HTML and CSS

Happy Coding!

HTML Introduction.

Objectives:

By the end of this session, you should be able to:

- Create simple pages using only HTML
- Explain the structure of an HTML Document

Happy Coding!

Intell Eyes {Countinfinite Technologies Pvt Ltd}

- Describe the difference between an element and an attribute

Getting started with HTML

HTML, which stands for **H**yper **T**ext **M**arkup **L**anguage, is one of the fundamental building blocks of the web. When you visit a webpage, your browser reads HTML and renders the document on the page. HTML is primarily responsible for holding the content of a page; the styling is defined by CSS, and much of the interactivity is controlled by JavaScript (we'll get to these technologies soon enough).

There are many reference materials on HTML and its history. If you'd like to learn more, check out Wikipedia or the MDN docs. For now, let's learn about HTML by actually writing some HTML!

The Structure of HTML

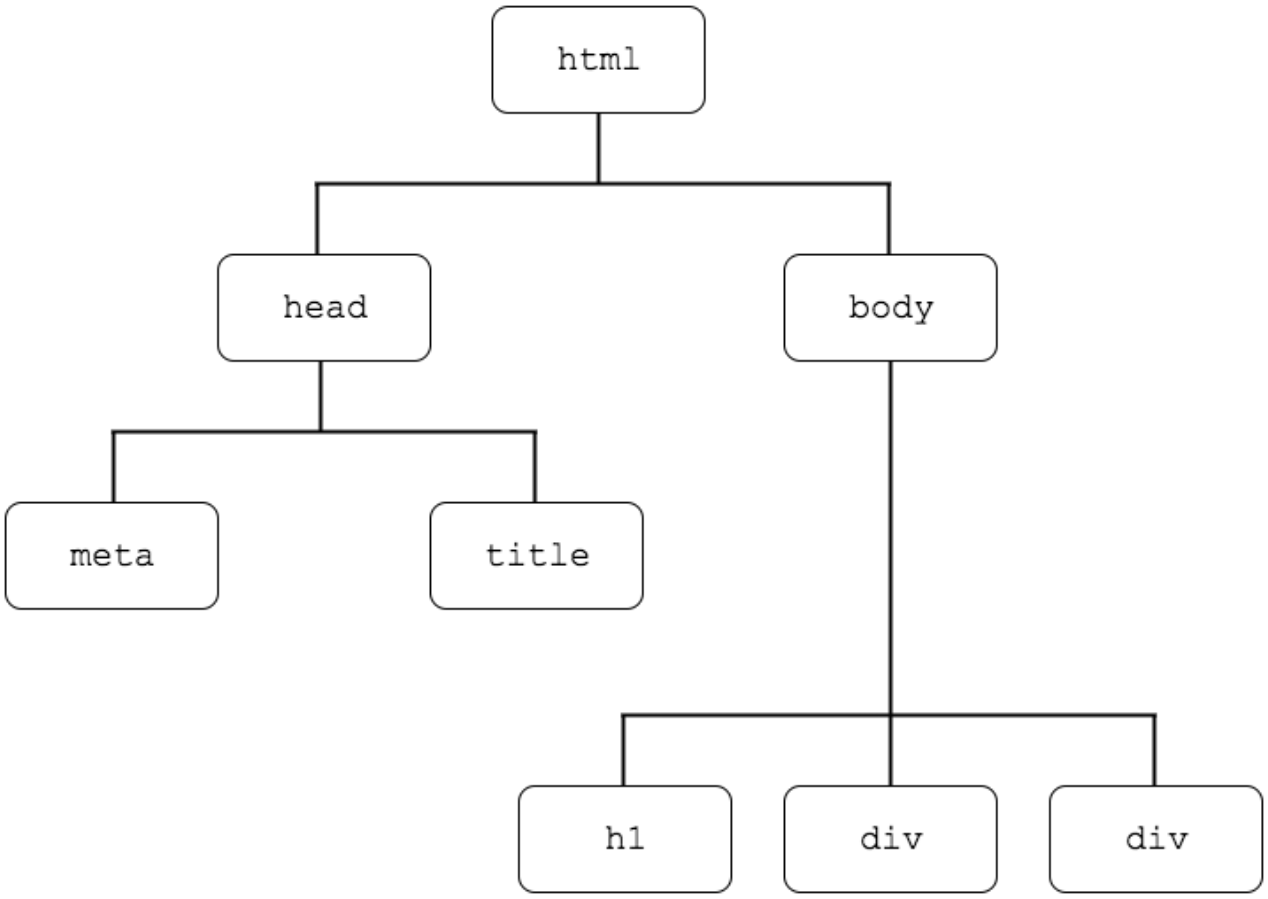
If you've never seen HTML before, one of the first things you'll notice is probably the number of elements, which are the building blocks of HTML. We specify the name of the element using its name along with opening and closing brackets (`<>`). To denote the ending of a tag, we use `</>`. Here's a basic HTML document:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My first HTML page!</title>
</head>
<body>
  <h1>Here's some important text!</h1>
  <div>Here's some content.</div>
  <div>See ya later!</div>
</body>
</html>
```

In the above example, we see a number of elements: `html`, `head`, `meta`, `title`, and so on. We'll see more elements as we learn more about HTML.

Notice that sometimes elements are nested inside of other elements. `meta` and `title` are inside of `head`, `h1` is inside of `body`, and everything is inside of `html`. These are commonly referred to in the language of parents and children: we say that the `h1` element is a child of the `body` element, for example, and that the `body` element is the parent of the `h1` element. Elements with the same parent are called siblings; the `meta` and `title` tags are siblings in the document above.

In fact, you can think of the entire structure of an HTML document in terms of these relationships. From this perspective, the document looks like a tree, with parent nodes higher up on the tree, and child nodes down below. Here's an example of how you could visualize the structure of the HTML file above:



Doctype and essential tags

Another essential building block for our web pages is to specify the `doctype` declaration, which tells the browser that the page is written in HTML. We will be using `<!DOCTYPE html>` to tell the page that we are using HTML5. This line about the `doctype` will be your first line in basically all of the HTML files that you write, and you won't need to modify it. If you'd like to read more about the `doctype`, check out this Stack Overflow post.

After the `doctype` declaration, you'll see a few essential tags across all of the HTML files you come across:

`<html>` - This is where we place all of our elements that comprise the contents of our page. `<head>` - This is the container for the elements that have content that does not need to be displayed on the page (like metadata, the title, or, as we will see, scripts and stylesheets). `<meta>` - This is a tag for providing metadata (data about our data) to the page. We can use meta tags to display what character set we are using or for SEO (Search Engine Optimization) purposes. `<title>` - This tag gives the page a title that can be displayed in the tab of your browser. `<body>` - This defines the main content of the HTML page.

We'll discuss the elements inside of the `body` tag in a later chapter.

Attributes and Content

Intell Eyes {Countinfinite Technologies Pvt Ltd}

Our sample HTML page contains a number of elements. Some of those elements contain content: for instance, the content of the `h1` tag is the text "Here's some important text!" Some elements also contain **attributes**, which are used to provide additional information about an element. The attributes are always set inside of the opening tag of the element, and take the form `attribute_name="attribute value"`. In our sample HTML, there are two attributes:

1. The `html` tag has a `lang` attribute set to `"en"`. This tells the browser that the HTML document is written in English. This will probably be the default you'll want for all of the web pages you create. For more on the `lang` attribute, check out this article.
2. The `meta` tag has a `charset` attribute of `"UTF-8"`. This specifies the character encoding for the file. You don't need to worry too much about this for now, but if you'd like to learn more about UTF-8, you can start with this Wikipedia article.

We'll see many other attributes as we start writing more HTML. For example, when you create a link using an `a` tag, you'll pass in the webpage the link should point to as an attribute. The same thing applies when you create an image using the `img` tag and need to point to the URL for the image.

Headings, Paragraphs, Breaks and horizontal rows

In HTML we have quite a few ways to display and separate text. We have heading tags `<h1></h1>`, `<h2></h2>`, ..., `<h6></h6>` (`h1` tags are the largest, `h6` are the smallest), paragraph tags `<p></p>`, line breaks `<br>` (notice that this tag does not close), and horizontal rows `<hr>` (this tag does not close as well). You'll commonly see heading tags used for titles of pages and sections, while `p` tags are used for larger chunks of text.

Lists in HTML

What if you don't want to write paragraphs, but instead just want to display some simple lists? To create list items in HTML, we can either use ordered lists or unordered lists:

```
<ol>
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ol>

<ul>
  <li>Something Not Ordered</li>
  <li>Something Not Ordered</li>
  <li>Something Not Ordered</li>
</ul>
```

By default, ordered lists will be numbered starting from 1. You can change the numbering by setting the `type` attribute on the list. What do you think the ordering will look like for each of the following lists?

```
<ol type="A">
```

Happy Coding!

```
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ol>

<ol type="a">
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ol>

<ol type="I">
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ol>

<ol type="i">
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ol>
```

Divs + Spans

In HTML there are two important tags which do not have much semantic significance, but are useful for laying out a page. These tags are `<div></div>` and `<span></span>`. Both can be used to format text, but they behave differently on the page. Try pasting the following into an HTML document, and take a look at how your browser renders the text:

```
<div>I'm in a div.</div>
<div>I'm in a div as well.</div>
<span>I'm in a span.</span>
<span>I'm in a span as well.</span>
```

What are some differences you notice? (If you need a hint, check out this article.)

Text modifications

To change the look of text on the page we can use tags like , , , , , ,

and . Take a look at what these do!

Anchor tags

Intell Eyes {Countinfinite Technologies Pvt Ltd}

When we want to create hyperlinks to other pages, we use an anchor tag `<a></a>`. To specify what link we want to go to we use the `href` attribute. For example, to add an anchor tag linking to Google, we would write something like `<a href="http://www.google.com">Go To Google!</a>`

Images

We can add images and specify their source using the `src` attribute. Another important attribute for image tags is the `alt` attribute which is what is displayed when the image fails to load and is very important for SEO purposes as well. It looks something like this: `<img src="http://lorempixel.com/200/200" alt="Some random 200x200 picture">`

Emmet.

Objectives:

By the end of this session, you should be able to:

- Use Emmet shortcuts to write HTML faster
- Understand how to add text inside of elements with Emmet
- Write nested statements to generate HTML quickly

Introduction and Installation

Sometimes typing all of these HTML tags is quite a pain. There are a lot of characters to type, some of them are awkward to type (especially the opening and closing brackets), and you have to remember to add closing tags as well!

Thankfully, there is a nice tool to help us out called Emmet, which makes writing HTML a breeze. We will be using VS Code as our text editor so to get started, follow these steps:

1. Install VS Code. You can find it here
2. Open VS Code. Create and save a file with a `.html` extension.

In your `.html` file - type in `!` and press tab and this should appear:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
</head>
<body>

</body>
</html>
```

Don't worry about trying to remember all of this HTML boilerplate; Emmet's got you covered!

Happy Coding!

Emmet Shortcuts

One of the nice things you can do with emmet is write HTML quickly! Get started by typing `h1` and pressing tab. You should see `<h1></h1>` with your cursor in the middle!

If you want to nest tags inside of others you can use the `>` symbol. For example, typing `div>span` and pressing tab should produce `<div><span></span></div>`. Once again, your cursor should show up in the middle.

If you want to add text inside a tag you can use the `{}` braces and then press tab. Emmet converts `h1{Hello World}` to `<h1>Hello World</h1>`.

If you want to create multiple tags you can use the `*` operator.

`p>div{inner}*2` produces:

```
<p>
  <div>inner</div>
  <div>inner</div>
</p>
```

You can also create sibling elements (remember, these are elements with the same direct parent) using the `+` operator.

`div+p>span` produces

```
<div></div>
<p>
  <span></span>
</p>
```

What about attributes? Emmet's got you covered there as well! You can pass in an attribute name and value between square brackets.

`div[id="hello"]{Hi everybody!}>span{Yo}*3` produces

```
<div id="hello">Hi everybody!
  <span>Yo</span>
  <span>Yo</span>
  <span>Yo</span>
</div>
```

There's a lot more that Emmet can do for you as well. You can always check out the documentation for all of the features, but there's also an Emmet cheat sheet which is a great reference when you're first trying to pick up these shortcuts.

Opening in browser

Happy Coding!

Intell Eyes {Countinfinite Technologies Pvt Ltd}

A handy extension for VS Code is open in browser. If you install this, when working with HTML files you can right click on the file and select "Open in Default Browser". This will open your page right in a browser tab! In order for this to work, you need to make sure your file is saved with the `.html` extension.

Emmet Exercises

Write the one-line Emmet commands to produce the following HTML:

```
<h1></h1>
<div>
    <p>
        <section>
            <h1>Nice!</h1>
        </section>
    </p>
</div>
<ul>outside
    <li>inside</li>
    <li>inside</li>
    <li>inside</li>
    <li>inside</li>
</ul>
<p>sibling 1</p>
<div>sibling 2</div>
<p>parent
    <div>child
        <div>grandchild</div>
    </div>
</p>
```

Basic Tags.

Objectives:

By the end of this session, you should be able to:

- Use heading tags, paragraph tags and breaks to separate content
- Create unordered and ordered lists
- Compare and contrast divs and spans
- Add anchor and image tags

Headings, Paragraphs, Breaks and horizontal rows

Happy Coding!

In HTML we have quite a few ways to display and separate text. We have heading tags `<h1></h1>`, `<h2></h2>`, ..., `<h6></h6>` (`h1` tags are the largest, `h6` are the smallest), paragraph tags `<p></p>`, line breaks `<br>` (notice that this tag does not close), and horizontal rows `<hr>` (this tag does not close as well). You'll commonly see heading tags used for titles of pages and sections, while `p` tags are used for larger chunks of text.

Lists in HTML

What if you don't want to write paragraphs, but instead just want to display some simple lists? To create list items in HTML, we can either use ordered lists or unordered lists:

```
<ol>
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ol>

<ul>
  <li>Something Not Ordered</li>
  <li>Something Not Ordered</li>
  <li>Something Not Ordered</li>
</ul>
```

By default, ordered lists will be numbered starting from 1. You can change the numbering by setting the `type` attribute on the list. What do you think the ordering will look like for each of the following lists?

```
<ol type="A">
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ol>

<ol type="a">
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ol>

<ol type="I">
  <li>First</li>
  <li>Second</li>
  <li>Third</li>
</ol>

<ol type="i">
  <li>First</li>
  <li>Second</li>
```

```
  <li>Third</li>
</ol>
```

Divs + Spans

In HTML there are two important tags which do not have much semantic significance, but are useful for laying out a page. These tags are `<div></div>` and `<span></span>`. Both can be used to format text, but they behave differently on the page. Try pasting the following into an HTML document, and take a look at how your browser renders the text:

```
<div>I'm in a div.</div>
<div>I'm in a div as well.</div>
<span>I'm in a span.</span>
<span>I'm in a span as well.</span>
```

What are some differences you notice? (If you need a hint, check out this article.)

Text modifications

To change the look of text on the page we can use tags like , , , , , ,

and . Take a look at what these do!

Anchor tags

When we want to create hyperlinks to other pages, we use an anchor tag `<a></a>`. To specify what link we want to go to we use the `href` attribute. For example, to add an anchor tag linking to Google, we would write something like `<a href="http://www.google.com">Go To Google!</a>`

Images

We can add images and specify their source using the `src` attribute. Another important attribute for image tags is the `alt` attribute which is what is displayed when the image fails to load and is very important for SEO purposes as well. It looks something like this: `<img src="http://lorempixel.com/200/200" alt="Some random 200x200 picture">`

HTML Comments

Sometimes we want to explain certain parts of our HTML, but not have it display on the page. We can put these explanations inside of comments, which will show up if someone looks at the HTML file, but won't be rendered on the page by a browser. Comments are helpful in all programming languages when we want to document our code. To add a comment in HTML, we simply include text inside of this syntax: ``. Here's an example:

```
<!-- Tell the reader to keep up the good work -->
<h1>
  Well done! You made it to the end of this section! Now go and practice some
HTML with the exercises below!
</h1>
```

If you render this code in the browser, you should see that the comment doesn't show up.

Semantic HTML Introduction.

Objectives:

By the end of this chapter, you should be able to:

- Define what semantic HTML is and what tags can be used to create it

Semantic HTML

When HTML5 was released, a number of new elements came along with it. These elements have given rise to a term called *Semantic HTML*, which refers to the practice of using these new elements to structure your HTML in such a way that the element name is descriptive of the content inside of it. Prior to HTML5 you may have seen much of the content of an HTML page inside of `div`, `span`, and `p` tags, but HTML5 allows us to be a little more precise in pairing our content with our elements. For more on Semantic HTML, check out this article.

Semantic HTML tags don't alter your page in any specific way, but make more sense to the developer and reader of the HTML as to what they do. Here are some semantic HTML5 tags (there are many more):

- `header` - used to display the header information on a page
- `nav` - used to display a navbar for navigating a page
- `section` - used to display a section of the page
- `article` - used to display independent, self-contained content.
- `aside` - used to display content aside from main content (on the side)
- `footer` - used to display the footer information on a page

For more on HTML5 (including some of the new semantic elements), check out the MDN docs.

Here's a quick example illustrating what Semantic HTML looks like. Take a look at this HTML document:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Non-semantic HTML</title>
</head>
<body>
```

```
  <div>
    <h1>My super cool blog</h1>
  </div>
  <div>
    <ul>
      <li>About Me</li>
      <li>Contact</li>
      <li>Search</li>
      <li>Store</li>
    </ul>
  </div>
  <div>
    <div>
      <h2>Important Title</h2>
    </div>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Expedita,
ullam maiores? Ex, eum! Ducimus ut velit ad ullam quisquam fugit vitae harum
eligendi, qui necessitatibus, laudantium reprehenderit nihil, sint aperiam!</p>
    <p>Recusandae fugiat, eum, modi odio, quis quibusdam a maxime impedit
accusamus ipsum nulla maiores commodi voluptas saepe nisi laudantium doloremque
perferendis suscipit! Ullam consequuntur veritatis, atque quidem reiciendis
facilis voluptatum!</p>
    <p>Animi sit, facere, molestiae vel consequuntur suscipit alias dolorem
adipisci hic non id sint illum, doloremque iure ut assumenda! Provident neque
asperiores, vitae ab. Ducimus recusandae sed rerum doloribus consequatur!</p>
  </div>
  <div>
    <div>
      <h2>Super Important Title</h2>
    </div>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Accusantium
consequuntur ab, aliquid facilis omnis praesentium recusandae. Aliquid itaque
omnis nam harum sit! Explicabo eum consectetur aut repellendus, nostrum hic
iste.</p>
    <p>Illo repellat quia, minima reiciendis. Laudantium, perferendis!
Laudantium consectetur impedit at nobis error aspernatur repellendus quaerat,
quas repudiandae maxime. Architecto consequatur, autem natus quaerat illo
accusamus! Voluptatem accusantium pariatur perspiciatis!</p>
    <p>Vitae corrupti, tempora itaque ab est facere eius optio fugiat, ex
possimus beatae nulla esse molestias suscipit quisquam nostrum sunt quod
mollitia. Mollitia reiciendis, aspernatur. Quas, eligendi temporibus earum
debitis?</p>
  </div>
  <div>
    <div>
      <h4>Recent Posts</h4>
    </div>
    <ul>
      <li>Post 1</li>
```

```
      <li>Post 2</li>
      <li>Post 3</li>
      <li>Post 4</li>
    </ul>
  </div>
  <div>
    <p>Copright MyAwesomeBlog.net. All Rights Reserved. Don't steal my stuff!
</p>
  </div>
</body>
</html>
```

There's nothing too complex going on here, but with so many divs floating around it can be hard to visualize the structure of this document.

Here's the same file, but refactored to use some HTML5 elements:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Non-semantic HTML</title>
</head>
<body>
  <header>
    <h1>My super cool blog</h1>
  </header>
  <nav>
    <ul>
      <li>About Me</li>
      <li>Contact</li>
      <li>Search</li>
      <li>Store</li>
    </ul>
  </nav>
  <article>
    <header>
      <h2>Important Title</h2>
    </header>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Expedita,
ullam maiores? Ex, eum! Ducimus ut velit ad ullam quisquam fugit vitae harum
eligendi, qui necessitatibus, laudantium reprehenderit nihil, sint aperiam!</p>
    <p>Recusandae fugiat, eum, modi odio, quis quibusdam a maxime impedit
accusamus ipsum nulla maiores commodi voluptas saepe nisi laudantium doloremque
perferendis suscipit! Ullam consequuntur veritatis, atque quidem reiciendis
facilis voluptatum!</p>
    <p>Animi sit, facere, molestiae vel consequuntur suscipit alias dolorem
adipisci hic non id sint illum, doloremque iure ut assumenda! Provident neque
asperiores, vitae ab. Ducimus recusandae sed rerum doloribus consequatur!</p>
```

```
  </article>
  <article>
    <header>
      <h2>Super Important Title</h2>
    </header>
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Accusantium consequuntur ab, aliquid facilis omnis praesentium recusandae. Aliquid itaque omnis nam harum sit! Explicabo eum consectetur aut repellendus, nostrum hic iste.</p>
    <p>Illo repellat quia, minima reiciendis. Laudantium, perferendis! Laudantium consectetur impedit at nobis error aspernatur repellendus quaerat, quas repudiandae maxime. Architecto consequatur, autem natus quaerat illo accusamus! Voluptatem accusantium pariatur perspiciatis!</p>
    <p>Vitae corrupti, tempora itaque ab est facere eius optio fugiat, ex possimus beatae nulla esse molestias suscipit quisquam nostrum sunt quod mollitia. Mollitia reiciendis, aspernatur. Quas, eligendi temporibus earum debitis?</p>
  </article>
  <article>
    <header>
      <h4>Recent Posts</h4>
    </header>
    <ul>
      <li>Post 1</li>
      <li>Post 2</li>
      <li>Post 3</li>
      <li>Post 4</li>
    </ul>
  </article>
  <footer>
    <p>Copright MyAwesomeBlog.net. All Rights Reserved. Don't steal my stuff!
</p>
  </footer>
</body>
</html>
```

In this particular case, we've been able to eliminate all the divs, and replace them with elements that have more semantic meaning regarding the layout of the page and the purpose of the content. We can look at this document and see where the footer his, what the header for each area is, and so on. That's the beauty of semantic HTML!

Tables and Forms.

Objectives:

By the end of this session, you should be able to:

- Build semantic HTML tables using HTML5 elements

Intell Eyes {Countinfinite Technologies Pvt Ltd}

- Build forms using various inputs and understand their specific attributes

Building Semantic HTML Tables

Although they are not as popular as they used to be (entire web pages used to be designed just using tables!), it's still quite valuable to know how to build tables. They are great for displaying certain kinds of data!

- This tag encompasses the entire table and wraps all of your HTML code for a table

`<tr>` - The table row element defines a row of cells which can include `td` and/or `th` tags

`<th>` - The table header tag defines a cell as the "header" for group of cells. This is traditionally placed inside of a `thead` tag.

`<td>` - The table data tag defines a cell of a table that contains some data.

`<thead>` - The table head tag defines a set of rows which define the columns of a table

`<tbody>` - From MDN:

> The HTML Table Body Element (`<tbody>`) defines one or more `<tr>` element data-rows to be the body of its parent `<table>` element (as long as no `<tr>` elements are immediate children of that table element.) In conjunction with a preceding `<thead>` and/or `<tfoot>` element, `<tbody>` provides additional semantic information for devices such as printers and displays. Of the parent table's child elements, `<tbody>` represents the content which, when longer than a page, will most likely differ for each page printed; while the content of `<thead>` and `<tfoot>` will be the same or similar for each page printed. For displays, `<tbody>` will enable separate scrolling of the `<thead>`, `<tfoot>`, and `<caption>` elements of the same parent `<table>` element.

`<tfoot>` - The table foot tag defines a set of rows summarizing the columns of the table.

Here is an example of what a semantic table looks like in HTML:

```
<table>
    <thead>
        <tr>
            <th>heading</th>
            <th>heading</th>
            <th>heading</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>content</td>
            <td>content</td>
            <td>content</td>
        </tr>
        <tr>
            <td>content</td>
```

Happy Coding!

```
            <td>content</td>
            <td>content</td>
        </tr>
    </tbody>
    <tfoot>
        <tr>
            <td>content</td>
            <td>content</td>
            <td>content</td>
        </tr>
    </tfoot>
</table>
```

Building HTML Forms

Forms are another essential part of HTML and will be used quite a bit when we get to server-side programming. We use forms to transmit data to other pages and servers.

From MDN:

> HTML Forms are one of the main points of interaction between a user and a web site or application. They allow users to send data to the web site. Most of the time that data is sent to the web server, but the web page can also intercept it to use it on its own.
>
> An HTML Form is made of one or more widgets. Those widgets can be text fields (single line or multi line), select boxes, buttons, check boxes, or radio buttons. Most of the time, those widgets are paired with a label that describes their purpose.

To get started with a form, we need a `<form></form>` tag and an `action` attribute (where the form should go when submitted). We will use `#` as the value of this attribute to signify that the form should not go anywhere. Forms also have another essential attribute called `method` which specifies how the form should be submitted (but don't worry about this one, we'll talk much more about it later).

`<form action="#"></form>`

Like MDN says, forms need to have widgets (or inputs) which can be paired with a label to describe their purpose. What does that mean, exactly? Let's take a look at this example form:

```
<form action="#">
    <div>
        <label for="username">Username</label>
        <input type="text" id="username">
    </div>
    <div>
        <label for="password">Password</label>
        <input type="password" id="password">
    </div>
    <div>
        <label>Favorite Instructor</label>
```

```
        <label for="elie">Elie</label>
        <input type="radio" name="favorite_instructor" id="elie">
        <label for="matt">Matt</label>
        <input type="radio" name="favorite_instructor" id="matt">
        <label for="tim">Tim</label>
        <input type="radio" name="favorite_instructor" id="tim">
    </div>
    <div>
        <label>Favorite Programming Language(s)</label>
        <label for="ruby">Ruby</label>
        <input type="checkbox" name="favorite_language" value="Ruby" id="ruby">
        <label for="javascript">JavaScript</label>
        <input type="checkbox" name="favorite_language" value="JavaScript"
id="javascript">
        <label for="python">Python</label>
        <input type="checkbox" name="favorite_language" value="Python"
id="python">
        <label for="go">Go</label>
        <input type="checkbox" name="favorite_language" value="Go" id="go">
    </div>
    <div>
        <label>Additional Comments</label>
        <textarea name="" id="" cols="30" rows="10"></textarea>
    </div>
    <div>
        <input type="submit" value="">
    </div>
</form>
```

See what this looks like

You can see that some of these inputs have `name` attribute. While we will not be discussing this in great detail for now (we will come back to this later), this input is necessary if we are trying to collect form values to be transmitted to a server/another page.

HTML5 Media.

Objectives:

By the end of this session, you should be able to:

- Embed audio and video into an HTML page
- Define what an iframe is and list use cases for using one

HTML5 Media

With HTML5 comes the ability to embed audio and video into our HTML. We can do this using the audio tag and video tag. You can read more about the audio tag here, and the video tag here.

iframe (inline frame)

A bit of an older tool that you will see sometimes is an `<iframe></iframe>` tag. The `iframe` tag is used to embed another document (HTML page) within the current page. You can read more about it here. For more on how they are (or are mostly not) used today, check out this Stack Overflow question.

Although you will rarely be creating your own `iframe` tags, it is very common to embed content from other websites on your page through an `iframe` tag. You can read how to embed an audio player from Spotify here and a YouTube video here.

CSS Syntax and Selectors.

Objectives:

By the end of this session, you should be able to:

- Define what CSS is and demonstrate three different ways to include it on your page
- Use selectors to target HTML
- Explain how specificity plays a large factor when using selectors

Getting started with CSS

When we first introduced HTML, we said that it was primarily responsible for holding the content of the page, while the styling was handled by CSS. So what is CSS, anyway, and how can we use it to style our web pages?

CSS stands for **C**ascading **S**tyle **S**heets, and it's a way to describe style rules that we'd like to set for our HTML elements. Using CSS we can alter things like colors, fonts, margins, text alignment, and much more! Here's some further reading on the background and history of CSS, if you're interested.

There are three ways to include CSS in our web pages:

1. Inline styling. This is done by adding a `style` attribute to an HTML tag. We will see shortly that this can have some unintended consequences.
2. `<style></style>`. This is done by using a `style` tag and placing CSS inside of it.
3. External stylesheets. **This is the most common way and almost always the best practice**

Since we will be using external stylesheets, we need to figure out how to link our CSS to our HTML. The answer is with a `link` tag! If we have a file called `style.css` in the same folder as our HTML file, we can link them with the following tag, which you should place inside of the `head` element:

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

Syntax, Selectors

Intell Eyes {Countinfinite Technologies Pvt Ltd}

Each CSS rule consists of a selector (how we find the element/elements) to target and a pair of curly braces, a.k.a. `{}`. Inside of `{}` we specify properties and values; we separate the two with a `:`, and after the value we add a `;`. To add multiple selectors we separate each one by a `,`. This looks something like this:

```
body {
    background-color: green;
}
```

In this rule, we are targeting the `<body></body>` element and applying a background color to it. While this works, it might be a bit too general (maybe we only want a background color of green on certain parts of the page, not the whole body). In order to be more specific, we can use attributes like `class` and `id` to target our elements. We'll talk more about these attributes in the next chapter; for now, you can think of them as a tool that makes it easier to find the element you're looking for on the page. The biggest difference is that `id`s must be unique: two elements should never have the same `id`. But you can have multiple elements share the same `class`.

Let's take a look at a sample HTML file. We will call this file `index.html`. Notice the `<link/>` tag we are using to link to an external stylesheet,which we will call `style.css`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Learn CSS</title>
    <link rel="stylesheet" href="style.css">
</head>
<body class="main_styles" id="container">
    <header class="main_heading">
        <h1>Welcome to our website</h1>
        <h4>We're so happy to have you</h4>
    </header>
    <section class="main">
        <ul> Here are some reasons you should keep reading....
            <li>You're learning things!</li>
            <li>CSS is fun!</li>
            <li>Well.....not always, but it is essential to know!</li>
        </ul>
    </section>
    <footer class="main_footer">
        <p>
            Thanks for checking out this page!
            <a href="#">Here is a link that goes nowhere! It's for showing you
stuff with CSS......we promise</a>
        </p>
    </footer>
</body>
</html>
```

Happy Coding!

We can see from this HTML that we have added some `class` and `id` attributes. Right now they do not alter anything about the page, but we can use them as selectors with CSS. To target a class with CSS, add a `.` before the name of the class. For an id, add a `#` before the name of the id.

This is what some CSS might look like for our page (imagine our filename is `style.css` and is in the same directory as our `index.html`:

```
#container {
    font-size: 16px;
}
.main_heading {
    background-color: grey;
}
.main {
    background-color: blue;
}
.main_footer {
    background-color: green;
}
.main_header,
.main_footer {
  color: red;
}
```

After typing this code into your `style.css` file, refresh the page. If you've set things up properly, you should see some big changes in the way your page looks!

Other useful selectors

Descendant Selector

The descendant selector matches all elements that are descendants of a specified element. The selector below will find all of the `p` tags inside of a `footer` tag (read the selectors from right to left).

```
footer p {
    background:red;
}
```

Adjacent Selector

The adjacent selector finds all elements that are directly adjacent to some other specified element. The selector below will find all the `h4` tags that are directly adjacent to an `h1` tag.

```
h1+h4 {
    background:blue;
}
```

Happy Coding!

Direct child selector

This is commonly confused with the descendant selector. The direct child selector matches all elements that are *direct children* - not just ancestors - of a specified element. The selector below will find all `li`s which are children of a `ul`.

```
ul > li {
    background:blue;
}
```

Attribute Selector

The attribute selector finds elements based on the value of some attribute. The selector below will find all `a` tags whose `href` attribute is set to "#".

```
a[href="#"] {
    background:blue;
}
```

First / Last Child Selector

The `first-child` and `last-child` selectors find all elements which are the first (or last, respectively) children of their parents. The selector below will find all the li's which are the first children of their parents.

```
li:first-child {
    background:blue;
}
```

n-th child selector

This is a sort of generalization of the selectors above. `nth-child` will find all elements which are the nth children of their parents, for some specified value of n. The selector below will find all the li's which are the second children of their parents.

```
li:nth-child(2) {
    background: teal;
}
```

These aren't the only selectors, but they are some of the more commonly used ones. For even more examples, check out MDN.

One important thing to know about CSS is the **C**, or **cascading** nature of CSS. This means that your code is read top to bottom, so if you have the same level of specificity, the rule closest to the bottom wins. For example, consider the following stylesheet:

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

p {
    font-size: 100px;
}
```

These two rules conflict, but because they have the same degree of specificity, cascading will kick in, and the latter rule will overwrite the former. The text on this page will be huge!

Specificity.

Objectives:

By the end of this session, you should be able to:

- Define what specificity is
- Understand the difference between using element, class, id, inline and !important styling
- Compare and contrast ids and classes

Specificity

So we've learned so far that CSS is built off of selecting HTML elements and their attributes and applying styling rules. But what happens if we have something like this (we are using a `style` tag just as an example)? One of the most important ideas around CSS is that of specificity, or how specific we are with our selectors. Imagine we have some CSS that looks like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        #main {
            background: green;
        }
        .container {
            background: blue;
        }
        div {
            background: red;
            height: 200px;
            width: 200px;
        }
    </style>
</head>
<body>
```

```
    <div class="container" id="main">
        I AM A DIV!
    </div>
</body>
</html>
```

`container` is a class on the single `div` element and `main` is an id on that `div` element... so we have 3 conflicting rules. Which one wins? The element styling? The class styling? The id styling?

Based on what you've learned, you might guess that the background of the `div` should be red, since CSS cascades and red is the last color to be set as the background color. However, if you open up this HTML page, you'll see that the background is green, not red! So what's going on?

To answer this question let's introduce a brief table for understanding. These weightings are not exact (10 element selectors does not equal a class selector), it is more an idea of showing you the magnitude of each kind of selector.

Selector	Weighting
element	1
class	10
id	100
inline-style	1000
!important	10000+

What this means is that a style rule on an id is weighed much more heavily than a style rule on a class. Similarly, a style rule on a class is weighed more heavily than a rule on an element. Put another way, id rules are more *specific* than class rules, and class rules are more *specific* than element rules.

This gives us another way to differentiate between classes and ids. Not only are ids meant to be unique, unlike classes, but also ids are more specific! This is also why the background color in our example is green: the id rule trumps the class rule, which in turn trumps the element rule. Cascading only applies when the rules have the same degree of specificity; in this case, rules about specificity determine the style, not rules about cascading.

Let's modify our earlier example a bit:

```
<!DOCTYPE html>
<html>
<head>
    <title>Specificity</title>
    <style>
        #main {
            background: green;
        }
```

```
        .container {
            background: blue;
        }
        div {
            background: red;
            height: 200px;
            width: 200px;
        }
        .winner {
            background: aqua !important;
        }
    </style>
</head>
<body>
    <div class="container winner" id="main" style="background:purple;">
        I AM A DIV!
    </div>
</body>
</html>
```

Here we see the two other types of selectors mentioned in the table above: an inline style rule and a style rule using the `!important` flag. Inline styles are styles declared using the `style` attribute on an HTML element. Both of these patterns should be avoided (we'll talk more about this down below), but for now, try to answer the following: In order of specificity (least to most specific), what is the background color?

Another example

Let's try something a bit trickier. What happens if we have some code like this?

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        div.container {
            background: green;
        }
        body div {
            background: red;
        }
        div.container#main {
            background: purple;
        }
        body #main {
            background: magenta;
        }
        .container {
```

```
            background: blue;
        }
        div {
            background: yellow;
            height: 200px;
            width: 200px;
        }
        body div.container#main {
            background: brown;
        }
    </style>
</head>
<body>
    <div class="container" id="main">
        I AM A DIV!
    </div>
</body>
</html>
```

See what this looks like

Which one wins? Try to calculate point values for each one! (1 for each element selector, 10 for each class selector and 100 for each id selector)

Selector	Weighting
div	1
body div	2
.container	10
div.container	11
body #main	101
div.container#main	111
body div.container#main	112

Therefore the order of colors (from least to most specific) is:

1. yellow
2. red
3. blue
4. green
5. magenta
6. purple
7. brown

We previously mentioned that using inline styling (a `<style></style>` tag) can have some unintended consequences. Strong specificity is one of them! Remember that inline styling is equal to 1000 points and is far more specific than any id, class or element! Even more, using `!important` will override ANY styling that you have (unless you have a more specific `!important`) so try your best to not use it. Not only does it make it more difficult to write future CSS, but it can get complicated quickly when working with other developers. If we catch you writing `!important` in your CSS, we'll ask you to refactor immediately!

Common Rules.

Objectives:

By the end of this session, you should be able to:

- Use common typography rules to add styling
- Include a custom Google font

CSS rules

Color

So far we have seen a few basic properties, such as `background-color`, which allows us to set the background color of an element. The value we assign to this property can either be a named color (`blue`, `green`, `red` etc.), a hex code (# followed by six values from 0 to F), an rgb value (red, green, blue), or an hsl value(hue, saturation, lightness) value. If you don't know what these are, you can mess around with them here. These same values can be assigned to the `color` property, which sets the font color on an element.

Typography

If we want to change the text on our page, we commonly use some of these rules:

- `font-size` - select the size of your font (we will discuss other units of measurement in the next unit)
- `text-align` - align the position of the text
- `font-family` - choose what kind of font you want to use
- `letter-spacing` - add positive or negative (less) space between characters
- `text-transform` - capitalize/uppercase/lowercase your text
- `line-height` - choose the amount of height between lines
- `font-variant` - useful for displaying a small-caps font
- `text-shadow` - add a shadow to your text
- `color` - select the color of the text

Here's an example of how you could set values for each of these properties:

```
.some_text {
    color: #BDBFEF;
    font-size: 12px;
```

```
    text-align: center;
    font-family: helvetica, sans-serif;
    letter-spacing: -1px;
    text-transform: lowercase;
    text-shadow: 0 2px 0 black;
    /* first value is horizontal/x offset */
    /* second value is vertical/y offset */
    /* third value is how much blur */
    /* fourth value is the color */
    /* you can even have multiple shadows separated by a comma! */
    line-height: 1.4;
    font-variant: small-caps;
}
```

There are hundreds of CSS properties, much more than we can cover here. You'll pick up more properties as you start working on your projects. If you'd like to see an exhaustive reference of CSS properties and other keywords, MDN has got you covered.

Using a custom google font

You can head over to https://www.google.com/fonts and select some fonts, then click "Use" at the bottom. It will then give you a tag as well as the CSS necessary. Here is an example using Open Sans. Create an HTML file, and inside of the `head` add the following `link`:

```
<link href='https://fonts.googleapis.com/css?family=Open+Sans' rel='stylesheet' type='text/css'>
```

Then, inside of your style sheet, add the following rule:

```
body {
    font-family: 'Open Sans', sans-serif;
}
```

If you put some text content in your HTML, you should see that the font has changed to Open Sans!

Box Model.

Objectives:

By the end of this session, you should be able to:

- Explain what the box model is
- Compare and contrast margin, padding and border
- Use properties like `box-sizing` to better calculate width and height

The Box Model

Happy Coding!

Intell Eyes {Countinfinite Technologies Pvt Ltd}

Now that we have an idea around basic color and typography, let's see how to add some space around our elements and the content inside. In order to do that, we first have to understand what the *box model* is. But before we explain the box model, let's make sure we understand some basic properties for adding space between elements and content.

`width` - The width CSS property specifies the width of the content area of an element. `height` - The height CSS property specifies the height of the content area of an element. `margin` - used to generate space around elements. `margin` is a shorthand for `margin-top`, `margin-right`, `margin-bottom` and `margin-left`. `padding` - The `padding` property in CSS defines the innermost portion of the box model, creating space around an element's content, inside of any defined margins and/or borders. `padding` is a shorthand for `padding-top`, `padding-right`, `padding-bottom` and `padding-left`. `border` - The `border` property defines the space between an element's padding and margin. `border` is shorthand for `border-top`, `border-right`, `border-bottom` and `border-left`.

```
div {
    /* This: */
    margin: 20px;
    /* Is the same as: */
    margin-top:20px;
    margin-right:20px;
    margin-bottom:20px;
    margin-left:20px;

    /* This: */
    padding: 10px 20px;
    /* Is the same as: */
    padding-top:10px;
    padding-right:20px;
    padding-bottom:10px;
    padding-left:20px;
}

h1 {
    /* This: */
    margin: 20px 10px 5px;
    /* Is the same as: */
    margin-top:20px;
    margin-right:10px;
    margin-bottom:5px;
    margin-left:10px;
}

h2 {
    /* This: */
    margin: 10px 20px;
    /* Is the same as: */
    margin-top: 10px;
```

Happy Coding!

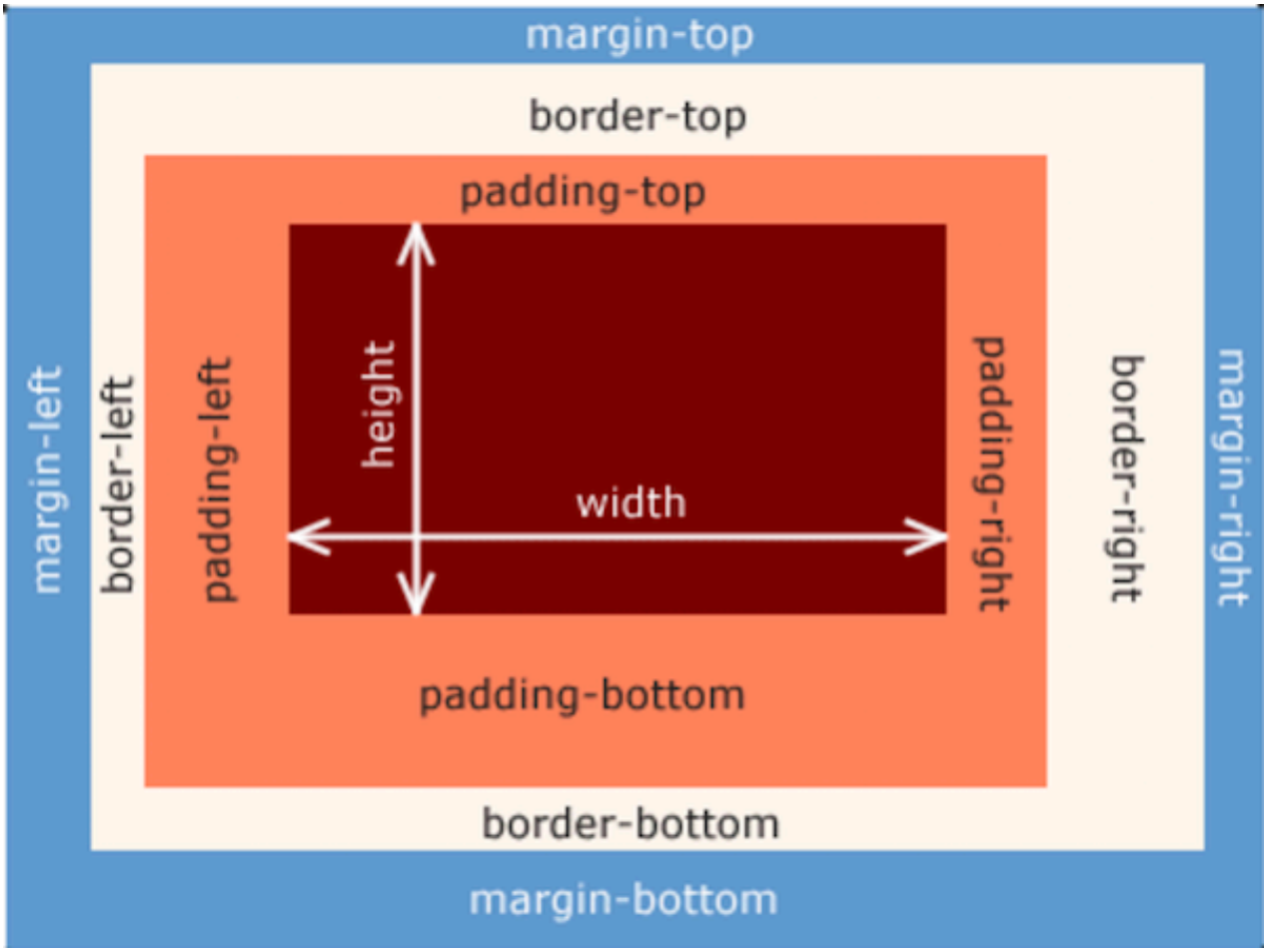
```
    margin-right: 20px;
    margin-bottom: 10px;
    margin-left: 20px;
}
```

Now that we understand this, we can think of every single element on a web page as a rectangular box.

From MDN:

In a document, each element is represented as a rectangular box. Determining the size, properties — like its color, background, borders aspect — and the position of these boxes is the goal of the rendering engine.

In CSS, each of these rectangular boxes is described using the standard box model. This model describes the content of the space taken by an element. Each box has four edges: the margin edge, border edge, padding edge, and content edge.



The **true** width/height of an element is comprised of its width/height + padding + border. **Margin is not counted when calculating the true width/height!**

Happy Coding!

```
div {
    width: 200px;
    height: 200px;
    margin: 20px;
    padding: 20px;
    border: 20px solid black;
}

/* True width =  width (200px) + padding-left(20px) + padding-right(20px) +
border-left (20px) + border-right (20px) = 280px
*/

/* True height =  width (200px) + padding-top(20px) + padding-bottom(20px) +
border-top (20px) + border-bottom (20px) = 280px
*/
```

Additional CSS properties

Let's quickly examine two more properties pertaining to the box model: `box-sizing` and `border-radius`. First, take a look at this example:

```
div {
    box-sizing: border-box;
}
```

By default, the `box-sizing` property is set to `content-box`, which means that the `width` and `height` property values correspond to the width and height of the content area only. When set to `border-box`, however, the `width` and `height` property values correspond to the width and height of content + border + padding.

Also, in CSS3 we can add rounded corners (and turn our boxes into rounded shapes) using the `border-radius` property:

```
div {
    background-color: blue;
    border-radius: 10px;
}
```

What happens as you increase/decrease the value of `border-radius`? Create a simple HTML page to explore this question!

Display.

Objectives:

By the end of this session, you should be able to:

Intell Eyes {Countinfinite Technologies Pvt Ltd}

- Explain the difference between `block`, `inline-block`, and `inline` elements
- Center elements vertically and horizontally using display

Layout

Since we have a basic understanding of the box model, let's take a stab at creating some layouts using the `display` property.

`display` is CSS's most important property for controlling layout. Every element has a default display value depending on what type of element it is. The default for most elements is usually `block` or `inline`. A block element is often called a block-level element. An inline element is always just called an inline element.

Here are four of the most commonly-used values for `display`:

- `none` - An element with a `display` property of none won't show up on the page.
- `block` - A block-level element starts on a new line and stretches out to the left and right as far as it can; that is, by default it takes up all available horizontal space. Common block-level elements are `div`, `p`, and `form`. New in HTML5 are `header`, `footer`, `section`, and more.
- `inline` - An inline element can wrap some text inside a paragraph without disrupting the flow of that paragraph. `span` is the standard inline element, but there are others too, like `strong`, `b`, and `em`. The `a` element is often the most common inline element, since they are used for links.
- `inline-block` - this value is sort of a hybrid of `block` and `inline`. To see how this works, let's look at an example.

`inline-block` vs. `inline` vs. `block`

There's one more important feature of `block` elements vs. `inline-elements` that we haven't discussed yet. To understand it, take a look at the following example in a web browser:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        /* Let's line up three 200x200 boxes together */

        /* since divs have a display:block; by default, they will stack on top
of each other like blocks */

        div {
            height: 200px;
            width: 200px;
            margin: 5px;
            background: green;
        }
```

Happy Coding!

```
        /* Let's change the display property for the last two divs; how do you
think this will affect the layout? /*

        #three, #four {
            display: inline;
        }
    </style>
</head>
<body>
    <div id="one">I AM A BOX</div>
    <div id="two">I AM A BOX</div>
    <div id="three">I AM A BOX</div>
    <div id="four">I AM A BOX</div>
</body>
</html>
```

See what this looks like

As you can see when you open up this page, the block-level elements both create new lines; were it not for the fixed width, they'd also take up all available horizontal space.

The inline elements don't create new lines: they share the same horizontal space. But you should notice something else as well: even though we're setting the `width` and `height` properties for these elements, they are only as large as the content inside of them requires! This is a general feature of `inline` elements: they don't respect the `width` and `height` property. If you set values for these properties on an inline element, those values will simply be ignored.

Knowing this, let's return to `inline-block`. Update the display on the last two divs in the above example so that their display is `inline-block` instead of `inline`. When you refresh the page, what happens? You should see that the last two divs now respect the `width` and `height` values! In fact, `inline-block` elements behave just like `inline` elements, except for the fact that you can set their `width` and `height`.

Table / Table Cell

While `block`, `inline`, `none`, and `inline-block` are four of the most common values for the `display` property, they aren't the only values. One family of values you'll sometimes see relates to table formatting. Even though tables are supported natively in HTML using `table`, `tr`, `td`, and related elements, sometimes you'll want to position elements as though they were tables without actually using these elements. For this, you can use a number of table-related `display` values (a full list can be found here.

Here's a quick example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
```

```
    <style>
        #table {
            display: table;
        }

        .row {
            display: table-row;
        }

        .cell {
            display: table-cell;
            border: 1px solid black;
            padding: 10px;
        }
    </style>
</head>
<body>
    <div id="table">
        <div class="row">
            <div class="cell">Data 1</div>
            <div class="cell">Data 2</div>
            <div class="cell">Data 3</div>
        </div>
        <div class="row">
            <div class="cell">Data 4</div>
            <div class="cell">Data 5</div>
            <div class="cell">Data 6</div>
        </div>
    </div>
</body>
</html>
```

See what this looks like

Open up this page in your browser, and you should see a table on the page, even though we didn't use the `table` element!

Vertical Align

The above example might seem a bit contrived: why use less semantic HTML to build a table? And indeed, most of the table values for the `display` property are not used very often.

One possible exception is the `table-cell` value, which can be used to vertically align an element to the middle of its container. In general, vertical alignment can be kind of a pain. But take a look at this example of an element which should be aligned to the middle of the page:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        #outer {
            display: table;
            width: 600px;
            height: 600px;
            background-color: blue;
            color: white;
        }

        #inner {
            display: table-cell;
            vertical-align: middle;
            text-align: center;
        }
    </style>
</head>
<body>
    <div id="outer">
        <div id="inner">
            WOAH I'M IN THE MIDDLE
        </div>
    </div>
</body>
</html>
```

See what this looks like

The key here is that when an element's display is set to `table-cell`, it respects a new property, called `vertical-align`, which lets you adjust the vertical alignment of the element. Try commenting out the `display` properties in the two divs above, and see how that affects the layout.

For more on table cells and vertical alignment, check out this article. And if all of this feels like a hack, you'll love learning about Flexbox at the end of this unit!

Floats.

Objectives:

By the end of this session, you should be able to:

- Use `float` to build simple layouts
- Define what document flow is and how floating changes the document flow
- Use `clear` to bring elements back into the document flow

Floats + Clearing

Intell Eyes {Countinfinite Technologies Pvt Ltd}

Another way to lay out a page, used specifically for moving elements to a certain side of the page, involves floats. Floats are used a bit less now that there are more advanced (and easier) means of laying out a page, but they are important to know about.

In order to understand floats, we first need to understand the idea of `document flow`. From here:

> The document flow is the model by which elements are rendered by default in the CSS specifications. In this model, elements are rendered according by their default display rule. In other words, block-level elements are displayed on a new line and inline elements on the same line. Everything is stacked in an ordered way from top to bottom.

When we float an element, we remove it from the document flow, let's check out this example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
        header {
            background: red;
        }

        article {
            background: green;
            width: 65%;
        }

        aside {
            float: left;
            background: yellow;
            width: 35%;
        }

        footer {
            background: brown;
        }
    </style>
</head>
<body>
    <header>
        header
    </header>
    <section>
        <aside>
            Sidebar
        </aside>
        <article>
```

Happy Coding!

```
            <div>Lorem ipsum dolor sit amet, consectetur adipisicing elit.
Deserunt enim porro praesentium in adipisci iste debitis reiciendis ipsum.
Minima harum, quisquam consequuntur nihil, error beatae ea dolorem blanditiis
molestias veniam.</div>
            <div>Quasi fuga provident neque repudiandae quo sed reprehenderit
vitae vel voluptatibus laboriosam quae eveniet quibusdam molestiae distinctio
quia saepe aspernatur, fugiat ipsum expedita repellendus, molestias harum? Sed
ex nostrum id.</div>
            <div>Fugiat neque, veritatis soluta modi voluptate hic vel
inventore, quod quaerat ad itaque enim aut sint quo earum! Magnam possimus,
autem dolorum laboriosam culpa quos, amet nostrum? At, maiores repellat.</div>
            <div>Ipsam, nulla. Laboriosam quam vero quas molestiae maiores.
Soluta distinctio officiis placeat necessitatibus fuga porro quis odio, odit
earum cupiditate natus iusto, ut alias, voluptatem amet maxime repudiandae
architecto deleniti.</div>
            <div>Dolorum debitis saepe, numquam sunt, et eum atque deleniti
dolore, culpa provident dolor nesciunt dignissimos. Corrupti quaerat dicta
aliquam voluptatum? Quaerat cumque odio, maiores, nesciunt dolores enim
suscipit earum sequi!</div>
            <div>Quos sit non deleniti fuga expedita quisquam unde asperiores.
Accusantium assumenda consequuntur ad dicta odit molestiae praesentium facere
impedit illo delectus nisi, quia asperiores dolorum necessitatibus saepe a
deserunt vero!</div>
        </article>
    </section>
    <footer>
        Footer
    </footer>
</body>
</html>
```

[See what this looks like]

Open this page up in your browser, and you should see that the sidebar is literally floated to the left. While the content in the `article` respects the sidebar, the `article` itself does not; instead, it looks as though the side bar is sitting right on top of the `article`.

Sometimes this is what you want, but sometimes you might want an element adjacent to a float to move down to the next line. In other words, you might want the element to be **cleared** below the float. To see what this looks like, try setting `clear: both;` on the `article`. How does this change the layout?

Next, let's make a two-column layout by floating the `article`. The `aside` is floated to the left, so let's remove the `clear` property on the `article` and instead set `float: right;` on it.

When you do this, you'll see that you've got two columns, but there's a problem: the `aside` is much shorter than the `article`, because it has less content! Consequently, the yellow background on the left is much shorter than the green background on the right. How can we fix this? If you know the exact height you need the `aside` to be you could set the `height` property, but very often you don't know what the height needs to be, as it may depend on the screen size.

Happy Coding!

In this case, there are a couple of workarounds. Here's the approach we'll use:

1. Set the `padding-bottom` to `1000px` (or some other large number) on the `aside`. This makes the `aside` tall enough: in fact, now it's too tall!
2. Set the `margin-bottom` to `-1000px` (or some other large number) on the `aside`.
3. Set the `overflow` property to `hidden` on the `section`.

(These last two steps ensure that the footer is in the right place, and that the extra height from the `aside` gets hidden.)

If you think this feels like a bit of a hack, you're not alone. With the advent of Flexbox (which will get to very soon), some people argue that floats should be avoided. However, they're still quite common, so it's important to be familiar with them and know how to deal with some of their quirks.

Positioning.

Objectives:

By the end of this session, you should be able to:

- Compare and contrast `static`, `relative`, `absolute`, and `fixed` positioning
- List the properties that positioning gives an element
- Use positioning to build more complex layouts

Positioning

So far in this unit we've examined the `display` and `float` properties, and looked at some examples of how to use these properties to lay out a page. In this chapter, we'll explore another important property for layouts: `position`.

Positioning allows you to take an element on the page and control where and how it's positioned relative to things such as its original starting position, other elements, or even the window itself. Depending on the type of positioning you use, the element will either remain in the document flow (relative) or be removed from the document flow (absolute, fixed), just like with floats.

Once `position` is set, offset values can be set for `top`, `right`, `bottom`, `left`, and `z-index` (a 3D axis that we can use to stack elements on top of each other).

By default, the position of an element is set to `static`. `static` positioning and `relative` positioning are basically the same, but with one important difference: a statically positioned element won't respond to the offset properties listed above. If you set `top: 10px` on an element that is statically positioned, this style rule will simply be ignored. This is analogous to how `inline` elements don't respond to `height` or `width`; elements that are positioned statically (which is the default positioning for elements) will not respond to `top`, `left`, `right`, `bottom`, or `z-index`.

Intell Eyes {Countinfinite Technologies Pvt Ltd}

Here's a quick example. The HTML page below has a single `div`, which is relatively positioned and has `top` and `left` values set. See what happens when you remove the `position: relative;` line. With this line removed, the `div` will revert to its default positioning (`static`), and it should ignore the positioning set by `top` and `left`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <style>
    div {
      width: 300px;
      height: 300px;
      background: orange;
      position: relative;
      /* move divs 100 pixels from the top, relative to where they'd normally
sit */
      top: 100px;
      /* move divs 200 pixels from the left, relative to where they'd normally
sit */
      left: 200px;
    }
  </style>
</head>
<body>
  <div>hello.</div>
</body>
</html>
```

See what this looks like

Absolute Positioning vs. Relative Positioning

With `relative` positioning, elements are not removed from the document flow, and any offsets you place on the element will be *relative* to its default position in the document flow. In the above example, for instance, `top` is set to `100px`; this means that the `div` will be 100 pixels below where it would otherwise be (in other words, you're offsetting the `div` 100 pixels from the top).

With `absolute` positioning, the situation is a little different. In this case, the element is removed from the document flow, and any offsets you place on the element will be relative to its parent, *provided its parent is not statically positioned!* If the parent element *is* statically positioned, then the offsets will be relative to the grandparent, provided the grandparent is not statically positioned. If the grandparent is statically positioned, we keep going up the chain until we find an element that is not statically positioned. If no such element exists, the offsets are relative to the `body`.

Here is an example with relative and absolute positioning:

Happy Coding!

Intell Eyes {Countinfinite Technologies Pvt Ltd}

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <style>
  body {
    margin: 0; /*REMOVE BROWSER MARGIN*/
  }

  .red {
    background-color: red;
    position:  absolute;
    width: 200px;
    height: 200px;
    bottom: 0;
    right: 0;
  }
  .wrapper {
    width: 500px;
    height: 400px;
    background: #e3e3e3;
    position: relative;
    margin-bottom: 10px;
  }
  </style>
</head>
<body>
  <div class="wrapper">
    <div class="red"></div>
  </div>
</body>
</html>
```

See what this looks like

When you open up this page, you should see that the red `div` is in the bottom-right corner of its parent. This makes sense: the parent has relative positioning, and the red div is set to be offset `0` pixels from the bottom and the right.

Now, what happens if you remove the `position: relative` rule from the `wrapper` class? In this case, the parent of the red `div` will have static positioning, which means that the position of the red `div` will be relative to the body! Refresh the page and you should see that the `red` div has moved to the bottom-right corner of the `body`, not of its parent!

Fixed positioning

Happy Coding!

The fourth type of positioning is `fixed`. This behaves similar to absolute positioning, but elements with fixed positioning are **ALWAYS** positioned relative to the active viewport. Now, what's that mean? If you position, for example, a fixed element with a top offset of 50 pixels and a right offset of 100 pixels, the top right corner of the element will be positioned 50 pixels over to the left and 100 pixels down. Unlike with absolute positioning, scrolling the page content would not affect this element's position at all. It will remain in that position, no matter how the screen was resized or scrolled. It truly is fixed.

Let's modify our previous example to see what fixed positioning looks like. We'll need to add some extra `div`s to the page so that it's long enough to scroll through. Let's also play around with the `z-index` a bit:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <style>
  body {
    margin: 0; /*REMOVE BROWSER MARGIN*/
  }
  .red {
    background-color: red;
    position: fixed;
    width: 200px;
    height: 200px;
    bottom: 0;
    left: 0;
    z-index: 1;
  }
  .wrapper {
    width: 500px;
    height: 400px;
    background: #e3e3e3;
    position: relative;
    margin-bottom: 10px;
  }
  #front {
    z-index: 2;
  }
  </style>
</head>
<body>
  <div class="wrapper">
    <div class="red"></div>
  </div>
  <div class="wrapper"></div>
  <div class="wrapper"></div>
  <div class="wrapper"></div>
```

```
  <div class="wrapper" id="front"></div>
  <div class="wrapper"></div>
</body>
</html>
```

See what this looks like

In this example, notice that the red `div` is *always* in the bottom-left corner of the screen, no matter where on the page you scroll. If you change the style so that the red `div` has absolute positioning, you'll see the difference.

Also note that with fixed positioning, the red `div` is in front of all the wrapper `div`s except for the one with an id of `front`. This is because of the `z-index`! The larger the `z-index`, the higher the element should be stacked. If two divs are positioned on top of one another, the one with the lower `z-index` will appear to be behind the one with the higher `z-index`.

Flexbox.

Objectives:

By the end of this chapter, you should be able to:

- Understand what flexbox is and how it differs from other forms of layout
- Use flexbox to build more complex layouts

Flexbox

We've mentioned flexbox a few times already in this unit. But what, exactly, is flexbox? Let's turn to the MDN documentation for an overview:

> The CSS3 Flexible Box, or flexbox, is a layout mode providing for the arrangement of elements on a page such that the elements behave predictably when the page layout must accommodate different screen sizes and different display devices. For many applications, the flexible box model provides an improvement over the block model in that it does not use floats, nor do the flex container's margins collapse with the margins of its contents.
>
> Many designers will find the flexbox model easier to use. Child elements in a flexbox can be laid out in any direction and can have flexible dimensions to adapt to the display space. Positioning child elements is thus much easier, and complex layouts can be achieved more simply and with cleaner code, as the display order of the elements is independent of their order in the source code. This independence intentionally affects only the visual rendering, leaving speech order and navigation based on the source order.

You can read more about flexbox here, here and here. For now, let's look at a simple example. Here's a fairly standard HTML file, with a few `div`s and some content. If you open this up in the browser (ignoring the stylesheet for now), you shouldn't see any surprises.

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
  <title>Document</title>
</head>
<body>
  <div class="container">
    <div class="first">
      First
    </div>
    <div class="second">
      Second
    </div>
    <div class="third">
      Third
    </div>
    <div class="fourth">
      Fourth
    </div>
    <div class="fifth">
      Fifth
    </div>
    <div class="sixth">
      Sixth
    </div>
  </div>
</body>
</html>
```

Now let's style this page using flexbox. Put the following code inside of a file named `style.css`. You'll probably notice quite a few new properties and values in this code. Don't worry, you'll learn about these momentarily!

```
div {
  width: 100px;
  height: 100px;
  color:white;
}

.container {
  display: flex;
  /*FLEX DIRECTION - can reverse etc*/
  flex-direction: column;
  background: #F60009;
  height: 500px;
  width: 500px;
}

.first {
```

```
  display: flex;
  background: #53007A;
  /*VERTICAL*/
  align-items: center;
  /*HORIZONTAL*/
  justify-content: space-around;
  /*ORDER  -1 first, highest last */
  order: 1;
}

.second {
  background: #FC9709;

}

.third {
  background: #A800DF;
  align-self: flex-end;
}

.fourth {
  background: #8BE7E6;
}

.fifth {
  background: #93B112;
}

.sixth {
  background: #E6F0BA;
  order: -1;
}
```

See what this looks like

Here's a quick rundown of some of the new things in this code (along with some other common flexbox properties and values):

`display` - We've seen this property before. It turns out that this property is of fundamental importance when you want to use flexbox. To start using flexbox, you need to set the `display` property of an element either to `flex` or `inline-flex` to make your elements inline. (The distinction here is similar to the distinction between `block` and `inline-block`; check out this Stack Overflow question for more details.)

`flex-direction` - with the `row` value this will layout items in a row, and with the `column` value it will lay elements out in a column. The default value is `row`. These values can also have a `-reverse` added to them (`row-reverse` / `column-reverse`) to reverse the ordering.

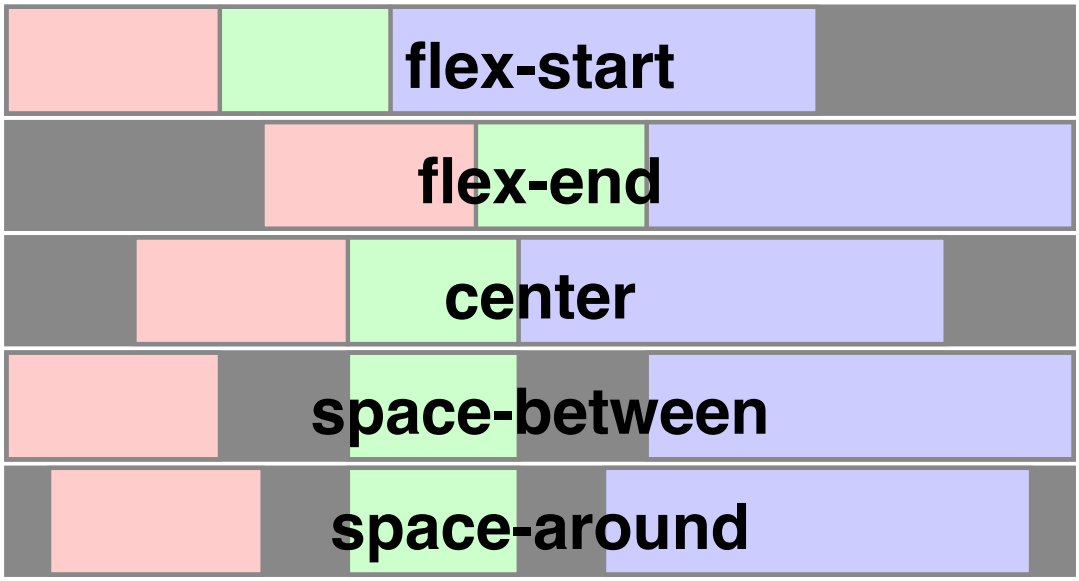
justify-content - this affects the space along the flex-direction. You can think of this as the horizontal alignment:

Happy Coding!

Intell Eyes {Countinfinite Technologies Pvt Ltd}

- left -> flex-start
- center -> center
- right -> flex-end
- even amount of space between elements -> space-between
- even amount of space around elements -> space-around

This image, taken from an CSS-Tricks article on `justify-content`, does a good job illustrating the difference between the values listed above:



Note that when the `flex-direction` is `column` or `column-reverse`, `justify-content` refers to the vertical axis, not the horizontal.

`align-items` - this affects the space perpendicular to `flex-direction`. You can think of this as the vertical alignment: - top -> `flex-start` - center -> `center` - bottom -> `flex-end`

(There are a few other possible values, check the docs if you're curious.)

Note that when the `flex-direction` is `column` or `column-reverse`, `align-items` refers to the horizontal axis, not the vertical.

`order` - this property allows you to order items based on their position in the DOM. The order will start at 0, so anything lower will go higher and anything higher will go lower in the order. See the HTML and CSS above for some examples of setting this property, and how it affects the layout of the page.

`align-self` - this property allows you to overwrite the value of `align-items` for particular elements. It accepts the same values as `align-items`.

`flex-basis` - This lets you manually set the size of a flexbox element's content box. At first glance it may seem like it's synonymous with the `width` property, but this isn't always the case. For more details, read this Stack Overflow question.

`flex-grow` - This lets you set proportional sizes of flex items within the same container. For example, an element with a `flex-grow` value of 2 will take up twice as much space as an element with a `flex-grow` value of 1.

Happy Coding!

`flex-shrink` - Similar to `flex-grow`, but for shrinking elements when there's not enough space. A useful demo of this property is available at CSS Tricks.

`flex` - This is a shorthand which lets you set `flex-grow`, `flex-shrink`, and `flex-basis` all in one line (similar to the `border` property, which lets you set `border-width`, `border-style`, and `border-color`).

`flex-wrap` - When items in a flex container get too wide, what should happen? This property lets you tell the browser whether it should allow flex elements to wrap onto a new line, or whether they should just shrink in order to fit on the same line. Fore more on this, check out MDN.

There are some great tools available if you're interested in learning more about flexbox. Check out the exercises for more!