Name:	What it Does:	Syntax:	Parameters:	Returns	Note:	Example:	Extra Info:
Document.querySelectorAll()	Document method: returns a static (not live) NodeList representing a list of the document's elements that match the specified group of selectors.	**elementList = parentNode.querySelectorAll(selectors);**	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	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.	`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).	This method is implemented based on the ParentNode mixin's querySelectorAll() method.

Document.getElementById()	Document method getElementById() returns an Element object representing the element whose id property matches the specified string. Since element IDs are required to be unique if specified, they're a useful way to get access to a specific element quickly.
	
	If you need to get access to an element which doesn't have an ID, you can use querySelector() to find the element using any selector.
		const element = document.getElementById(id);	id
			The ID of the element to locate. The ID is case-sensitive string which is unique within the document; only one element may have any given ID.
				An Element object describing the DOM element object matching the specified ID, or null if no matching element was found in the document.		**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>` (×7)
						In Javascript, we can reference all seven of these elements and store them in a single variable.
							Usage notes
							The capitalization of "Id" in the name of this method must be correct for the code to function; getElementById() is not valid and will not work, however natural it may seem.
							
							Unlike some other element-lookup methods such as Document.querySelector() and Document.querySelectorAll(), getElementById() is only available as a method of the global document object, and not available as a method on all element objects in the DOM. Because ID values must be unique throughout the entire document, there is no need for "local" versions of the function.
							Using `forEach()` on a NodeList:
							Javascript
							`const cloudySpans = document.querySelectorAll("span.cloudy");`
							`cloudySpans.forEach(span => { console.log("Cloudy!"); });`

Document.createElement()	In an HTML document, the document.createElement() method creates the HTML element specified by tagName, or an HTMLUnknownElement if tagName isn't recognized.	**let element = document.createElement(tagName[, options]);**	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.
				The new Element.		**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); }`
						Hi there and greetings!
						The text above has been created dynamically.	

Element.setAttribute()	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().
		Element.setAttribute(name, value);	**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.
				undefined.	**In the example to the right:**
					The first call to setAttribute() above shows changing the name attribute's value to "helloButton". You can see this using your browser's page inspector (Chrome, Edge, Firefox, Safari).
					To set the value of a Boolean attribute, such as disabled, you can specify any value. An empty string or the name of the attribute are recommended values. All that matters is that if the attribute is present at all, regardless of its actual value, its value is considered to be true. The absence of the attribute means its value is false. By setting the value of the disabled attribute to the empty string (""), we are setting disabled to true, which results in the button being disabled.
						In the following example, `setAttribute()` is used to set attributes on a `<button>`.
						HTML
						`<button>Hello World</button>`
						JavaScript
						`var b = document.querySelector("button");`
						`b.setAttribute("name", "helloButton");`
						`b.setAttribute("disabled", "");`
							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().

Document.createTextNode()	Creates a new Text node. This method can be used to escape HTML characters.	**var text = document.createTextNode(data);** **text is a Text node.**	**data** is a string containing the data to be put in the text node.	None		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, Buried!"); // 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;` `<!DOCTYPE html> <html> <body> Click the button to create a h1 element with some text. <button onclick="myFunction()">Try it</button> <script> function myFunction() { var h = document.createElement("H1"); var t = document.createTextNode("Hello World"); h.appendChild(t); document.body.appendChild(h); } </script> </body> </html>` Click the button to create a h1 element with some text. Try it **Hello World**	

Node.appendChild()	The Node.appendChild() method adds a node to the end of the list of children of a specified parent node. If the given child is a reference to an existing node in the document, appendChild() moves it from its current position to the new position (there is no requirement to remove the node from its parent node before appending it to some other node).
	
	This means that a node can't be in two points of the document simultaneously. So if the node already has a parent, the node is first removed, then appended at the new position. The Node.cloneNode() method can be used to make a copy of the node before appending it under the new parent. Note that the copies made with cloneNode will not be automatically kept in sync.
	
	If the given child is a DocumentFragment, the entire contents of the DocumentFragment are moved into the child list of the specified parent node.
		element.appendChild(aChild)	**aChild**
			The node to append to the given parent node (commonly an element).
				The returned value is the appended child (aChild), except when aChild is a DocumentFragment, in which case the empty DocumentFragment is returned.	The ParentNode.append() method supports multiple arguments and appending strings.	`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, Buried!"); // 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;`
						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.
						
						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.
						`// Create a new paragraph element, and append it to the end of the document body let p = document.createElement("p"); document.body.appendChild(p);`
							Notes
							Chaining may not work as expected, due to `appendChild()` returning the child element:
							`let aBlock = document.createElement('block').appendChild( document.createElement('b') );`
							Sets `aBlock` to `<b></b>` only, which is probably not what you want.

GlobalEventHandlers.onload	The onload property of the GlobalEventHandlers mixin is an EventHandler that processes load events on a Window, XMLHttpRequest, <img> element, etc.
	
	The load event fires when a given resource has loaded.
		target.onload = functionRef;	**Value**
			functionRef is the handler function to be called when the window's load event fires.
					Notes
					The load event fires at the end of the document loading process. At this point, all of the objects in the document are in the DOM, and all the images, scripts, links and sub-frames have finished loading.
					
					There are also DOM Events like DOMContentLoaded and DOMFrameContentLoaded (which can be handled using EventTarget.addEventListener()) which are fired after the DOM for the page has been constructed, but do not wait for other resources to finish loading.
						HTML
						`<!DOCTYPE html> <html> <head> <script type="text/javascript" src="window-load-script.js"></script> </head> <body></body> <html></html> </html>`
						JS
						`window.onload = () => { console.log( "This script loaded when all the resources and the DOM were ready." ); };`
						DOMContentLoaded ensures that a script will run when the document has been loaded without waiting for stylesheets, images and subframes to load. However, if we wanted to wait for everything in the document to load before running the script, we could instead use the window object method
							window.onload fires when the document's window is ready for presentation and document.onload fires when the DOM tree (built from the markup code within the document) is completed.
							
							Ideally, subscribing to DOM-tree events, allows offscreen-manipulations through Javascript, incurring almost no CPU load. Contrarily, window.onload can take a while to fire, when multiple external resources have yet to be requested, parsed and loaded.