**The <script> tag**

The <script> element allows you to add JavaScript code inside an HTML file. Below, the <script> element embeds valid JavaScript code:

<h1>This is an embedded JS example</h1>

<script>

function Hello() {

alert ('Hello World');

}

</script>

Frankly, without the <script> tag, websites would be unclickable and a bit boring.

The <script> element, like most elements in *HTML*, has an opening and closing angle bracket. The closing tag marks the end of the content inside of the <script> element. Just like the <style> tag used to *embed* CSS code, you use the <script> tag to *embed* valid JavaScript code.

**Instructions**

**1.**

Copy this JavaScript code and paste it between the opening and closing <script> tags.

function blooming() {

var image = document.getElementById('myImage');

if (image.src.match("normal")) {

image.src = "flower.png";

} else {

image.src = "normal.png";

}

}

Then, click **Run** when you are finished and click on the Codecademy logo.

<!DOCTYPE html>

<html>

<head>

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

</head>

<body>

  <section class = "container">

    <img src = "normal.png" id= "myImage">

    <p onclick="blooming()">Codecademy</p>

  </section>

  <!-- Paste your code in the script element below:  -->

  <script>

    function blooming() {

  var image = document.getElementById('myImage');

  if (image.src.match("normal")) {

    image.src = "flower.png";

  } else {

    image.src = "normal.png";

  }

}

  </script>

</body>

</html>

**The src attribute**

Since you know how to use a <script> element with embedded code, let’s talk about linking code. Linking code is preferable because of a programming concept called Separation of Concerns (SoC). Instead of having messy code that is all in the same file, web developers separate their code into different files, making each “concern” easier to understand and more convenient when changes must be made.

For this exercise, instead of writing JavaScript in our HTML file, we are going to write it in its own file, and then reference this code with a *file path name*. We will do this using an attribute that may jog your memory: the src attribute!

If this seems familiar, that’s because you may have been linking to external files with the <img> and <link> elements. The attribute is exactly the same, but now its value specifies the location of your script file.

If the file is in the same project folder, the src value will be a *relative path* name. Below is an example of a providing a relative path for a JavaScript file.

<script src="./exampleScript.js"></script>

The <script> above would look for a file called **exampleScript.js** that is in the same folder/directory as our **index.html** file.

If you must refer to JavaScript hosted externally, or in a [CDN](https://developer.mozilla.org/en-US/docs/Glossary/CDN), you can also link to that file location.

**Instructions**

**1.**

Add a <script> element with a src attribute that points to **script.js**.

Hint

Here’s an example on how to include a <script> element with an src attribute:

<script src="example.js"></script>

**2.**

Now, click on the Codecademy logo repeatedly to see random font sizes and font colors. The source attribute allows our HTML file to access all of this magical JavaScript with little code.

<!DOCTYPE html>

<html>

  <head>

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

    <!-- Add your script tag here -->

    <script src="./script.js"></script>

  </head>

  <body onclick="newStyle();">

    <p class="centered" id="logo">Codecademy</p>

  </body>

</html>

# How are scripts loaded?

A quick recap: the <script> element allows HTML files to load and execute JavaScript. The JavaScript can either go embedded inside of the <script> tag or the script tag can reference an external file. Before we dive deeper, let’s take a moment to talk about how browsers parse HTML files into web pages. This informs where to include a <script> element inside your HTML file.

Browsers come equipped with HTML parsers that help browsers render the elements accordingly. Elements, including the <script> element, are by default, parsed in the order they appear in the HTML file. When the HTML parser encounters a <script> element, it loads the script then executes its contents before parsing the rest of the HTML. The two main points to note here are that:

* The HTML parser does NOT process the next element in the HTML file until it loads and executes the <script> element, thus leading to a delay in load time and resulting in a poor user experience.
* Additionally, scripts are loaded sequentially, so if one script depends on another script, they should be placed in that very order inside the HTML file.

The GIF below displays two scripts being loaded. The first script makes a Watering Can appear, the second script makes a Flower appear. This shows how scripts are loaded sequentially, and how they pause the HTML parser, which is why “Blooming” appears at the end.

**Instructions**

**1.**

Oops! In the code editor **script2.js** depends on a variable in **script1.js** causing an error (we actually want our text to be blue instead of pink). Switch the order of the scripts so that they load appropriately.

Hint

Rearrange the order of <script>s so that the one with src="script1.js" is above src="script2.js".

To see the error, right click on the page and select **inspect**. You’ll see a panel show up with multiple tabs at its top. Click on the **console** tab to see the error. For more information check out [this article on dev tools](https://www.codecademy.com/articles/use-devtools), specifically the section “Debug your JavaScript Using the Console Tool”.

<!DOCTYPE html>

<html>

  <head>

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

  </head>

  <body>

    <p class="centered" id="logo">Codecademy</p>

    <script src="script1.js"></script>

    <script src="script2.js"></script>

  </body>

</html>

# Defer attribute

When the HTML parser comes across a <script> element, it stops to load its content. Once loaded, the JavaScript code is executed and the HTML parser proceeds to parse the next element in the file. This can result in a slow load time for your website. HTML4 introduced the defer and async attributes of the <script> element to address the user wait-time in the website based on different scenarios.

The defer attribute specifies scripts should be executed after the HTML file is completely parsed. When the HTML parser encounters a <script> element with the defer attribute, it loads the script but defers the actual execution of the JavaScript until after it finishes parsing the rest of the elements in the HTML file.

Here is an example of the defer tag:

<script src="example.js" defer></script>

When is defer useful?

When a script contains functionality that requires interaction with the DOM, the defer attribute is the way to go. This way, it ensures that the entire HTML file has been parsed before the script is executed.

**Instructions**

**1.**

We want the “Codecademy” to be blue, not yellow! Each script tag re-styles the Codecademy logo, but the **turnYellow.js** executes last, making the font color 'yellow'. Add a defer attribute to the **turnBlue.js** script to make it the last script that is downloaded and executed.

Hint

Even though **turnBlue.js** appears before **turnYellow.js**, because we added the defer attribute, **turnBlue.js** loads after because of the defer attribute.

Here’s an example of syntax:

<script defer scr="sampleFile.js"></script>

<!DOCTYPE html>

<html>

  <head>

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

    <!--  Add the defer attribute to the script below:  -->

    <script id="blue" src="turnBlue.js" defer></script>

  </head>

  <body>

    <p class="centered" id="logo">Codecademy</p>

    <script id="yellow" src="turnYellow.js"></script>

  </body>

</html>

# Async attribute

The async attribute loads and executes the script asynchronously with the rest of the webpage. This means that, similar to the defer attribute, the HTML parser will continue parsing the rest of the HTML as the script is downloaded in the background. However, with the async attribute, the script will not wait until the entire page is parsed: it will execute immediately after it has been downloaded. Here is an example of the async tag:

<script src="example.js" async></script>

When is it useful?

async is useful for scripts that are independent of other scripts in order to function accordingly. Thus, if it does not matter exactly at which point the script file is executed, asynchronous loading is the most suitable option as it optimizes web page load time.

**Instructions**

**1.**

Currently our text is pink because in our **style.css** file, we have a ruleset for that.

Add async attribute to the **turnBlue.js** script to optimize load speed. Notice that once the **turnBlue.js** script is loaded, the text turns blue!

Hint

Similar to defer, we can add the async attribute to a <script> inside the <head> and have it load after elements below load first.

Depending on the speed of your internet, the loading of the page should be pretty fast so you probably won’t even see the text change from pink to blue!

<!DOCTYPE html>

<html>

  <head>

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

    <!-- Add the async attribute to the script below: -->

   <script id="blue" src="turnBlue.js" async></script>

  </head>

  <body>

    <p class="centered" id="logo">Codecademy</p>

  </body>

</html>

# Review

Let’s review.

* HTML creates the skeleton of a webpage, but JavaScript introduces interactivity
* The <script> element has an opening and closing tag. You can embed JavaScript code inbetween the opening and closing <script> tags.
* You link to external JavaScript files with the **src** attribute in the opening <script> tag.
* By default, scripts are loaded and executed as soon as the HTML parser encounters them in the HTML file, the HTML parser waits to load the entire script before from proceeding to parse the rest of the page elements.
* The defer attribute ensures that the entire HTML file has been parsed before the script is executed.
* The async attribute will allow the HTML parser to continue parsing as the script is being downloaded, but will execute immediately after it has been downloaded.

The old convention was to put scripts right before the </body> tag to prevent the script from blocking the rest of the HTML content. Now, the convention is to put the script tag in the <head> element and to use the defer and async attributes.

# What is the DOM?

The Document Object Model, abbreviated DOM, is a powerful tree-like structure that allows programmers to conceptualize hierarchy and access the elements on a web page.

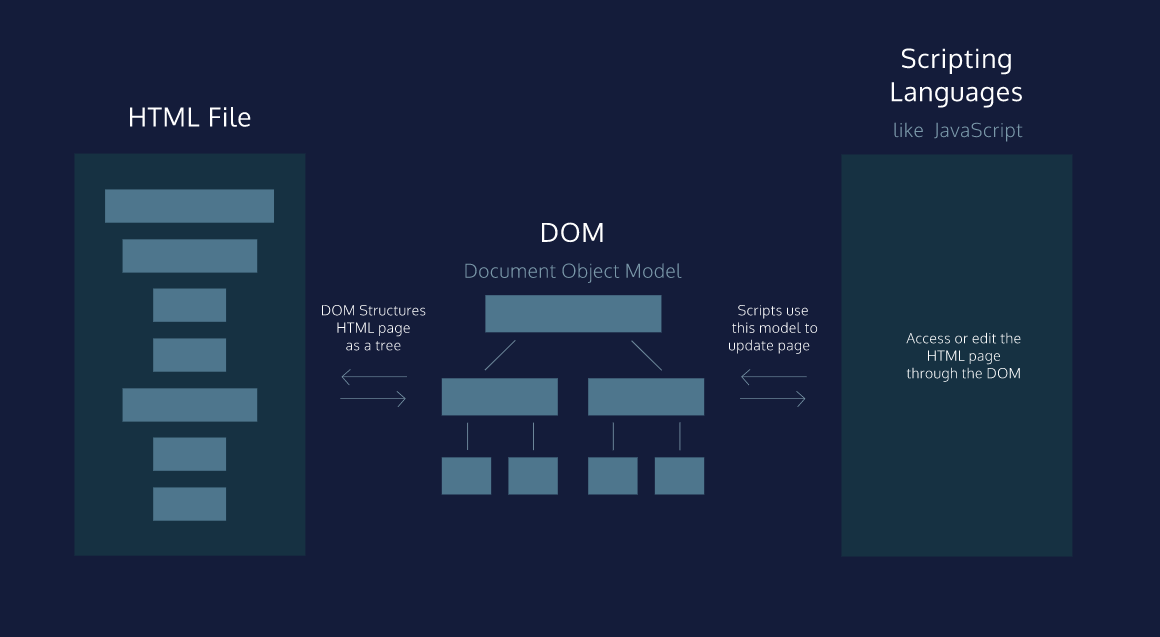
The DOM is one of the better-named acronyms in the field of Web Development. In fact, a useful way to understand what DOM does is by breaking down the acronym but out of order:

* The DOM is a logical tree-like **M**odel that organizes a web page’s HTML **D**ocument as an **O**bject.

Note: There are other types of documents, such as XML and SVG, that are also modeled as DOM structures.

The DOM is a language-agnostic structure implemented by browsers to allow for web scripting languages, like JavaScript, to access, modify, and update the structure of an HTML web page in an organized way.

For this reason, we like to think of the DOM as the link between an HTML web page and scripting languages.



# The DOM as a Tree Structure

Tree-like modeling is used in many fields, including evolutionary science and data analytics. Perhaps you’re already familiar with the concept of family trees: these charts represent the familial relationships amongst the descendants of a given family name.

The DOM tree follows similar logic to that of a family tree. A family tree is made up of family members and their relationships to the family name. In computer science, we would call each family member a node.

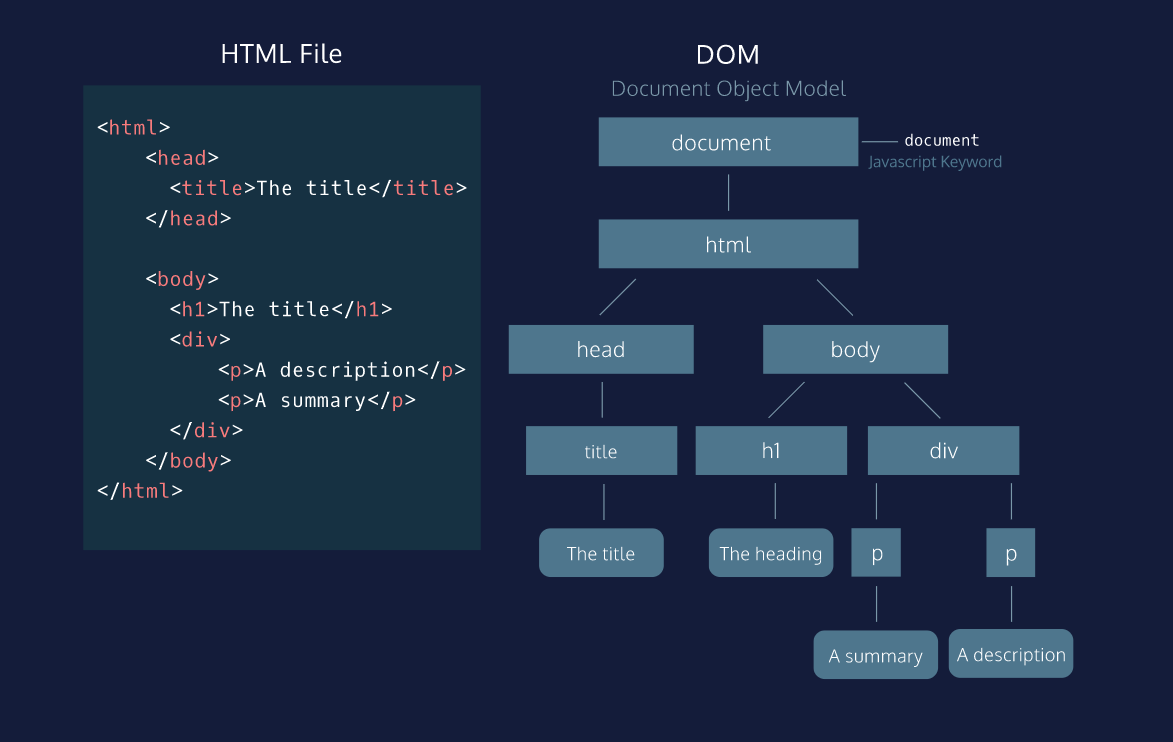
We define a node as an intersecting point in a tree that contains data.

In the DOM tree, the top-most node is called the root node, and it represents the HTML document. The descendants of the root node are the HTML tags in the document, starting with the <html> tag followed by the <head> and <body> tags and so on.

**Instructions**

The diagram to the right models the HTML document and labels the root element, which is the document.

Observe the difference in the rectangular boxes and the curved boxes. These denote a difference in the types of nodes in the DOM structure.



# Parent Child Relationships in the DOM

Following the metaphor of a family tree, let’s define some key terminology in the DOM hierarchy:

A parent node is the closest connected node to another node in the direction towards the root.

A child node is the closest connected node to another node in the direction away from the root.

Knowing these terms will allow you to understand and discuss the DOM as a tree-like structure. In fact, you will also see this terminology used when referring to the nesting structure of HTML code. Programmers refer to elements nested inside other elements as the children elements and parent elements respectively.

**Instructions**

**1.**

Add the appropriate HTML elements to the **index.html** file so that it reflects the tree-diagram on the far right.

Start by adding the <title> element. Make sure to nest the tags correctly, so that the nodes follow the illustrated parent-child relationships.

**2.**

Add the <div> element to **index.html** so that it reflects the DOM diagram to the right.

<!DOCTYPE html>

<html>

  <head>

    <title>The Title

      </title>

  </head>

  <body>

    <h1>The heading</h1>

    <div>

      <p>A summary</p>

      <p>A description</p>

    </div>

  </body>

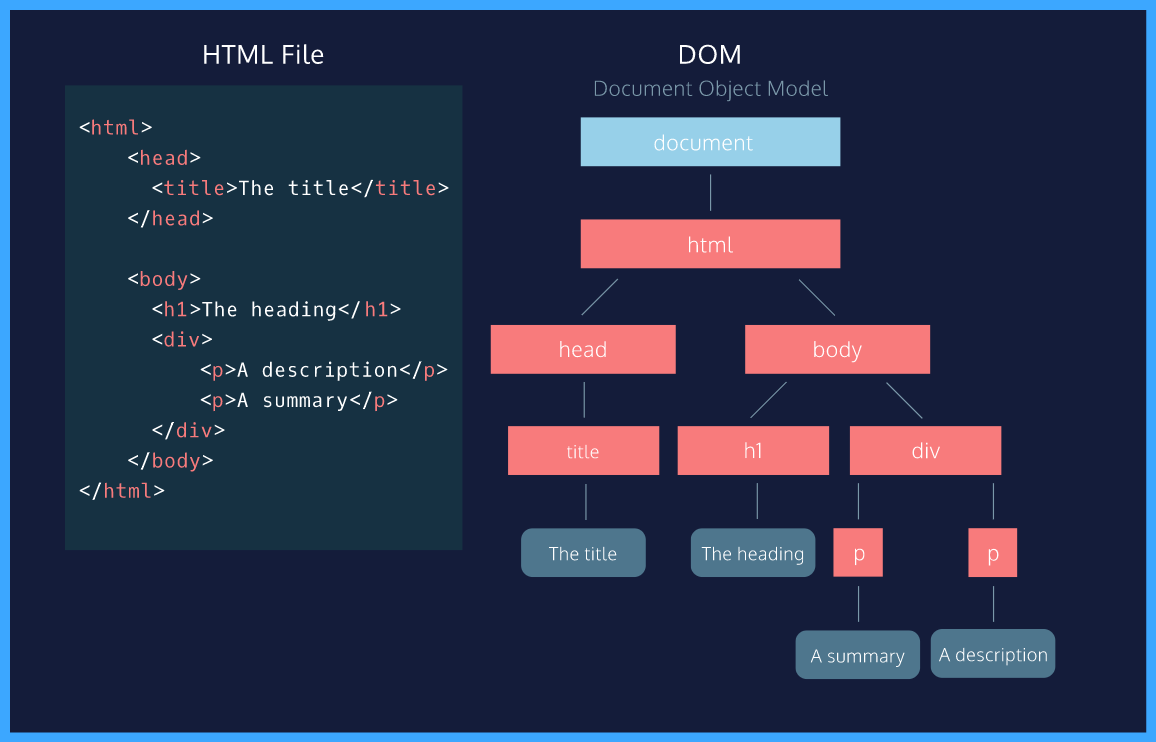
</html>

# Nodes and Elements in the DOM

As mentioned, a node is the equivalent of each family member in a family tree. A node is an intersecting point in a tree that also contains data.

There are nine different types of node objects in the DOM tree. In our diagram, the node objects with the sharp-edge rectangles are of the type [Element](https://developer.mozilla.org/en-US/docs/Web/API/Element), while the rounded edge rectangles are of type [Text](https://developer.mozilla.org/en-US/docs/Web/API/Text), because they represent the text inside the HTML paragraph elements.

When trying to modify a web page, the script will mostly interact with the DOM nodes of type element. Elements are the building units of HTML web pages, they contain everything between an opening tag and a closing tag. If the tag is a self-closing tag, then that is the element itself.

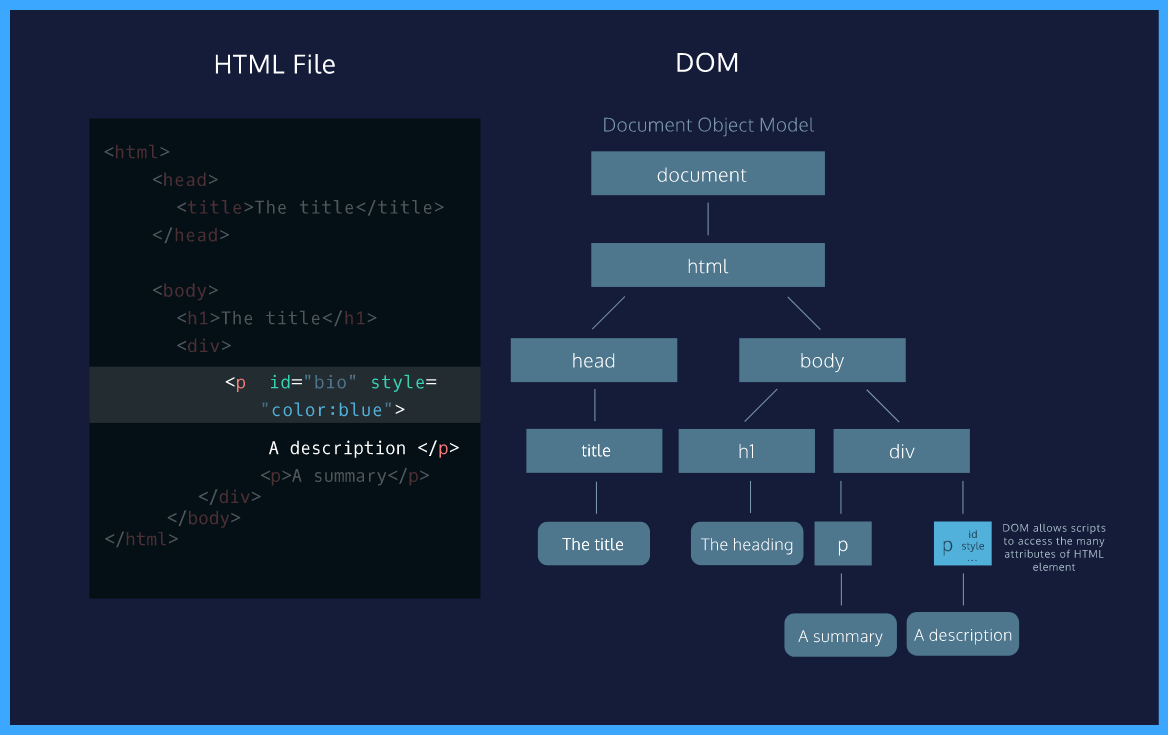


# Attributes of Element Node

DOM element nodes model elements in an HTML document.

Much like an element in an HTML page, the DOM allows us to access a node’s attributes, such as its class, id, and inline style.

In the diagram to the right, we have highlighted the paragraph element with an id of “bio” in the HTML document. If we were accessing that element node in our script, the DOM would allow us to tweak each of those attributes, or simply access them to check their value in the code.



# Review

Congratulations on completing our introduction to the Document Object Model, or DOM, as a structure!

Let’s review what you’ve learned so far:

* The DOM is a structural model of a web page that allows for scripting languages to access that page.
* The system of organization in the DOM mimics the nesting structure of an HTML document.
* Elements nested within another are referred to as the children of that element. The element they are nested within is called the parent element of those elements.
* The DOM also allows access to the regular attributes of an HTML element such as its style, id, etc.

With this understanding, you can begin to leverage the power of scripting languages to create, update, and maintain webpages!

# The document keyword

The Document Object Model, abbreviated DOM, is a powerful tree-like structure that organizes the elements on a web page and allows scripting languages to access them. This lesson will focus on some of the most useful methods and properties of the [DOM Interface](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model) in JavaScript. This interface is implemented by every modern browser.

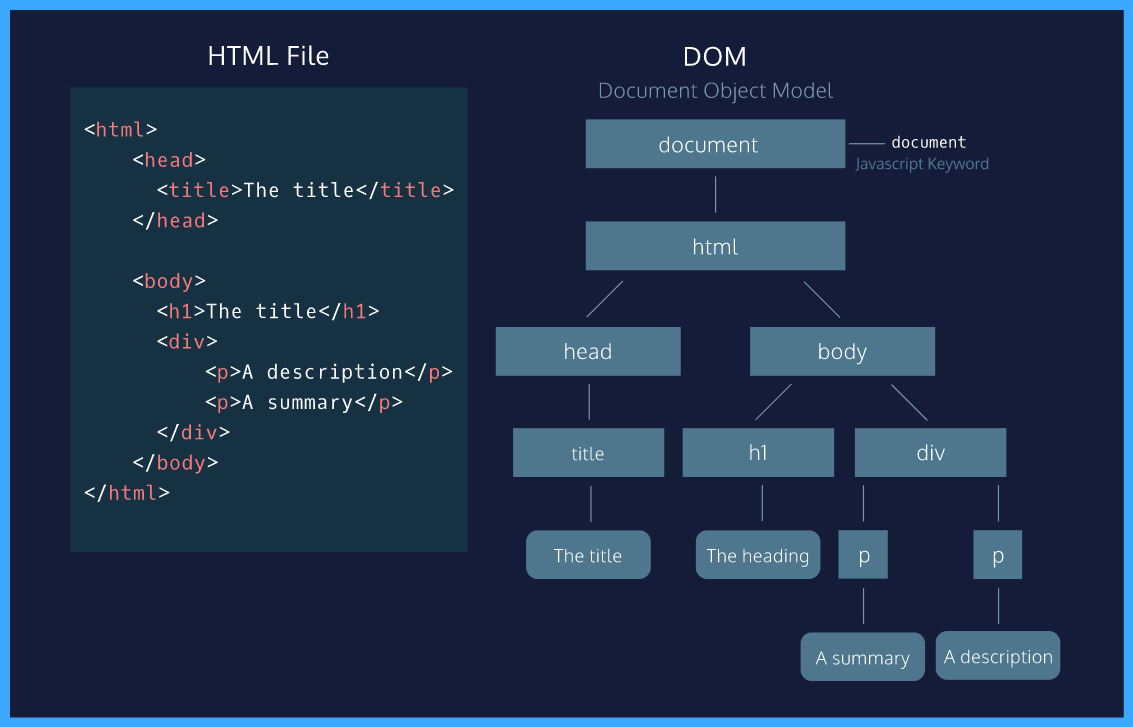
First things first! The document object in JavaScript is the door to the DOM structure. The document allows you to access the root node of the DOM tree. Before you can access a specific element in the page, first you must access the document structure itself. The document allows scripts to access children of the DOM as properties.

For example, if you wanted to access the <body> element in your script, you could access it as a property of the document by typing document.body. This property will return the body element of that DOM.

Similarly, you could access the <title> element with the .title property. See a [comprehensive list](https://developer.mozilla.org/en-US/docs/Web/API/Document) of all document properties.

**Instructions**

The diagram to the right illustrates that the document keyword points to the root node of the Document Object Model (DOM). Thedocument.body and document.head properties act as though you were directly accessing the html DOM element. Click Next when you’re ready!



# Tweak an Element

When using the DOM in your script to access an HTML element, you also have access to all of that element’s properties.

This includes the ability to modify the contents of the element as well as its attributes and properties— that can range from modifying the text inside a p element to assigning a new background color to a div.

You can access and set the contents of an element with the .innerHTML property.

For example, the following code reassigns the inner HTML of the body element to the text ‘The cat loves the dog’:

document.body.innerHTML = 'The cat loves the dog.';

The .innerHTML property can also add any valid HTML, including properly formatted elements. The following example assigns an h2 element as a child inside the <body> element:

document.body.innerHTML = '<h2>This is a heading</h2>';

**Instructions**

**1.**

Use the .innerHTML property to modify the content of the body to display the text ‘This is the text of the body element’.

Notice that the previous content inside of body has been replaced

Hint

document.body.innerHTML = 'desired content';

document.body.innerHTML = 'This is the text of the body element';

# Select and Modify Elements

In the previous exercise, we accessed the webpage elements with the document keyword!

What if we wanted to select a specific element? The DOM interface allows us to access a specific element with CSS selectors. CSS selectors define the elements to which a set of CSS rules apply, but we can also use these same selectors to access DOM elements with our script! Selectors can include the name of the tag, a class, or an ID.

The .querySelector() method allows us to specify a CSS selector and then returns the first element that matches that selector. The following code would return the first paragraph in the document.

document.querySelector('p');

You can also use other CSS selectors such as an element’s . class or its # ID.

Another option, if you want to access elements directly by their id, you can use the aptly named .getElementByID() function:

document.getElementById('bio').innerHTML = 'The description';

The example chains so that it selects the element with an ID of ‘bio’ and set its .innerHTML to the text ‘The description’.

**Instructions**

**1.**

Call the .querySelector() method to select the tag of h1 from the DOM. Access that element’s .innerHTML property to change the h1 title to ‘Most popular TV show searches in 2016’.

**2.**

Use the .getElementById method to access the element with an id of "fourth". Access that element’s .innerHTML to replace the content in the fourth item in the list to read “Fourth element”.

document.querySelector("h1").innerHTML = "Most popular TV show searches in 2016";

document.getElementById('fourth').innerHTML = "Fourth element";

# Style an element

Another way to modify an element is by changing its CSS style. The .style property of a DOM element provides access to the inline style of that HTML tag.

The syntax follows an element.style.property format, with the property representing a CSS property.

For example, the following code selects the first element with a class of blue and assigns blue as the background-color:

let blueElement = document.querySelector('.blue');

blueElement.style.backgroundColor = 'blue';

Unlike CSS, the DOM style property does not implement a hyphen such as background-color, but rather camel case notation backgroundColor.

The following chaining syntax would also work:

document.querySelector('.blue').style.fontFamily = 'Roboto';

**Instructions**

**1.**

Style the background-color of the body in the blog-post document to match the color of the Codecademy text editor, '#201F2E' by using the .style property.

Hint

The following code will select a DOM element with a class of 'blue'. Then it will assign the element a blue background color.

let blueElement = document.querySelector('.blue');

blueElement.style.backgroundColor = 'blue';

document.body.style.backgroundColor = '#201F2E';

**Create and Insert Elements**

Just as the DOM allows scripts to modify existing elements, it also allows for the creation of new ones.

The .createElement(tagName) method creates a new element based on the specified tag name passed into it as an argument. However, it does not append it to the document. It creates an empty element with no inner HTML.

In order to create an element and add it to the web page, you must assign it to be the child of an element that already exists on the DOM. We call this process appending. The .appendChild() method will add a child element as the last child node.

The following code creates a new paragraph element, gives it an id, adds text to the new element’s .innerHTML, and appends it to the body of the document:

let paragraph = document.createElement('p');

paragraph.id = 'info';

paragraph.innerHTML = 'The text inside the paragraph';

document.body.appendChild(paragraph);

Unlike the .innerHTML property, the .appendChild() method does not replace the content inside of the parent, in this case body. Rather, it appends the element as the last child of that parent.

**Instructions**

**1.**

* Create an li element using the .createElement() method and save it in a variable called newDestination.
* On the following line, assign the element an id of 'oaxaca' by using the id property on newDestination.
* Lastly, assign the element the text 'Oaxaca, Mexico' by using the .innerHTML property on newDestination.

Hint

First, declare a variable that holds a new li element:

let yourVariableName = document.createElement('li');

Next, set the id:

yourVariableName.id = 'Name of id goes here';

Lastly, add text using the .innerHTML property:

yourVariableName.innerHTML = 'The list element text goes here';

**2.**

Append the new element you created as the last child of the list with the ID more-destinations.

Scroll to the bottom of the page in the browser to see your new element.

Hint

Use the .getElementById() method to target the element with the id, more-destinations.

Append the new element you created as the last child of the list using .appendChild():

document.getElementById('more-destinations').appendChild(yourVariableName);

let newDestination = document.createElement('li');

newDestination.id = 'oaxaca';

newDestination.innerHTML = 'Oaxaca, Mexico';

document.getElementById('more-destinations').appendChild(newDestination);

# Remove an Element

In addition to modifying or creating an element from scratch, the DOM also allows for the removal of an element. The .removeChild() method removes a specified child from a parent.

Because the .querySelector() method returns the first paragraph, the following code would remove the first paragraph in the document:

let paragraph = document.querySelector('p');

document.body.removeChild(paragraph);

It’s possible to also specify a different parent with the .querySelector() method, as long as you remove an element nested within that parent element.

If you want to hide an element because it does not need to be loaded initially, the .hidden property allows you to hide it by assigning it as true or false:

document.getElementById('sign').hidden = true;

The code above did not remove the element from the DOM, but rather hid it. This is not the same as setting the CSS visibility property to hidden. For a list of the best use cases for this property, read a list in the [MDN documentation](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/hidden).

**Instructions**

**1.**

* Select the element with the ID oaxaca and save it to a variable.
* Select its parent, assigned an ID of more-destinations.
* Remove the element using the .removeChild() method and passing in the variable containing the oaxaca element.

Hint

let paragraph = document.querySelector('p');

document.body.removeChild(paragraph);

const parent = document.querySelector("#more-destinations");

const child = document.querySelector("#oaxaca");

//code

parent.removeChild(child);

const parent = document.querySelector("#more-destinations");

const child = document.querySelector("#oaxaca");

parent.removeChild(child);

# Interactivity with onclick

You can add interactivity to DOM elements by assigning a function to run based on an [event](https://developer.mozilla.org/en-US/docs/Web/Events).

Events can include anything from a click to a user mousing over an element.

The .onclick property allows you to assign a function to run on a click event on an element:

let element = document.getElementById('interact');

element.onclick = function() { element.style.backgroundColor = 'blue' };

**Instructions**

**1.**

Modify the body of the turnButtonRed() function so that it modifies the button as follows:

1. Assigns the .style.backgroundColor to 'red'
2. Assigns the style.color to 'white'
3. Modifies the .innerHTML to 'Red Button'

**2.**

Add interactivity to the button element by adding a function that turns its .backgroundColor to 'red' and its font color to 'white' when the button is clicked.

Hint

let element = document.getElementById('interact');

element.onclick = function() { element.style.backgGround = 'red' };

let element = document.querySelector("button");

function turnButtonRed (){

  element.style.backgroundColor = "red";

  element.style.color = "white";

  element.innerHTML = "Red Button";

}

element.onclick = turnButtonRed;

# Traversing the DOM

In the DOM hierarchy, parent and children relationships are defined in relation to the position of the root node.

A parent node is the closest connected node to another node in the direction towards the root.

A child node is the closest connected node to another node in the direction away from the root.

These relationships follow the nesting structure present in HTML code. Elements nested within one HTML tag are children of that parent element.

Each DOM element node has a .parentNode and .children property. The property will return a list of the element’s children and return null if the element has no children.

The .firstChild property will grant access to the first child of that parent element.

**Instructions**

**1.**

Access the .firstChild of the body and save it to a variable named first. Then modify first‘s .innerHTML to:

'I am the child!'

Take a moment to note which element has now been modified.

Hint

To access the body of the page, you have to use document.body. After that, define the variable first and assign it the .firstChild of document.body. Assign the .innerHTML property of first to 'I am the child!'.

When done correctly you should see the first header text changed to: I am the child!

**2.**

Use the .parentNode property to access the parent element of the variable first and modify its .innerHTML to:

'I am the parent and my inner HTML has been replaced!'

Take a moment to notice the change in the web page.

Hint

Chain .innerHTML to first.parentNode to access the HTML. Then use = to assign the correct string as its value.

let first = document.body.firstChild;

first.innerHTML = 'I am the child!';

first.parentNode.innerHTML = 'I am the parent and my inner HTML has been replaced!';

# Review

In this lesson, you manipulated a webpage structure by leveraging the Document Object Model interface in JavaScript.

Let’s review what we learned:

* The document keyword grants access to the root of the DOM in JavaScript
* The DOM Interface allows you to select a specific element with CSS selectors by using the .querySelector() method
* You can also access an element directly by its ID with .getElementById()
* The .innerHTML and .style properties allow you to modify an element by changing its contents or style respectively
* You can create, append, and remove elements by using the .createElement(),.appendChild() and .removeChild() methods respectively
* The .onclick property can add interactivity to a DOM element based on a click event