Web System Development

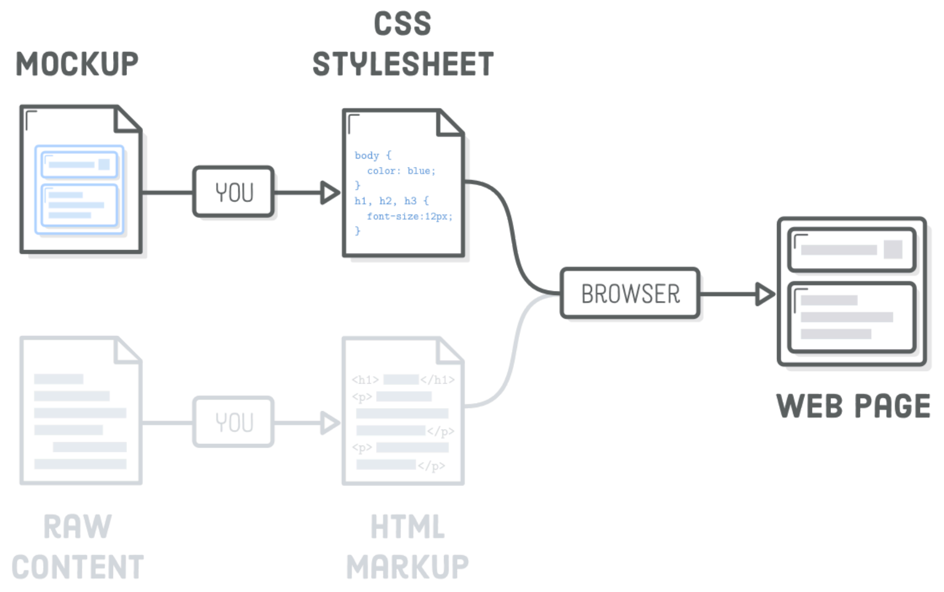
Module 2: CSS

CSS

- CSS (Cascading Style Sheets) is a language that describes the presentation of a document written in HTML.
- You can think of CSS as defining the **design** of a web page.
- It determines things like font size, margins, and colors using a language entirely separate from HTML.

Why is it a separate language?

- CSS serves a completely different purpose than HTML.
- HTML represents the content of your web page
- CSS defines how that content is presented to the user.



- CSS provides the vocabulary to tell a web browser things like, “I want my headings to be really big and my sidebar to appear on the left of the main article.”
- HTML doesn't have the terminology to make those kinds of layout decisions—all it can say is, “that's a heading and that's a sidebar.”

Setup

To keep things simple, let's create a simple vite server in a folder called `hello-css` .

```
# In the terminal
mkdir hello-css
cd hello-css
npm init -y # to create a package.json file with default values
npm install --save-dev vite
```

```
// Add the following to the package.json file
"scripts": {
  "dev": "vite",
  "build": "vite build",
  "preview": "vite preview"
}
```

```
# Start the dev server
npm run dev
```

```
<!-- index.html -->
<!DOCTYPE html>
<html lang='en'>
  <head>
    <meta charset='UTF-8'/>
    <title>Hello, CSS</title>
  </head>
  <body>
    <h1>Hello, CSS</h1>

    <p>CSS lets us style HTML elements. There's also
      <a href='dummy.html'>another page</a> associated with this example.</p>

    <h2>List Styles</h2>

    <p>You can style unordered lists with the following bullets:</p>

    <ul>
      <li>disc</li>
      <li>circle</li>
      <li>square</li>
    </ul>

    <p>And you can number ordered lists with the following:</p>

    <ol>
      <li>decimal</li>
      <li>lower-roman</li>
      <li>upper-roman</li>
      <li>lower-alpha</li>
      <li>upper-alpha</li>
      <li>(and many more!)</li>
```

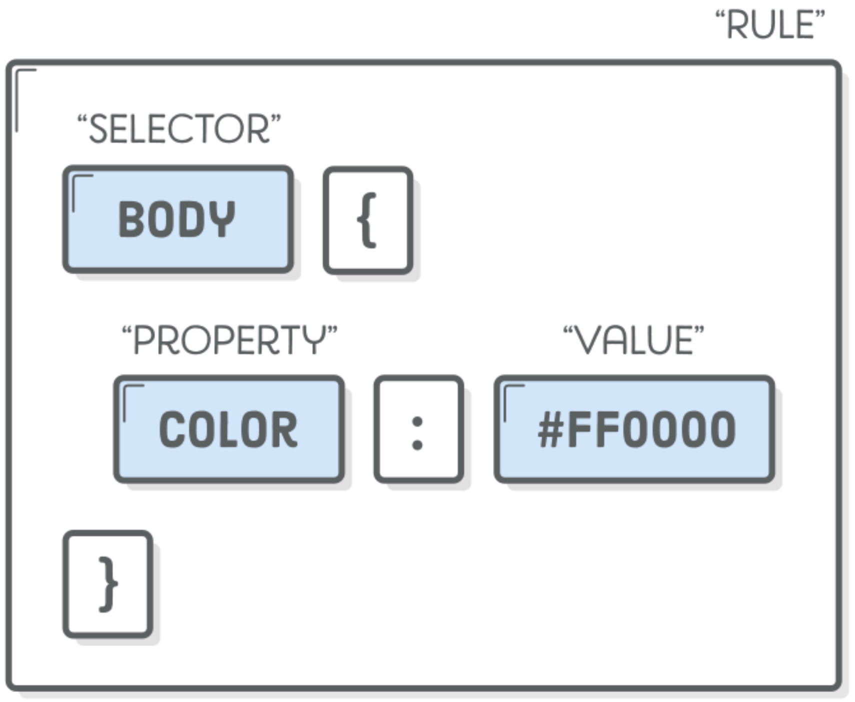
In addition, we'll need a small dummy page to learn how CSS styles can be applied to multiple web pages. Create `dummy.html` and add the following:

```html
<!DOCTYPE html>
<html lang='en'>
  <head>
    <meta charset='UTF-8'/>
    <title>Dummy</title>
  </head>
  <body>
    <h1>Dummy</h1>

    <p>This is a dummy page that helps us demonstrate reusable CSS
      stylesheets. <a href='index.html'>Go back</a>.</p>

    <p>Want to try crossing out an <a href='nowhere.html'>obsolete link</a>? This
      is your chance!</p>
  </body>
</html>
```

CSS stylesheets

- CSS stylesheets reside in plaintext files with a `.css` extension.
- Create a new file called `styles.css` in our `hello-css` folder.
- Let's add one CSS rule so that we can tell if our stylesheet is hooked up to our HTML pages properly.

```css
body {
  color: #FF0000;
}
```



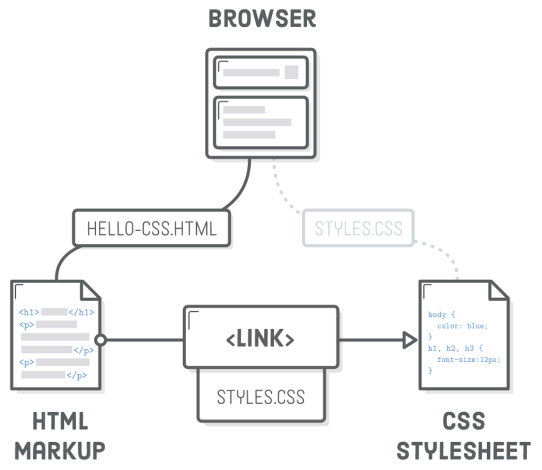
- A CSS **rule** always start with a **selector** that defines which HTML elements it applies to.
- In this case, we're trying to style the `<body>` element.
- After the selector, we have the **declarations block** inside of some curly braces.
- Any **properties** we set in here will affect the `<body>` element.

- `color` is a built-in property defined by the CSS specification that determines the text color of whatever HTML elements have been selected.
- It accepts a hexadecimal value representing a color. `#FF0000` means bright red.
- CSS properties are kind of like HTML attributes in that they both deal with key-value pairs.
- Except, here we're defining **presentational** information instead of contributing to the **semantic** meaning of the underlying content.

Linking a CSS Stylesheet

- If you try loading either of the HTML pages in a browser, you won't see our stylesheet in action.That's because we didn't link them together yet.
- This is what the HTML `<link/>` element is for.
- In `index.html` , update `<head>`
- This `<link/>` element is how browsers know they need to load `styles.css` when they try to render our `index.html` page.
- We should now see **blindingly red text** everywhere.

```html
<head>
  <meta charset='UTF-8'/>
  <title>Hello, CSS</title>
  <link rel='stylesheet' href='styles.css'/>
</head>
```



- The `<link/>` element is just like the `<a>` element, but it's only meant to be used inside of `<head>` .
- Since it's in the head of the document, `<link/>` connects to **metadata** that's defined outside of the current document.
- Also notice that it's an **empty element** , so it doesn't need a closing tag.
- The `rel` attribute defines the relationship between the resource and the HTML document.
- By far the most common value is `stylesheet` , but there are a few other options .
- The `href` attribute should point to a `.css` file instead of another web page. The value for `href` can be an absolute, relative, or root-relative link .

Setting Multiple Properties

You can stick as many properties as you want in the declarations block of a CSS rule. Try setting the background color of the entire web page by changing our rule to the following:

```css
body {
  color: #414141;
  background-color: #EEEEEE;
}
```

The `background-color` property is very similar to the `color` property, but it defines the background color of whatever element you selected.

Different Elements

Of course, you'll want to apply styles to elements other than `<body>` . For that, simply add **more CSS rules** with different selectors.

```css
body {
  color: #414141;
  background-color: #EEEEEE;
}

h1 {
  font-size: 36px;
}

h2 {
  font-size: 28px;
}
```

Selecting Multiple Elements

What if we want to add some styles to **all** our headings? We don't want to have redundant rules, since that would eventually become a nightmare to maintain:

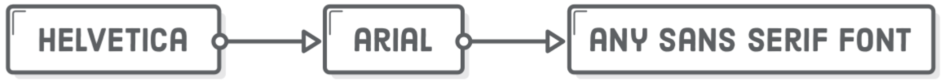
```css
h1 {
  font-family: "Helvetica", "Arial", sans-serif;
}

h2 {
  font-family: "Helvetica", "Arial", sans-serif;
}

h3 {
  font-family: "Helvetica", "Arial", sans-serif;
}
```

Instead, we can select **multiple HTML elements in the same CSS rule** by separating them with commas:

```css
h1, h2, h3, h4, h5, h6 {
  font-family: "Helvetica", "Arial", sans-serif;
}
```

Defining Fonts

- `font-family` is another built-in CSS property that defines the typeface for whatever element you selected.
- It accepts multiple values because not all users will have the same fonts installed.
- With the example before, the browser tries to load the left-most one first (`Helvetica`), falls back to `Arial` if the user doesn't have it, and finally chooses the system's default sans serif font.



- Relying on the user's built-in fonts has historically been incredibly limiting for web designers. Nowadays, system fonts have been largely superseded by web fonts .

CSS Comments

```
body {
  color: #414141;               /* Dark gray */
  background-color: #EEEEEE;    /* Light gray */
}
```

- Comments in CSS are a little different than their HTML counterparts.
- Instead of the `<!-- -->` syntax, CSS ignores everything between `/*` and `*/` characters.

List Styles

The `list-style-type` property lets you alter the bullet icon used for `<li>` elements. You'll typically want to define it on the parent `<ul>` or `<ol>` element:

```css
ul {
  list-style-type: circle;
}

ol {
  list-style-type: lower-roman;
}
```

Check out the MDN documentation for more information on the different values you can use.

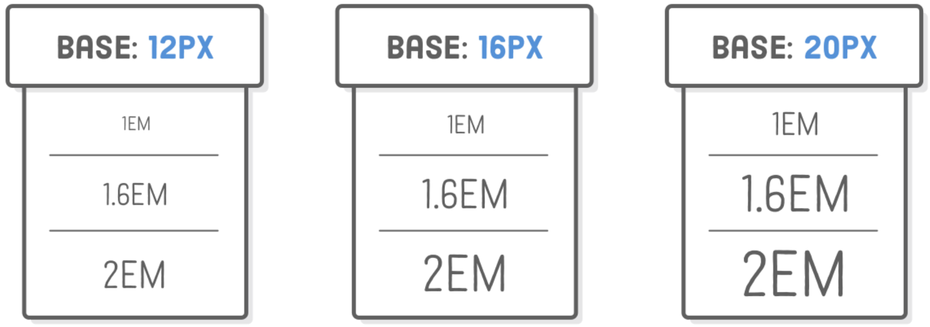
You can even create custom bullets for `<li>` elements with the `list-style-image` property (see MDN for details).

Units Of Measurement

- Many CSS properties require a unit of measurement.
- There's a lot of units available, but the most common ones you'll encounter are `px` (pixel) and `em` (pronounced like the letter *m*).
  - `px` is what you would intuitively call a pixel, regardless of whether the user has a retina display or not.
  - `em` is the current font size of the element in question.

EM Units

The `em` unit is very useful for defining sizes relative to some base font.



In the above diagram, you can see `em` units scaling to match a base font size of `12px` , `16px` , and `20px` .

Example

Consider the following alternative to the previous code snippet:

```
body {
  color: #414141;               /* Dark gray */
  background-color: #EEEEEE;    /* Light gray */
  font-size: 18px;
}

h1 {
  font-size: 2em;
}

h2 {
  font-size: 1.6em;
}
```

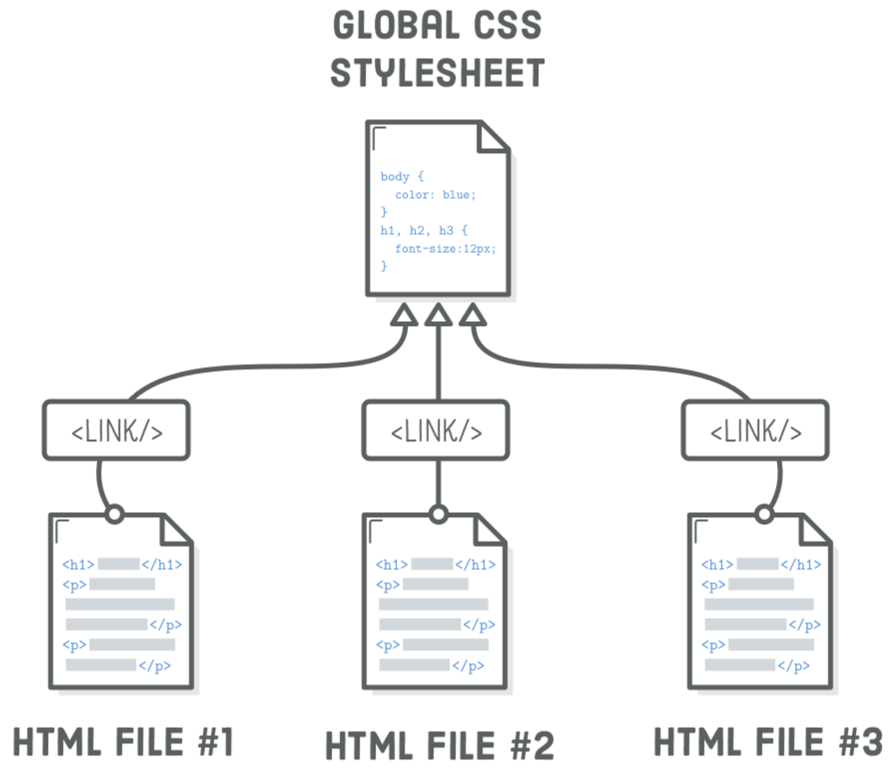
- This sets our base font size for the document to `18px` .
- Then, it says that our `<h1>` elements should be **twice** that size and our `<h2>` 's should be **1.6 times bigger** .
- If we (or the user) ever wanted to make the base font bigger or smaller, `em` units would allow our entire page to scale accordingly.

Reusable Stylesheets

- So, we just defined some basic styles for one of our web pages.
- It would be really convenient if we could reuse them on our other page, too.
- All we need to do is add the same `<link/>` element to any other pages we want to style.
- Let's add the following line to the `<head>` of `dummy.html` :

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

Now, our `dummy.html` pages should match our `hello-css.html` styles.



Whenever we change a style in `styles.css` , those changes will automatically be reflected in both of our web pages.

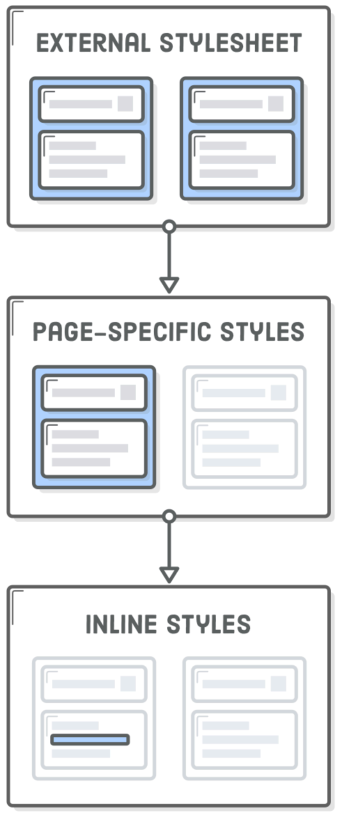
The Cascade

- The *cascading* part of CSS (Cascading Style Sheets) is due to the fact that rules cascade down from multiple sources.
- So far, we've only seen one place where CSS can be defined: external `.css` files.
- However, external stylesheets are just one of many places you can put your CSS code.

The CSS hierarchy for every web page looks like this:

- The browser's default stylesheet
- External stylesheets (that's us)
- Page-specific styles (that's also us)
- Inline styles (that could be us, but it never should be)

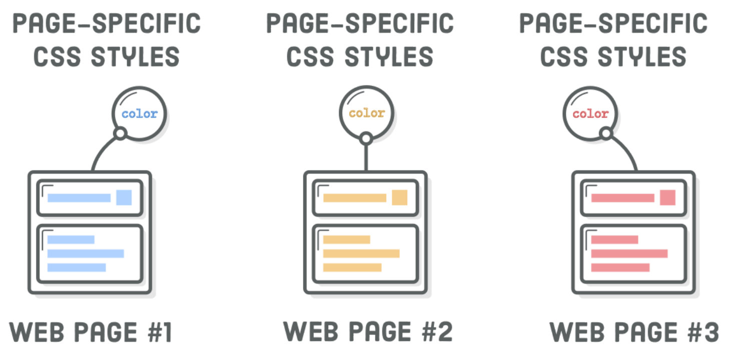
This is **ordered** from **least** to **most** precedence, which means styles defined in each subsequent step override previous ones.

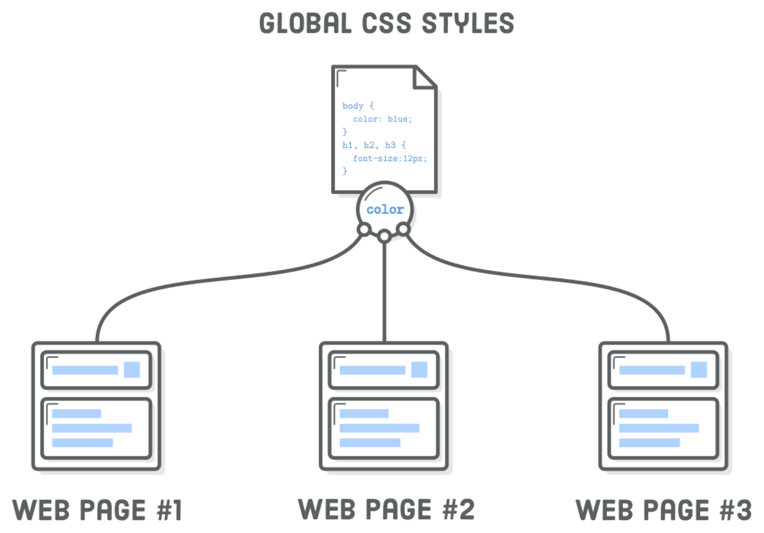


Page-Specific Styles

- The `<style>` element is used to add page-specific CSS rules to individual HTML documents.
- The `<style>` element always lives in the `<head>` of a web page, which makes sense because it's metadata, not actual content.
- The example below will apply only to `dummy.html` . Our `hello-css.html` page won't be affected. If you did it right, you should see bright blue text when you load `dummy.html` in a browser.
- Anything you would put in our `styles.css` file can live in this `<style>` element.

```
<head>
  <meta charset='UTF-8'/>
  <title>Dummy</title>
  <link rel='stylesheet' href='styles.css'/>
  <style>
    body {
      color: #0000FF;     /* Blue */
    }
  </style>
</head>
```





```
body {
  color: blue;
}
h1, h2, h3 {
  font-size:12px;
}
```

- Page-specific styles use the exact same CSS syntax as an external stylesheet, but everything here will override rules in our `styles.css` file.
- In this case, we’re telling the browser to ignore the `color` property we defined for `<body>` in our external stylesheet and use `#0000FF` instead.
- The problem with page-specific styles is that they're incredibly **difficult to maintain** . As shown in the diagram on the left, when you want to apply these styles to another page, you have to copy-and-paste them into that document's `<head>` .

Inline Styles

You can also stick CSS rules in the `style` attribute of an HTML element.

```html
<p>
  Want to try crossing out an
  <a href='nowhere.html' style='color: #990000; text-decoration: line-through;'>obsolete link</a>?
  This is your chance!
</p>
```

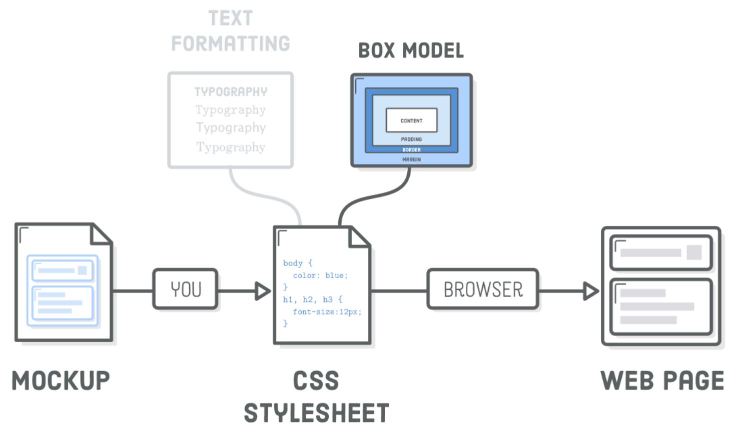
- Inline styles should be **avoided at all costs** because they make it impossible to alter styles from an external stylesheet.
- If you ever wanted to re-style your website down the road, you can't just change a few rules in your global `styles.css` file—you'd have to go through every single page and update every single HTML element that has a `style` attribute.
- That said, there will be many times when you need to apply styles to only a specific HTML element. For this, you should always use **CSS classes** instead of inline styles. We'll explore classes later.

Multiple Stylesheets

- CSS rules can be spread across several external stylesheets by adding multiple `<link/>` elements to the same page.
- A common use case is to separate out styles for different sections of your site. This lets you selectively apply consistent styles to distinct categories of web pages.
- The order of the `<link/>` elements matters. Stylesheets that come later will override styles in earlier ones. Typically, you'll put your *base* or *default* styles in a global stylesheet (`styles.css`) and supplement them with section-specific stylesheets (`product.css` and `blog.css`). This allows you to organize CSS rules into manageable files while avoiding the perils of page-specific and inline styles.

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

The Box Model



- So far we've talked about the basic text formatting properties of CSS, but that was only one aspect of styling pages. Defining the **layout** of a web page is an entirely different beast.
- The **CSS box model** is a set of rules that define how every web page on the Internet is rendered.
- CSS treats each element in your HTML document as a **box** with a bunch of different properties that determine where it appears on the page.
- So far, all of our web pages have just been a bunch of elements rendered one after another. The box model is our toolkit for customizing this default **layout** scheme.

Setup

```
<!DOCTYPE html>
<html lang='en'>
  <head>
    <meta charset='UTF-8'/>
    <title>Boxes Are Easy!</title>
    <link rel='stylesheet' href='box-styles.css'/>
  </head>
  <body>
    <h1>Headings Are Block Elements</h1>

    <p>Paragraphs are blocks, too. <em>However</em>, &lt;em&gt; and &lt;strong&gt;
      elements are not. They are <strong>inline</strong> elements.</p>

    <p>Block elements define the flow of the HTML document, while inline elements
      do not.</p>
  </body>
</html>
```

Block Elements and Inline Elements

- All the HTML elements that we've been working with have a default type of box.
- For instance, `<h1>` and `<p>` are block-level elements, while `<em>` and `<strong>` are inline elements.
- Let's get a better look at our boxes by adding the following to `box-styles.css`

```css
h1, p {
  background-color: #DDE0E3;     /* Light gray */
}

em, strong {
  background-color: #B2D6FF;     /* Light blue */
}
```

- The background-color property only fills in the background of the selected box, so this will give us a clear view into the structure of the current sample page.
- Our headings and paragraphs should have gray backgrounds, while our emphasis and strong elements should be light blue.

Block and Inline Behaviours

- **Block boxes** always appear **below** the previous block element. This is the “natural” or “static” flow of an HTML document when it gets rendered by a web browser.
- The **width of block boxes** is set automatically based on the width of its parent container. In this case, our blocks are always the width of the browser window.
- The default **height of block boxes** is based on the content it contains. When you narrow the browser window, the `<h1>` gets split over two lines, and its height adjusts accordingly.
- **Inline boxes** don’t affect **vertical spacing** . They’re not for determining layout—they’re for styling stuff *inside* of a block.
- The **width of inline boxes** is based on the content it contains, not the width of the parent element.

Changing Box Behaviour

- We can override the default box type of HTML elements with the CSS `display` property.
- For example, if we wanted to make our `<em>` and `<strong>` elements blocks instead of inline elements, we could update our rule in `box-styles.css` like so:

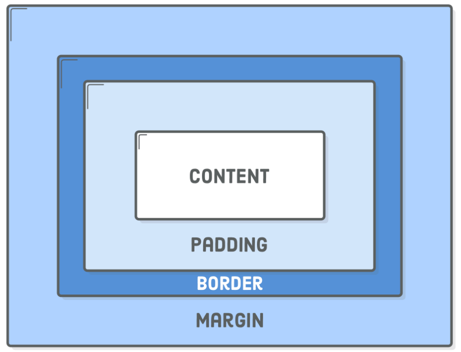
```css
em, strong {
  background-color: #B2D6FF;
  display: block;
}
```

- Now, these elements act like our headings and paragraphs: they start on their own line, and they fill the entire width of the browser.
- This comes in handy when you're trying to turn `<a>` elements into buttons or format `<img/>` elements (both of these are inline boxes by default).

Rendering an element's box

The **CSS box model** is a set of rules that determine the dimensions of every element in a web page. It gives each box (both inline and block) four properties:

- **Content** - The text, image, or other media content in the element.
- **Padding** - The space between the box's content and its border.
- **Border** - The line between the box's padding and margin.
- **Margin** - The space between the box and surrounding boxes.



Together, this is everything a browser needs to render an element's box.

Padding

The **padding** property defines the padding for the selected element:

```css
h1 {
  padding: 50px;
}
```

This adds 50 pixels to *each side* of the `<h1>` heading. Notice how the background color expands to fill this space. That's always the case for padding because it's inside the border, and everything inside the border gets a background.

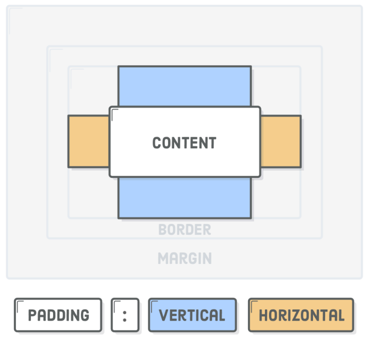
Sometimes you¡ll only want to style one side of an element. For that, CSS provides the following properties:

```css
p {
  padding-top: 20px;
  padding-bottom: 20px;
  padding-left: 10px;
  padding-right: 10px;
}
```

Shorthand Formats

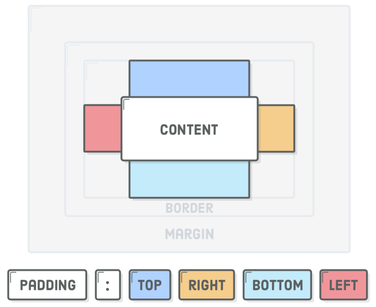
When you provide **two** values to the `padding` property, it's interpreted as the vertical and horizontal padding values, respectively.

```css
p {
  padding: 20px 10px;  /* Vertical  Horizontal */
}
```



If you provide **four** values, you can set the padding for each side of an element individually. The values are interpreted clockwise, starting at the top:

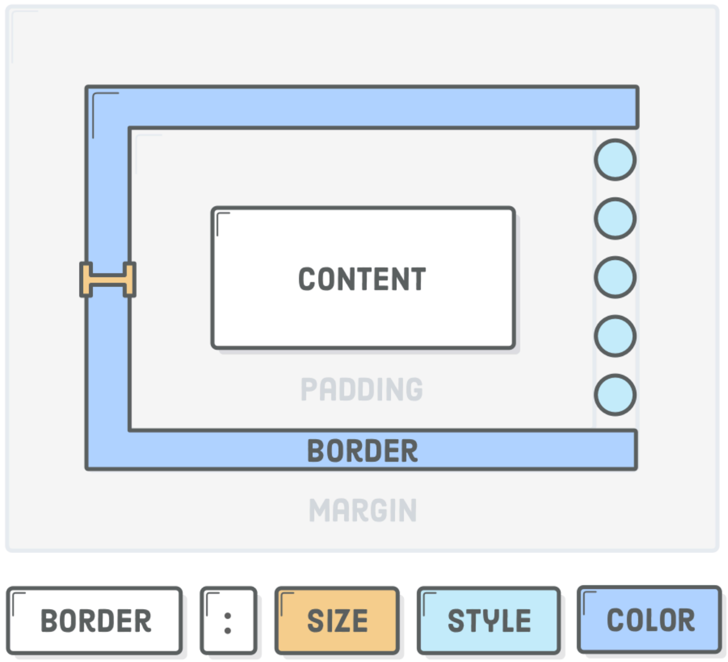
```css
p {
  padding: 20px 0 20px 10px;  /* Top  Right  Bottom  Left */
}
```



Border

- The border is a line drawn around the content and padding of an element.
- The `border` property requires a new syntax that we've never seen before. First, we define the **stroke width** of the border, then its **style** , followed by its **color** .
- Please refer to the Mozilla Developer Network for more information about border styles.

```
h1 {
  padding: 50px;
  border: 1px solid #5D6063;
}
```



Margins

Margins define the space outside of an element's border. Or, rather, the space between a box and its surrounding boxes.

```
p {
  padding: 20px 0 20px 10px;
  margin-bottom: 50px;
}
```

Margins and padding can accomplish the same thing in a lot of situations, making it difficult to determine which one is the “right” choice. The most common reasons why you would pick one over the other are:

- The padding of a box has a background, while margins are always transparent.
- Padding is included in the click area of an element, while margins aren't.

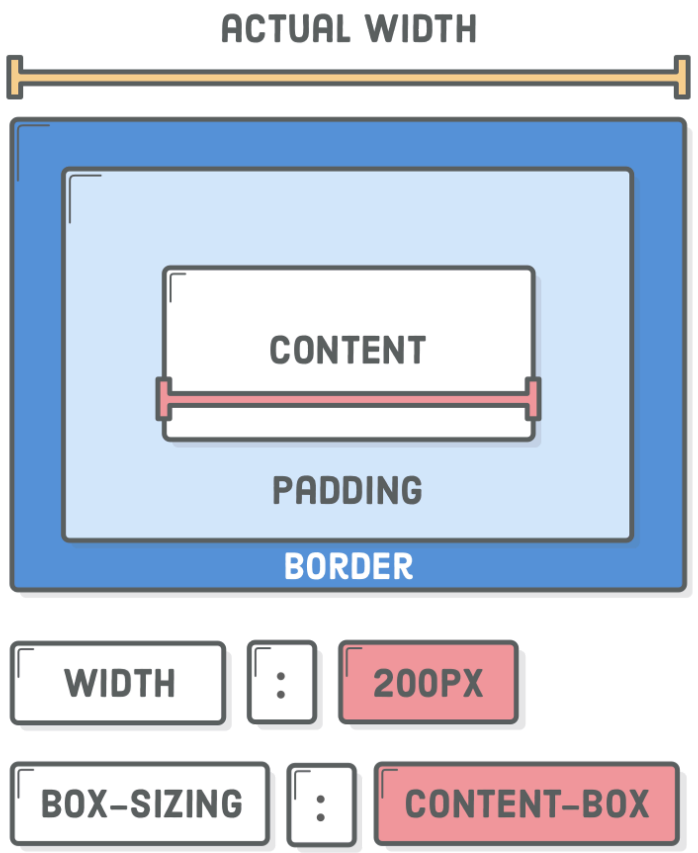
Generic Boxes

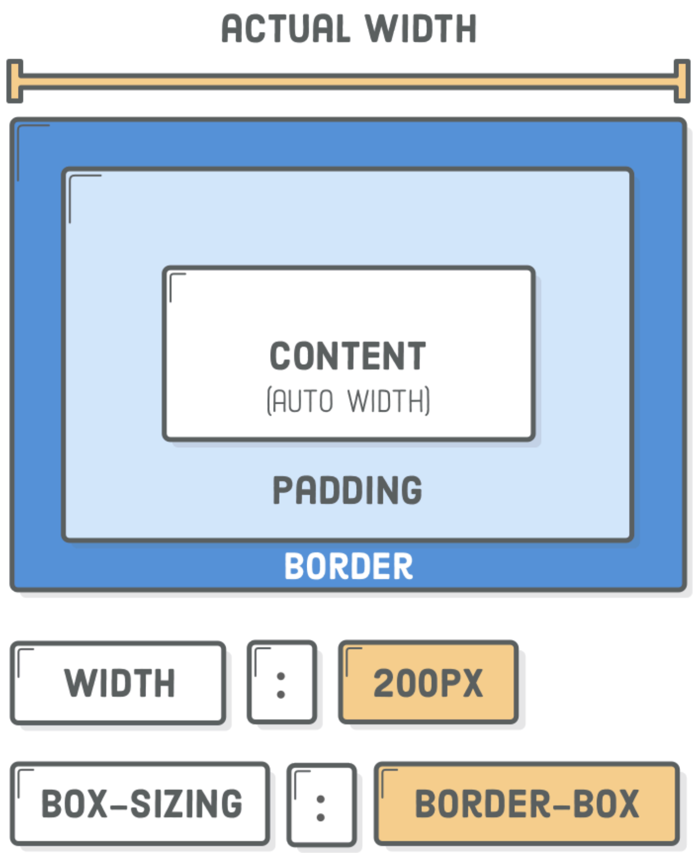
- So far, every HTML element we've seen lends additional meaning to the content it contains.
- Indeed, this is the whole point of HTML, but there are many times when we need a generic box purely for the sake of styling a web page.
- This is what `<div>` and `<span>` are **container elements** that don't have any affect on the semantic structure of an HTML document. They provide a hook for adding CSS styles to arbitrary sections of a web page.
- For example, sometimes you need to add an invisible box to prevent a margin collapse, or maybe you want to group the first few paragraphs of an article into a synopsis with slightly different text formatting.
- The following will give us a big blue button that spans the entire width of the browser:

```
<div>Button</div>


div {
  color: #FFF;
  background-color: #5995DA;
  font-weight: bold;
  padding: 20px;
  text-align: center;
  border: 2px solid #5D6063;
  border-radius: 5px;
}
```

Content Box vs. Border Box

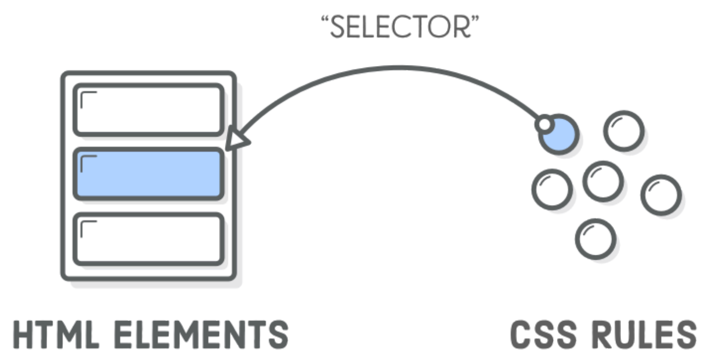
The `width` and `height` properties only define the size of a box's *content* . Its padding and border are both added *on top of* whatever explicit dimensions you set. This explains why you'll get an image that's 244 pixels wide when you take a screenshot of our button, despite the fact that it has a `width: 200px` declaration attached to it.





CSS Selectors

- Before we learned how to connect an HTML document to other files in our project.
- **CSS selectors** are similar, except instead of navigating between whole files, they let us map a single CSS rule to a specific HTML element.
- This makes it possible to *selectively* style individual elements while *ignoring* others.
- Unless you want every section of your website to look exactly the same, this is a crucial bit of functionality for us.
- It's how we say things like “I want this paragraph to be blue and that other paragraph to be yellow.” Until now, we've only been able to turn *all* our paragraphs blue (or yellow).
- The only CSS selector we've seen so far is called the “type selector”, which targets all the matching elements on a page.



Setup

We'll only need one HTML file and a CSS stylesheet for our example this chapter. Create a new folder called `css-selectors` and new web page called `selectors.html` with the following markup:

```html
<script type="text/plain" class="language-html">
  <!DOCTYPE html>
  <html lang='en'>
    <head>
      <meta charset='UTF-8'/>
      <title>CSS Selectors</title>
      <link rel='stylesheet' href='styles.css'/>
    </head>
    <body>
      <h1>CSS Selectors</h1>

      <p>CSS selectors let you <em>select</em> individual HTML elements in an HTML
        document. This is <strong>super</strong> useful.</p>

      <p>Classes are ridiculously important, since they allow you to select
        arbitrary boxes in your web pages.</p>

      <p>We’ll also be talking about links in this example, so here’s
        <a href='https://internetingishard.com'>Interneting Is Hard</a> for us to
        style.</p>

      <div>Button One</div>

    </body>
  </html>
</script>
```

Go ahead and create that `styles.css` stylesheet in the same folder, too.

Class Selectors

- Class selectors let you apply CSS styles to a **specific HTML element** .
- They let you differentiate between HTML elements of the same type, like when we had two `<p>` elements.
- Class selectors require two things:
  - A `class` attribute on the HTML element in question.
  - A matching CSS class selector in your stylesheet.

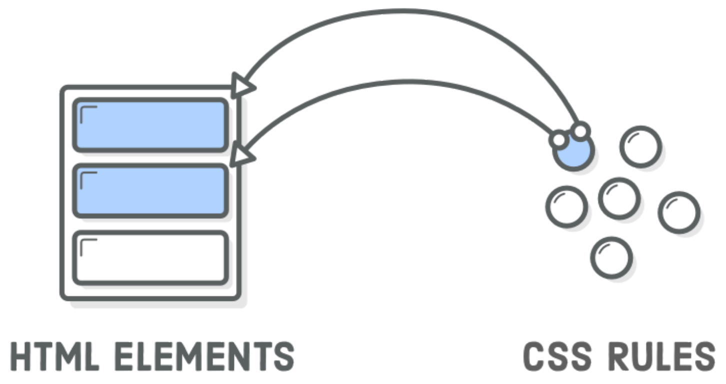
Example

- Let's add a `class` attribute to the desired paragraph and pluck out that `<p class='synopsis'>` element in our CSS with the following:

```
<p class="synopsis">
  CSS selectors let you <em>select</em> individual HTML
  elements in an HTML document. This is <strong>super</strong> useful.
</p>


.synopsis {
  color: #7E8184;        /* Light gray */
  font-style: italic;
}
```

- This rule is *only* applied to elements with the corresponding `class` attribute.
- Notice the dot (`.`) prefixing the class name. This distinguishes class selectors from the type selectors that we've been working with before.
- The `class` attribute isn't limited to `<p>` elements—it can be defined on *any* HTML element.

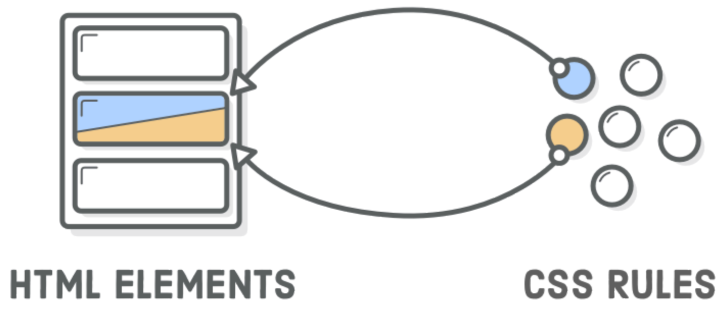
Reusing Class Styles



- The same class can be applied to **multiple elements** in a single HTML document.
- This means that we can now reuse arbitrary CSS declarations wherever we want.
- To create another similar paragraph, all we have to do is add another HTML element with the same class
- This gives us a second paragraph that looks just like the first one—without writing a single line of CSS!

```
<p class="synopsis">First paragraph</p>

<p class="synopsis">Second paragraph</p>
```

Modifying Class Styles



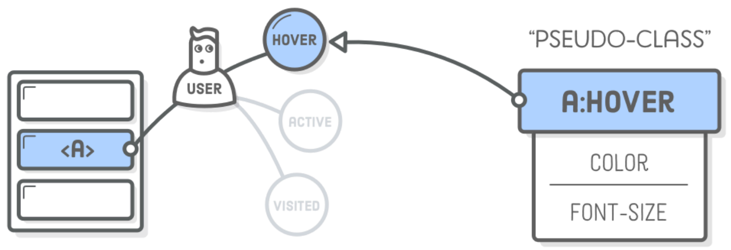
- What if we want to alter our second paragraph a little bit?
- Fortunately, we can apply multiple classes to the *same* HTML element, too.
- The styles from each class will be applied to the element, giving us the opportunity to both reuse styles from `.synopsis` and override some of them with a new class.

```
<p class="synopsis">First paragraph</p>

<p class="synopsis call-to-action">Second paragraph</p>


.call-to-action {
  font-style: italic;
  background-color: #EEB75A;
}
```

Pseudo-classes

- So far, all the CSS selectors we've seen map directly to a piece of HTML markup that we wrote.
- However, there's more going on in a rendered web page than just our HTML content.
- There's **stateful** information about what the user is doing.

Pseudo-classes for Links



- The classic example is a link. As a web developer, you create an `<a href>` element. After the browser renders it, the user can interact with that link. They can **hover** over it, **click** it, and **visit** the URL.
- CSS **pseudo-classes** provide a mechanism for hooking into this kind of temporary user information.
- At any given time, an `<a href>` element can be in a number of **different states** , and you can use pseudo-classes to style each one of them individually.
- Think of them as class selectors that you don't have to write on your own because they're built into the browser.

Basic Link Styles

Pseudo-classes begin with a colon followed by the name of the desired class. The most common link pseudo-classes are as follows:

- `:link` - A link the user has never visited.
- `:visited` - A link the user has visited before.
- `:hover` - A link with the user's mouse over it.
- `:active` - A link that's being pressed down by a mouse (or finger).

```css
a:link {
  color: blue;
  text-decoration: none;
}
a:visited {
  color: purple;
}
a:hover {
  color: aqua;
  text-decoration: underline;
}
a:active {
  color: red;
}
```

Pseudo-classes for Structure

- Link states are just one aspect of pseudo-classes.
- There's also a bunch of other pseudo-classes that provide extra information about an element's surroundings.
- For example, the `:last-of-type` pseudo-class selects the final element of a particular type in its parent element.
- This gives us an alternative to class selectors for selecting specific elements.
- For instance, we could use `:last-of-type` to add some space after the last paragraph of our example page
- This avoids selecting the first two `<p>` elements *without* requiring a new `class` attribute on the last paragraph:

```css
p:last-of-type {
  margin-bottom: 50px;
}
```

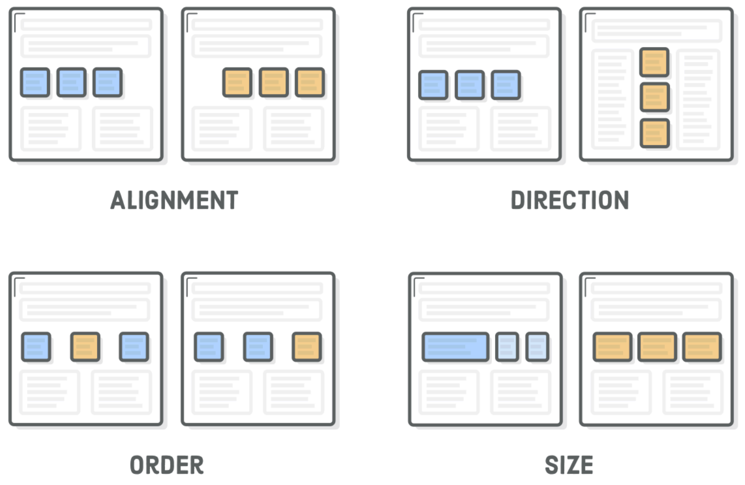
ID Selectors

- "ID selectors" are a more stringent alternative to class selectors.
- They work pretty much the same way, except you can only have **one** element with the same ID per page, which means **you can't reuse styles** at all.
- Instead of a `class` attribute, they require an `id` attribute on whatever HTML element you're trying to select. Try adding one to our second button
- The corresponding CSS selector must begin with a hash sign (`#`) opposed to a dot.

```
<a id="button-2" class="button" href="nowhere.html">Button Two</a>


#button-2 {
  color: #5D6063;  /* Dark gray */
}
```

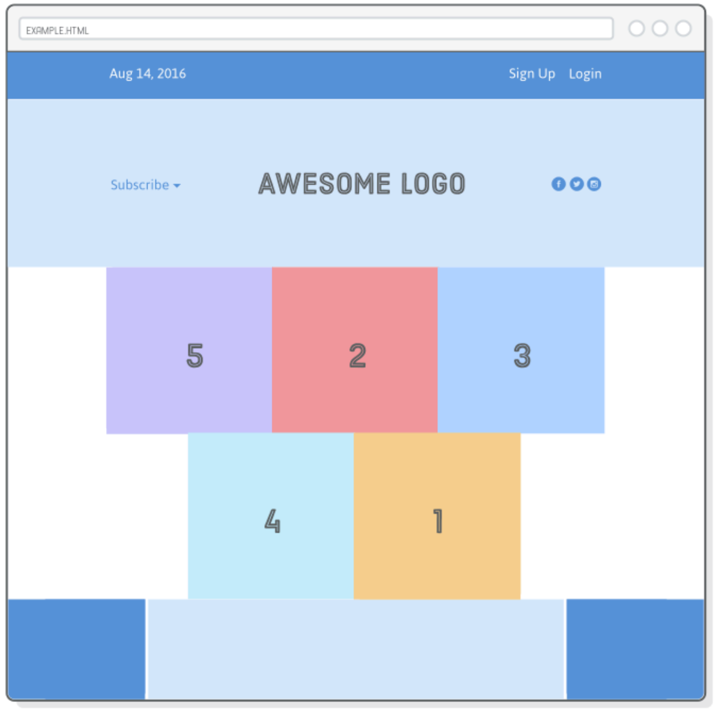
Flexbox

- The Flexbox layout mode offers a way for defining the overall appearance of a web page.
- It gives us *complete* control over the alignment, direction, order, and size of our boxes.



Setup

The goal is to wind up with something that looks like this:

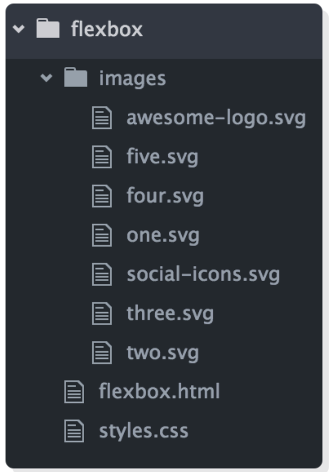


Setup

- Make a new folder called `flexbox` to house all the example files for this chapter.
- Then, create `flexbox.html` and `styles.css` and add the following markup and styles.
- Finally, download some images for use by our example web page.

```html
<!-- flexbox.html -->

<!DOCTYPE html>
<html lang='en'>
  <head>
    <meta charset='UTF-8'/>
    <title>Some Web Page</title>
    <link rel='stylesheet' href='styles.css'/>
  </head>
  <body>
    <div class='menu-container'>
      <div class='menu'>
        <div class='date'>Aug 14, 2016</div>
        <div class='signup'>Sign Up</div>
        <div class='login'>Login</div>
      </div>
    </div>
  </body>
</html>
```

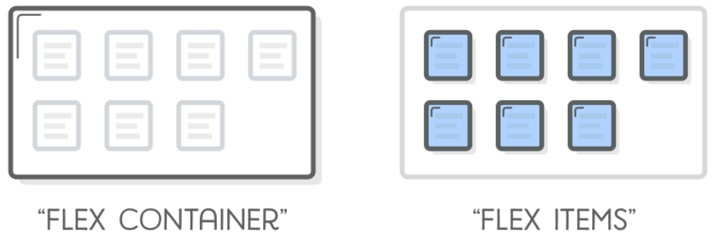
```css
/* styles.css */

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}

.menu-container {
  color: #fff;
  background-color: #5995DA;  /* Blue */
  padding: 20px 0;
}

.menu {
  border: 1px solid #fff;  /* For debugging */
  width: 900px;
}
```



```
flexbox
  images
    awesome-logo.svg
    five.svg
    four.svg
    one.svg
    social-icons.svg
    three.svg
    two.svg
  flexbox.html
  styles.css
```

Flexbox Overview

- Flexbox uses two types of boxes that we've never seen before: **flex containers** and **flex items** .
- The job of a **flex container** is to group a bunch of flex items together and define how they're positioned.
- Every HTML element that's a direct child of a flex container is an **item** . Flex items can be manipulated individually, but for the most part, it's up to the container to determine their layout. The main purpose of flex items are to let their container know how many things it needs to position.
- Defining complex web pages with flexbox is all about **nesting boxes** . You align a bunch of flex items inside a container, and, in turn, those items can serve as flex containers for their own items.



Flex Containers

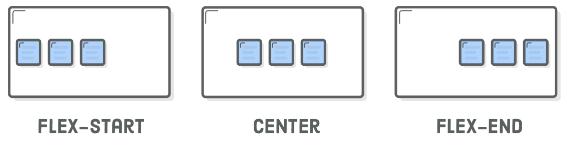
- The first step in using flexbox is to turn one of our HTML elements into a flex container.
- We do this with the `display` property. By giving it a value of `flex` , we're telling the browser that everything in the box should be rendered with flexbox instead of the default box model.

```css
.menu-container {
  /* ... */
  display: flex;
}
```

Aligning a Flex Item

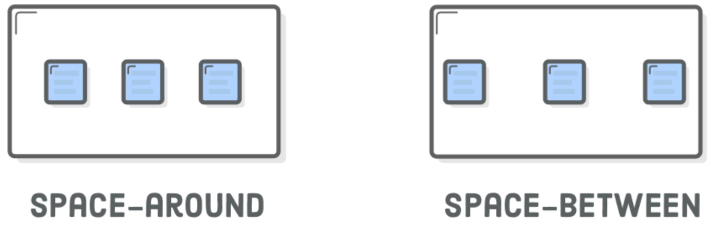
- After you've got a flex container, your next job is to define the horizontal alignment of its items.
- That's what the `justify-content` property is for. We can use it to center our `.menu`
- Other values for `justify-content` are:
  - `center`
  - `flex-start`
  - `flex-end`
  - `space-around`
  - `space-between`

```
.menu-container {
  /* ... */
  display: flex;
  justify-content: center;     /* Add this */
}
```



Distributing Multiple Flex Items

The `justify-content` property also lets you distribute items equally inside a container.



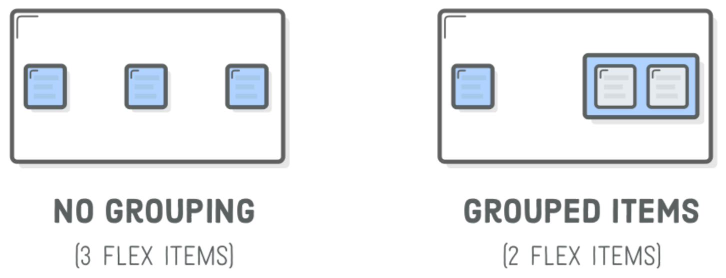
Change our `.menu` rule to match the following:

```
.menu {
  border: 1px solid #fff;
  width: 900px;
  display: flex;
  justify-content: space-around;
}
```

The flex container automatically distributes extra horizontal space to either side of each item. The `space-between` value is similar, but it only adds that extra space *between* items. This is what we actually want for our example page, so go ahead and update the `justify-content` line

Grouping Flex Items

- Flex containers only know how to position elements that are one level deep (i.e., their child elements).
- They don't care one bit about what's inside their flex items.
- This means that grouping flex items is another weapon in your layout-creation arsenal.
- Wrapping a bunch of items in an extra `<div>` results in a totally different web page.

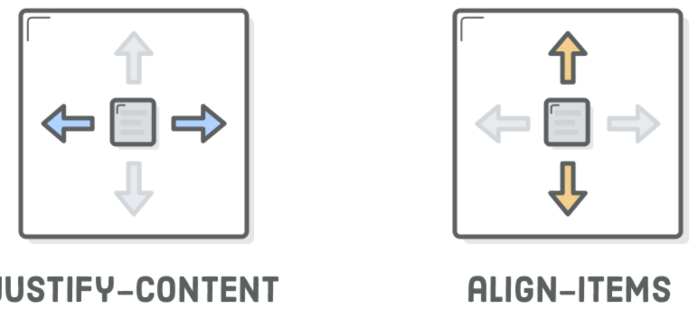


```html
<div class="menu">
  <div class="date">Aug 14, 2016</div>
  <div class="links">
    <div class="signup">Sign Up</div>      <!-- This is nested now -->
    <div class="login">Login</div>         <!-- This one too! -->
  </div>
</div>
```

```css
.links {
  border: 1px solid #fff;  /* For debugging */
  display: flex;
  justify-content: flex-end;
}

.login {
  margin-left: 20px;
}
```

Cross-Axis (Vertical) Alignment

So far, we've been manipulating horizontal alignment, but flex containers can also define the vertical alignment of their items.



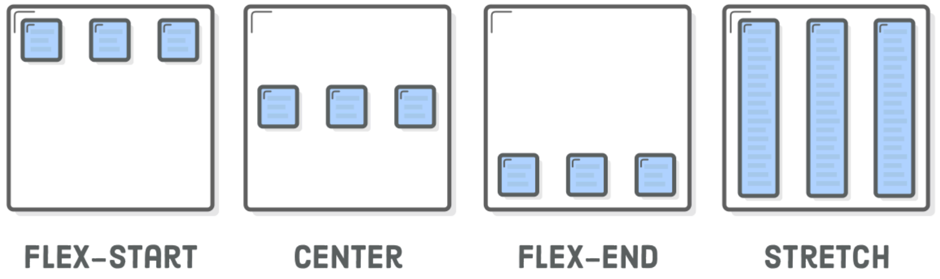
```html
<div class="menu-container">
  <div class="menu">
    <div class="date">Aug 14, 2016</div>
    <div class="links">
      <div class="signup">Sign Up</div>
      <div class="login">Login</div>
    </div>
  </div>
</div>
<div class="header-container">
  <div class="header">
    <div class="subscribe">Subscribe ▾</div>
    <div class="logo"><img src="images/awesome-logo.svg"></div>
    <div class="social"><img src="images/social-icons.svg"></div>
  </div>
</div>
```

```css
.header-container {
  color: #5995DA;
  background-color: #D6E9FE;
  display: flex;
  justify-content: center;
}

.header {
  width: 900px;
  height: 300px;
  display: flex;
  justify-content: space-between;
  align-items: center;
}
```

Aligning a Flex Item

The available options for `align-items` is similar to `justify-content` :

- `center`
- `flex-start` (top)
- `flex-end` (bottom)
- `stretch`
- `baseline`



The Stretch Option

The `stretch` option is worth a taking a minute to play with because it lets you display the background of each element. Let's take a brief look by adding the following to `styles.css` :

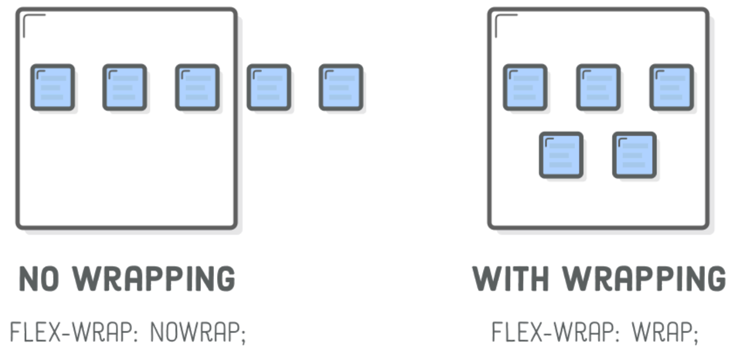
```css
.header {
  /* ... */
  align-items: stretch;    /* Change this */
}

.social,
.logo,
.subscribe {
  border: 1px solid #5995DA;
}
```

The box for each item extends the full height of the flex container, regardless of how much content it contains.

A common use case for this behavior is creating equal-height columns with a variable amount of content in each one

Wrapping Flex Items

Not only Flexbox can render items as a grid, it can change their alignment, direction, order, and size, too. To create a **grid** , we need the `flex-wrap` property.



Add a row of photos to `flexbox.html` . This should go inside of `<body>` , under the `.header-container` element, along with the corresponding CSS:

```
<div class="photo-grid-container">
  <div class="photo-grid">
    <div class="photo-grid-item first-item">
      <img src="images/one.svg">
    </div>
    <div class="photo-grid-item">
      <img src="images/two.svg">
    </div>
    <div class="photo-grid-item">
      <img src="images/three.svg">
    </div>
  </div>
</div>
```

```
.photo-grid-container {
  display: flex;
  justify-content: center;
}

.photo-grid {
  width: 900px;
  display: flex;
  justify-content: flex-start;
}

.photo-grid-item {
  border: 1px solid #fff;
  width: 300px;
```

This should work as expected, but watch what happens when we add more items than can fit into the flex container. Insert an extra two photos into the `.photo-grid` :

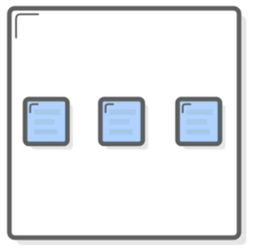
```
<div class="photo-grid-item">
  <img src="images/four.svg">
</div>
<div class="photo-grid-item last-item">
  <img src="images/five.svg">
</div>
```

By default, they flow off the edge of the page! Adding the following `flex-wrap` property forces items that don't fit to get bumped down to the next row:

```
.photo-grid {
  /* ... */
  flex-wrap: wrap;
  justify-content: center;    /* if we want to center the items */
}
```

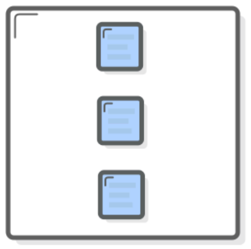
Flex Container Direction

Direction refers to whether a container renders its items horizontally or vertically. So far, all the containers we've seen use the default horizontal direction, which means items are drawn one after another in the same row before popping down to the next column when they run out of space.



ROW

FLEX-DIRECTION: ROW;



COLUMN

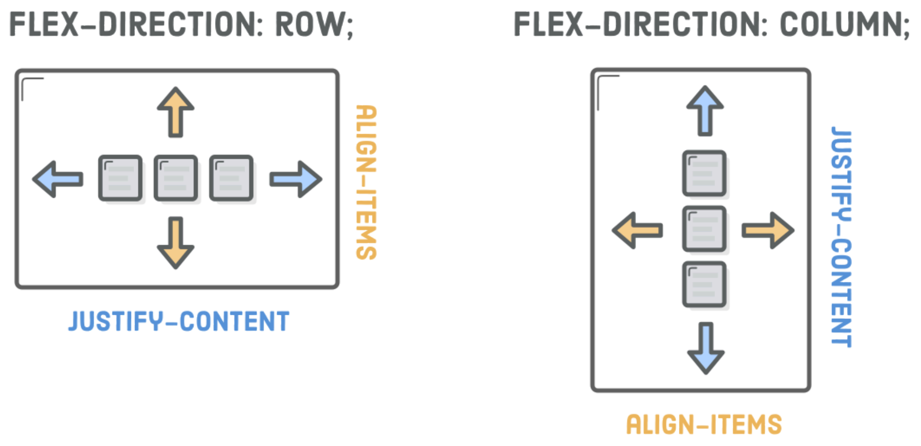
FLEX-DIRECTION: COLUMN;

To change the direction, we need the `flex-direction` property.

```css
.photo-grid {
  /* ... */
  flex-direction: column;
}
```

Alignment Considerations

- Notice that the column is hugging the left side of its flex container despite our `justify-content: center;` declaration.
- When you rotate the direction of a container, you also rotate the direction of the `justify-content` property.
- It now refers to the container's vertical alignment—not its horizontal alignment.



References

- Mozilla's CSS tutorial
- CSS Reference
- Flexbox Adventure

Module 02 Exercise

Styled Personal Page with CSS

Create a Vite project, replicate the HTML page you built in Exercise 01 and enhance it with CSS. You must:

- Place all styles in a separate `style.css` file
- Link the CSS file correctly in your `index.html`
- Use CSS to:
    - Set a background color and font for the page
    - Style your `<header>` , `<main>` , and `<footer>`
    - Style the list and its items with colors, padding, or borders
    - Customize the `<a>` link styles (hover effect, color)
    - Add a `<div>` with a class name `card` and style it as a simple card with a background, padding, border-radius, and shadow
- Deliver your project using Vite in the folder: `/exercises/02/`
- **Deadline:** To be announced