UNIVERSITÀ DI PARMA

Client-side JavaScript

Execution
Window Object
DOM

Tecnologie Internet
a.a. 2022/2023

Execution

Client-side JavaScript in a nutshell

The **Window object** (identifier: `window`) is the main entry point to all client-side JavaScript features and APIs. It represents a web browser window or frame. We are going to explore its properties, methods and constructurs.

Then, we will see techniques for querying, traversing and modifying document content, for scripting CSS and for handling events. The combination of scriptable content, presentation and behavior is called **Dynamic HTML (DHTML)**.

Most importantly, we will focus on the development of **web applications**, i.e., web pages that use JavaScript to access the more advanced services offered by browsers - networking, graphics and data storage.

Web applications

JavaScript is more central to web applications that it is to web documents.

JavaScript enhances web documents, but a well-designed document will continue to work with JavaAcript disabled.

Web applications are, by definition, JavaScript programs that use the OS-type services provided by the web browser, and they would not be expected to work with JavaScript disabled.

Synchronous, asynchronous and deferred scripts

When the HTML parser of the web browser encounters a `<script>` element, it must, by default, run the script before it can resume parsing and rendering the document.

This is not much of a problem for the inline scripts, but, if the script source code is in an external file specified with a `src` attribute, this means that the portions of the document that follow the script will not appear until the script has been downloaded and executed. Only these **synchronous scripts** can use `document.write()` to insert text into the input stream.

The `<script>` tag can have **`defer`** and **`async`** attributes, which cause scripts to be executed differently. The `defer` attribute causes the browser to defer execution of the script until after the document has been loaded and parsed and is ready to be manipulated. The `async` attribute causes the browser to run the script asap but not to block document parsing.

Threading model

The core JavaScript language does not contain any threading mechanism. Thus, two event handlers will never run at the same time.

Single-threaded execution means that web browsers must stop responding to user input while scripts and event handlers are executing. This means that scripts and event handlers must not run for too long.

If the application must perform enough computation to cause a noticeable delay, you should allow the document to load fully before performing the computation, and you should be sure to notify the user that computation is underway and that the browser is not hung (e.g., by means of a progress indicator).

Client-side JavaScript timeline

1. The web browser creates a **Document object** and begins parsing the web page, adding **Element objects** and **Text nodes** to the document as it parses the HTML elements and their textual content. The `document.readyState` property has the value "loading" at this stage.

2. When the HTML parser encounters `<script>` elements that have neither the `async` nor `defer` attributes, it adds those elements to the document and executes the inline or external scripts. Synchronous scripts often simply define functions and register event handlers for later use, but they can traverse and manipulate the document tree *as it exists* when they run.

Client-side JavaScript timeline

3. When the parser encounters a `<script>` element that has the `async` attribute set, it begins downloading the script text and continues parsing the document. The script will be executed asap after it has been downloaded.

4. When the document is completely parsed, the `document.readyState` property changes to “interactive”.

5. Any script that had the `defer` attribute set are executed, in the order in which they appeared in the document. Deferred scripts have access to the complete document tree and must not use the `document.write()` method.

Client-side JavaScript timeline

6. The browser fires a **DOMContentLoaded event** on the Document object. This marks the transition from **synchronous script execution phase** to the **asynchronous event-driven phase** of program execution. Note, however, that there may still be `async` scripts that have not yet executed at this point.

7. The document is completely parsed, but the browser may still be waiting for additional content (e.g., images) to load. When all such content finishes loading and all `async` scripts have executed, `document.readyState` changes to "complete".

What JavaScript can't do

Client-side JavaScript **does not provide any way to write or delete arbitrary files or list arbitrary directories on the client computer**.

However, we will see how JavaScript can obtain a **secure private filesystem** within which it can read and write files.

The same-origin policy

The same-origin policy governs the interactions of JavaScript code in one window or frame with the content of other windows or frames. Specifically, **a script can read only the properties of windows and documents that have the same origin (protocol, host and port of the URL) as the document that contains the script**.

Example:

a script hosted by host A is included in a web page served by host B; the origin of the script is host B and the script has full access to the content of the document that contains it;

if the script opens a new window and loads a second document from host B, the script has full access to the content of that second document;

but if the script opens a third window and loads a document from host C (or even one from host A) into it, the same-origin policy comes into effect and prevents the script from accessing this document.

The same-origin policy

The same-origin policy applies to scripted HTTP requests made with the **XMLHttpRequest object**, which allows client-side JavaScript code to make arbitrary HTTP requests to the web server from which the containing document was loaded, but it does not allow scripts to communicate with other web servers.

For cross-origin requests, resource owner and clients have to enable Cross-Origin Resource Sharing.

The CORS standard is now included in the **Fetch** specification:

https://fetch.spec.whatwg.org/

Window Object

The Window object in a nutshell

The Window object is the global object for client-side JavaScript programs. Its main properties are related to

1) Timers
2) Browser location and navigation
3) Browsing history
4) Browser and screen information
5) Dialog boxes
6) Document elements
7) Cross-window communication

Timers

The **`setTimeout()`** method of the Window object schedules a function to run after a specified number of milliseconds elapses.

`setTimeout()` returns a value that can be passed to **`clearTimeout()`** to cancel the execution of the scheduled function.

`setInterval()` is like `setTimeout()` except that the specified function is invoked repeatedly at intervals in the specified number of milliseconds:

`setInterval(updateClock, 60000);` // call updateClock() every 60s

`setInterval()` returns a value that can be passed to **`clearInterval()`** to cancel the execution of the scheduled function.

Timers

```
/*
 * Schedule an invocation or invocations of f() in the future.
 * Wait start milliseconds, then call f() every interval milliseconds,
 * stopping after a total of start+end milliseconds.
 * If interval is specified but end is omitted, then never stop invoking f.
 * If both interval and end are omitted, then invoke f once after start ms.
 * If only f is specified, behave as if start was 0.
 * Note that the call to invoke() does not block: it returns right away.
 */
function invoke(f, start, interval, end) {
    if (!start) start = 0;          // Default to 0 ms
    if (arguments.length <= 2)      // Single-invocation case
        setTimeout(f, start);       // Single invocation after start ms.
    else {                          // Multiple invocation case
        setTimeout(repeat, start);  // Repetitions begin in start ms
        function repeat() {         // Invoked by the timeout above
            let h = setInterval(f, interval); // Invoke f every interval ms.
            // And stop invoking after end ms, if end is defined
            if (end) setTimeout(function() { clearInterval(h); }, end);
        }
    }
}
```

UNIVERSITÀ DI PARMA

Browser location and navigation

The `location` property of the Window object refers to a **Location object**, which represents the current URL of the document displayed in the window, and which also defines methods for making the window load a new document.

The **`href`** property of the Location object is a string that contains the complete text of the URL.

Other properties of the Location object are: **`protocol`**, **`host`**, **`hostname`**, **`port`**, **`pathname`**, **`search`** and **`hash`**.

The `hash` property returns the “fragment identifier” portion of the URL, if there is one: a hash mark (#) followed by an element ID. The `search` property returns the portion of the URL that starts with a question mark (?): often some sort of query string.

Browser location and navigation

```
/*
 * This function parses &-separated name=value argument pairs from
 * the query string of the URL. It stores the name=value pairs in
 * properties of an object and returns that object.
 * Use it like this:
 * let args = urlArgs();  // Parse args from URL
 */
function urlArgs() {
    let args = {};                              // Start with an empty object
    let query = location.search.substring(1);   // Get query string, minus '?'
    let pairs = query.split("&");               // Split at ampersands
    for(let i = 0; i < pairs.length; i++) {     // For each fragment
        let pos = pairs[i].indexOf('=');        // Look for "name=value"
        if (pos == -1) continue;                // If not found, skip it
        let name = pairs[i].substring(0,pos);   // Extract the name
        let value = pairs[i].substring(pos+1);  // Extract the value
        value = decodeURIComponent(value);      // Decode the value
        args[name] = value;                     // Store as a property
    }
    return args;                                // Return the parsed arguments
}
```

Browser location and navigation

The **`assign()`** method of the Location object makes the window load and display the document at the specified URL.

The **`replace()`** method is similar, but it removes the current document from the browsing history before loading the new document.

The **`reload()`** method makes the browser reload a document.

Alternative navigation approaches:

```
location = “http://www.oreilly.com”;
location = “page2.html”;
location = “#top”;
location.search = “?page=” + (pagenum+1);
```

Browsing history

The **`history`** property of the Window object refers to the **History object**, which models the browsing history of the window as a list of documents and document states.

The **`length`** property of the History object specifies the number of elements in the browsing history list.

The History object has **`back()`** and **`forward()`** methods that make the browser go backward or forward one step in its browsing history (unless the *same-origin policy* prevents it).

The **`go()`** method takes an integer argument and can skip any number of pages forward (for positive arguments) or backward (for negative arguments) in the history list:

```
history.go(-2);
```

Browser and screen information

The **`navigator`** property of a Window object refers to a **Navigator object** that contains browser vendor and version number information.

In the past, the Navigator object was commonly used by scripts to determine if they were running in Internet Explorer or Netscape.

Today, **feature testing** is preferred: rather than making assumptions about particular browser versions and their features, it is better to simply test for the needed feature (i.e., method or property).

Browser and screen information

The Navigator object has four properties that provide information about the browser that is running:

`appName`: the full name of the web browser (always Netscape, because browser detection is bad practice; use feature detection instead)

`appVersion`: browser vendor and version information (the format is not standard)

`userAgent`: the string that the browser sends in its USER-AGENT HTTP header; contains all the information in `appVersion` and additional details, possibly (the format is not standard)

`platform`: a string that identifies the operating system (and possibly the hardware) on which the browser is running

Browser and screen information

Other nonstandard properties of the Navigator object are:

`onLine`: specifies whether the browser is currently connected to the network

`geolocation`: a Geolocation object that defines an API for determining the user's geographical location

`javaEnabled()`: should return `true` if the browser can run Java applets

`cookieEnabled`: should be `true` if the browser can store persistent cookies

Browser and screen information

The **`screen`** property of a Window object refers to a **Screen object** that provides information about the size of the user's display and the number of colors available on it.

The **`width`** and **`height`** properties specify the size of the display in pixels.

The **`availWidth`** and **`availHeight`** properties specify the display size that is actually available (they exclude the space required by features such as a desktop taskbar).

The **`colorDepth`** property specifies the bits-per-pixel value of the screen.

Dialog boxes

The Window object provides three methods for displaying simple dialog boxes to the user (to be used for *debugging*, mostly).

`alert()` displays a message to the user and waits for the user to dismiss the dialog

`confirm()` displays a message, waits for the user to click an OK or Cancel button and returns a boolean value

`prompt()` displays a message, waits for the user to enter a string, and returns that string

```
<script>
do {
let name = prompt("What is your name?");
let correct = confirm("You entered '" + name + "'.\n" + "Click Okay to
proceed or Cancel to re-enter.");
} while (!correct)

alert("Hello, " + name);
</script>
```

Document elements

Every element of an HTML document provided with the **`id`** attribute becomes a Window object’s property, whose name is the value of the `id` attribute and whose value is the **HTMLElement object** representing that document element (unless the Window object already has a property by that name).

However, it is recommended to use **`document.getElementById()`** to look up elements explicitly. The use of this method is less onerous if we give it a simpler name:

```
let $ = function(id) { return document.getElementById(id); };
ui.prompt = $(“prompt”);
```

```
<div id="foo">Foo<div>
```

`window.document.getElementById("foo") === window.foo` ⟶ `returns true`

Cross-window communication

A single web browser window may contain several tabs, each one being an independent browsing context, **with its own Window object**.

A script in one browser window or tab can open new browser windows or tabs, and when a script does this, the windows can interact with one another and with one another’s documents (subject to the constraints of the *same-origin policy*).

Opening and closing windows

The `open()` method of the Window object loads a specified URL into a new or existing browsing context (window, iframe or tab) and returns the Window object that represents the window. It takes three optional arguments:

- the URL

- a window name

- a comma-separated list of size and feature attributes for the new window to be opened

The `close()` method closes the window it is invoked on.

UNIVERSITÀ DI PARMA

Opening and closing windows

```
<!DOCTYPE html>
<html>

<body>

<p>Click the button to open a new browser window.</p>

<button onclick="myFunction()">Try it</button>

<script>
function myFunction() {
    let w = window.open("https://www.w3schools.com", "smallwin",
   "width=400,height=350,status=yes,resizable=yes", false);
}
</script>

</body>

</html>
```

Accessing an iframe contents

`iframe.contentWindow` is a reference to the window object of an <iframe>.

`iframe.contentDocument` is a reference to the document object of an <iframe>.

When we try to access an embedded document, the browser checks if the iframe has the same origin of the document that executes the code. If not, then the access is denied.

An example with the same origin:

```
<iframe src="/" id="iframe"></iframe>

<script>
  iframe.onload = function() {
    // just do anything
    iframe.contentDocument.body.prepend("Hello, world!");
  };
</script>
```

JavaScript in interacting windows

Suppose a window has two iframes named “A” and “B”, containing documents from the same server.

Suppose a script in iframe A defines a variable i:

```
let i = 3;
```

Then, a script in iframe B can refer to that variable by means of the **`parent`** property of the Window object associated to iframe B:

```
parent.A.i = 4; // the script in iframe B changes the value of i in iframe A
```

Similarly, the script in iframe A can refer to a function defined in iframe B:

```
parent.B.f(); // the script in iframe A invokes function f() defined in iframe B
```

DOM

The HTML DOM (Document Object Model)

The **DOM** is a W3C standard for accessing documents:

"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

The W3C DOM standard is separated into 3 different parts:

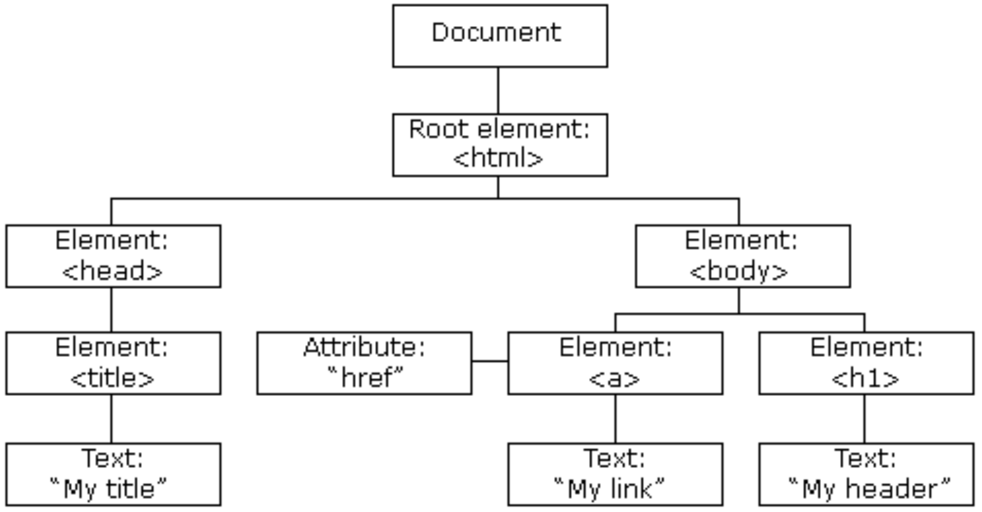
- **Core DOM** - standard model for all document types
- **XML DOM** - standard model for XML documents
- **HTML DOM** - standard model for HTML documents

The HTML DOM (Document Object Model)

The HTML DOM is a standard object model and **programming interface** for HTML. It defines:

- The HTML elements as **objects**, organized in a tree
- The **properties** of all HTML elements
- The **methods** to access all HTML elements
- The **events** for all HTML elements

In other words: **the HTML DOM is a standard for how to get, change, add, or delete HTML elements.**



The HTML DOM (Document Object Model)

With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements in the page
- JavaScript can change all the HTML attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

UNIVERSITÀ DI PARMA

HTML DOM programming interface

The HTML DOM can be accessed with JavaScript (and with other programming languages).
In the DOM, all HTML elements are defined as **objects**.
The programming interface is the properties and methods of each object.

A **property** is a value that you can get or set (like changing the content of an HTML element).

A **method** is an action you can do (like add or deleting an HTML element).

HTML DOM programming interface

To access any element in an HTML page, always start with accessing the `document` object.

The following example changes the content (the innerHTML) of the `<p>` element with `id="demo"`:

```
<html>
<body>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>

</body>
</html>
```

HTML DOM programming interface

The most common way to access an HTML element is to use the **`id`** of the element.
In the example above the **`getElementById()`** method used `id`="demo" to find the element.

The easiest way to get the content of an element is by using the **`innerHTML`** property.
The `innerHTML` property is useful for getting or replacing the content of HTML elements.

UNIVERSITÀ DI PARMA

HTML DOM programming interface

To find HTML elements:

Method	**Description**
document.getElementById()	Find an element by element id
document.getElementsByTagName()	Find elements by tag name
document.getElementsByClassName()	Find elements by class name

To change HTML elements:

Method	**Description**
element.innerHTML=	Change the inner HTML of an element
element.attribute=	Change the attribute of an HTML element
element.setAttribute*(attribute,value)*	Change the attribute of an HTML element
element.style.*property*=	Change the style of an HTML element

HTML DOM programming interface

To add and delete elements:

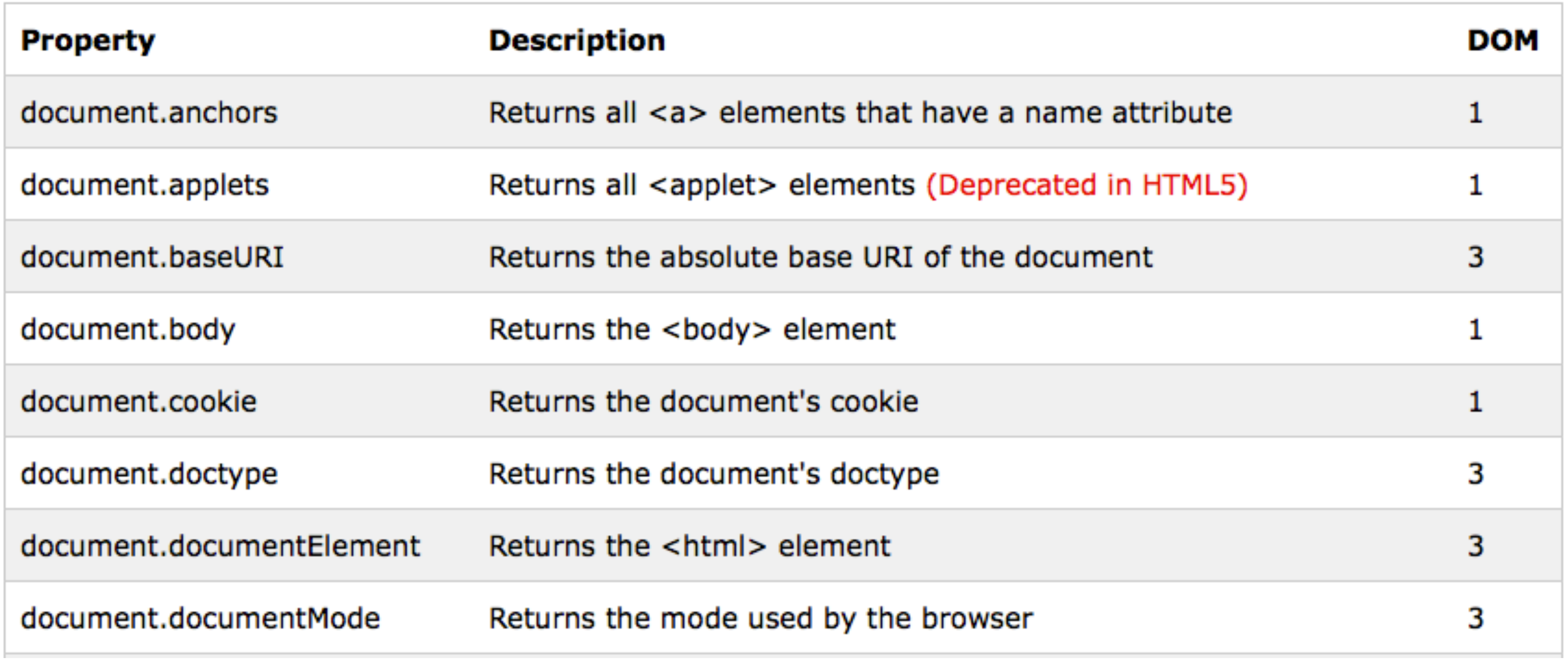
Method	**Description**
document.createElement()	Create an HTML element
document.removeChild()	Remove an HTML element
document.appendChild()	Add an HTML element
document.replaceChild()	Replace an HTML element
document.write(*text*)	Write into the HTML output stream

To add event handlers:

Method	**Description**
document.getElementById(*id*).onclick=function(){*code*}	Adding event handler code to an onclick event

HTML DOM programming interface

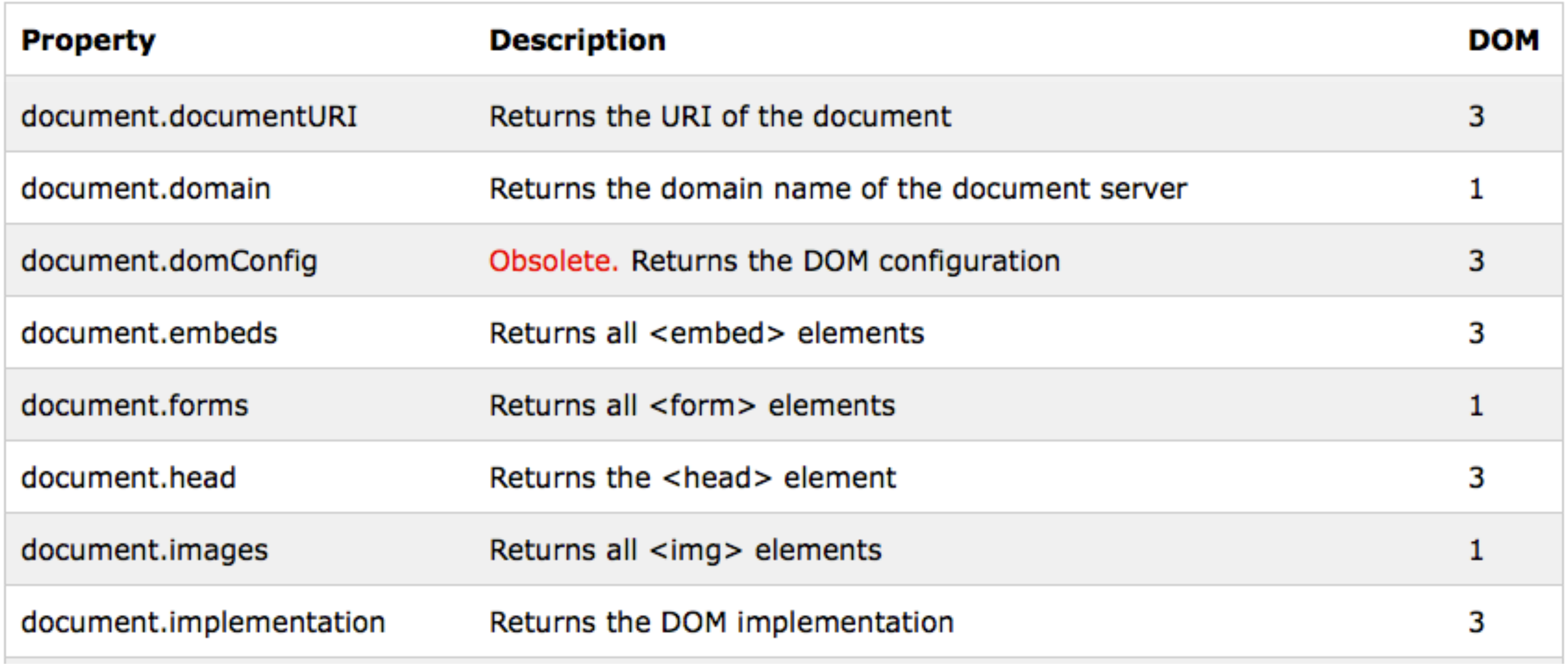
To find HTML objects:



Property	Description	DOM
document.anchors	Returns all <a> elements that have a name attribute	1
document.applets	Returns all <applet> elements (Deprecated in HTML5)	1
document.baseURI	Returns the absolute base URI of the document	3
document.body	Returns the <body> element	1
document.cookie	Returns the document's cookie	1
document.doctype	Returns the document's doctype	3
document.documentElement	Returns the <html> element	3
document.documentMode	Returns the mode used by the browser	3

HTML DOM programming interface

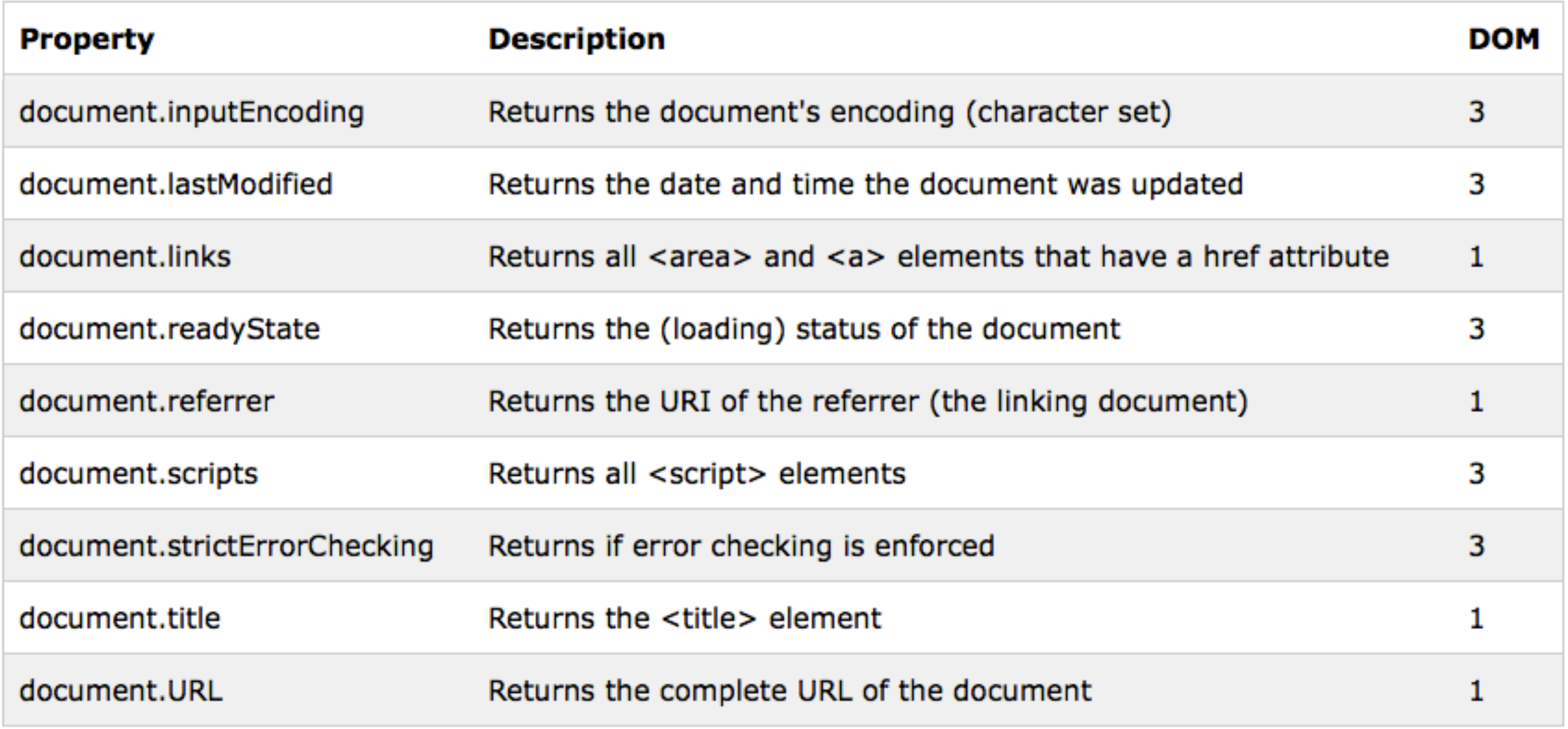
To find HTML objects:



Property	Description	DOM
document.documentURI	Returns the URI of the document	3
document.domain	Returns the domain name of the document server	1
document.domConfig	Obsolete. Returns the DOM configuration	3
document.embeds	Returns all <embed> elements	3
document.forms	Returns all <form> elements	1
document.head	Returns the <head> element	3
document.images	Returns all <img> elements	1
document.implementation	Returns the DOM implementation	3

HTML DOM programming interface

To find HTML objects:



Property	Description	DOM
document.inputEncoding	Returns the document's encoding (character set)	3
document.lastModified	Returns the date and time the document was updated	3
document.links	Returns all <area> and <a> elements that have a href attribute	1
document.readyState	Returns the (loading) status of the document	3
document.referrer	Returns the URI of the referrer (the linking document)	1
document.scripts	Returns all <script> elements	3
document.strictErrorChecking	Returns if error checking is enforced	3
document.title	Returns the <title> element	1
document.URL	Returns the complete URL of the document	1

Finding HTML elements

The easiest way to find an HTML element in the DOM, is by using the element **`id`**.

This example finds the element with `id="intro"`:

```
let myElement = document.getElementById("intro");
```

If the element is found, the method will return the element as an object (in myElement).
If the element is not found, myElement will contain null.

Finding HTML elements

Another approach to find an HTML element in the DOM, is by using the tag name.

This example finds all `<p>` elements:

```
let x = document.getElementsByTagName("p");
```

This example finds the element with `id="main"`, and then finds all `<p>` elements inside `"main"`:

```
let x = document.getElementById("main");
let y = x.getElementsByTagName("p");
```

Finding HTML elements

To find all HTML elements with the same class name, use **`getElementsByClassName()`**.
This example returns a list of all elements with `class="intro"`.

```
let x = document.getElementsByClassName("intro");
```

To find all HTML elements that matches a specified CSS selector (id, class names, types, attributes, values of attributes, etc), use the **`querySelectorAll()`** method.
This example returns a list of all <p> elements with class="intro".

```
let x = document.querySelectorAll("p.intro");
```

The `querySelectorAll()` method does not work in Internet Explorer 8 and earlier versions.

Changing the HTML output stream

In JavaScript, `document.write()` can be used to write directly to the HTML output stream:

```
<!DOCTYPE html>
<html>
<body>

<script>
document.write(Date());
</script>

</body>
</html>
```

Never use `document.write()` after the document is loaded. It will overwrite the document.

Changing HTML content

The easiest way to modify the content of an HTML element is by using the `innerHTML` property.
To change the content of an HTML element, use this syntax:

```
document.getElementById(id).innerHTML = new HTML
```

This example changes the content of a <p> element:

```
<html>
<body>

<p id="p1">Hello World!</p>
<script>
document.getElementById("p1").innerHTML = "New text!";
</script>

</body>
</html>
```

Changing HTML content

This example changes the content of an <h1> element:

```
<!DOCTYPE html>
<html>
<body>

<h1 id="header">Old Header</h1>
<script>
let element = document.getElementById("header");
element.innerHTML = "New Header";
</script>

</body>
</html>
```

- The HTML document above contains an <h1> element with id="header"
- We use the HTML DOM to get the element with id="header"
- A JavaScript changes the content (innerHTML) of that element

Changing the value of an attribute

To change the value of an HTML attribute, use this syntax:

```
document.getElementById(id).attribute=new value
```

This example changes the value of the src attribute of an <img> element:

```
<!DOCTYPE html>
<html>
<body>

<img id="myImage" src="smiley.gif">
<script>
document.getElementById("myImage").src = "landscape.jpg";
</script>

</body>
</html>
```

Changing CSS

The HTML DOM allows JavaScript to change the style of HTML elements. Use this syntax:

```
document.getElementById(id).style.property=new style
```

The following example changes the style of a <p> element:

```
<html>
<body>

<p id="p2">Hello World!</p>
<script>
document.getElementById("p2").style.color = "blue";
</script>
<p>The paragraph above was changed by a script.</p>

</body>
</html>
```

Changing CSS

This example changes the style of the HTML element with id="id1", when the user clicks a button:

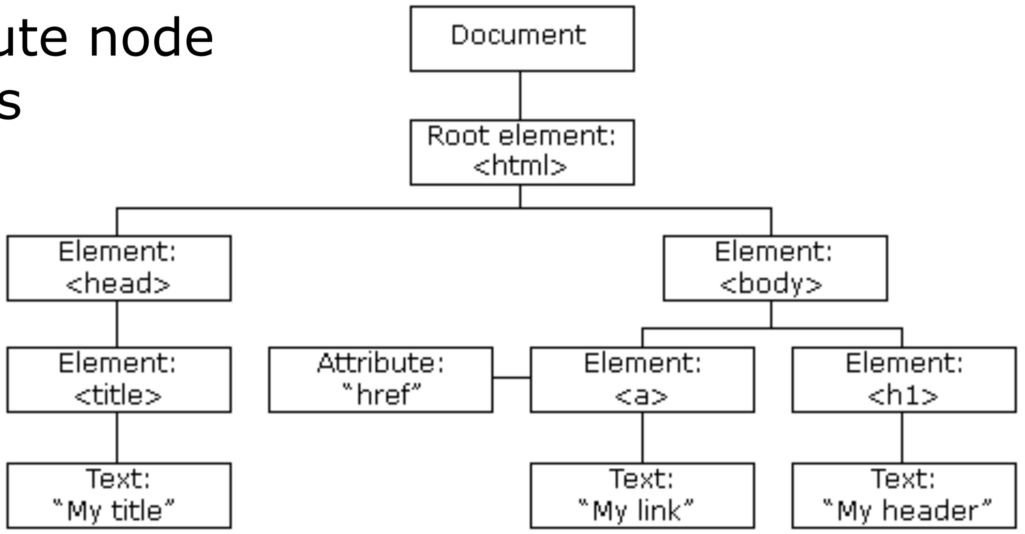
```
<!DOCTYPE html>
<html>
<body>

<h1 id="id1">My Heading 1</h1>

<button type="button"
onclick="document.getElementById('id1').style.color = 'red'">
Click Me!</button>

</body>
</html>
```

HTML DOM Navigation

With the HTML DOM, you can navigate the node tree using node relationships.

According to the W3C HTML DOM standard, **everything in an HTML document is a node**:

- The entire document is a document node
- Every HTML element is an element node
- The text inside HTML elements are text nodes
- Every HTML attribute is an attribute node
- All comments are comment nodes



HTML DOM Navigation

The nodes in the node tree have a hierarchical relationship to each other.

The terms parent, child, and sibling are used to describe the relationships.

- In a node tree, the top node is called the **root** (or root node)
- Every node has exactly one **parent**, except the root (which has no parent)
- A node can have a number of **children**
- **Siblings** (brothers or sisters) are nodes with the same parent

HTML DOM Navigation

```
<html>

  <head>
    <title>DOM Tutorial</title>
  </head>

  <body>
    <h1>DOM Lesson one</h1>
    <p>Hello world!</p>
  </body>

</html>
```

- <html> is the root node
- <html> has no parents
- <html> is the parent of <head> and <body>
- <head> is the first child of <html>
- <body> is the last child of <html>
- <head> has one child: <title>
- <title> has one child (a text node): "DOM Tutorial"
- <body> has two children: <h1> and <p>
- <h1> has one child: "DOM Lesson one"
- <p> has one child: "Hello world!"
- <h1> and <p> are siblings

HTML DOM Navigation

To navigate between nodes with JavaScript, use the following node properties:

- `parentNode`
- `childNodes[`*`nodenumber`*`]`
- `firstChild`
- `lastChild`
- `nextSibling`
- `previousSibling`

A common error in DOM processing is to expect an element node to contain text.
In this example: **<title>DOM Tutorial</title>**, the element node <title> does not contain text. It contains a **text node** with the value "DOM Tutorial".
The value of the text node can be accessed by the node's **`innerHTML`** property, or the **`nodeValue`**.

HTML DOM Navigation

In addition to the `innerHTML` property, you can also use the **`childNodes`** and nodeValue properties to get the content of an element.
The following example collects the node value of an <h1> element and copies it into a <p> element:

```
<html>
<body>

<h1 id="intro">My First Page</h1>
<p id="demo">Hello!</p>
<script>
let myText =
document.getElementById("intro").childNodes[0].nodeValue;
document.getElementById("demo").innerHTML = myText;
</script>

</body>
</html>
```

`getElementById()` is a method, while `childNodes` and `nodeValue` are properties

HTML DOM Navigation

Using the **`firstChild`** property is the same as using **`childNodes[0]`**:

```
<html>
<body>

<h1 id="intro">My First Page</h1>
<p id="demo">Hello World!</p>
<script>
myText = document.getElementById("intro").firstChild.nodeValue;
document.getElementById("demo").innerHTML = myText;
</script>

</body>
</html>
```

HTML DOM Navigation

There are two special properties that allow access to the full document:

- **`document.body`** - The body of the document
- **`document.documentElement`** - The full document

```
<html>
<body>

<p>Hello World!</p>
<div>
<p>The DOM is very useful!</p>
<p>This example demonstrates the <b>document.body</b>
property.</p>
</div>
<script>
alert(document.body.innerHTML);
</script>

</body>
</html>
```

HTML DOM Navigation

```
<html>
<body>

<p>Hello World!</p>
<div>
<p>The DOM is very useful!</p>
<p>This example demonstrates the <b>document.documentElement</b>
property.</p>
</div>

<script>
alert(document.documentElement.innerHTML);
</script>

</body>
</html>
```

HTML DOM Navigation

The **`nodeName`** property specifies the name of a node.

- `nodeName` is read-only
- `nodeName` of an element node is the same as the tag name
- `nodeName` of an attribute node is the attribute name
- `nodeName` of a text node is always #text
- `nodeName` of the document node is always #document

Note: `nodeName` always contains the uppercase tag name of an HTML element.

The **`nodeValue`** property specifies the value of a node.

- `nodeValue` for element nodes is undefined
- `nodeValue` for text nodes is the text itself
- `nodeValue` for attribute nodes is the attribute value

HTML DOM Navigation

The **`nodeType`** property returns the type of node. nodeType is read only.
The most important node types are:

Element type	NodeType
Element	1
Attribute	2
Text	3
Comment	8
Document	9

Adding and removing HTML elements

To add a new element to the HTML DOM, you must create the element (element node) first, and then append it to an existing element.

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
let para = document.createElement("p");
let node = document.createTextNode("This is new.");
para.appendChild(node);

let element = document.getElementById("div1");
element.appendChild(para);
</script>
```

Adding and removing HTML elements

The previous code creates a new <p> element:

```
let para = document.createElement("p");
```

To add text to the <p> element, create a text node first. This code creates a text node:

```
let node = document.createTextNode("This is a new paragraph.");
```

Then append the text node to the <p> element:

```
para.appendChild(node);
```

Finally, append the new element to an existing element. This code finds an existing element and appends the new element to it:

```
let element = document.getElementById("div1");
element.appendChild(para);
```

Adding and removing HTML elements

In the previous example, the **`appendChild()`** method appended the new element as the last child of the parent.
Alternatively, you can use the **`insertBefore()`** method:

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
let para = document.createElement("p");
let node = document.createTextNode("This is new.");
para.appendChild(node);

let element = document.getElementById("div1");
let child = document.getElementById("p1");
element.insertBefore(para,child);
</script>
```

UNIVERSITÀ DI PARMA

Adding and removing HTML elements

To remove an HTML element, you must know the parent of the element:

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
let parent = document.getElementById("div1");
let child = document.getElementById("p1");
parent.removeChild(child);
</script>
```

Adding and removing HTML elements

This HTML document contains a `<div>` element with two child nodes (two `<p>` elements):

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>
```

Find the element with `id="div1"`:

```
let parent = document.getElementById("div1");
```

Find the `<p>` element with `id="p1"`:

```
let child = document.getElementById("p1");
```

Remove the child from the parent:

```
parent.removeChild(child);
```

Adding and removing HTML elements

It would be nice to be able to remove an element without referring to the parent.

Unfortunately, this is not possible: the DOM needs to know both the element you want to remove, and its parent.

Here is a common workaround: find the child to be removed, and use its **`parentNode`** property to find the parent:

```
let child = document.getElementById("p1");
child.parentNode.removeChild(child);
```

UNIVERSITÀ DI PARMA

Adding and removing HTML elements

To replace an element to the HTML DOM, use the **`replaceChild()`** method:

```
<div id="div1">
<p id="p1">This is a paragraph.</p>
<p id="p2">This is another paragraph.</p>
</div>

<script>
let para = document.createElement("p");
let node = document.createTextNode("This is new.");
para.appendChild(node);

let parent = document.getElementById("div1");
let child = document.getElementById("p1");
parent.replaceChild(para,child);
</script>
```

HTML DOM Node List

The **`getElementsByTagName()`** method returns a **node list**. A node list is an array-like collection of nodes.
The following code selects all <p> nodes in a document:

```
let x = document.getElementsByTagName("p");
```

The nodes can be accessed by an index number:

```
y = x[1];
```

The **`length`** property defines the number of nodes in a node list. In the following example, we get all <p> elements in a node list and we display the length of the node list:

```
let myNodelist = document.getElementsByTagName("p");
document.getElementById("demo").innerHTML = myNodelist.length;
```

HTML DOM Node List

The `length` property is useful when you want to loop through the nodes in a node list.

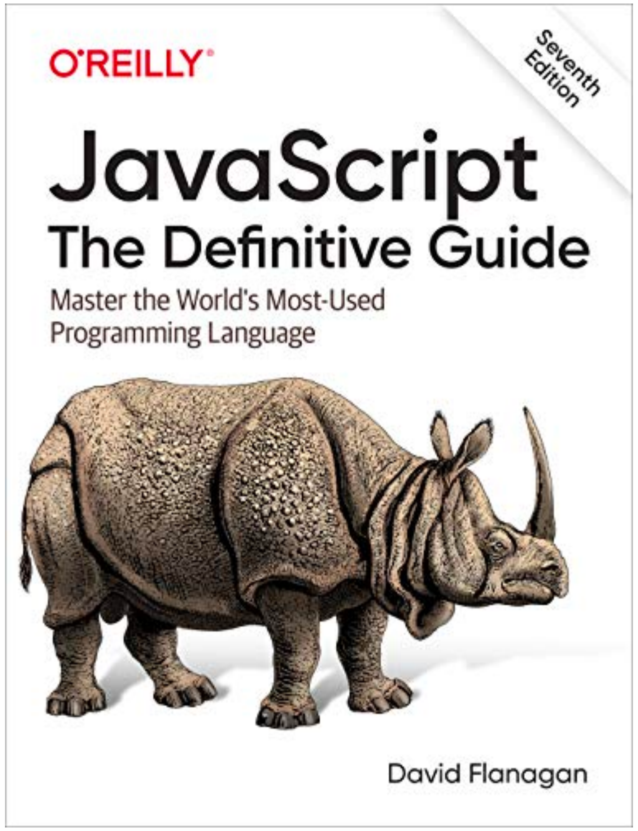
In the following example, we change the background color of all <p> elements in a node list:

```
let myNodelist = document.getElementsByTagName("p");
let i;
for (i = 0; i < myNodelist.length; i++) {
    myNodelist[i].style.backgroundColor = "red";
}
```

A node list may look like an array, but it is not. You can loop through the node list and refer to its nodes like an array. However, you cannot use array methods, like `valueOf()` or `join()` on the node list.

References

D. Flanagan
JavaScript - The Definitive Guide
ed. O'Reilly, 2020



http://www.w3schools.com/js/default.asp