimson); }

```
```