DOM – part 2

IIKG1002/IDG1011 – Front-end web development

Johanna Johansen
johanna.johansen@ntnu.no

Accessing and updating the DOM tree involves

1) Locate the node that represents the element you want to work with

2) Use its text content, child elements, and attributes

- To work with the content of elements we can
  - navigate to the text nodes
    - best for when the element contains only text nodes
  - work with the containing element
    - access its text nodes and child elements → if the element contains both text nodes and child elements

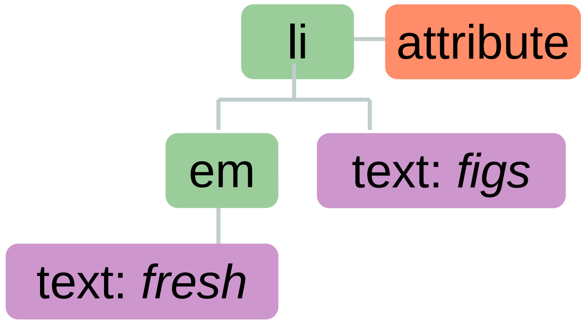
Working with text content

nodeValue

- `nodeValue` → works with the **text node** of an element
- returns the text in a text node
- we can both retrieve and amend its content

- Steps to follow to retrieve/update text content
  1) select first an element node, using methods such as `querySelector()`
  2) use properties such as `firstChild / lastChild` that allows us to locate text nodes
  3) access/update the contents of the text node using `nodeValue`

nodeValue



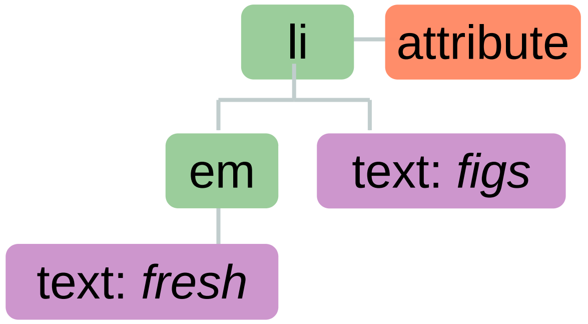
HTML

```
<li id='one'><em>fresh</em> figs</li>
```

JS

```
document.querySelector('#one').firstChild.nextSibling.nodeValue;
// returns the string ' figs'
```

nodeValue



HTML

```
<li id='one'><em>fresh</em> figs</li>
```

JS

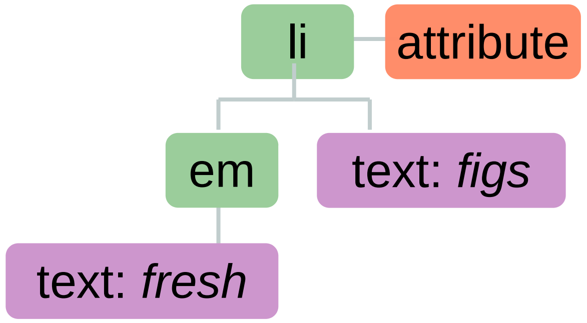
we use the `replace()` method of the String object to replace the text content of our text node

```
let firstItem = document.querySelector('#one');           // gets first list item
let elText = firstItem.firstChild.nextSibling.nodeValue;  // gets its text content
elText = elText.replace(' figs', ' kale');                // change ' figs' to ' kale'
firstItem.firstChild.nextSibling.nodeValue = elText;      // update the list item
```

textContent

- `textContent` → works with **HTML content**
- gets/sets text only
- replaces the content of an element with a text
  - the element node can contain mark-up, not only text
  - the updating string will replace the entire contents (including mark-up)
    - the mark-up is deleted

textContent



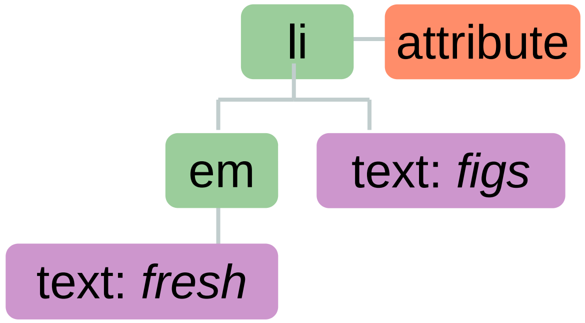
HTML

```
<li id='one'><em>fresh</em> figs</li>
```

JS – **getting** the element's text content

```
document.querySelector('#one').textContent; // returns the
string 'fresh figs'
```

textContent



HTML

```
<li id='one'><em>fresh</em> figs</li>
```

JS – **setting** the content of an element to a string

```
document.querySelector('#one').textContent = 'kale'; // printing
out document.getElementById('#one').textContent; returns 'kale'
```

innerText

- `innerText` → works with **HTML content**
  - similarly to `textContent`
    - gets/sets text only
  - differently from `textContent`
    - it will not show any content that was hidden by CSS
    - reading the value of `innerText` triggers a reflow* to ensure up-to-date computed styles
      - is computationally expensive
  - it is part of the standard: https://html.spec.whatwg.org/multipage/dom.html#the-innertext-idl-attribute
  - **USAGE NOTE:** It is not well specified and not implemented compatibly between browsers and should no longer be used

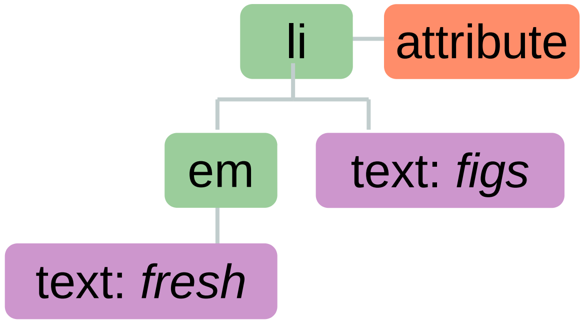
---

* the browser processes and draws part or all of a web page again

innerHTML

- `innerHTML` → works with **HTML content**
- gets/sets text & markup
- to update an element, new content is provided as a string
  - the string can contain markup as well
  - we can add as much markup to the DOM tree as we wish
- to remove all content from an element we can set *innerHTML* to an empty string

innerHTML



HTML

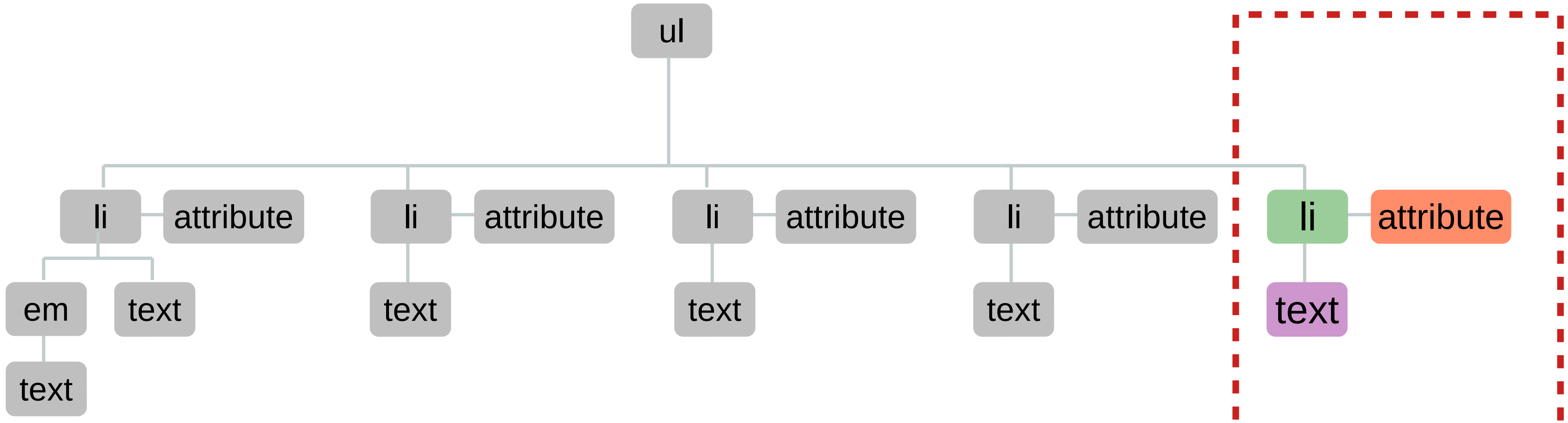
```
<li id='one'><em>fresh</em> figs</li>
```

JS – **getting** the element's content

```
document.querySelector('#one').innerHTML; // returns a string that contains both the text and the mark-up: '<em>fresh</em> figs'
```

```
document.querySelector('#one').textContent; // returns ONLY the the text 'fresh figs'
```

innerHTML


ul
li
attribute
em
text
text
li
attribute
text
li
attribute
text
li
attribute
text
li
attribute
text

innerHTML

initial HTML

```
<ul id='shoppingList'>
  <li id='one' class='hot'><em>fresh</em>
  figs</li>
  <li id='two' class='hot'>pine nuts</li>
  <li id='three' class='hot'>honey</li>
  <li id='four'>balsamic vinegar</li>
</ul>
```

resulting HTML

```
<ul id='shoppingList'>
  <li id='one' class='hot'><em>fresh</em>
  figs</li>
  <li id='two' class='hot'>pine nuts</li>
  <li id='three' class='hot'>honey</li>
  <li id='four'>balsamic vinegar</li>
  <li id='five'>kale</li>
</ul>
```

JS – **setting** the element's content

```
let elContent = document.querySelector('#shoppingList').innerHTML; // we are getting the content of the ul element node

elContent +=  '<li id=\"five\">kale</li>'; // to the initial content we add a fifth element

document.querySelector('#shoppingList').innerHTML = elContent; // we assign the elContent to the ul element node
```

Exercise

initial HTML

```
<ul id='shoppingList'>
  <li id='one' class='hot'><em>fresh</em>
  figs</li>
  <li id='two' class='hot'>pine nuts</li>
  <li id='three' class='hot'>honey</li>
  <li id='four'>balsamic vinegar</li>
</ul>
```

JS – **setting** the element's content

```
let elContent =  "<li
id=\'five\'>kale</li>";

document.querySelector('#shoppingList')
.innerHTML = elContent;
```

Exercise

resulting HTML

```
<ul id='shoppingList'>
   <li id='five'>kale</li>
</ul>
```

NOTE: the new item will overwrite the entire contents of the element, both text and markup

innerHTML

initial HTML

```
<ul id='shoppingList'>
   <li id='one' class='hot'><em>fresh</em>
   figs</li>
   <li id='two' class='hot'>pine nuts</li>
   <li id='three' class='hot'>honey</li>
   <li id='four'>balsamic vinegar</li>
</ul>
```

resulting HTML

```
<ul id='shoppingList'>
   <li id='one' class='hot'><em>fresh</em>
   figs</li>
   <li id='two' class='hot'>pine nuts</li>
   <li id='three' class='hot'>honey</li>
</ul>
```

JS – **setting** the element's content

```
let elContent =  "<li id=\'one\' class=\'hot\'><em>fresh</em> figs</li><li
id=\'two\' class=\'hot\'>pine nuts</li><li id=\'three\'
class=\'hot\'>honey</li>"; // we add all the content we want minus the item we do
not want

document.querySelector('#shoppingList').innerHTML = elContent;
```

innerHTML

- if you want to only to retrieve or write **text** inside an element use `textContent` instead of `innerHTML`
  - `textContent` has better performance because its value is not parsed as HTML
- `innerHTML` is prone to Cross-Site Scripting Attacks (XSS)
  - attackers can place malicious code into a site
  - it should not be used to add content that has come from a user e.g., filled in forms or comments

innerHTML and XSS

- The XSS can give attackers access to information in
  - the DOM, e.g., form data (account data)
  - the website's cookies
  - session tokens: information that identifies one specific user from other users when loged into a site
- This information allows the attacker to
  - make purchases with that account
  - retrieve private information about your users and their behavior/ actions they did on our web site
- The attacker can simply just inject defamatory content in your website

innerHTML and XSS

- You can safely use `innerHTML` to add markup to a page if you have written that code
- Content from any **untrusted sources**
  - should be escaped and added as text, not markup, using properties such as `textContent`
  - limit to where on the page this content will be shown
  - **never** include data from untrusted sources in your JavaScript

innerHTML and XSS

- **Escaping** user content
  - stripping out unwanted data, like malformed HTML or script tags, preventing this data from being seen as code
  - convert possible harmful characters into something that is not harmful for the computer to translate (with e.g., `textContent`)
    - & – &amp; – ampersand
    - < – &lt; – Less than (or open angled bracket)
    - > – &gt; – Greater than (or close angled bracket)

innerHTML and XSS

- **Escaping** user content
  - …
  - there are tools such as https://github.com/cure53/DOMPurify
    - an HTML sanitation library strips anything that could lead to script execution from HTML
    - so you can safely inject complete sets of HTML nodes from a remote source into your DOM
  - `encodeURIComponent()` method can be used to encode user input of URLs
    - encodes the following characters `, / ? : @ & = + $ #`

innerHTML and XSS

- Limit where user content goes
  - the text from users should be placed in elements that are visible in the viewport
  - never place this content inside script tags, HTML comments, as tag names, attributes, or CSS values
  - HTML5 specifies that a <script> tag inserted with *innerHTML* should not execute
    - there are other ways to execute JavaScript without using script tags, such as using the event handler attribute

innerHTML and XSS

- **untrusted sources** → all content that is created by other people then yourself:
  - if your website has users that can add comments or/and create profiles
  - multiple authors contribute to writing articles
  - includes data that comes from third-party sites such as social media content or RSS feeds
  - allows for file uploading such as images and videos

innerHTML and XSS

- content coming from forms
  - validate input from the users so that users supply only the characters they are required, not HTML markup or JavaScript
    - characters such as angled brackets (used in HTML tags, comments), ampersands (&), or parentheses
  - validation should be done both at the UI with HTML / JS
  - and on the server, in case the users have JS turned off
    - most server-side languages offer helper functions that strip out or escape malicious code

DOM manipulation

- The safe approach is to create the nodes separately and assign text content to them using `textContent`
  - **DOM manipulation** methods that can be used to
  - create – `createElement()`,
  - insert – `append(), prepend(), after(), before()` – and
  - delete nodes – `remove(), replaceWith()`
- DOM manipulation
  - though safer, it require more code and thus is slower
  - fit to work with individual nodes
  - `innerHTML` is better suited to updating larger fragments of code
    - we need to write more code to achieve the same thing when using DOM manipulation

Adding element/text nodes

- a set of DOM methods that allow us to add elements to the DOM following the steps:
  1) create one element node at a time, with `createElement()`,
  2) append strings of text or other elements to it with its `append()` and `prepend()` methods
     - we can add as many arguments we want
     - string arguments are automatically converted to *text nodes*

Adding element/text nodes

- `prepend() / append()`
  - used to add nodes to an element at the start / end of the child list
  - both work only on *element nodes* only
- If we want to reuse one node that we have just created
  - we need to copy it first with `cloneNode()`
    - with no argument it copies only the element node
    - with an argument of true, copies also all its content
  - otherwise the node is just moved, instead of copied

Adding element/text nodes

- `before() / after()`
  - they work with both *element* and *text nodes*
  - when we want to insert an *element/text node* into the middle of the containing element's child list
  - we need first to obtain a reference to a sibling node

Adding element/text nodes

- `elNode.remove();`
  - method attached to the *element node* to be removed
- `elNode.replaceWith(newNode);`
  - method attached to the *element node* to be replaced
  - the argument is the new *element/text node* the element will be replaced with

Old generation methods

- In the syllabus book you are presented an older generation of methods for inserting and removing content
  - `createTextNode()`, `appendChild()`, `insertBefore()`, `replaceChild()`, and `removeChild()`
  - they are harder to use
  - they should never be needed

Adding element/text nodes

initial HTML

```
<ul id='shoppingList'>
  <li>balsamic vinegar</li>
  <li>pine nuts</li>
  <li>honey</li>
</ul>
```

resulting HTML

```
<ul id='shoppingList'>
  <li><em>fresh</em> kale</li>
  <li>balsamic vinegar</li>
  <li>pasta</li>
  <li>pine nuts</li>
  <li>honey</li><!-- replaced with
  sugar -->
  <li><em>fresh</em> figs</li>
</ul>
```

Exercise

Starting from this HTML

```
<ul id='shoppingList'>
    <li>balsamic vinegar</li>
    <li>pine nuts</li>
    <li>honey</li>
</ul>
```

resulting HTML

```
<ul id='shoppingList'>
  <li><em>fresh</em> kale</li>
  <li>balsamic vinegar</li>
  <li>pasta</li>
  <li>pine nuts</li>
  <li>honey</li><!-- replaced with sugar -->
  <li><em>fresh</em> figs</li>
</ul>
```

- Add **one** of list elements marked with red in the “resulting HTML” using the methods we have just learned:
  - `createElement()`, and
  - one of the following – depending on which element you want to include – `append(), prepend(), before(), after()`
- Use the tool of your choice for implementing

```
let shoppingList = document.querySelector('#shoppingList'); // the
element node to which we want to append or prepend new elements

let firstElement = document.createElement('li'); // create the
list item which we want to place at the start of the list

let emphasis = document.createElement('em'); // create an emphasis
node element that we want to include in the first list item


emphasis.append('fresh'); // we add the text node that we want to
have emphasized
```

```
firstElement.append(emphasis, ' kale'); // we append the emphasized text and the text ' kale' to the first element

shoppingList.prepend(firstElement); // the element with its children, both text and element nodes are added to the start of the list


// shoppingList.append(firstElement); // if we do this, the element will not be copied but moved at the end of the list


let lastElement = firstElement.cloneNode(); // create the list item which we want to place at the end of the list
```

```
//lastElement.append(emphasis, ' figs');  // if we do this, the element will
not be copied but moved to the new element appended at the end of the list

// The append() method can take as many arguments as we want; it takes two
arguments in this case

lastElement.append(emphasis.cloneNode(true), ' figs'); // we use
cloneNode(true) if we want to copy an already created node with its content
that we want to reuse: <em>fresh</em>

shoppingList.append(lastElement);


let printOuts = document.querySelector('#printOuts');

printOuts.textContent = `Nr. elements in the shopping list:
${shoppingList.querySelectorAll('li').length}`;
```

```
// add another item after the "balsamic vinegar"
let thirdElement = firstElement.cloneNode(); // create the list
item which we want to place as the third element in the list

thirdElement.append('pasta');
let secondElement = document.querySelectorAll('li')[1]; // locate
the element node after which we wish the new element to be added

secondElement.after(thirdElement);
```

```
let fifthElement = document.querySelectorAll('li')[4];
printOuts.textContent += `Fifth element: ${fifthElement.textContent}`; // check the content of the fifth element to see that is 'honey', so that we replace the right item

let newFifthElement = firstElement.cloneNode();
newFifthElement.append('sugar');
fifthElement.replaceWith(newFifthElement);
```

Attribute nodes

- HTML elements consist of a tag name and a set of name/value pairs known as **attributes**

```
<li id="one" class="hot"><em>fresh</em>figs</li>
```

- `getAttribute()`, `setAttribute()`, `hasAttribute()`, and `removeAttribute()`
  - for getting (querying), setting, testing, and removing the attributes of an element

```html
<ul id='shoppingList'>
   <li id='one' class='hot'><em>fresh</em> figs</li>
   <li id='two' class='hot'>pine nuts</li>
   <li id='three' class='hot'>honey</li>
   <li id='four'>balsamic vinegar</li>
</ul>
```

Attribute nodes

- we first query the DOM for an element

```
let firstItem =
document.querySelector('#one');
```

- attach one of the methods such as `getAttribute()` to work with that element's attributes

```
firstItem.getAttribute('class');
```

- `getAttribute()`
  - if the given attribute does not exist, the value returned will either be `null` or `""` (the empty string)
  - pseudo-code:

    ```
    let attribute =
    element.getAttribute(attributeName);
    ```
  - Example:

    ```
    firstItem.getAttribute('class'); // returns 'hot'
    ```

- `setAttribute()`
  - if the attribute already exists, the value is updated
  - otherwise a new attribute is added with the specified name and value.
  - pseudo-code:

    `Element.setAttribute(name, value);` → *name* is the name of the attribute, *value* is the value of the attribute
  - Example:

    ```
    firstItem.setAttribute('class', 'cool');  // the html will
    be <li id="one" class="cool"><em>fresh</em> figs</li>
    ```

- `hasAttribute()`
  - returns a Boolean value (true or false) indicating whether the specified element has the specified attribute or not
  - pseudo-code:

    `let result = element.hasAttribute(name);` → *name* is the name of the attribute
  - Example:

    ```
    firstItem.hasAttribute('class'); // returns true
    ```

- `removeAttribute()`
  - removes the attribute with the specified name from the element
  - pseudo-code:

    ```
    element.removeAttribute(attrName);
    ```
  - Example:

    ```
    firstItem.removeAttribute('class');  // the html will now be <li id="one"><em>fresh</em> figs</li>
    ```

Attribute nodes

- Good practice
  - check first with `hasAttribute()` if the respective attribute exists, before we work with the respective attribute
    - this saves resources if the attribute cannot be found

```
let firstItem =  document.querySelector('#one');
if (firstItem.hasAttribute('class')) {
    // do something with the respective attribute
}
```

Attribute nodes

- the DOM treats each HTML element as an object in the DOM tree
- the attributes of HTML element correspond to the properties of the object
- therefore we can use the property names on the element to get and set values for the attributes

Attribute nodes

- some of the attribute names are reserved words in JavaScript
  - the name of the property is usually prefixed with html
    - for attribute of the <label> element → `htmlFor`
  - exceptions
    - class attribute → `className`
    - value attribute of the <input> (the user's current input) → `defaultValue`
    - checked attribute of the <input> for a checkbox or radio button → `defaultChecked`
- if the attribute is more than one word long the lowerCamelCase rule is used
  - tabindex → use to indicate that its element can be focused, and where it participates in sequential keyboard navigation (usually with the Tab key, hence the name) → `tabIndex`
  - event handlers such as `onclick` are exceptions from this rule

Attribute nodes

- the `className` property
  - returns the value of the class attribute as a *string*
  - its value can be a list of classes, not only one value

```
let cName = elementNodeReference.className;
elementNodeReference.className = cName;
```

  - `cName` is a string variable representing the value of the class attribute or space-separated values of the class attribute of the current element

- for cases where we want to add and remove individual classes from a list of classes, use the `classList` property instead

Attribute nodes

- `classList` property
  - allows you to treat the class attribute as a list
  - it is an iterable Array like object
  - `add(), remove(), contains(), toggle()`

Attribute nodes

`add()`

```
add(token0);
```

```
add(token0, token1);
```

```
add(token0, token1, /* ... ,*/
tokenN)
```

can add one or more tokens

adds the given tokens* to the list, omitting any that are already present

----
* Tokens are the smallest individual words, phrases, or characters that JavaScript can understand.

HTML:

```
<span class="a b c"></span>
```

JavaScript:

```
let span = document.querySelector("span");

let classes = span.classList;

classes.add("d");

span.textContent = classes;
```

Output:

```
a b c d
```

Attribute nodes

```
remove()
remove(token);
remove(token, token);
remove(token, token, token);
...
```

`token`

- a string representing the token we want to remove from the list
- if the string is not in the list, no error is thrown, and nothing happens

HTML

```
<div id="ab" class="a b c"></div>
```

JavaScript

```
let span = 
document.getElementById("ab");
let classes = span.classList;
classes.remove("c"); // remove several 
with classes.remove("c", "b");
span.textContent = classes;
```

Output

```
a b
```

Attribute nodes

`contains()`

`contains(token);`

Returns true if the list contains the given token, otherwise false.

HTML

```
<span class="a b c"></span>
```

JavaScript

```
let span = document.querySelector("span");
let classes = span.classList;
if (classes.contains("c")) {
  span.textContent = "The classList contains 'c'";
} else {
  span.textContent = "The classList does not contain 'c'";
}
```

Output

```
The classList contains 'c'
```

Attribute nodes

`toggle()`

`toggle(token);`

- removes an existing token from the list and returns false
- if the token doesn't exist it's added and the function returns true
- useful for toggling class names based on user interaction with the page, such as clicking on an icon to expand/hide details

HTML

```
<span class="a b">classList is 'a b'</span>
```

JavaScript

```
let span = document.querySelector("span");
let classes = span.classList;

span.addEventListener('click', function() {
  let result = classes.toggle("c");

  if (result) {
    span.textContent = `'c' added; classList is
now "${classes}".`;
  } else {
    span.textContent = `'c' removed; classList is
now "${classes}".`;
  }
})
```

Output

classList is 'a b'

- it will change each time we click on the text to add/remove c