codecademy

JavaScript Interactive Websites

HTML script element src attribute

The `src` attribute of a `<script>` element is used to point to the location of a script file.
The file referenced can be local (using a relative path) or hosted elsewhere (using an absolute path).

```
<!-- Using a relative path -->
<script src="./script.js"></script>

<!-- Using an absolute path -->
<script
src="https://code.jquery.com/jquery-
3.3.1.min.js"></script>
```

HTML script element defer attribute

The `defer` attribute of a `<script>` tag is a boolean attribute used to indicate that the script can be loaded but not executed until after the HTML document is fully parsed. It will only work for externally linked scripts (with a `src` attribute), and will have no effect if it is applied to an inline script.
In the example code block, the `<h1>` tag will be loaded and parsed before the script is executed due to the `defer` attribute.

```
<body>
  <script src="main.js" defer></script>
  <h1>Hello</h1>
</body>
```

HTML script tag async attribute

Scripts are loaded synchronously as they appear in an HTML file, before the following HTML is loaded and parsed. The `async` attribute can be used to load the scripts asynchronously, such that they will load in the background without blocking the HTML parser from continuing.
In the example code block, the script will load asynchronously in the background, without blocking the HTML parser.

```
<body>
  <script src="main.js" async></script>
  <h1>Hello world!</h1>
</body>
```

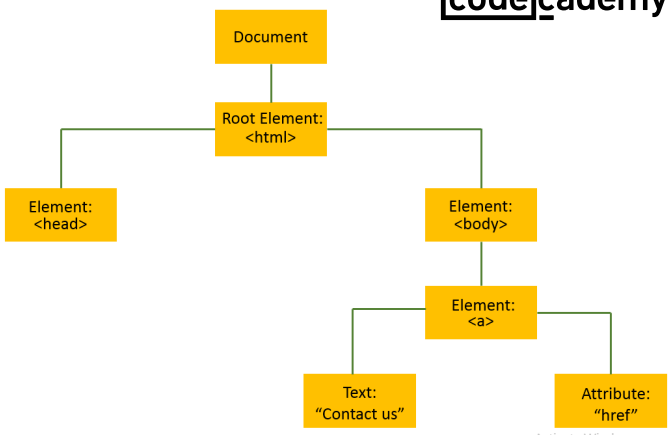
HTML script element

The HTML `<script>` element can contain or reference JavaScript code in an HTML file. The `<script>` element needs both an opening and a closing tag, and JavaScript code can be *embedded* between the tags.

```
<script>
  console.log("Hello world!");
</script>
```

Nodes in DOM tree

A *node* in the DOM tree is the intersection of two branches containing data. Nodes can represent HTML elements, text, attributes, etc. The *root node* is the top-most node of the tree. The illustration shows a representation of a DOM containing different types of nodes.



HTML DOM

The DOM is an interface between scripting languages and a web page's structure. The browser creates a Document Object Model or DOM for each webpage it renders. The DOM allows scripting languages to access and modify a web page. With the help of DOM, JavaScript has the ability to create dynamic HTML.

Accessing HTML attributes in DOM

The DOM nodes of type Element allow access to the same attributes available to HTML elements. For instance, for the given HTML element, the `id` attribute will be accessible through the DOM.

```
<h1 id="heading">Welcome!</h1>
```

The Document Object Model

The *Document Object Model*, or DOM is a representation of a document (like an HTML page) as a group of objects. While it is often used to represent HTML documents, and most web browsers use JavaScript interfaces to the DOM, it is language agnostic as a model.
The DOM is tree-like and heirarchical, meaning that there is a single top-level object, and other objects descend from it in a branching structure.

The DOM Parent-Child Relationship

The parent-child relationship observed in the DOM is reflected in the HTML nesting syntax.
Elements that are nested inside the opening and closing tag of another element are the children of that element in the DOM.

```
<body>
  <p>first child</p>
  <p>second child</p>
</body>
```

In the code block, the two `<p>` tags are children of the `<body>`, and the `<body>` is the parent of both `<p>` tags.

The `removeChild()` Method

The `.removeChild()` method removes a specified child from a parent element. We can use this method by calling `.removeChild()` on the parent node whose child we want to remove, and passing in the child node as the argument.

In the example code block, we are removing `iceCream` from our `groceryList` element.

```
const groceryList =
document.getElementById('groceryList');
const iceCream =
document.getElementById('iceCream');

groceryList.removeChild(iceCream);
```

The `element.parentNode` Property

The `.parentNode` property of an element can be used to return a reference to its direct parent node. `.parentNode` can be used on any node.

In the code block above, we are calling on the `parentNode` of the `#first-child` element to get a reference to the `#parent` `div` element.

```
<div id="parent">
  <p id ="first-child">Some child
text</p>
  <p id ="second-child">Some more child
text</p>
</div>
<script>
  const firstChild =
document.getElementById('first-child');
  firstChild.parentNode;  // reference to
the #parent div
</script>
```

The `document.createElement()` Method

The `document.createElement()` method creates and returns a reference to a new Element Node with the specified tag name.

`document.createElement()` does not actually add the new element to the DOM, it must be attached with a method such as `element.appendChild()`.

```
const newButton =
document.createElement("button");
```

The `element.InnerHTML` Property

The `element.innerHTML` property can be used to access the HTML markup that makes up an element's contents.

`element.innerHTML` can be used to access the current value of an element's contents or to reassign it.

In the code block above, we are reassigning the `box` element's inner HTML to a paragraph element with the text "Goodbye".

```
<box>
  <p>Hello there!</p>
</box>

<script>
```

```
  const box =
document.querySelector('box');
  // Outputs '<p>Hello there!</p>':
  console.log(box.innerHTML)
  // Reassigns the value:
  box.innerHTML = '<p>Goodbye</p>'
</script>
```

The document Object

The `document` object provides a Javascript interface to access the DOM. It can be used for a variety of purposes including referencing the `<body>` element, referencing a specific element with ID, creating new HTML elements, etc.
The given code block can be used to obtain the reference to the `<body>` element using the `document` object.

```
const body = document.body;
```

The document.getElementById() Method

The `document.getElementById()` method returns the element that has the `id` attribute with the specified value.
`document.getElementById()` returns `null` if no elements with the specified ID exists.
An ID should be unique within a page. However, if more than one element with the specified ID exists, the `.getElementById()` method returns the first element in the source code.

```
// Save a reference to the element with
id 'demo':
const demoElement =
document.getElementById('demo');
```

The .querySelector() Method

The `.querySelector()` method selects the first child/descendant element that matches its selector argument.
It can be invoked on the `document` object to search the entire document or on a single element instance to search that element's descendants.
In the above code block, we are using `.querySelector()` to select the first `div` element on the page, and to select the first element with a class of `button`, inside the `.main-navigation` element.

```
// Select the first <div>
const firstDiv =
document.querySelector('div');

// Select the first .button element
inside .main-navigation
const navMenu =
document.getElementById('main-
navigation');
const firstButtonChild =
navMenu.querySelector('.button');
```

The document.body Object

`document.body` returns a reference to the contents of the `<body>` HTML element of a document/HTML

page. The `<body>` element contains all the visible contents of the page.

The `element.onclick` Property

The `element.onclick` property can be used to set a function to run when an element is clicked. For instance, the given code block will add an `<li>` element each time the element with ID `addItem` is clicked by the user.

```
let element =
document.getElementById('addItem');
element.onclick = function() {
  let newElement =
document.createElement('li');

document.getElementById('list').appendChi
ld(newElement);
};
```

The `element.appendChild()` Method

The `element.appendChild()` method appends an element as the last child of the parent.
In the given code block, a newly created `<li>` element will be appended as the last child of the HTML element with the ID `list`.

```
var node1 = document.createElement('li');
document.getElementById('list').appendChi
ld(node1);
```

The `element.style` Property

The `element.style` property can be used to access or set the CSS style rules of an element. To do so, values are assigned to the attributes of `element.style`.
In the example code, `blueElement` contains the HTML element with the ID `colorful-element`. By setting the `backgroundColor` attribute of the `style` property to blue, the CSS property `background-color` becomes blue.
Also note that, if the CSS property contains a hyphen, such as `font-family` or `background-color`, Camel Case notation is used in Javascript for the attribute name, so `background-color` becomes `backgroundColor`.

```
let blueElement =
document.getElementById('colorful-
element');
blueElement.style.backgroundColor =
'blue';
```