Questions:
1. What is span in HTML and what is a class?
`const cloudySpans = document.querySelectorAll("span.cloudy");`
2. What is span.cloudy and why is it surrounded by quotes?
3. <script type="text/javascript"> …. What is script type? And what does "text/javascript"> mean? Why not just javascript?

`const getRandomInt = max => {`
`  return Math.floor(Math.random() * Math.floor(max));`
`};`

4. What is going on here?
5. What is id and does it always go in quotes?
6.
7.

Using JavaScript, we can reference this element by scanning the document and finding the element by its id with the method document.getElementById(). We then assign the reference to a variable.

Javascript

```
const divOfInterest = document.getElementById(“catch-me-if-you-can”)
```

Now let’s say that our HTML file contains seven `span` elements that share a class name of `cloudy`, like below:

HTML

```
<span class=""cloudy”"></span>
<span class=""cloudy”"></span>
<span class=""cloudy”"></span>
<span class=""cloudy”"></span>
<span class=""cloudy”"></span>
<span class=""cloudy”"></span>
<span class=""cloudy”"></span>
```

In Javascript, we can reference all seven of these elements and store them in a single variable.

Javascript

```
const cloudySpans = document.querySelectorAll("span.cloudy");
```

While `getElementById` allows us to reference a single element, `querySelectorAll` references all elements with the class name “cloudy” as a static `NodeList` (*static* meaning that any changes in the DOM do not affect the content of the collection).

`NodeList` objects are collections of nodes, usually returned by properties such as `Node.childNodes` and methods such as `document.querySelectorAll()`.

Although `NodeList` is not an `Array`, it is possible to iterate over it with `forEach()`. It can also be converted to a real `Array` using `Array.from()`. However, some older browsers have not implemented `NodeList.forEach()` nor `Array.from()`. This can be circumvented by using `Array.prototype.forEach()` — see this document's Example.

Using `forEach()` on a NodeList:

Javascript

```
const cloudySpans = document.querySelectorAll("span.cloudy");

cloudySpans.forEach(span => {
  console.log("Cloudy!");
});
```

The `Document` method **`querySelectorAll()`** returns a static (not live) `NodeList` representing a list of the document's elements that match the specified group of selectors.

Note: This method is implemented based on the `ParentNode` mixin's `querySelectorAll()` method.

Syntax

```
elementList = parentNode.querySelectorAll(selectors);
```

Parameters

`selectors`

A `DOMString` containing one or more selectors to match against. This string must be a valid CSS selector string; if it's not, a `SyntaxError` exception is thrown. See Locating DOM elements using selectors for more information about using selectors to identify elements. Multiple selectors may be specified by separating them using commas.

Note: Characters which are not part of standard CSS syntax must be escaped using a backslash character. Since JavaScript also uses backslash escaping, special care must be taken when writing string literals using these characters. See Escaping special characters for more information.

Return value

A non-live `NodeList` containing one `Element` object for each element that matches at least one of the specified selectors or an empty `NodeList` in case of no matches.

Note: If the specified `selectors` include a CSS pseudo-element, the returned list is always empty.

Exceptions

`SyntaxError`
The syntax of the specified `selectors` string is not valid.

Examples

Obtaining a list of matches

To obtain a `NodeList` of all of the `<p>` elements in the document:

```
var matches = document.querySelectorAll("p");
```

This example returns a list of all `<div>` elements within the document with a class of either `note` or `alert`:

```
var matches = document.querySelectorAll("div.note, div.alert");
```

Here, we get a list of `<p>` elements whose immediate parent element is a `<div>` with the class `highlighted` and which are located inside a container whose ID is `test`.

```
var container = document.querySelector("#test");
```

```
var matches = container.querySelectorAll("div.highlighted > p");
```

This example uses an attribute selector to return a list of the `<iframe>` elements in the document that contain an attribute named `data-src`:

```
var matches = document.querySelectorAll("iframe[data-src]");
```

Here, an attribute selector is used to return a list of the list items contained within a list whose ID is `userlist` which have a `data-active` attribute whose value is `1`:

```
var container = document.querySelector("#userlist");
var matches = container.querySelectorAll("li[data-active='1']");
```

Accessing the matches

Once the `NodeList` of matching elements is returned, you can examine it just like any array. If the array is empty (that is, its `length` property is 0), then no matches were found.
Otherwise, you can simply use standard array notation to access the contents of the list. You can use any common looping statement, such as:

```
var highlightedItems = userList.querySelectorAll(".highlighted");

highlightedItems.forEach(function(userItem) {
  deleteUser(userItem);
});
```

Creating New DOM Elements

Now that we know how to reference DOM elements, let's try creating new elements. First we'll set up a basic HTML file with the appropriate structure and include a reference to a Javascript file that exists in the same directory in the `head`.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
    <script type="text/javascript" src="example.js"></script>
  </head>
  <body></body>
</html>
```

In our example.js file, we'll write a function to create a new `h1` element, assign it an id, give it content, and attach it to the body of our HTML document.

Javascript

```
const addElement = () => {
  // create a new div element
  const newElement = document.createElement("h1");

  // set the h1's id
  newElement.setAttribute("id", "sleeping-giant");

  // and give it some content
  const newContent = document.createTextNode("Jell-O, Burled!");

  // add the text node to the newly created div
  newElement.appendChild(newContent);

  // add the newly created element and its content into the DOM
  document.body.appendChild(newElement);
};
// run script when page is loaded
window.onload = addElement;
```

If we open up our HTML file in a browser, we should now see the words `Jell-O Burled!` on our page. If we use the browser tools to inspect the page (right-click on the page and select “inspect”, or hotkeys fn + f12), we notice the new `h1` with the id we gave it.

In an HTML document, the **`document.createElement()`** method creates the HTML element specified by *tagName*, or an `HTMLUnknownElement` if *tagName* isn't recognized.

Syntax

```
let element = document.createElement(tagName[, options]);
```

Parameters

tagName

A string that specifies the type of element to be created. The `nodeName` of the created element is initialized with the value of *tagName*. Don't use qualified names (like "html:a") with this method. When called on an HTML document, `createElement()` converts *tagName* to lower case before creating the element. In Firefox, Opera, and Chrome, `createElement(null)` works like `createElement("null")`.

options Optional

An optional `ElementCreationOptions` object, containing a single property named `is`, whose value is the tag name of a custom element previously defined via `customElements.define()`. See Web component example for more details.

Return value

The new `Element`.

Examples

Basic example

This creates a new `<div>` and inserts it before the element with the ID "`div1`".

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title>||Working with elements||</title>
</head>
<body>
  <div id="div1">The text above has been created dynamically.</div>
</body>
</html>
```

JavaScript

```
document.body.onload = addElement;

function addElement () {
  // create a new div element
  const newDiv = document.createElement("div");

  // and give it some content
  const newContent = document.createTextNode("Hi there and greetings!");

  // add the text node to the newly created div
  newDiv.appendChild(newContent);

  // add the newly created element and its content into the DOM
  const currentDiv = document.getElementById("div1");
```

```
  document.body.insertBefore(newDiv, currentDiv);
}
```

Sets the value of an attribute on the specified element. If the attribute already exists, the value is updated; otherwise a new attribute is added with the specified name and value.
To get the current value of an attribute, use `getAttribute()`; to remove an attribute, call `removeAttribute()`.

Syntax

```
Element.setAttribute(name, value);
```

Parameters

`name`
: A `DOMString` specifying the name of the attribute whose value is to be set. The attribute name is automatically converted to all lower-case when `setAttribute()` is called on an HTML element in an HTML document.

`value`
: A `DOMString` containing the value to assign to the attribute. Any non-string value specified is converted automatically into a string.

Boolean attributes are considered to be `true` if they're present on the element at all, regardless of their actual `value`; as a rule, you should specify the empty string (`""`) in `value` (some people use the attribute's name; this works but is non-standard). See the example below for a practical demonstration.
Since the specified `value` gets converted into a string, specifying `null` doesn't necessarily do what you expect. Instead of removing the attribute or setting its value to be `null`, it instead sets the attribute's value to the string `"null"`. If you wish to remove an attribute, call `removeAttribute()`.

Return value

`undefined`.

Hello, World DOMination: Inserting Elements in JS File and Script Block

Let's practice adding new elements to our page. We'll create a second element, a `div` with an id of `lickable-frog`, and append it to the `body` like we did the first time. Update the Javascript function to append a second element to the page.

Javascript

```
const addElements = () => {
  // create a new div element
  const newElement = document.createElement("h1");

  // set the h1's id
  newElement.setAttribute("id", "sleeping-giant");

  // and give it some content
  const newContent = document.createTextNode("Jell-O, Burled!");

  // add the text node to the newly created div
  newElement.appendChild(newContent);

  // add the newly created element and its content into the DOM
  document.body.appendChild(newElement);

  // append a second element to the DOM after the first one
  const lastElement = document.createElement("div");
  lastElement.setAttribute("id", "lickable-frog");
  document.body.appendChild(lastElement);
};
// run script when page is loaded
window.onload = addElements;
```

Notice that our function is now called `addElements`, plural, because we're appending two elements to the `body`. Save your Javascript file and refresh the HTML file in the browser. When you inspect the page, you should now see two elements in the `body`, the `h1` and the `div` we added via Javascript.

Referencing a JS File vs. Using a Script Block

In our test example above, we referenced an external JS file, which contained our function to add new elements to the DOM. Typically, we would keep Javascript in a separate file, but we could also write a script block directly in our HTML file. Let's try it. First, we'll delete the script source so that we have an empty script block.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript">
      //Javascript goes here!
    </script>
  </head>
  <body></body>
</html>
```

Inside of our script block, we'll:

- create a `ul` element with no id
- create an `li` element with the id `dreamy-eyes`
- add the `li` as a child to the `ul` element
- add the `ul` element as the first child of the `body` element.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Cool Website</title>
    <script type="text/javascript">
      const addListElement = () => {
        const listElement = document.createElement("ul");
        const listItem = document.createElement("li");
        listItem.setAttribute("id", "dreamy-eyes");
        listElement.appendChild(listItem);
        document.body.prepend(listElement);
      };
      window.onload = addListElement;
    </script>
  </head>
  <body></body>
```

```
</html>
```

Refresh the HTML in your browser, inspect the page, and notice the `ul` and `li` elements that were created in the script block.

Hello, World DOMination: Adding a CSS Class After DOM Load Event

In our previous JS examples, we used `window.onload` to run a function after the window has loaded the page, which ensures that all of the objects are in the DOM, including images, scripts, links, and subframes. However, we don't need to wait for stylesheets, images, and subframes to finish loading before our JavaScript runs because JS isn't dependent on them. And, some images may be so large that waiting on them to load before the JS runs would make the user experience feel slow and clunky. There is a better method to use in this case: `DOMContentLoaded`.
According to MDN, "the DOMContentLoaded event fires when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading."
We'll use DOMContentLoaded to add CSS classes to page elements immediately after they are loaded. Let's add the CSS class "i-got-loaded" to the `body` element when the window fires the DOMContentLoaded event. We can do this in the script block or in an external JS file, as we did in the examples above.

Javascript

```
window.addEventListener("DOMContentLoaded", event => {
  document.body.className = "i-got-loaded";
});
```

After adding the Javascript above, refresh the HTML in your browser, inspect the page, and notice that the `body` element now has a class of "i-got-loaded".

Hello, World DOMination: Console.log, Element.innerHTML, and the Date Object

In this section, we'll learn about how to use `console.log` to print element values. We'll also use `Element.innerHTML` to fill in the HTML of a DOM element. Finally, we'll learn about the Javascript Date object and how to use it to construct a clock that keeps the current time.

Console Logging Element Values

Along with the other developer tools, the console is a valuable tool Javascript developers use to debug and check that scripts are running correctly. In this exercise, we'll practice logging to the console.

Create an HTML file that contains the following:

HTML

```
<!DOCTYPE html>
<html>
  <head> </head>
  <body>
    <ul id="your-best-friend">
      <li>Has your back</li>
      <li>Gives you support</li>
      <li>Actively listens to you</li>
      <li>Lends a helping hand</li>
      <li>Cheers you up when you're down</li>
      <li>Celebrates important moments with you</li>
    </ul>
  </body>
</html>
```

In the above code, we see an id with which we can reference the `ul` element. Recall that we previously used `document.querySelectorAll()` to store multiple elements with the same class name in a single variable,

as a NodeList. However, in the above example, we see only one id for the parent element. We can reference the parent element via its id to get access to the content of its children.

Javascript

```
window.addEventListener("DOMContentLoaded", event => {
  const parent = document.getElementById("your-best-friend");
  const childNodes = parent.childNodes;
  for (let value of childNodes.values()) {
    console.log(value);
  }
});
```

In your browser, use the developer tools to open the console and see that the values of each `li` have been printed out.

Using Element.innerHTML

Thus far, we have referenced DOM elements via their id or class name and appended new elements to existing DOM elements. Additionally, we can use the inner HTML property to get or set the HTML or XML markup contained within an element.

In an HTML file, create a `ul` element with the id "your-worst-enemy" that has no children. We'll add some JavaScript to construct a string that contains six `li` tags each containing a random number and set the inner HTML property of `ul#your-worst-enemy` to that string.

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="example.js"></script>
  </head>
  <body>
    <ul id="your-worst-enemy"></ul>
  </body>
</html>
```

The **HTML `<ul>` element** represents an unordered list of items, typically rendered as a bulleted list.

Javascript

```
// generate a random number for each list item
const getRandomInt = max => {
  return Math.floor(Math.random() * Math.floor(max));
};

// listen for DOM ready event
window.addEventListener("DOMContentLoaded", event => {
  // push 6 LI elements into an array and join
  const liArr = [];
  for (let i = 0; i < 6; i++) {
    liArr.push("<li>" + getRandomInt(10) + "</li>");
  }
  const liString = liArr.join(" ");

  // insert string into the DOM using innerHTML
  const listElement = document.getElementById("your-worst-enemy");
  listElement.innerHTML = liString;
});
```

Save your changes, and refresh your browser page. You should see six new list items on the page, each containing a random number.

Inserting a Date Object into the DOM

We've learned a lot about accessing and manipulating the DOM! Let's use what we've learned so far to add extra functionality involving the Javascript Date object.

Our objective is to update the title of the document to the current time at a reasonable interval such that it looks like a real clock.

We know we'll be starting with an HTML document that contains an empty title element. We've learned a couple of different ways to fill the content of an element so far. We could create a new element and append it to the title element, or we could use `innerHTML` to set the HTML of the title element. Since we don't need to create a new element nor do we care whether it appears last, we can use the latter method.

Let's give our title an id for easy reference.

HTML

```
<title id="title"></title>
```

In our Javascript file, we'll use the Date constructor to instantiate a new Date object.

```
const date = new Date();
```

Javascript

```
window.addEventListener("DOMContentLoaded", event => {
  const title = document.getElementById("title");
  const time = () => {
    const date = new Date();
    const seconds = date.getSeconds();
    const minutes = date.getMinutes();
    const hours = date.getHours();

    title.innerHTML = hours + ":" + minutes + ":" + seconds;
  };
  setInterval(time, 1000);
});
```

Save your changes and refresh your browser. Observe the clock we inserted dynamically keeping the current time in your document title!

The new operator lets developers create an instance of a user-defined object type or of one of the built-in object types that has a constructor function.

Syntax

new constructor[([arguments])]

Parameters

constructor

A class or function that specifies the type of the object instance.

arguments

A list of values that the constructor will be called with.