UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 07

JAVASCRIPT IN A BROWSER ENVIRONMENT

Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

https://luistar.github.io

https://www.docenti.unina.it/luigiliberolucio.starace

PREVIOUSLY, ON WEB TECHNOLOGIES

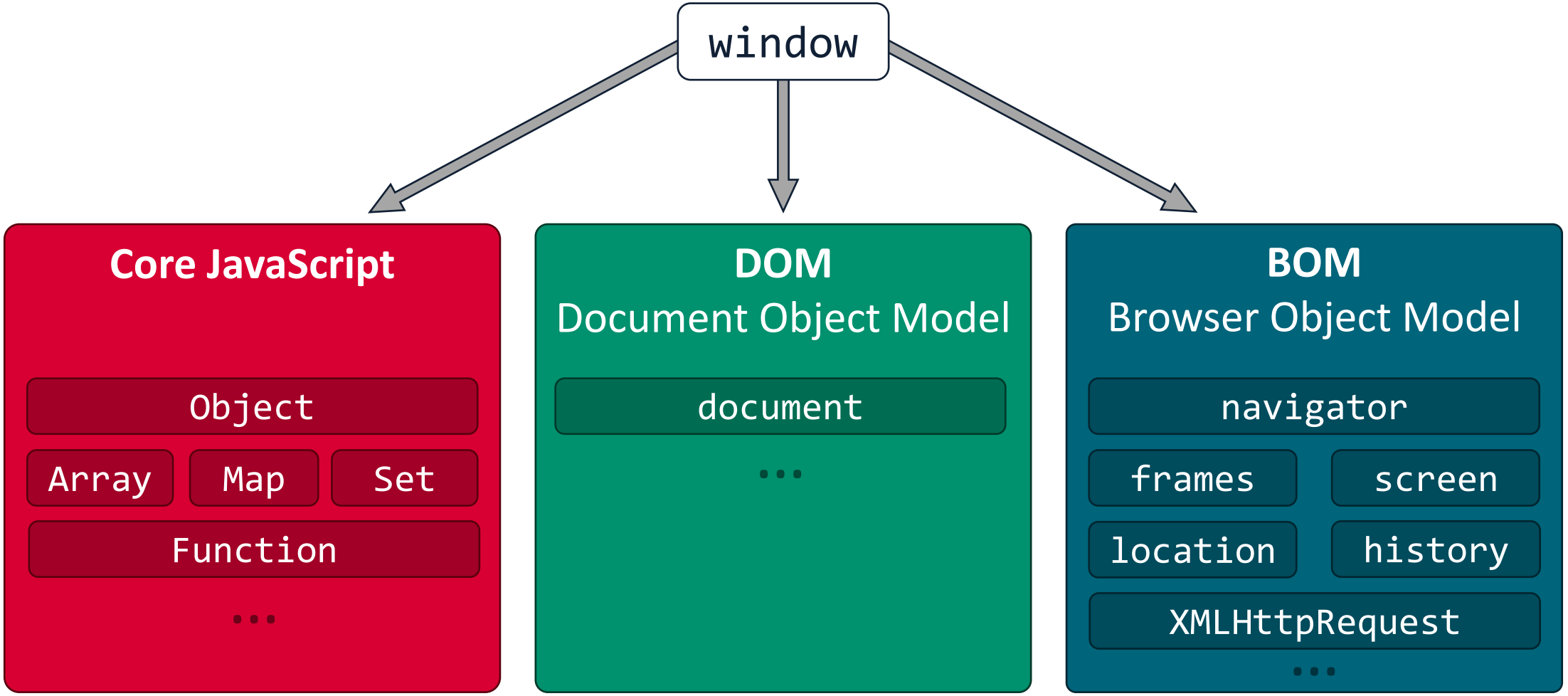
So far we've learned the core concepts of the JavaScript language.

JavaScript can run on a number of different **host environments**:

- **Web Browsers** (for which JavaScript was originally created)
- **Web Servers** (e.g.: via the Node.js runtime)

- Host environments provide their own **objects** and **functions**, in addition to those included in the **language core**
- Today, we'll learn about the **web browser host environment**

THE BROWSER ENVIRONMENT: OVERVIEW



THE BROWSER ENVIRONMENT: WINDOW

- The Browser Environment feature a «**root**» object called **`window`**
- The **`window`** object:
  1. Is a global object for JavaScript code
  2. Represents the «browser window», and provides method to control it
- Global functions and variables are properties of the window object

```
let msg = "Hello!";
function greet(name){
  console.log(`${msg}, ${name}!`);
}

greet("Web Technologies"); //Hello, Web Technologies!
window.greet("window object"); //Hello, window object!
```

THE DOCUMENT OBJECT MODEL (DOM)

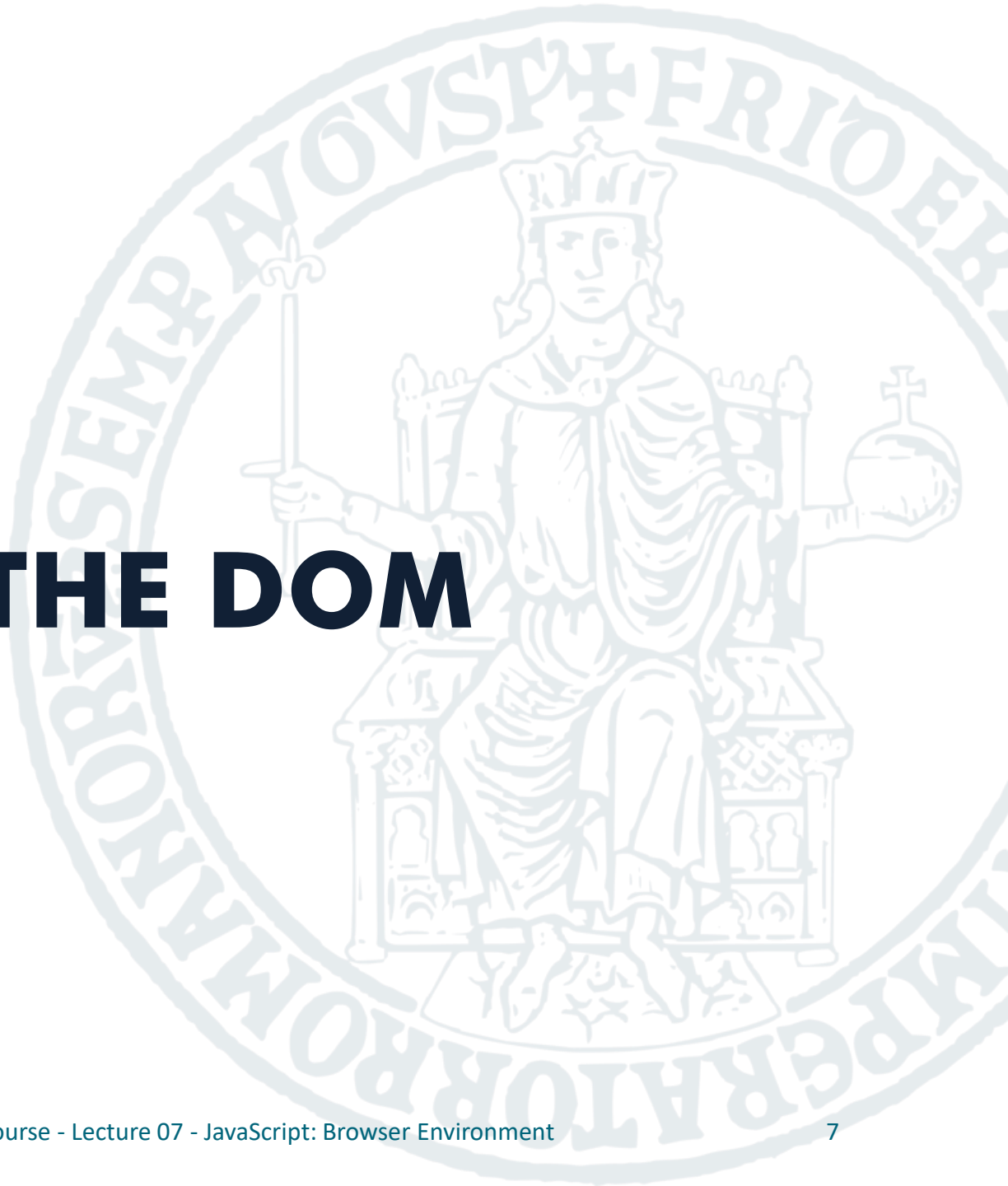
- The DOM represents the content of the current document
- The **`document`** object is the main entry point to the web page
- Provides ways to **access** and **manipulate** the contents

THE BROWSER OBJECT MODEL (BOM)

- The BOM includes additional objects provided by the Browser for working with the browser itself, not with the document contents

```
//The location object contains information on the URL of the current document
console.log(location.href); //http://localhost:3000/08-JavaScript...

//screen contains information about the display used by the user
console.log(screen.availWidth); //2048

//navigator contains additional details about the web browser and platform
console.log(navigator.userAgent); //Mozilla/5.0 (Windows NT 10.0...

//history represents the navigation history
history.back(); //same as clicking on the "back" button in the browser GUI
```

WORKING WITH THE DOM

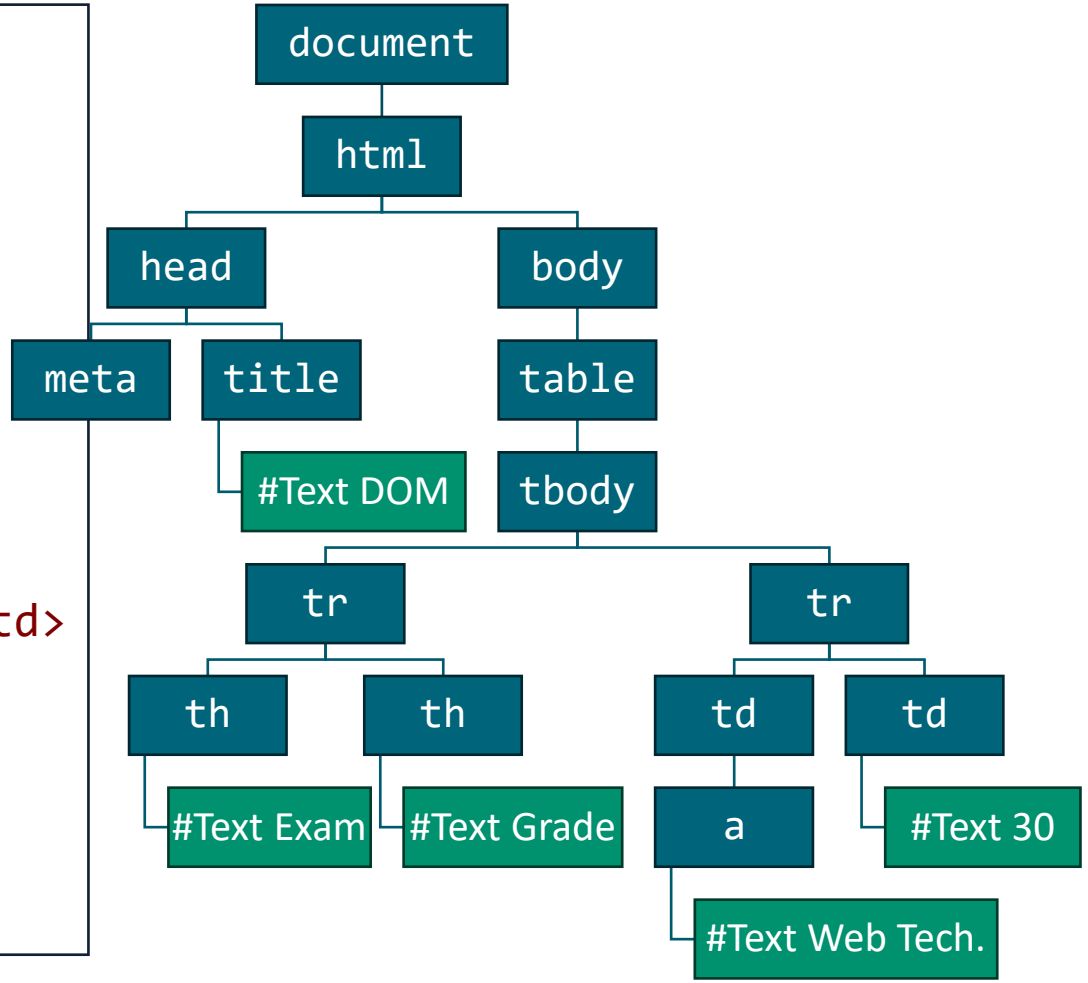


THE DOM

- When we learned about CSS selectors, we hinted at the fact that HTML documents could be seen as trees
  - We talked about **descendants**, **children**, **siblings**…
- The **DOM** formalizes that intuition
  - Every HTML tag is an **object** in the DOM
  - Nested tags are children of the enclosing ones
  - The text content of a tag is an object as well, and a child of the tag
  - Attributes of a tag are accessible as properties of the corresponding object

THE DOM: EXAMPLE

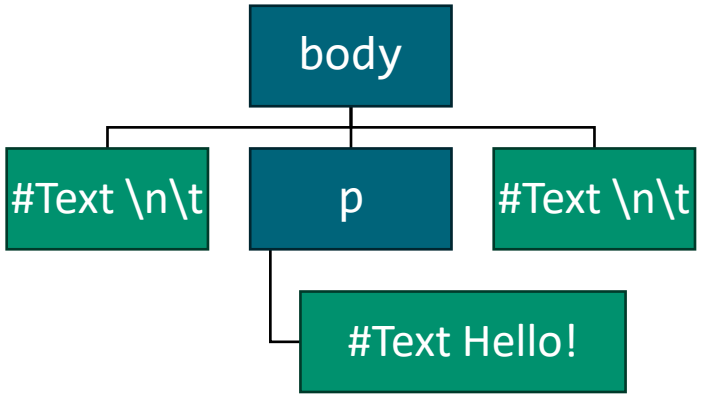
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>DOM</title>
</head>
<body>
  <table>
    <tbody>
      <tr><th>Exam</th><th>Grade</th></tr>
      <tr><td><a href="/wt">Web Tech.</a></td>
          <td>30</td></tr>
    </tbody>
  </table>
</body>
</html>
```



THE DOM: A NOTE ABOUT THE EXAMPLE

- The example in the previous slide is a bit of a simplification
- The actual DOM includes also **spaces** and **newlines** between tags as text nodes. For the sake of brevity, we will generally omit such nodes

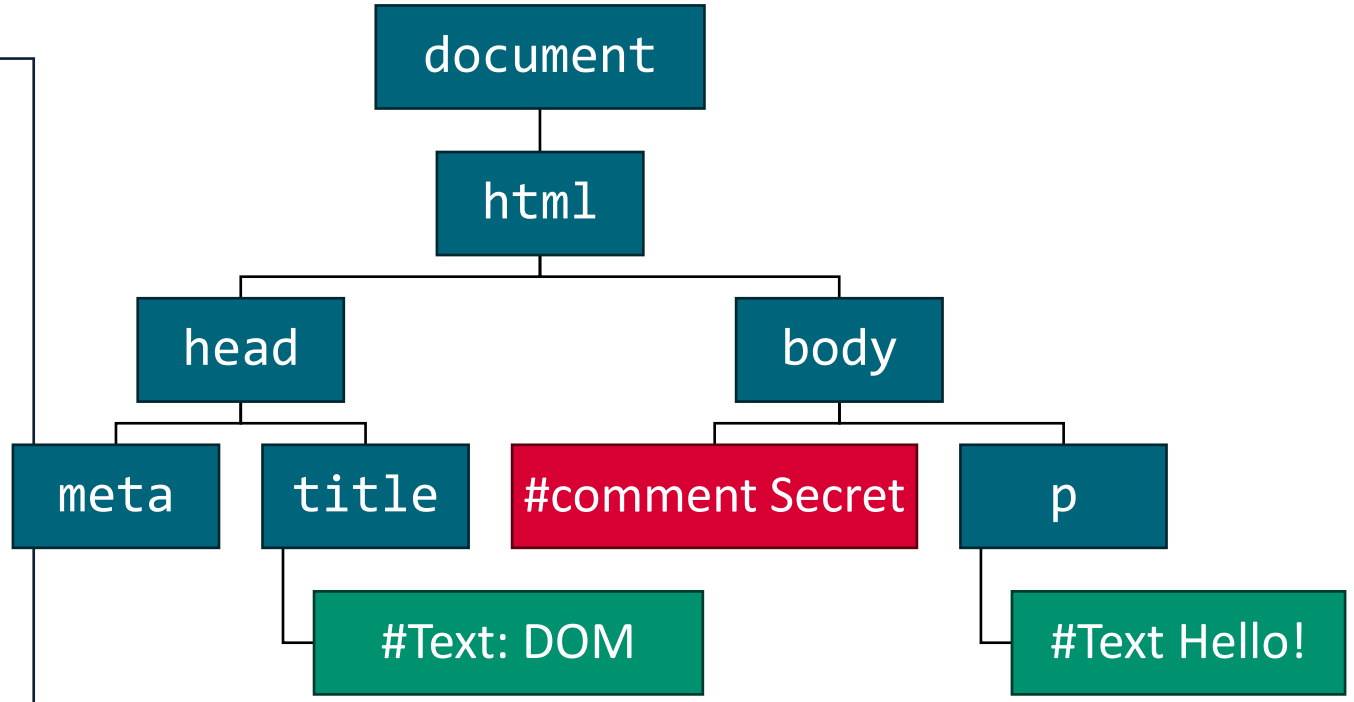
```
<!DOCTYPE html>
<html>
  <head><meta charset="utf-8">
    <title>JavaScript</title>
  </head>
  <body>
    <p>Hello!</p>
  </body>
</html>
```



THE DOM: COMMENTS

- HTML comments are not displayed by web browsers
- Nonetheless, they are still parsed and added to the DOM as special **comment nodes**

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>JavaScript</title>
  </head>
  <body>
    <!-- Secret -->
    <p>Hello!</p>
  </body>
</html>
```



THE DOM: NODE TYPES

- The DOM specification defines 12 different node types
- Typically, we will work with four types of nodes:
  - **`document` node**: is the root and the entry point of the DOM
  - **Element nodes**: represent HTML elements
  - **Text nodes**: represent text content
  - **Comment nodes**: represent comments

THE DOM: FINDING ELEMENTS

- The **document** object provides several methods to select elements from an HTML document

```
<ul>
  <li id="a">A</li>
  <li class="foo bar">B</li>
  <li name="c">C</li>
</ul>
<script>
  console.log(document.getElementById("a")); //select the first list item
  console.log(document.getElementsByClassName("foo")); //HTMLCollection (1)
  console.log(document.getElementsByName("c")); //NodeList (1)
  console.log(document.getElementsByTagName("li")); //HTMLCollection (3)
</script>
```

THE DOM: FINDING ELEMENTS (2)

`document.querySelector(css)` and **`querySelectorAll(css)`** are the most versatile methods, and take as input a **css selector**

- **`querySelector`** returns only the first element (if any) matching the selector, while **`querySelectorAll`** returns a list of matches
- All methods return **`null`** when they fail to locate elements

```
<ul>
  <li id="a">A</li> <li class="foo bar">B</li> <li name="c">C</li>
</ul>
<script>
  console.log(document.querySelector("ul > li")); //first li element
  console.log(document.querySelectorAll("ul > li")); //NodeList (3)
  console.log(document.querySelector("ul > li.myclass")); //null
</script>
```

THE DOM: FINDING ELEMENTS (3)

All methods of the **`getElementsBy*`** family return an **HTMLCollection**, also known as a «**live collection**»

- Such collections are **automatically updated** and **always reflect the current state of the document**

```
<a href="/">First link</a>
<script>
  let links = document.getElementsByTagName("a");
  console.log(links); //HTMLCollection (1)
</script>
<a href="/">Second Link</a> <a href="/">Third Link</a>
<script>
  console.log(links); //HTMLCollection (3)
</script>
```

THE DOM: FINDING ELEMENTS (4)

When invoked on **`document`**, the methods to search for element search the entire HTML document for matches.

- If the elements we need are descendants of another element, it might be more efficient to invoke the search methods on the ancestor element itself.

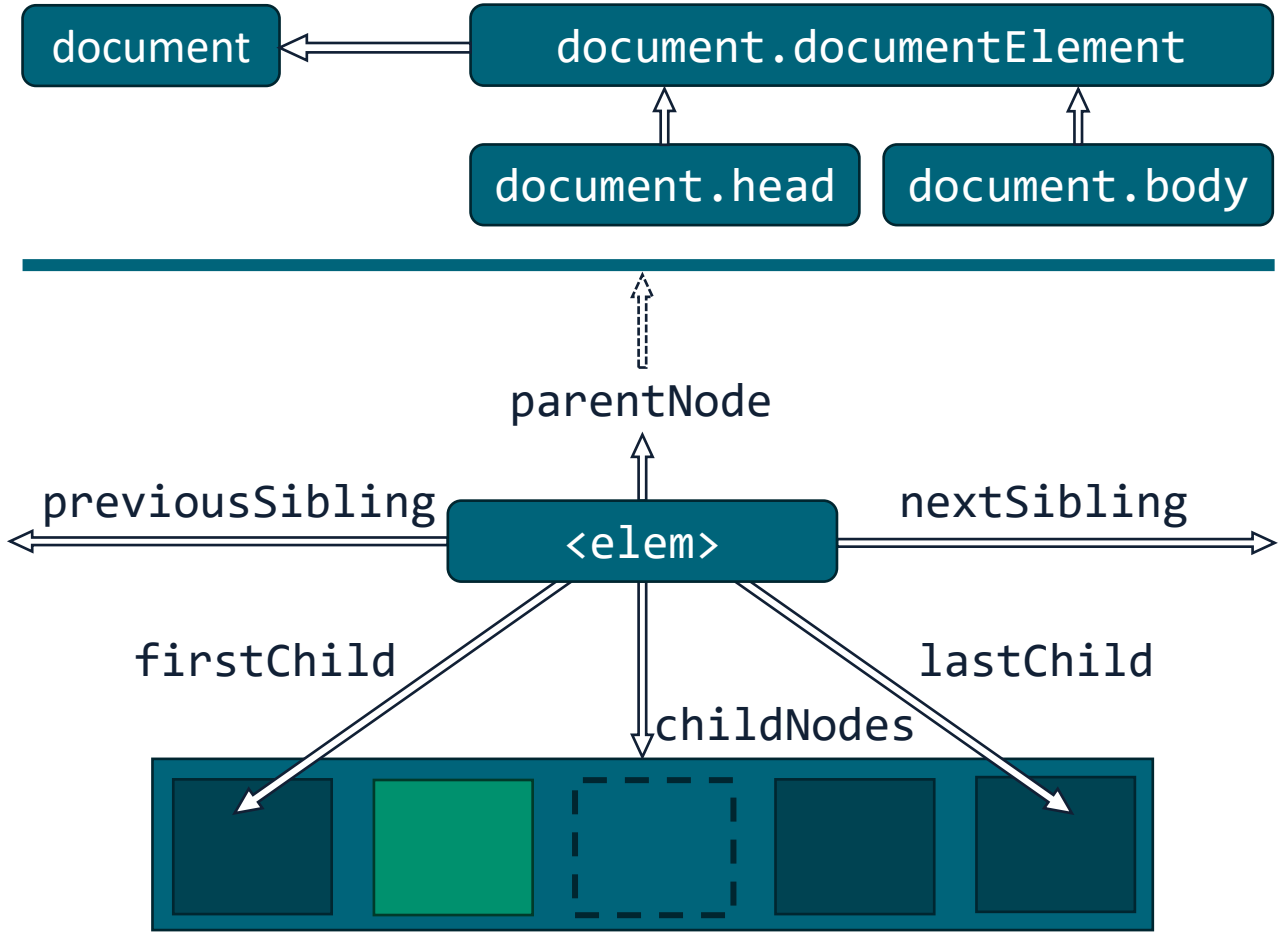
```
<a href="/">First link</a>
<script>
  let sidebar = document.getElementsById("sidebar");
  let sidebarLinks = sidebar.querySelectorAll("a"); //only searches within sidebar
</script>
```

THE DOM: FINDING ELEMENTS RECAP

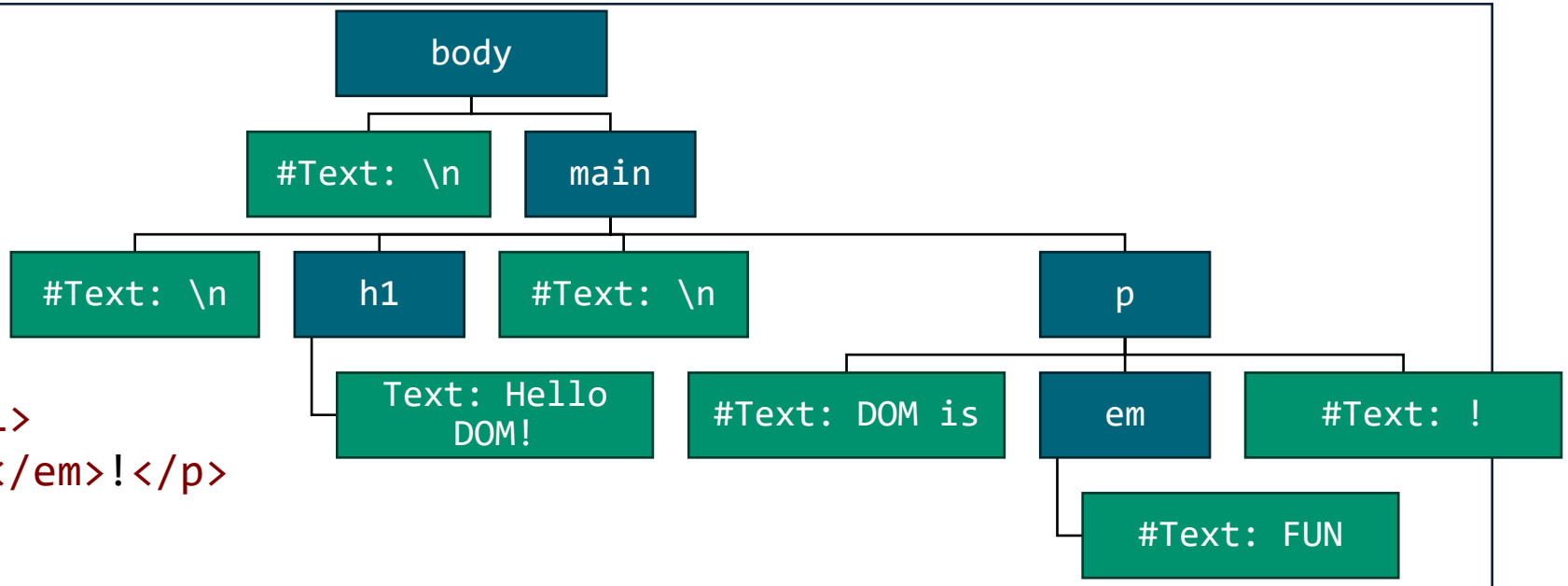
Method	Searches by...	Can be called on elements?	Returns Live Collection?
querySelector	CSS selector	✓	-
querySelectorAll	CSS selector	✓	-
getElementById	Id attribute	-	-
getElementsByName	Name attribute	-	✓
getElementsByTagName	Tag name or '*'	✓	✓
getElementsByClassName	Class attribute	✓	✓

NAVIGATING THE DOM

- Topmost HTML elements (html, head, body) are also available as properties of document
- General DOM node objects contain references to their **parent**, **siblings**, and **children**
- These references are **read-only!**



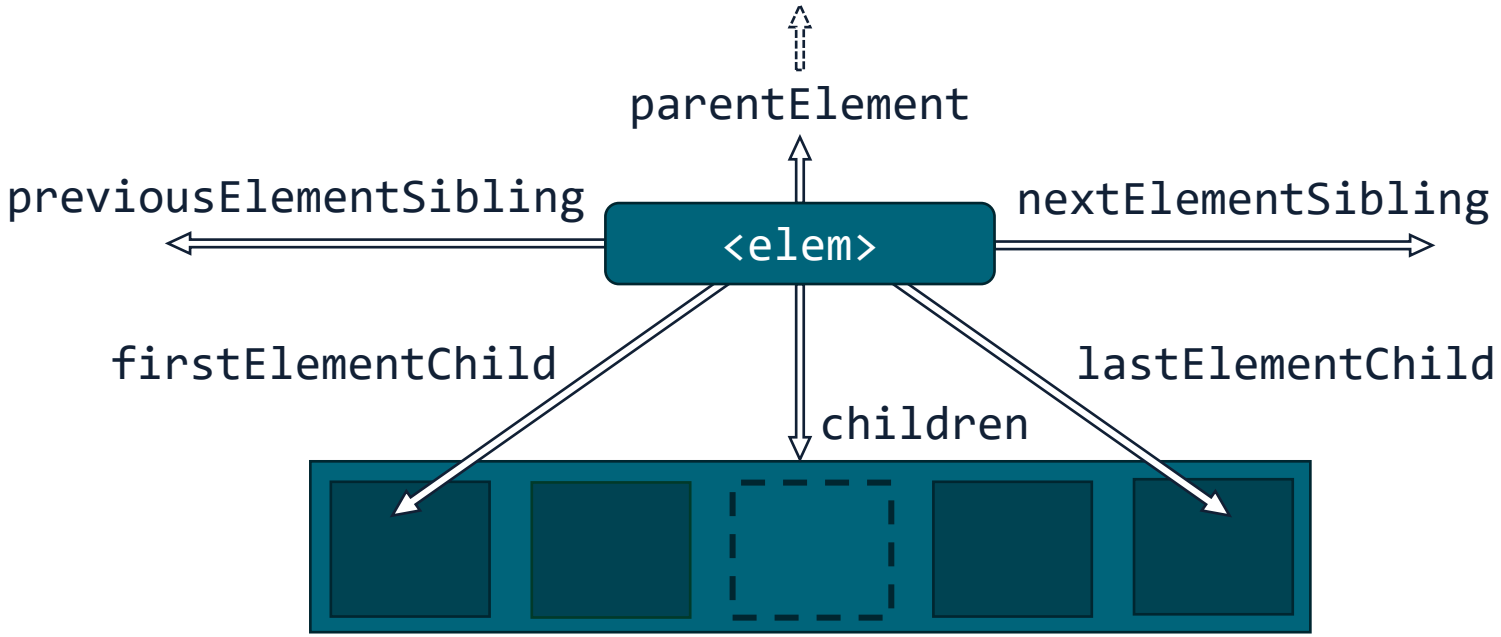
NAVIGATING THE DOM: EXAMPLE




```
<body>
  <main>
    <h1>Hello DOM!</h1>
    <p>DOM is <em>FUN</em>!</p>
  </main>
  <script>
    console.log(document.body.firstChild.nextSibling); //main elem
    console.log(document.body.childNodes[1]); //main elem
    console.log(document.querySelector("em").parentNode); //p elem
  </script>
</body>
```

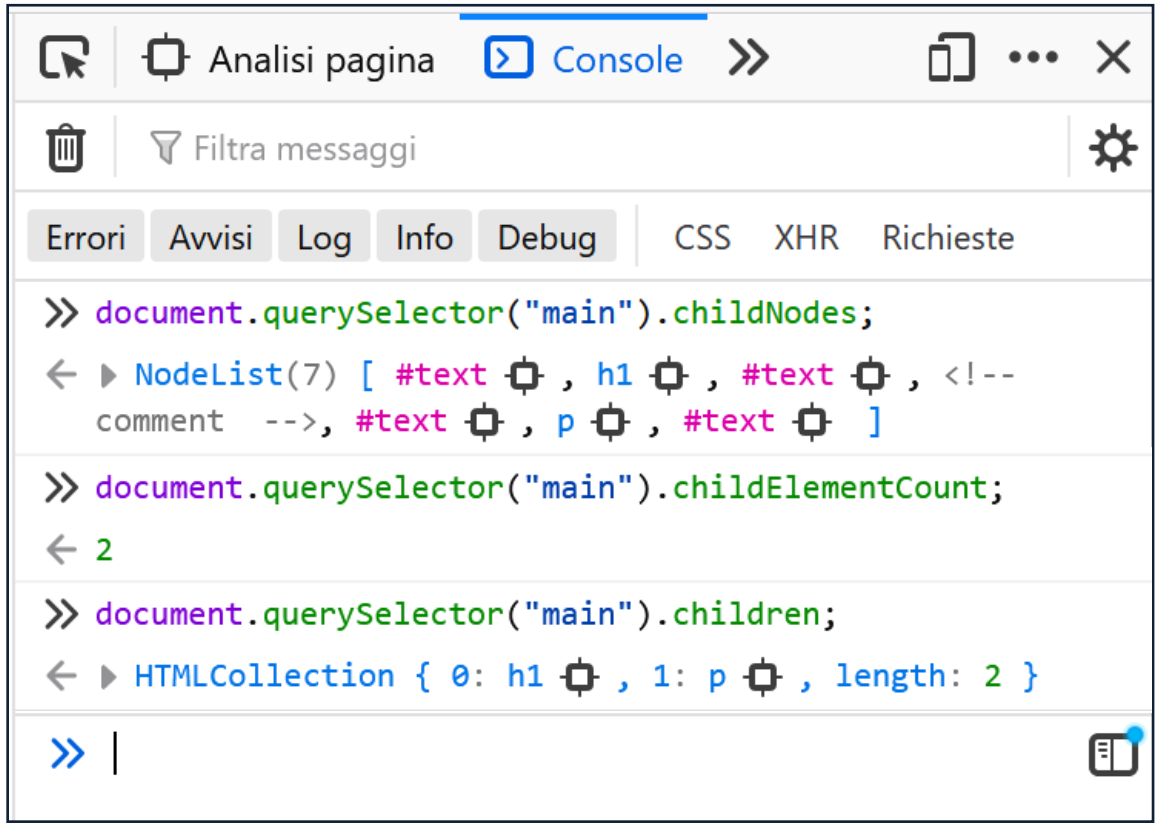
ELEMENT–ONLY DOM NAVIGATION

- We might want to ignore text or comment nodes when navigating the DOM, and focus only on DOM Elements.
- The following navigation properties only consider DOM Elements

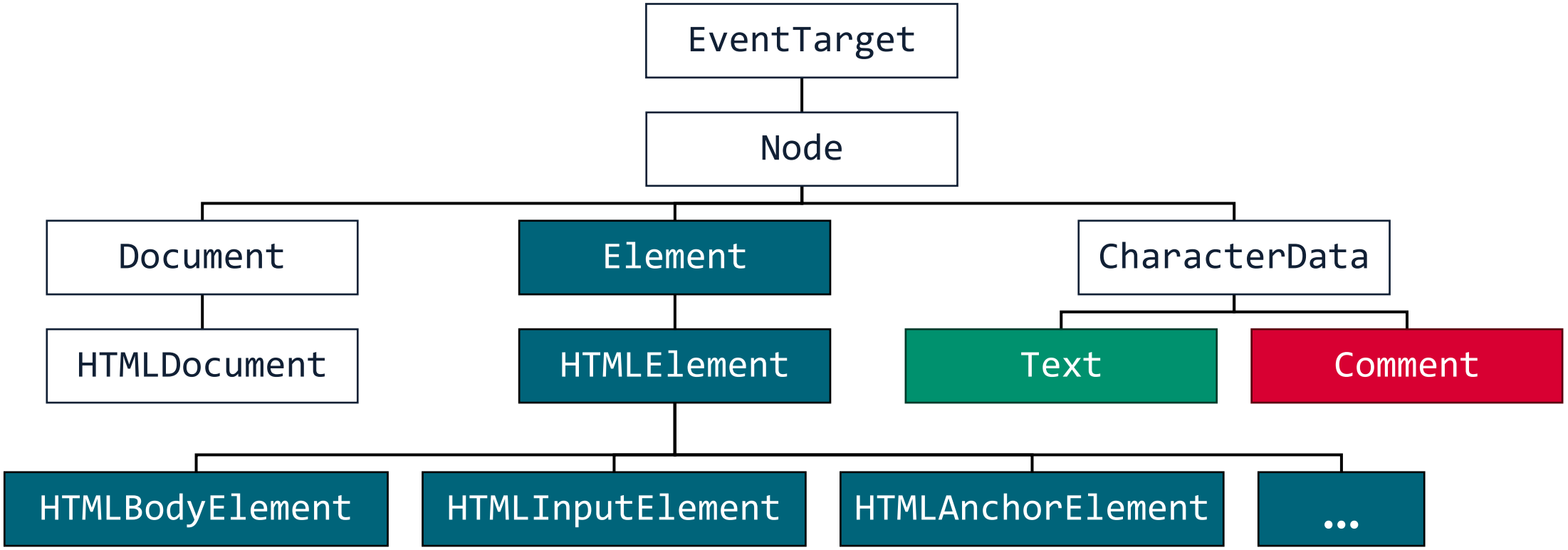


NAVIGATING THE DOM

```
<main>
  <h1>Hello DOM!</h1>
  <!-- comment -->
  <p>DOM is <em>Fun</em>!</p>
</main>
```



DOM NODES CLASS HIERARCHY



NODE PROPERTIES

DOM nodes have some useful properties we can access:

- **nodeName/tagName** can be used to access information about the type of a node. These properties are **read-only.**

```
<body>
  <h1 id="h">Hello <em>DOM!</em></h1>
  <!-- comment -->
  <script>
    let body = document.body;
    console.log(body.firstChild.nodeName);         //#text
    console.log(body.firstChild.tagName);          //undefined
    console.log(body.childNodes[3].nodeName);      //#comment
    console.log(body.firstElementChild.nodeName); //H1
    console.log(body.firstElementChild.tagName);  //H1
  </script>
</body>
```

NODE PROPERTIES

Standard **attributes** are also parsed and made available as properties of the corresponding elements

```
<body>
  <input id="x" class="inp ok" name="field" custom="y">
  <script>
    let i = document.body.firstElementChild;
    console.log(i.id); //x
    console.log(i.classList); //DOMTokenList ["inp", "ok"]
    console.log(i.name); //field
    console.log(i.custom); //undefined, because custom is not a standard attrib.
    console.log(i.getAttribute("custom")); //y
  </script>
</body>
```

NODE PROPERTIES

These properties are also **writable**!

```
<body>
  <input id="x" class="inp ok" name="field" custom="y">
  <script>
    let i = document.body.firstElementChild;

    i.id = "w";
    i.classList.remove("ok"); i.classList.add("err");
    i.setAttribute("custom","k");
    i.removeAttribute("name");
  </script>
</body>

<!-- the input after the script executes -->
<input id="w" class="inp err" custom="k">
```

MODIFYING THE DOM STRUCTURE

- We’re not limited to modifying existing DOM elements
  - E.g.: by changing their attributes
- We can dynamically **create** new elements and **remove** existing ones, and change the **DOM structure**
- Dynamically updating the DOM is the key to creating reactive, «live» web pages

NODE PROPERTIES: innerHTML / outerHTML

- **innerHTML** can be used to access the HTML code contained in an HTMLElement
- **outerHTML** can be used to access the entire HTML code of an HTMLElement

```
<body>
  <h1 id="h">Hello <em>DOM!</em></h1>
  <!-- comment -->
  <script>
    let body = document.body;
    console.log(body.firstElementChild.innerHTML); //Hello <em>DOM!</em>
    console.log(body.firstChild.innerHTML);        //undefined (not an HTMLElement)
    console.log(body.firstElementChild.outerHTML); //<h1 id="h">Hello <em>DOM!</em></h1>
  </script>
</body>
```

NODE PROPERTIES: innerHTML / outerHTML

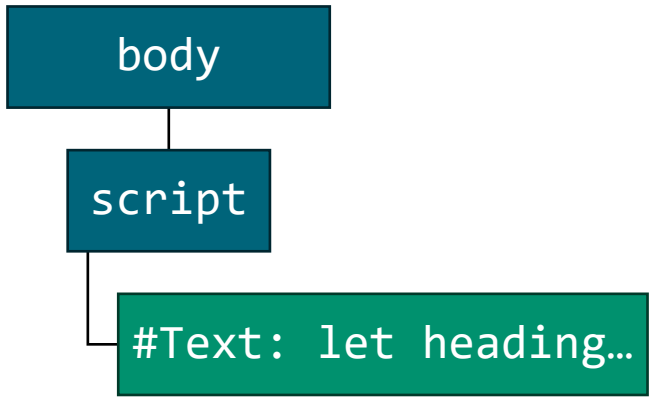
A fun thing about **innerHTML** and **outerHTML** is that they are **writable**

- **We can assign new values to them to change the content of the page!**

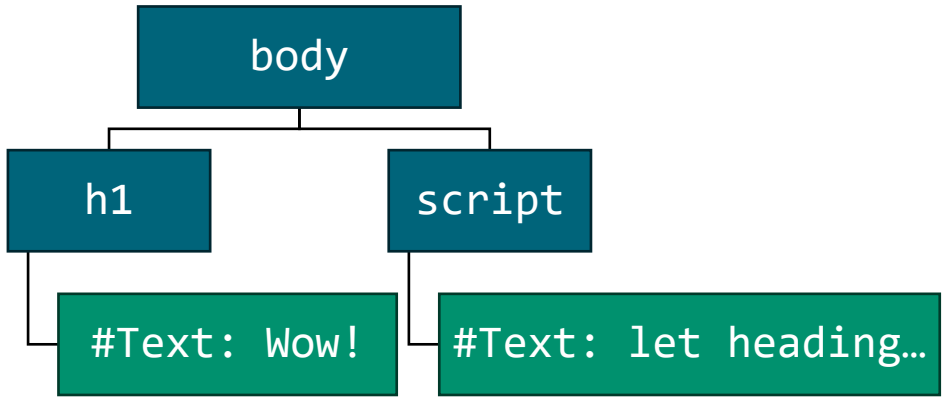
```
<body>
  <h1 id="h">Hello <em>DOM!</em></h1>
  <!-- comment -->
  <script>
    let elem = document.body.firstElementChild;

    elem.innerHTML = "DOM: <em>Wow!</em>";
    //We changed the content of the <h1> element

    elem.outerHTML = `<p>${elem.innerHTML}</p>`;
    // Now the <h1> element is a <p>, with the same content as before!
  </script>
</body>
```

CREATING NEW DOM ELEMENTS

```
<body>
  <script>
    let heading = document.createElement("h1"); //create a new <h1> elem
    // the new <h1> is not yet in the DOM
    let title = document.createTextNode("Wow!"); //create a new #text node
    // the new text node is not yet in the DOM
    heading.append(title); //add the #text node as a child of the <h1>
    document.body.prepend(heading); //<h1> (and its child) at the beginning of <body>
  </script>
</body>
```

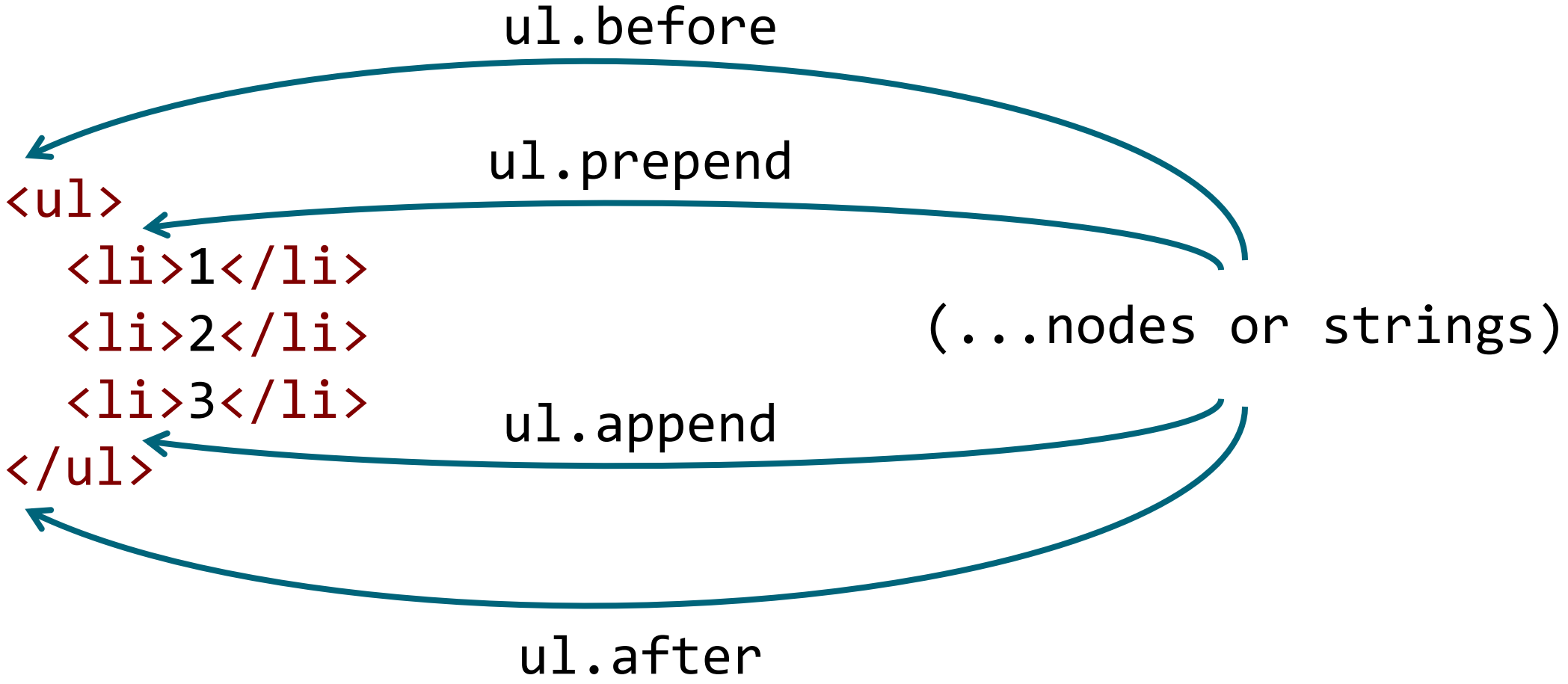


DOM Tree **before** executing the script



DOM Tree **after** executing the script

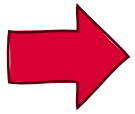
DOM MANIPULATION METHODS



DOM MANIPULATION: EXAMPLE

```
<body>
  <ul>
    <li class="kw">CSS</li>
  </ul>
</body>
```

```
let ul = document.querySelector("ul");
let html = ul.querySelector("li").cloneNode();
let js = document.createElement("li");
html.innerHTML = "HTML";
js.innerHTML = "JavaScript";
ul.before("Keywords:");
ul.prepend(html);
ul.append(js);
ul.after("End of keyword list");
```

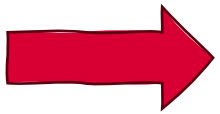


```
<body>
  Keywords:
  <ul>
    <li class="kw">HTML</li>
    <li class="kw">CSS</li>
    <li>JavaScript</li>
  </ul>
  End of keyword list
</body>
```

DOM MANIPULATION: EXAMPLE (2)

```
<body>
  <ul>
    <li class="kw">CSS</li>
  </ul>
</body>
```

```
let ul = document.querySelector("ul");
let ol = document.createElement("ol");

ol.append(ul.firstElementChild);
ul.replaceWith(ol);
```



```
<body>
  <ol>
    <li class="kw">CSS</li>
  </ol>
</body>
```

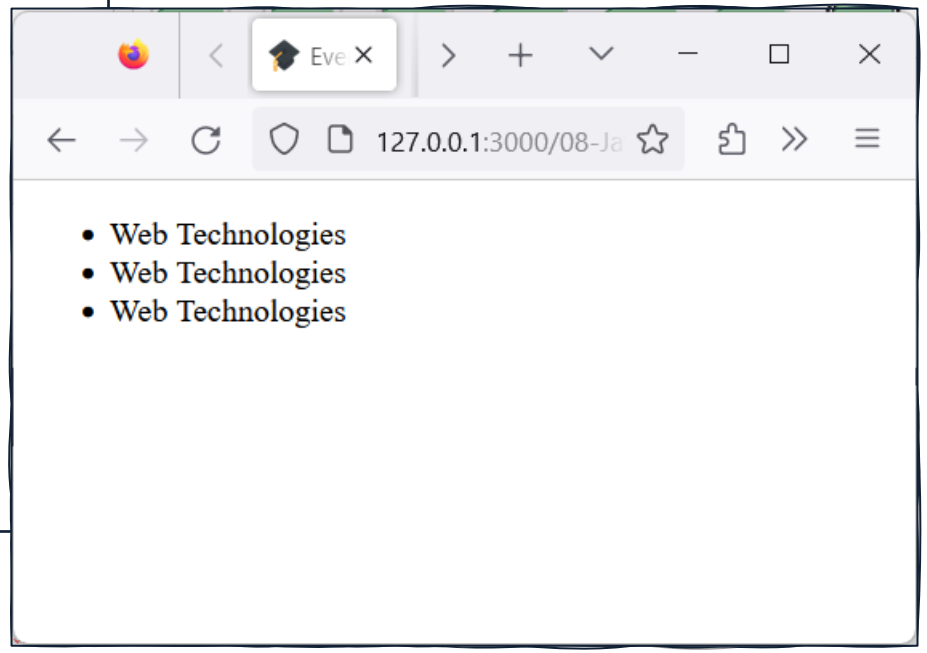
DOM MANIPULATION: RECAP

Method	Description
`document.createElement(tag)`	creates an element with the given tag
`document.createTextNode(value)`	creates a text node (rarely used)
`elem.cloneNode(deep)`	clones the element, if `deep==true` with all descendants
`node.append(...nodes or strings)`	insert into node, at the end
`node.prepend(...nodes or strings)`	insert into node, at the beginning
`node.before(...nodes or strings)`	insert right before node
`node.after(...nodes or strings)`	insert right after node
`node.replaceWith(...nodes or strings)`	replace node
`node.remove()`	remove the node

INTERACTING WITH USERS

```
<ul></ul>
<script>
  alert("This page will ask for your input");
  if(confirm("Do you want to continue?") === true) {
    let name = prompt("State your name:");
    let num = parseInt(prompt("Insert a number:"));
    for(let i = 0; i < num; i++) {
      let li = document.createElement("li");
      li.innerHTML = name;
      document.querySelector("ul").append(li);
    }
  } else {
    console.log("user cancelled");
  }
</script>
```

alert, **prompt**, and **confirm** are useful ways to interact with users in a web browser.



EVENTS

BROWSER EVENTS

Events are signals that **something** happened. They might be related to:

- **User behaviour**:
  - Clicks, key presses on the keyboard, mouse movements…
- **Forms**:
  - Submission, focus (or loss thereof) on an input field
- **Document events**:
  - Document or elements loaded, network requests completed…

USEFUL BROWSER EVENTS

Mouse	`click`	User clicks (taps, for touchscreen devices) on element
	`contextMenu`	Mouse right-click on element
	`mouseover/mouseout`	Mouse cursor goes over / leaves an element
	`mousedown/mouseup`	The main mouse button is pressed / released over an element
	`mousemove`	The mouse cursor moves
Keyboard	`keydown/keyup`	A keyboard keys is pressed / released
Forms	`submit`	User submits a form
	`focus`	User focuses on an element (e.g.: <input>)
Document	`DOMContentLoaded`	The HTML has been parsed, DOM is fully built
Window	`load`	The web pages has been completely loaded and rendered

EVENT HANDLERS

- To react to events, **handler** functions can be defined
- They are functions that are executed when an event fires
- The simplest way to define handlers is to use `on<event>` HTML attributes. The value of the attribute is the JavaScript code to run

```
<input type="button" value="Click me" onclick="handleClick();">

<script>
  function handleClick(){
    console.log("Click handled");
  }
</script>
```

EVENT HANDLERS: DYNAMIC DEFINITION

- Event handlers can also be defined **dynamically**, via JavaScript
- This can be done by writing the **on<event>** property on HTML elements, or by using the **addEventListener** method

```
<input type="button" value="Click me" onclick="handleClick();">

<script>
  function handleClick(){ console.log("Click handled"); }

  let input = document.querySelector("input");
  input.onclick = () => {console.log("Tap");}; // overrides HTML-attrib. handler

  input.addEventListener("click", handleClick); // adds new handler function
</script>
```

EVENT HANDLERS: VALUE OF THIS

- Inside an event handler, **`this`** is evaluated to the element to which the invoked handler is assigned

```
<input type="button" value="Click me">
<p>Hello Events!</p>

<script>
  function handleClick(){
    console.log(this.outerHTML);
  }
  document.querySelector("input").addEventListener("click", handleClick);
  // <input type="button" value="Click me">
  document.querySelector("p").addEventListener("click", handleClick);
  // <p>Hello Events!</p>
</script>
```

EVENT HANDLERS: THE EVENT OBJECT

To properly handle events, we might need **more details** on them

- In a `keydown` event, which key was pressed on the keyboard?
- In a `click` event, at which coordinated did the click happen? Did the user also press CTRL on their keyboard, while clicking?

All details on the event are provided as an **event object**, which is passed to event handlers

- Different kind of events have different properties. For example, keyboard-related events have a `key` property representing the key that has been pressed

THE EVENT OBJECT: EXAMPLE

```
<input type="text" placeholder="Write here" autofocus>
<script>
  function handler(event){
    console.log(`You pressed ${event.key}`);
  }
  document.querySelector("input").addEventListener("keyup", handler);
</script>
```

EVENT BUBBLING

When an event is fired on an element **elem**:

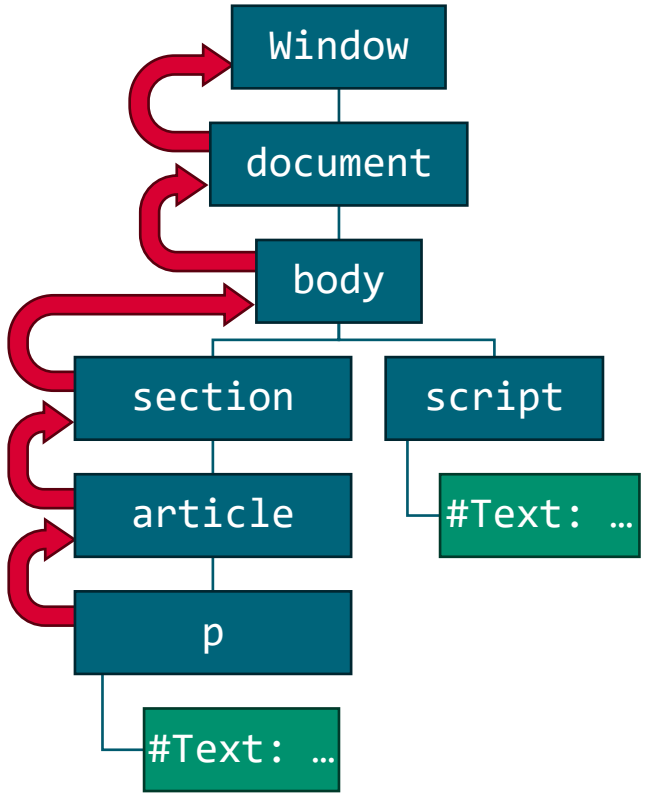
- First, all handlers registered on **elem** for that event are executed
- Then, all the handlers for the event are executed on **elem**'s parent
- Handlers are executed all the way up on other ancestors until the **root** node of the DOM is reached

- This mechanism is known as event **bubbling**
- **It applies to most events,** but not to all of them (e.g.: **focus** events do not bubble)

EVENT BUBBLING: EXAMPLE

```
<body>
  <section onclick="console.log('section');">
    <article onclick="console.log('article');">
      <p onclick="console.log('p');">Bubbling</p>
    </article>
  </section>
  <script>
    document.onclick = () => console.log("document");
  </script>
</body>
```

```
Console output after a click on <p>:
p
article
section
document
```



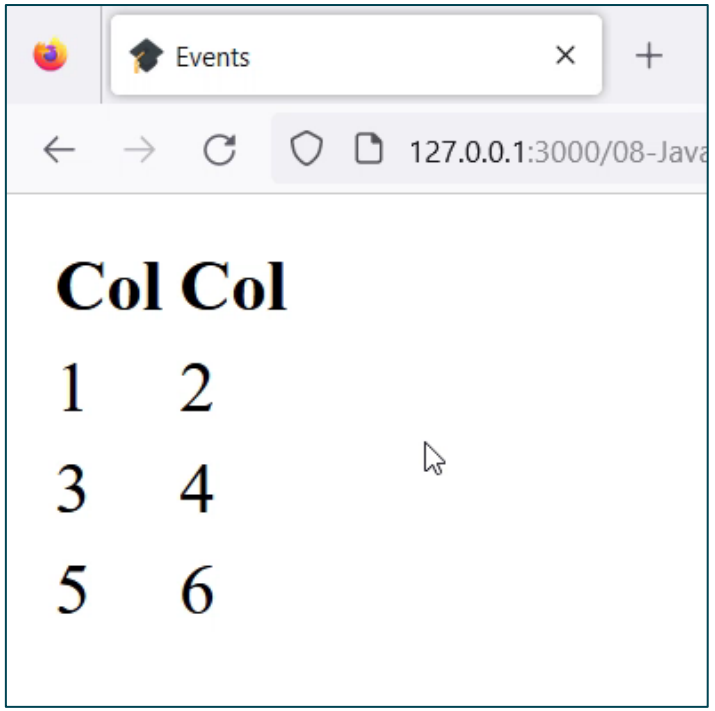
EVENT DELEGATION

Event Delegation is a powerful pattern for handling events

- **Idea**: if many elements can generate similar events, instead of assigning an handler to each of them, we use a single handler on their common ancestor
  - The elements **delegate** event handling to an ancestor
  - The **target** property on the event object can be used to determine what element the event was generated on

EVENT DELEGATION: EXAMPLE

```
<table onclick="handler(event);">
  <tr><th>Col</th><th>Col</th></tr>
  <tr><td>1</td><td>2</td></tr>
  <tr><td>3</td><td>4</td></tr>
  <tr><td>5</td><td>6</td></tr>
</table>
<script>
  function handler(event){
    if(event.target.nodeName === "TD"){
      let oldValue = parseInt(event.target.innerHTML);
      event.target.innerHTML = ++oldValue;
    }
  }
</script>
```

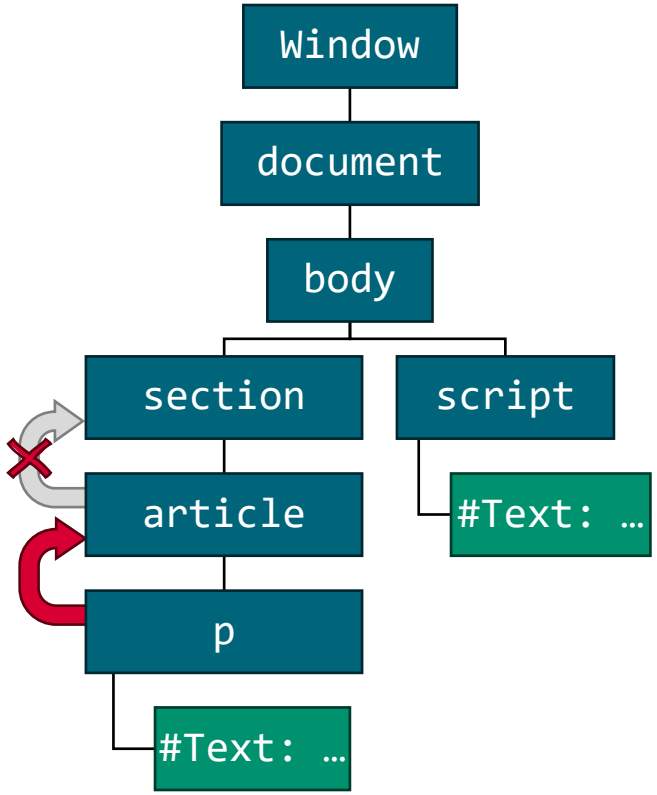


STOPPING EVENT BUBBLING

- A bubbling event originates from a certain elements and **propagates** (bubbles up) all the way up to the global **window** object.
- In some cases, an handler might decide that the event has been fully processed, and there is no need to propagate further
  - Bubbling can be **stopped** by invoking the **`stopPropagation()`** method on the event object
- **Rule of thumb**: do not stop propagation unless you have a good reason. You might need to catch events on an ancestors in the future!

STOPPING EVENT BUBBLING: EXAMPLE

```
<body>
  <section onclick="console.log('section');">
    <article>
      <p onclick="console.log('p');">Bubbling</p>
    </article>
  </section>
  <script>
    let art = document.querySelector("article");
    art.addEventListener("click", handler);
    function handler(event){
      console.log(this.tagName);
      event.stopPropagation();
    }
  </script>
</body>
```



DISPATCHING EVENTS

- From JavaScript, it is also possible to **generate** events
- The **isTrusted** event property is set to **false** for «artificial» events

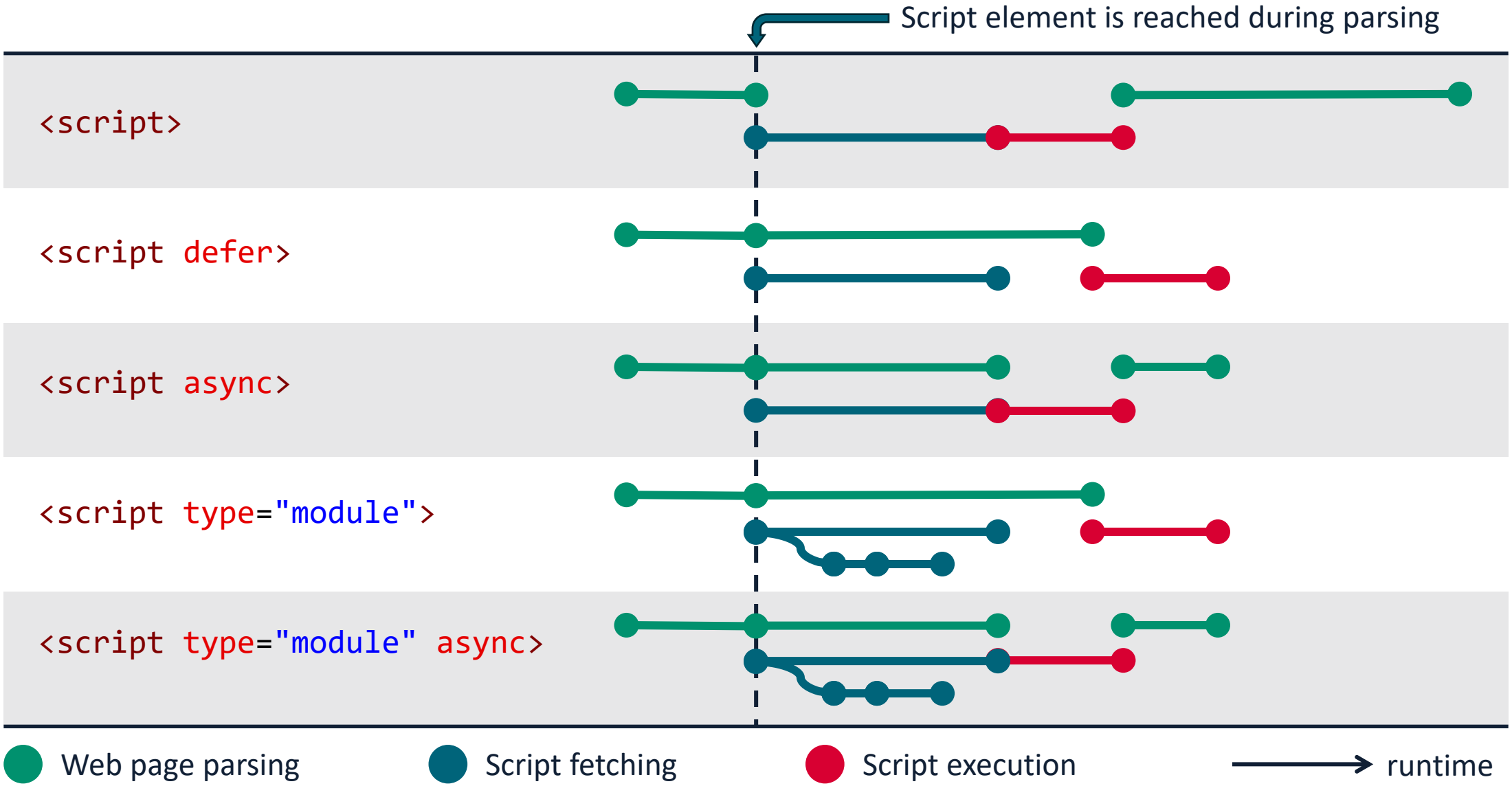
```
<input type="button" value="Click me">
<script>
  function handler(event){
    console.log(`${event.type} event. Trusted: ${event.isTrusted}`);
  }
  let input = document.querySelector("input");
  input.onclick = handler;

  let click = new Event("click");
  input.dispatchEvent(click); //isTrusted is false for events generated via JS
</script>
```

JAVASCRIPT: EXECUTION ORDER

As our scripts become more complex, interact with the DOM and dynamically generate event handlers, we need to be aware of how JavaScript code is executed when a web page is loaded.

- Generally, scripts are fetched and executed in the order they appear, while web page parsing is paused
- External scripts with the **defer** attribute are downloaded in parallel, and executed after the page has finished parsing
- External scripts with the **async** attribute are downloaded in parallel, and executed as soon as they are downloaded
- **Module** scripts **defer** by default, and can be declared as **async**

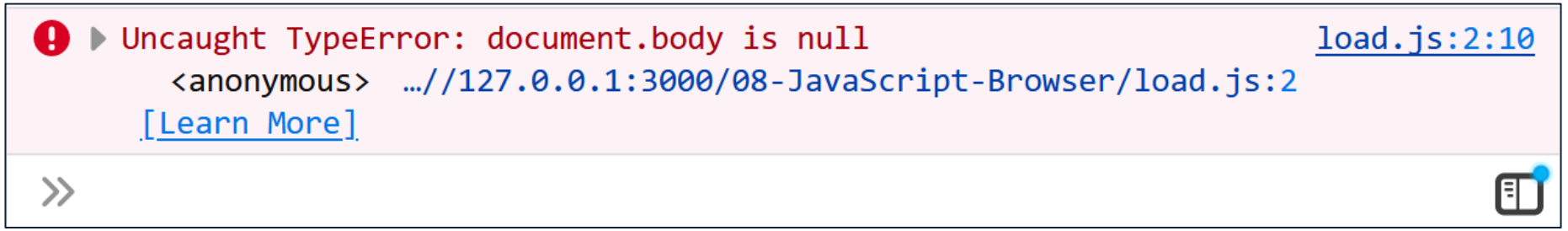
JAVASCRIPT: EXECUTION ORDER



EXECUTION ORDER MATTERS: EXAMPLES

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script src="load.js"></script>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```js
//load.js file
let h1 = document.body.querySelector("h1");
console.log(h1.innerText);
```



EXECUTION ORDER MATTERS: EXAMPLES

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script defer src="load.js"/>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```js
//load.js file
let h1 = document.body.querySelector("h1");
console.log(h1.innerText);
```

```
JavaScript                                    load.js:3:9
»
```

EXECUTION ORDER MATTERS: EXAMPLES

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script async src="load.js"/>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```js
//load.js file
let h1 = document.body.querySelector("h1");
console.log(h1.innerText);
```

We can’t really know for sure. It depends on how much time it takes to fetch the script!

EXECUTION ORDER MATTERS: EXAMPLES

```html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script src="load.js"></script>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```js
//load.js file
document.addEventListener("DOMContentLoaded",
() => {
  let h1 = document.body.querySelector("h1");
  console.log(h1.innerText);
});
```

REFERENCES

- **The Modern JavaScript Tutorial**
  Freely available at https://javascript.info/ or on GitHub
  Part 2: Document (1.1 to 1.7), Introduction to events, UI events, Document and resource loading
- **Eloquent JavaScript (3rd edition)**
  By Marijn Haverbeke
  Freely available at https://eloquentjavascript.net/
  Chapters 13 to 15
- **JavaScript Reference**
  MDN web docs
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference