**CHALLENGE**

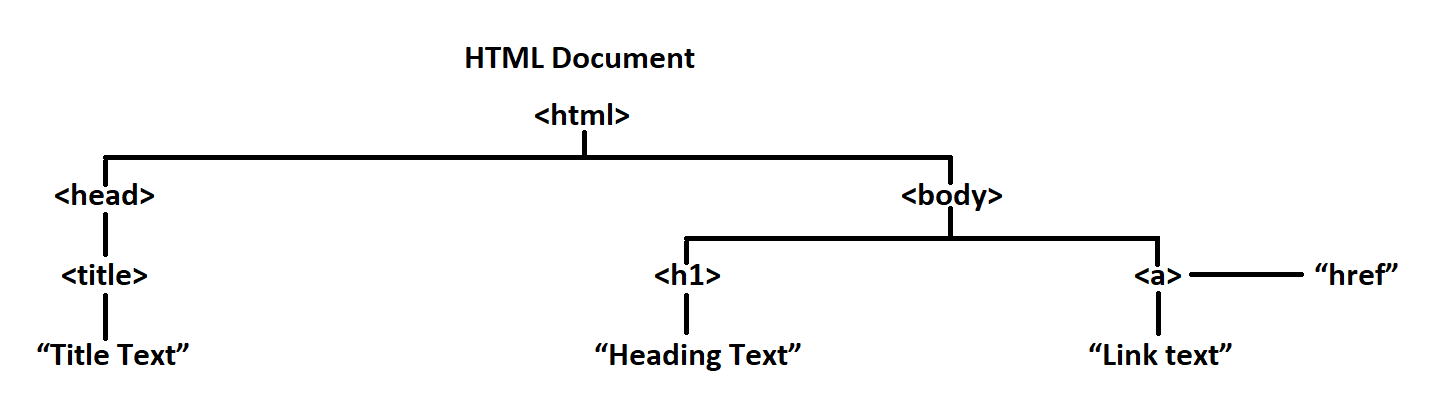
**NODES AND THE DOM**

In the DOM, every item within an HTML document is a node.

This means: every element is a node (an element node), text inside elements are nodes (text nodes) and comments are nodes (comment nodes). Even the document itself is considered a node (document node).

These nodes have object-oriented relationships (i.e. parent and child).

In this DOM tree:

****

In this tree, the <html> element is the parent node.

The <head> and <body> elements are the first child nodes of the <html> node and the <title>, <h1> and <a> elements are the last child nodes.

Additionally, the following is true as well:

* The <title> element is the child node of the <head> node
* The “Title Text” is a text node, and the child of the <title> node
* The <body> element has two child nodes: <h1> and <a>

And so on.

The childNodes[indexnumber] property can be used to navigate child nodes. For example: childNodes[0] could be used to access the <h1> element when used in relation to the <body> element

**DOM VS HTML**

Remember, you can use Google Dev Tools to view the DOM:

[**https://developers.google.com/web/tools/chrome-devtools/dom/**](https://developers.google.com/web/tools/chrome-devtools/dom/)

The DOM looks very similar to your HTML code. But your HTML file and the DOM are different things.

Here are some of the differences between HTML and the DOM:

1. The DOM tree is built as a result of your HTML code being parsed by the browser. It represents your HTML elements as objects – thereby making the code a sort of object-oriented language which HTML inherently isn’t (it’s a markup language).

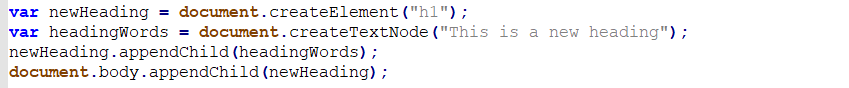
2. The DOM is used by browsers as a first step towards determining what to render in the browser window.

3. The DOM is an abstraction of the HTML code. As a reminder, “abstraction” refers to the hiding away of the implementation details, and only providing the description of the behavior to be performed.

4. Sometimes browsers correct errors in HTML code when building DOM trees. For example:

If you accidentally leave out the <body> element in your code, the browser may add this to the DOM to ensure the code is valid (note: this change would only be included in the DOM, not your actual code).

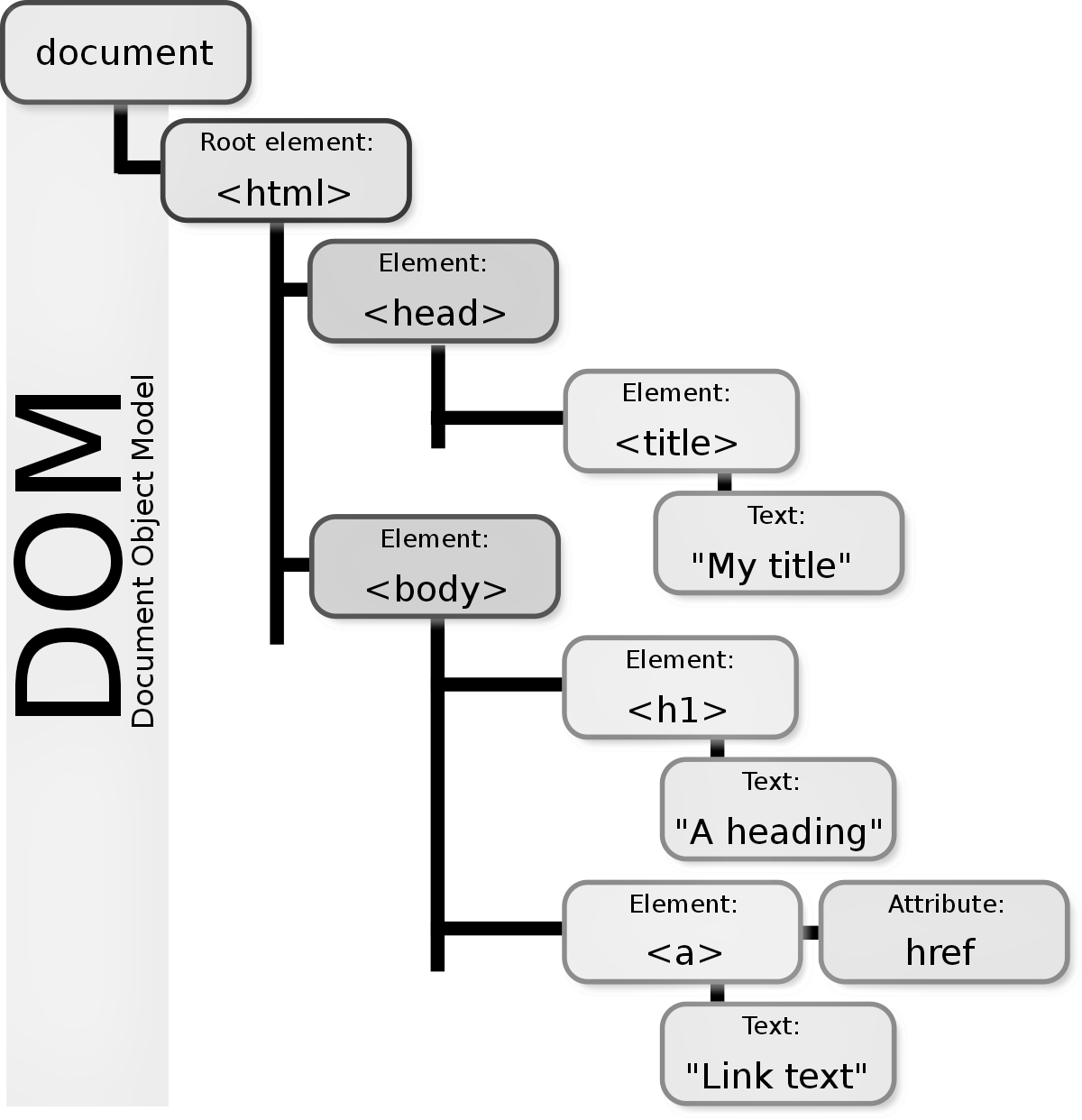
5. The DOM can be modified. For example: nodes (individual parts like elements or text) can be added to the DOM using JavaScript. For example:

****

These updates to the DOM would not change our HTML file. JavaScript can be used to modify the content, structure, or styling of the DOM.

**HTML OBJECTS**As we’ve covered, the DOM is the Document Object Model. Here is a picture we’ve shown a few times during your training thus far:

As we’ve covered, the DOM is the Document Object Model. Here is a picture we’ve shown a few times during your training thus far:

****

As far as the DOM is concerned, all of the elements in this image are objects. All elements are objects.

They are objects because properties and methods can be used with them.

Here is a great resource that lists out all of the properties/methods that can be used on HTML elements:

[**https://www.w3schools.com/jsref/dom\_obj\_document.asp**](https://www.w3schools.com/jsref/dom_obj_document.asp)

Using HTML and CSS, perform the following actions with an image:

1. Center the image. Successfully run your code in the browser.

HELPFUL LINK:

[**https://www.w3schools.com/howto/howto\_css\_image\_center.asp**](https://www.w3schools.com/howto/howto_css_image_center.asp)

2. Apply a CSS filter to the image. Successfully run your code in the browser.

HELPFUL LINK:

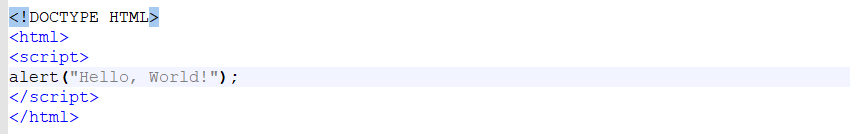
[**https://www.w3schools.com/cssref/css3\_pr\_filter.asp**](https://www.w3schools.com/cssref/css3_pr_filter.asp)

3. Give the image rounded edges. Successfully run your code in the browser.

HELPFUL LINK:

[**https://www.w3schools.com/cssref/css3\_pr\_border-radius.asp**](https://www.w3schools.com/cssref/css3_pr_border-radius.asp)

:



https://www.typingclub.com/<!DOCTYPE HYML

<!DOCTYPE HTML>

<!DOCTYPE HTML>

<html>

<script>

alert(“Mmmm pretzel bites”);

</script>

</html>

--------------------------------------------------------------

<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Alert</h2>

<button onclick="myFunction()">Try it</button>

<script>

function myFunction() {

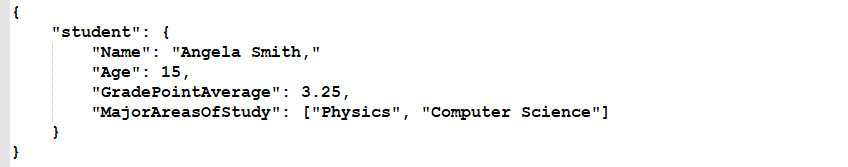
alert("I am an alert box!");

}

</script>

</body>

</html>

  
<!DOCTYPE HTML>

<html>

<body>

<script>

alert("Mmmm pretzel bites");

</script>

</body>

</html>

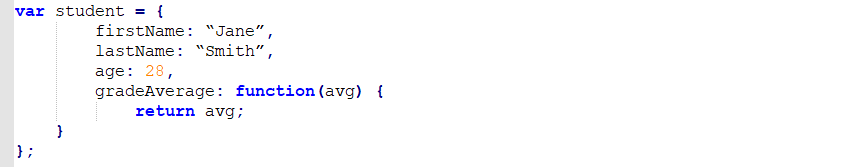


An example could look like this:



Other JavaScript code elements could call this “add” function by specifying its name and passing it two numbers. That could look like this:

https://techacademystorage.blob.core.windows.net/htmlandcss/function2.png



Here, we aren’t setting the property “gradeAverage” to a fixed number of 3.5. Instead, we are setting that property to the value returned by a set of code. Specifically, that code will take in a number (the variable “avg”) and set the value of the “gradeAverage” property to the value of that variable.

Executing that code could look like this:

https://techacademystorage.blob.core.windows.net/htmlandcss/method3.PNG

An “attribute” is a specification that assigns a property (name; characteristic) to something. Basically, an attribute is metadata (data that describes other data).

The element returned by document.getElementById has an Id attribute with a specific value assigned to it. The Id is a variable.

For an example:

https://techacademystorage.blob.core.windows.net/htmlandcss/h1.png

We have assigned the h1 element the Id “Header\_1”. To run and display (return) this element, we write:

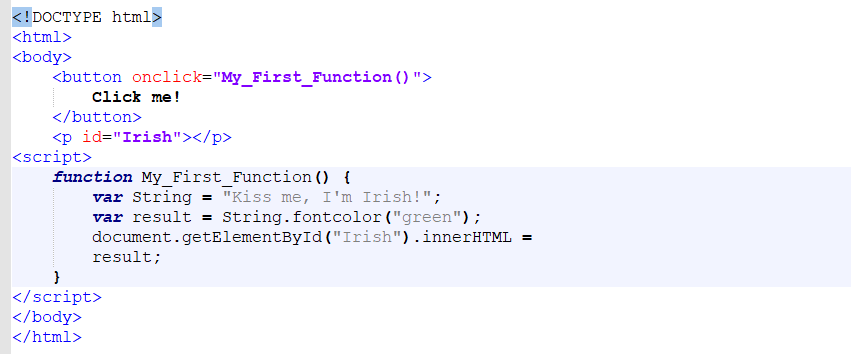
https://techacademystorage.blob.core.windows.net/htmlandcss/get_element.png

“innerHTML” gets or sets the HTML markup contained within the element.

Document.getElementById is used mainly to control or get information from an element within your code. If it can’t find the element with the specified Id, it will return “null.”

**CREATING A FUNCTION**

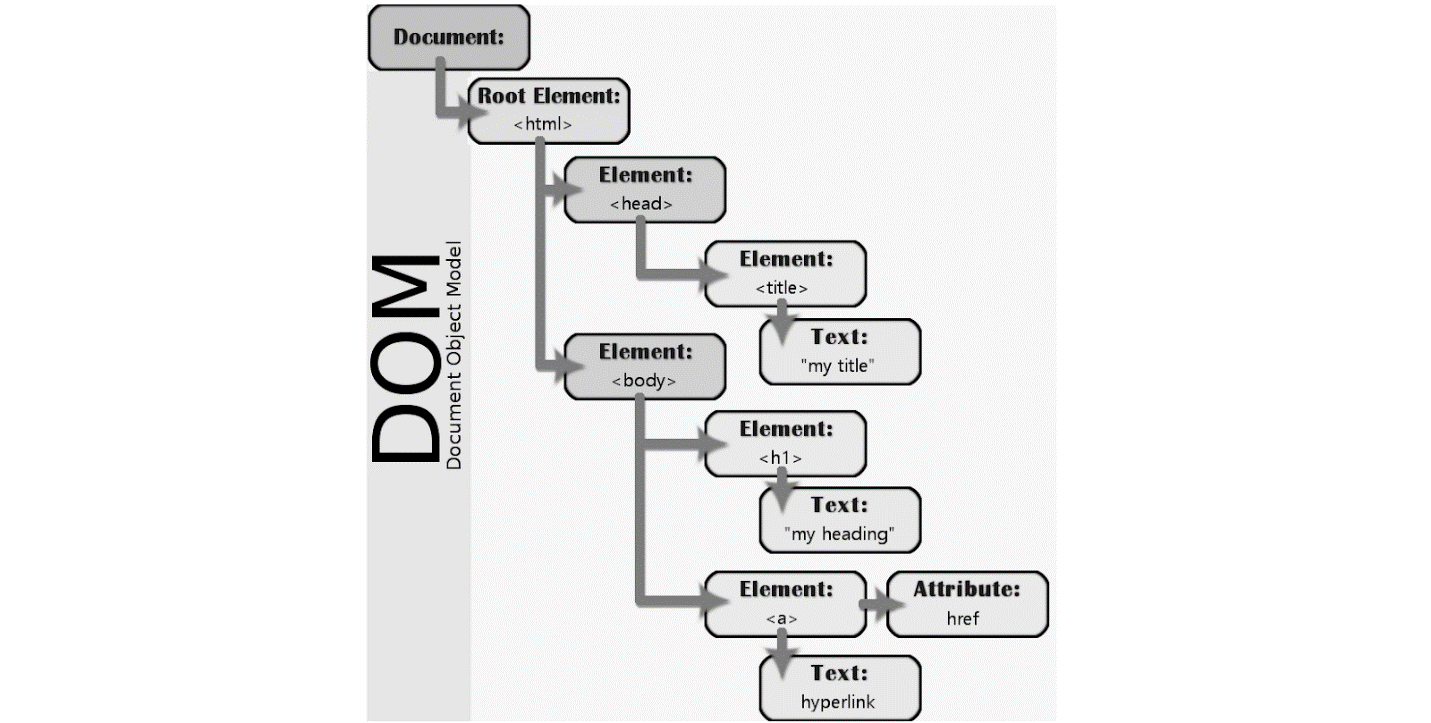
Let’s look at creating a function using the document.getElementById method:



The above code accomplished the following:

* Created a button element.
* Utilized the ID attribute and assigned the button element the value “Irish”.
* Returned the button element by calling the “Irish” value that we assigned earlier (when writing the ID attribute).

Everything inside our curly brackets { } is our function.



Suppose you wanted to find all paragraph elements in an HTML document. You might use code like this:

https://techacademystorage.blob.core.windows.net/htmlandcss/get_element2.png

The variable called “foo” would now contain a collection of all the paragraph tags in the document.

Many different programming languages can be used to work with the DOM. One of the most common is JavaScript; in fact, the code example above is actually JavaScript.

olor like:

https://techacademystorage.blob.core.windows.net/htmlandcss/color.PNG

2. Specify the color using its hex code (a way of specifying color using hexadecimal values). For example:

https://techacademystorage.blob.core.windows.net/htmlandcss/color2.PNG

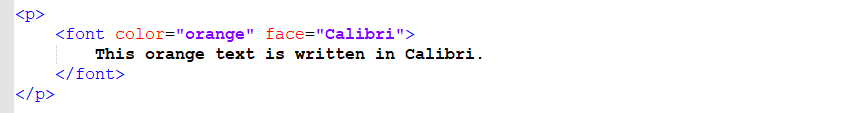
[**Here is a useful chart to find the hex code for various colors.**](https://www.rapidtables.com/web/color/html-color-codes.html)

For example: red would be written as

https://techacademystorage.blob.core.windows.net/htmlandcss/color3.PNG

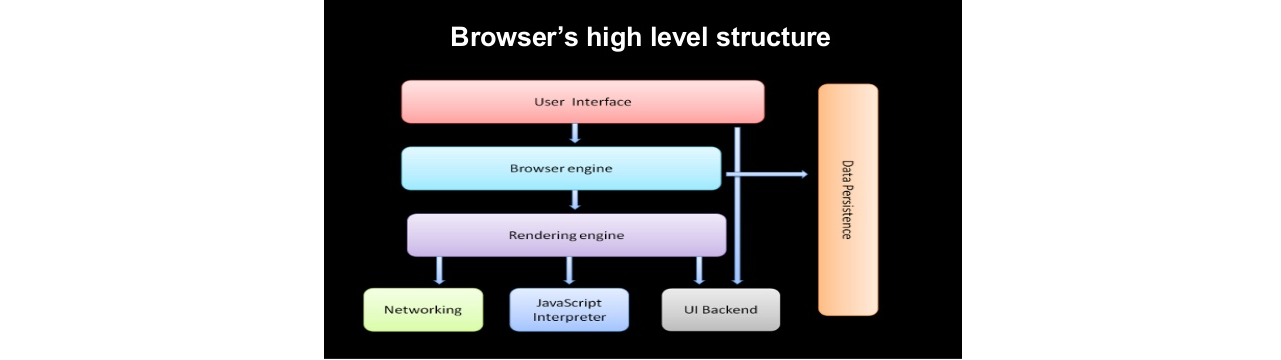
To assign a font color to an element (such as a paragraph) you write your code as follows:

https://techacademystorage.blob.core.windows.net/htmlandcss/color4.png



https://techacademystorage.blob.core.windows.net/htmlandcss/italics2.png

There are seven main components to any web browser:



1. Layout Engine: This takes input from the browser (URL bar, search box, mouse clicks, and keyboard input) and passes them to the rendering engine.

2. Rendering Engine: “Render” literally means “to cause to be; make”. A rendering engine takes HTML code and interprets it into what you see visually. For example: a <bold> tag would be interpreted by the rendering engine as a set of instructions to make the text inside the element bold.

3. User Interface: This is the visual presentation of controls in the browser, for instance the back and forward buttons.

4. JavaScript Engine: This engine takes JavaScript code, parses (analyzes and reads the code) it, executes it, and returns the results.

5. Network Layer: This is a function of the browser that happens behind the scenes and handles network functions such as encryption (concealing data by altering it into a secure format), requests, and all network settings such as HTTP.

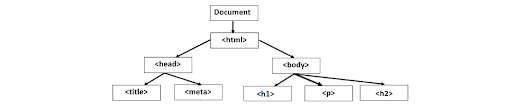
6. Storage: Browser’s must store some data which can include cached files and cookies.

7. Operating System Interface: The browser interacts with the operating system to display several elements of the page, like drop down boxes and some icons on the window (close, maximize, and minimize buttons).

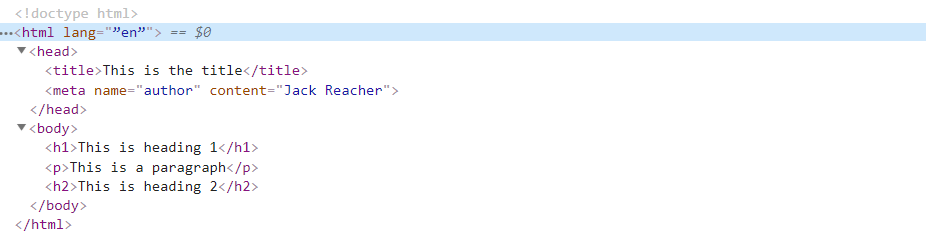
**RENDERING**

The rendering engine displays what you see on your screen. It receives HTML code and other items passed from a server and then creates the Document Object Model (DOM).

The rendering engine creates a DOM tree like this from the code received:



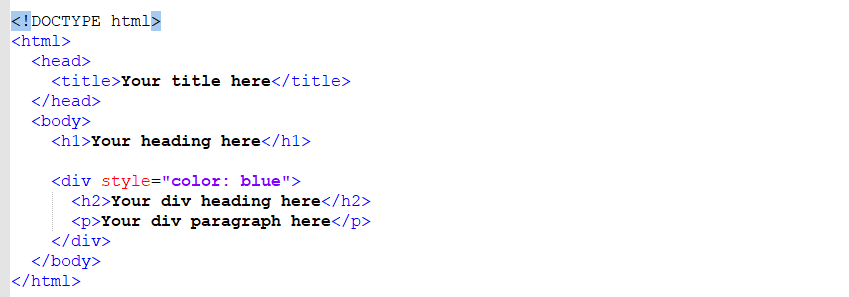
Though, technically, the DOM (render tree) would look like this:



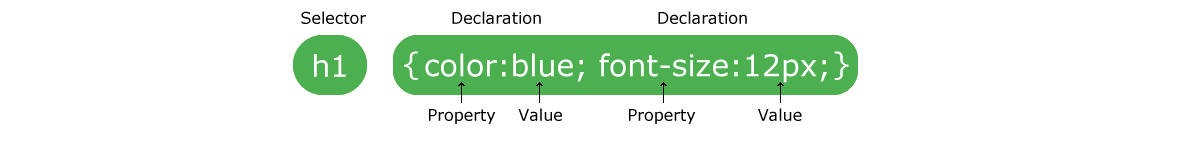
**DIV**

The <div> tags define a division or a section in an HTML document. The div element is often used as a container for other HTML elements to style them or make specific changes to the content within that particular div tag.

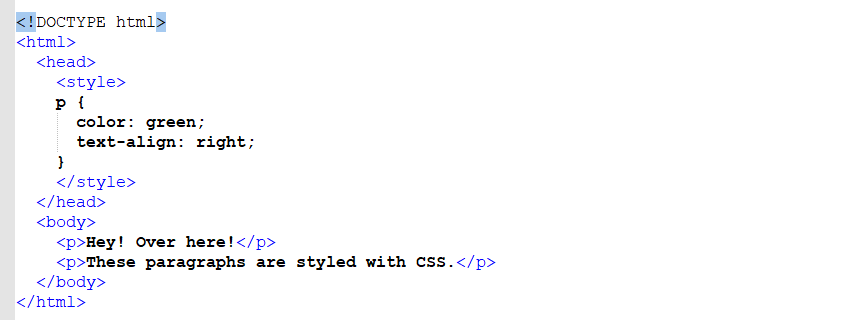
EXAMPLE:



All of the elements contained within the <div> element will be written in blue.



Review the following code:



The output of this code would be:

https://techacademystorage.blob.core.windows.net/htmlandcss/review2.PNG

**EXTERNAL STYLE SHEETS**

As we mentioned earlier, CSS can be saved in a different file than one’s HTML code. This is actually recommended. We do this in a similar way to how we did it with JavaScript:

https://techacademystorage.blob.core.windows.net/htmlandcss/stylesheet2.png

In HTML, there’s a <link> element that defines a link between a document and an external resource. It is common to use <link> tags to link to a style sheet.

“Rel” is short for “relationship”. Rel is an attribute (word used inside an element’s opening tag that controls its behavior) that specifies the relationship between two documents. For example: if you wanted to specify that you were linking to the help page on your website, you’d write:

https://techacademystorage.blob.core.windows.net/htmlandcss/rel.PNG

“Stylesheet” is a common value used following the rel attribute. The stylesheet value imports the specified stylesheet

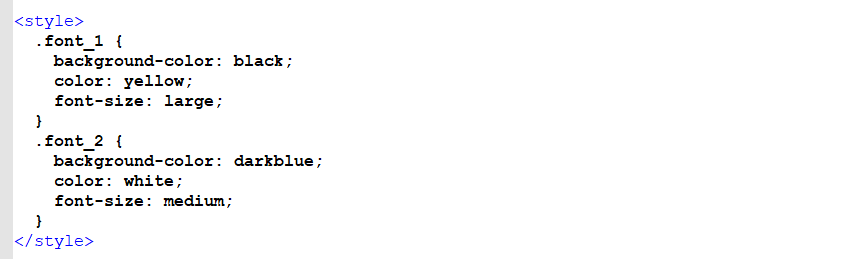
**CLASS ATTRIBUTE**

The HTML class attribute is used to define equal styles for all elements with the same class name. HTML elements with the same class attribute have the same format and style.

For example: we could utilize the class attribute if we wanted to have two different styles of paragraphs.

You assign a class name by writing .classname (a period followed by a class name of your choosing).

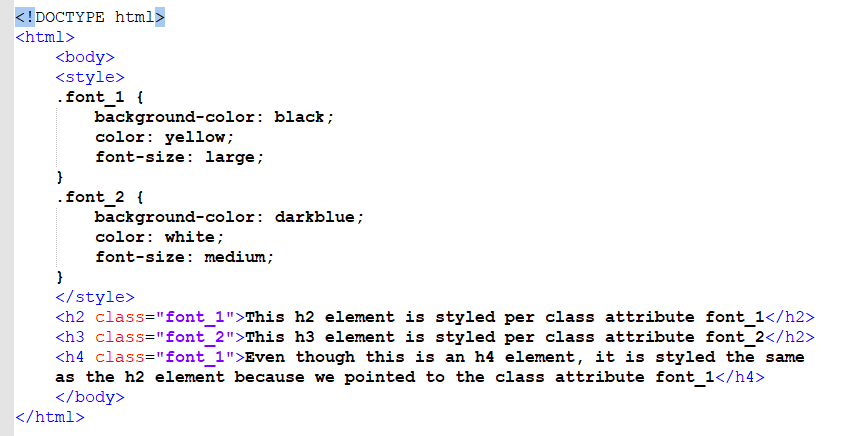
Here is an example of how to create two styles using the class attribute:



We created two class names: font\_1 and font\_2. The class attribute specified the class names.

By pointing to the class name “font\_1” or “font\_2” we can specify the styling of our font.

This is how:



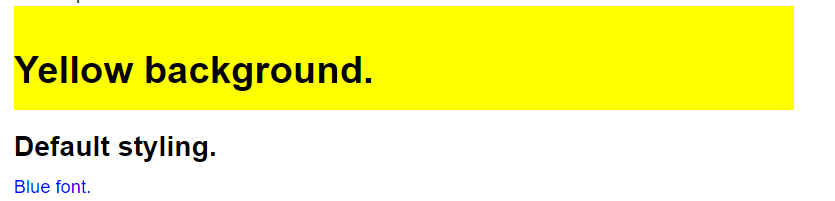
**ASSIGNMENT**

Complete the following:

1. Open the Basic\_HTML\_3.html file and add a line of text (e.g. a paragraph or heading).

2. Using CSS, set the color and alignment of the text.

Save your code and successfully run it in the browser.



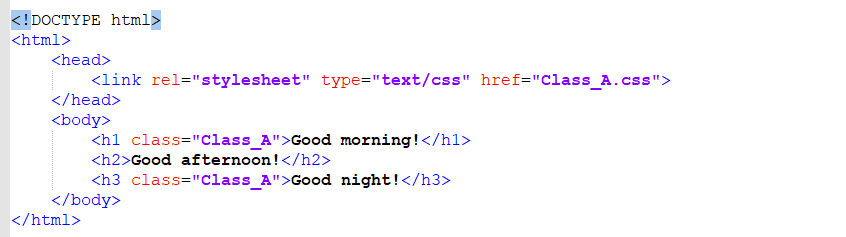
**CLASS IN EXTERNAL FILE**

Let’s try keeping our class in a separate file. To do this, we would first write a new css file as follows:

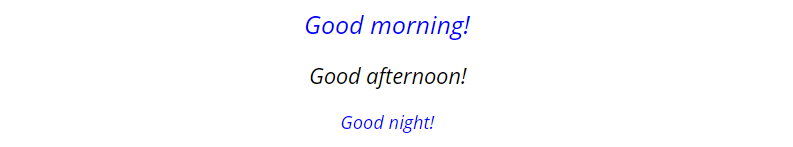


This code states that anything within the <body> element will be written in italics and centered on the page. We’ve also created are our class name (Class\_A) and stated that any element connected to that class name will have its text written in blue. Let’s say we saved the file as Class\_A.css.

Here is what the separate HTML file would look like:



The output of this code would look like this:



Notice that “Good afternoon!” is written in default black (not blue) because we didn’t utilize the class name with it

**ASSIGNMENT**

Complete the following actions:

1. Create a new CSS document and write your own class attribute that causes all <p> elements to have the following properties (you can set the values as you choose):

a. Color

b. Background color

c. Font-size

d. Text-align

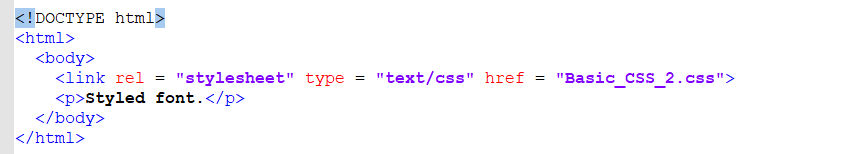
For example:



Save this file as Basic\_CSS\_2.css within your Basic\_HTML\_and\_CSS folder.

2. Create a new HTML document that minimally includes a <p> element. Save this file as Basic\_HTML\_5.html in your Basic\_HTML\_and\_CSS folder.

For example:



3. Successfully run the Basic\_HTML\_5.html file in the browser.

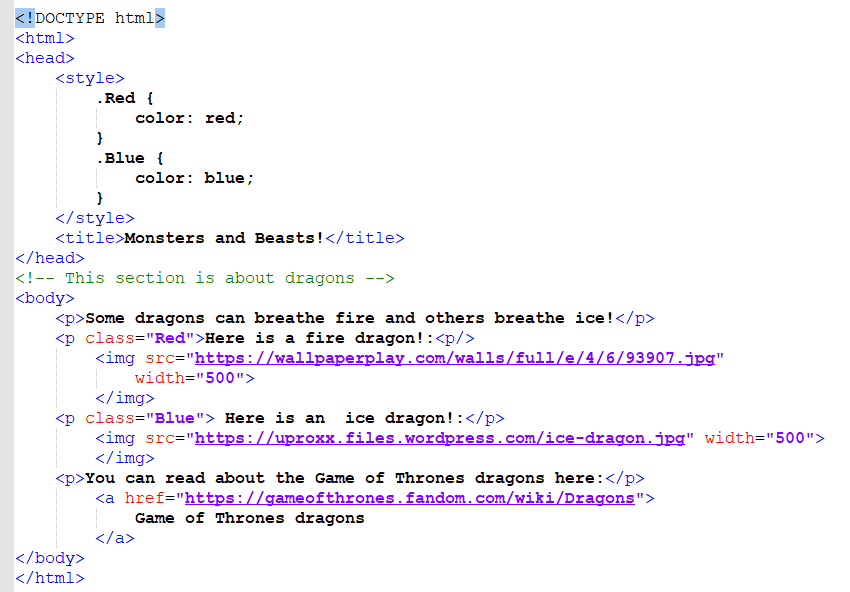
**COMMENTING CODE**

It is highly recommended that you write comments throughout your code that explains what each section of it is meant to do. This enhances the ability of other developers to read through and understand your code, and can also remind you of things in the future.

An HTML comment begins with <!– – and closes with – –>. HTML comments are visible to anyone that views the page source code, but are not rendered when the HTML document is rendered by a browser.

Meaning, you don’t see the comments when viewing the website in the browser but you can see them when you utilize Chrome’s dev tools.

An example of commenting HTML code is as follows:



The output of this code would be:

**HTML TABLES**

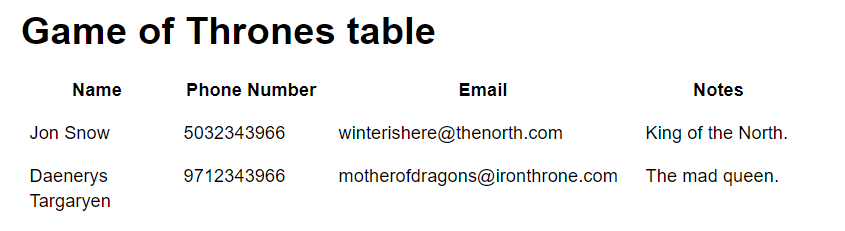
To create a table in HTML, we use the <table> tag

Rows are created with the <tr> tag (table row).

Table headers are created with the <th> tag. Table headers are bold and centered by default.

Table data (cells) are created with the <td> tag.

For example, look at this table:



To create this in HTML, we would write the following code:

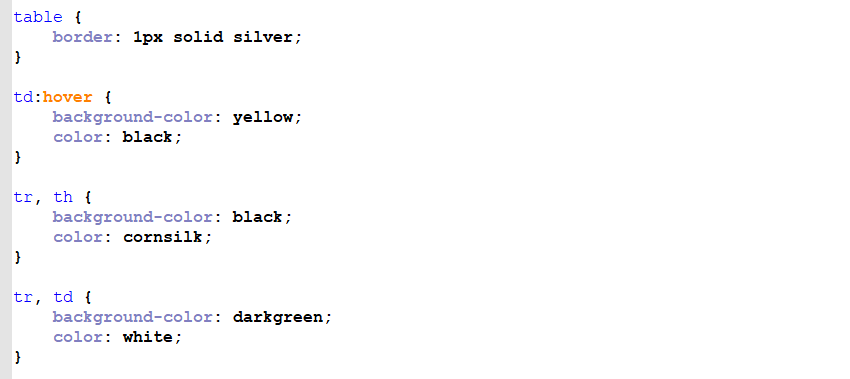


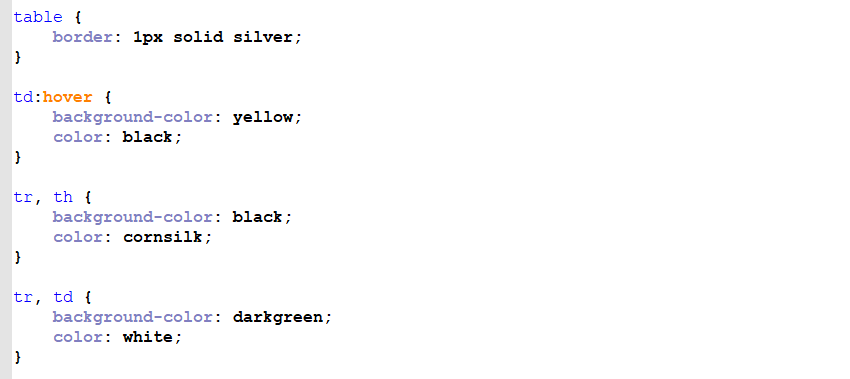
To create an ordered list, we use the <ol> tag. Each item on the list is contained within the list element <li>. In the earlier example, this would be written as:



To create an underordered list, we use the <ul> tag. The earlier example would be written as



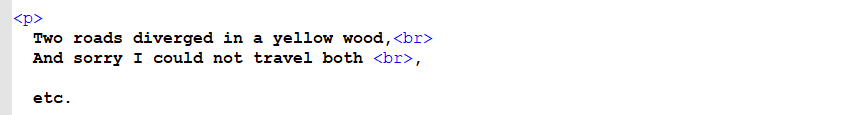




https://techacademystorage.blob.core.windows.net/htmlandcss/styling7.png

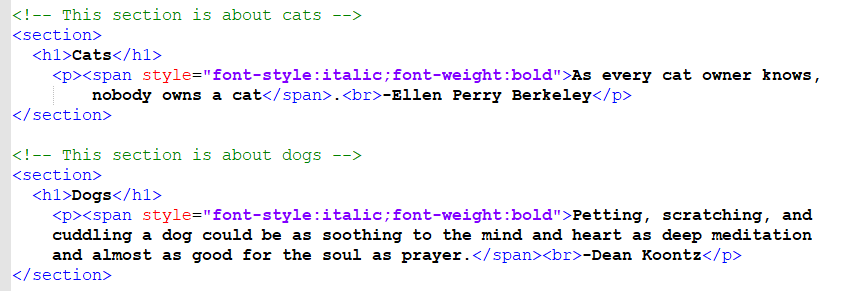


Now, that code is not very orderly, so for appearance’s sake, we’d write it more like:



As the name sounds, the <section> tag designates sections in a document, such as headers, footers, chapters, or any other sections of the document.

Here’s an example:



The output of this code looks like this:

**NON-BREAKING SPACE**

A common character entity used in HTML is the non-breaking space – written like this: &nbsp;

A non-breaking space is a space that will not break into a new line. Two words separated by a non-breaking space will stick together. This can be useful when breaking the words might be disruptive.

For example:

https://techacademystorage.blob.core.windows.net/htmlandcss/space.PNG

would ensure that the output is:

3:11 p.m.

As opposed to:

3:11

p.m. (or something)

Additionally, browsers automatically default multiple spaces down to 1. For example: if you write 12 spaces, the browser will delete 11 of them.

You can try it out for yourself but here’s an example:

https://techacademystorage.blob.core.windows.net/htmlandcss/space2.PNG

will display as:

Here’s Johnny !

&nbsp; can tell the browser to leave extra spaces in place.

If we wanted to start every sentence with two spaces, we would write:

https://techacademystorage.blob.core.windows.net/htmlandcss/space3.PNG

<!DOCTYPE html>

<html lang=”en”>

<body><body>,<body><body><body><body><body><body><body><body><body><body><body><body>

<html land=”en”><html lang=”en”><html lang=”en”><html lang=”en”><html lang=”en”><html lang=”en”><html lang=”en”>

<h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1><h1>

</html></</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</</

<h1><h1>

<html><html><html><html>

<!DOCTYPE><!DOCTYPE html><!DOCTYPE html>

<!DOCTYPE html>

<!DOCTYPE html>

<!DOCTYPE html>

</body></body></body></body></body></body></body></body></body></body></body></body>

<html></html></html></html></html></html></html></html></html></

**QUERY STRING**

This data is included in the HTTP message as a “query string”. This is a series of key/value pairs.

A "key/value pair" is a set of two pieces of data. The first, the "key," can be text data or numeric data. It might be something like "Age" or "Name." It is used to uniquely identify a particular key/value pair, so it must be unique within a collection of key/value pairs.

The second, the "value," is the actual data associated with that key. This is data like text, decimal numbers, integers, true/false data, etc. If we write “Animal=Dog”, “animal” is the key and “dog” is the value.

The key/value pairs send with the query string are written in this format:

[key]=[value]

For example:

firstName=Harry

A question mark is used to show where the query string begins, and key/value pairs are separated by an ampersand (& – symbol for “and”). The format is:

?[key1]=[value]&[key2]=[value]

For example:

?firstName=Harry&lastName=Potter

Let’s take a look at how this is used.

Say there is a program out on our web server that can retrieve the data for a particular student if it is given a valid student ID – a unique identifier for the student. The program is called “get\_student”.

We have given the user a web page that lists out all the students in the school. Next to each student name is a link that says “Details”. The URL might look like this:

https://techacademystorage.blob.core.windows.net/htmlandcss/query_string.PNG

The actual HTTP request generated by clicking on the “Details” hyperlink would look something like this:

GET /adminportal/get\_student?studentId=23

Host: www.exampleschool.com

Accept-Language: en-us

Let’s break down the first line of this HTTP message – the remaining lines are identical to what we saw in the earlier example.

* The first section is the “Start Line”, which is used to describe the message. In this case, the resource to which the GET method will be applied is the program called “get\_student”. The query string contains the key/value pair that will be given to that program.

request that took the data from a form and sent it to a program on the server called “updateStudent”:

PUT /admin/updateStudent HTTP/1.1

Host: www.exampleschool.com

studentId=23&GPA=3.74

This would result in the change of the GPA of the student with ID 23 from 3.58, as above, to 3.74.

On the other hand, if you were creating a new student, you wouldn’t want it to actually happen multiple times – that would result in more than one identical student being created.

The POST verb is non-idempotent. If you wanted to create a new student, you might send an HTTP request that took the data from a form and sent it to a program on the server called “createStudent”:

POST /admin/createStudent HTTP/1.1

Host: www.exampleschool.com

firstName=John&lastName=Doe

This would result in a new student named John Doe being created on the server.

If you used a browser to access that first API, the HTTP request might look like this:

GET /api/oneday/ZIP/95437

Host: www.weatherforcast.com

Accept-Language: en-us

The HTTP response might look like this:

HTTP/1.1 200 OK

Date: Sat, 25 May 2019 12:38:53 GMT

{

“ZIP”: 95437

“Date”: “05 25 2019”

“High Temperature”: “77F”

}

There are many APIs available on the Internet. You will likely create and use APIs in your work.

GET: retrieve information about a specified resource

POST: create a specified resource

PUT: modify a specified resource

DELETE: delete a specified resource

The GET verb is pretty straightforward. Let’s use it in an example where we want to allow other programs to get a list of all students. The URL for that might be:

http://www.exampleschool.com/api/StudentService/Students

If we send an HTTP message to that URL with an HTTP verb of GET, we will receive a list of all students.

Let’s look at an example where we want to get data about a specific student. The URL for that might be:

http://www.exampleschool.com/api/StudentService/Student/:studentId

Here, the HTTP request would use a GET verb. The request would also include the studentId for the student whose data we want.

**CSS LISTS**

We can also style lists with CSS. For example: the following CSS code can change an ordered list from 1., 2., 3. to Roman numerals:



And to make an unordered list from circular bullet points to squares, we write:



To add these effects to our code, we would first save the above CSS code as a separate .css file.

Then we would link our HTML document to the new CSS file.

Non-breaking Space

A common character entity used in HTML is the non-breaking space: **&nbsp;**

A non-breaking space is a space that will not break into a new line.

Two words separated by a non-breaking space will stick together (not break into a new line). This is handy when breaking the words might be disruptive.

Examples:

* § 10
* 10 km/h
* 10 PM
* **CREATING A RESOURCE**
* Now let’s look at creating a resource – in our case, a new student.
* Again, in RESTful we identify a resource and use the HTTP verb to indicate what we want to happen with that resource.
* In this case, we identify the resource by calling the appropriate endpoint in our service:
* http://www.exampleschool.com/api/StudentService/Student
* In this case, however, we will change the HTTP verb to POST, so that we are creating a new resource, and we will put the information needed to create that new resource in the body of the HTTP message. The HTTP message might look like this:
* POST /api/StudentService/Student
* Host: www.weatherforcast.com
* Accept-Language: en-us
* firstName=Billy&lastName=Bart&GPA=2.43
* When this message is received by the server, the data in the body of the message will be used to create a new student. This happens because the programmer who made the service sets it up that way, as part of making their program conform with this idea of a RESTful system.
* The response often includes data that will be useful in tracking the state of the various resources that the program is tracking. In this case, the HTTP response would likely include the studentId for the newly-created student “Billy Bart”, as that studentId would have just been created by the program that has the Student service.

Finally, let’s look at the use of the PUT verb to modify the state of a specified resource. If we wanted to change the first name of a specified student from “Billy” to “William”, the HTTP message might look like this:

PUT /api/StudentService/Student

Host: www.weatherforcast.com

Accept-Language: en-us

studentId=24&firstName=William

When this message is received by the server, the data in the body of the message will be used to change the first name of the student with an ID of 24 from “Billy” to “William”.

By creating the APIs of your programs to conform to RESTful principles, you enable the tracking and control of the state of the things in your program, even though the HTTP protocol you’re using isn’t stateful by itself.

**DELETE**

DELETE is pretty straightforward - it would use the same URL, but include the HTTP verb DELETE. Additionally, a piece of data that would identify the specific resource to be deleted would be in the body of the message. The HTTP message might look like this:

DELETE /api/StudentService/Student

Host: www.weatherforcast.com

Accept-Language: en-us

studentId=24

When this message is received by the server, the data in the body of the message will be used to delete the student with an ID of 23.

Finally, let’s look at the use of the PUT verb to modify the state of a specified resource. If we wanted to change the first name of a specified student from “Billy” to “William”, the HTTP message might look like this:

PUT /api/StudentService/Student

Host: www.weatherforcast.com

Accept-Language: en-us

studentId=24&firstName=William

When this message is received by the server, the data in the body of the message will be used to change the first name of the student with an ID of 24 from “Billy” to “William”.

By creating the APIs of your programs to conform to RESTful principles, you enable the tracking and control of the state of the things in your program, even though the HTTP protocol you’re using isn’t stateful by itself.

**HTML FORMS**

The <form> tag is used to create a form that allows users to input data. Here is a typical way to write the <form> tag:

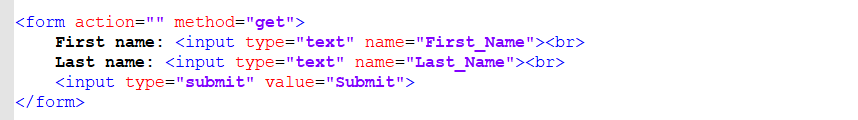
https://techacademystorage.blob.core.windows.net/htmlandcss/forms.PNG

The action attribute specifies where to send form-data when a form is submitted.

The method attribute specifies how to send form-data (while the action attribute gives the page to send the form-data to). The method attribute can contain GET or POST.

**CREATING A FORM**

Here is the HTML code for creating a basic form:



We left the quotation marks after the action attribute blank because we aren’t sending the form data anywhere at this point.

The <input> tag specifies a field where a user can enter data.

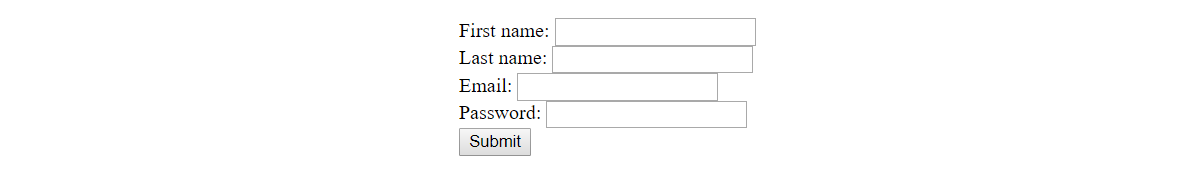
The type attribute specifies the type of user input (text, checkbox, password, etc.). In the above code, we use the type attribute “submit”, which creates a submit button.

https://techacademystorage.blob.core.windows.net/htmlandcss/submit.png

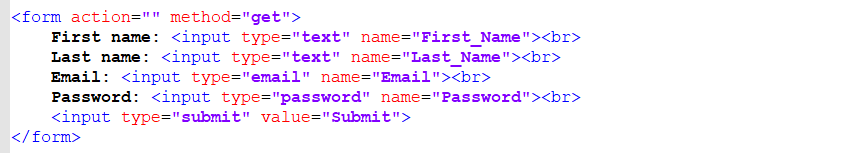
The name attribute specifies the name of an <input> element.

The value attribute specifies the value of an <input> element. For example: if you use a button, the value attribute defines what text will be displayed on the button.

To create a form that looks like this:



We would write this code:

ds

To create a fixed top menu, use position:fixed and top:0. Note that the fixed menu will overlay your other content. To fix this, add a margin-top (to the content) that is equal or larger than the height of your menu.