The script tag

The `<script>` element allows you to add JavaScript code inside an HTML file.

Linking code, instead of adding the js code into HTML, 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.

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

How are scripts loaded?

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.

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.

defer and async

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.

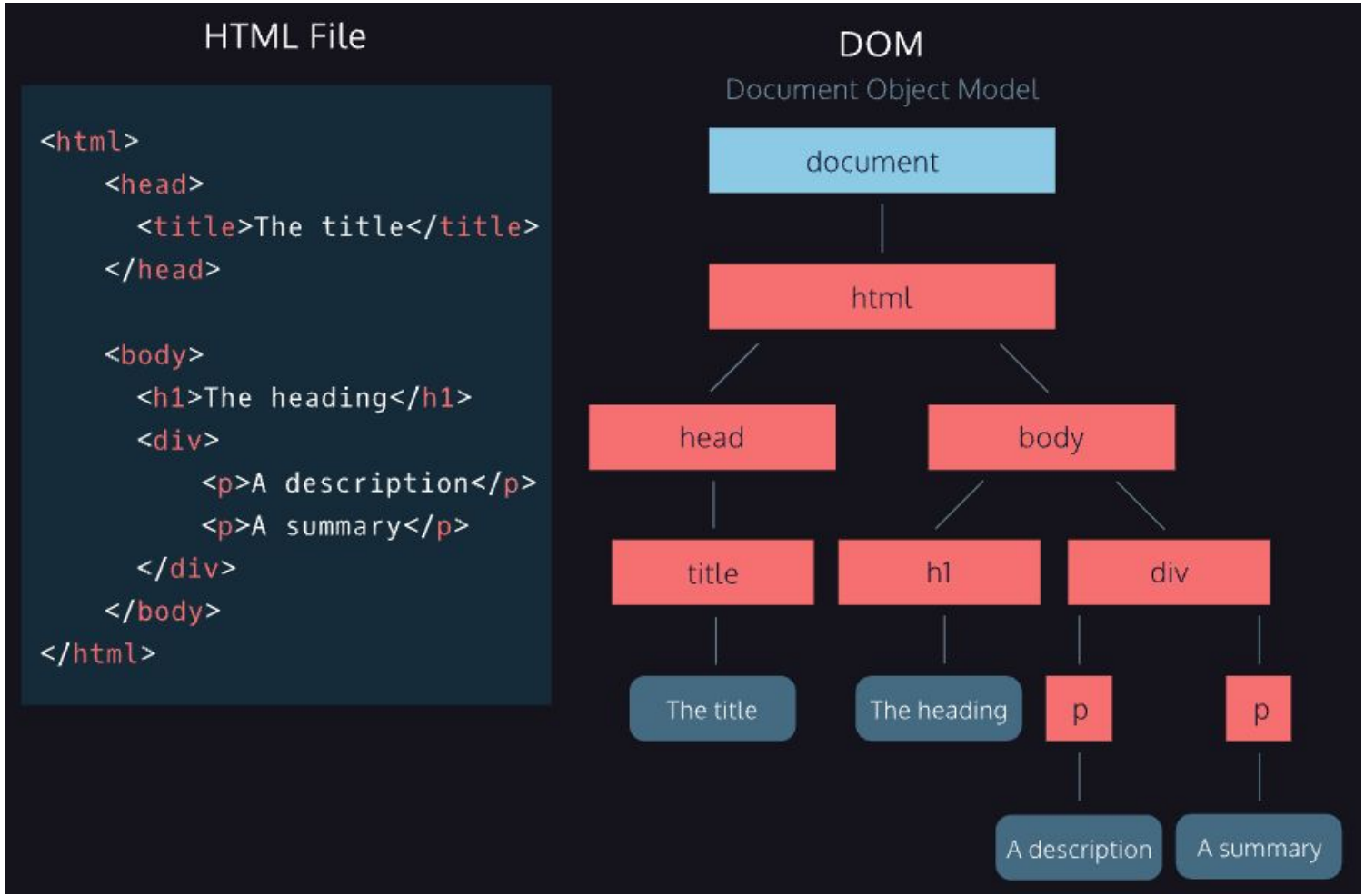
The `async` attribute loads and executes the script asynchronously with the rest of the webpage. This means that, similar to the `defer` attribute, the HTML parser will continue parsing the rest of the HTML as the script is downloaded in the background. However, with the `async` attribute, the script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

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 implemented by browsers to allow JavaScript to access, modify, and update the structure of an HTML web page in an organized way.

DOM


HTML File
<html>
<head>
<title>The title</title>
</head>
<body>
<h1>The heading</h1>

A description
A summary

</body>
</html>
DOM
Document Object Model
document
html
head
body
title
h1
div
The title
The heading
p
p
A description
A summary

The document keyword

The `document` object in JavaScript is the door to the DOM structure. The `document` object 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` object allows scripts to access children of the DOM as properties.

For example, if you want to access the `<body>` element from your script, you can access it as a property of the `document` object by using `document.body`. This property will return the body element of that DOM.

Similarly, you could access the `<title>` element with the `.title` property. Here is a comprehensive list of all `document` properties.

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, which can range from modifying the text inside a `<p>` element to assigning a new background color to a `<div>`. For example, the `.innerHTML` property allows you to access and set the contents of an element.

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

Select and modify elements

What if we wanted to select a specific element besides the entire `<body>` element? The DOM interface allows us to access a specific element with CSS selectors.

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

```
document.querySelector('p');
```

Along with `.querySelector()`, JavaScript has more targeted methods that select elements based on their class, id, or tag name:
`.getElementById(),.getElementsByClassName(),.getElementsByTagName()`

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.

```
let blueElement = document.querySelector('.blue');
blueElement.style.backgroundColor = 'blue';
```

Traversing the DOM

Each element has a `.parentNode` and `.children` property. The `.parentNode` property returns the parent of the specified element in the DOM hierarchy. Note that the `document` element is the *root node* so its `.parentNode` property will return `null`. The `.children` property returns an array of the specified element's children. If the element does not have any children, it will return `null`.

```
<ul id='groceries'>
  <li id='must-have'>Toilet Paper</li>
  <li>Apples</li>
  <li>Chocolate</li>
  <li>Dumplings</li>
</ul>
```

```
let parentElement = document.getElementById('must-
have').parentNode; // returns <ul> element
let childElements
= document.getElementById('groceries').children; // returns an
array of <li> elements
```

Create an Element

The `.createElement()` 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.

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

We can assign values to the properties of the newly created element like how we've done previously with existing elements.

```
paragraph.id = 'info';
paragraph.innerHTML = 'The text inside the paragraph';
```

Insert an Element

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, referred to as the parent element. We call this process *appending*. The `.appendChild()` method will add a child element as the parent element's last child node.

```
document.body.appendChild(paragraph);
```

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.

```
let paragraph = document.querySelector('p');
document.body.removeChild(paragraph);
```

If you want to hide an element rather than completely deleting it, the `.hidden` property allows you to hide it by setting the property as `true` or `false`:

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

Add Click Interactivity

You can add interactivity to DOM elements by assigning a function to run based on an event. 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 when a click event happens on an element:

```
let element
= document.querySelector('button');

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

```
let element
= document.querySelector('button');

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

element.onclick = turnBlue;
```

Activity