



# DOM (Document Object Model)

---

.NET CORE

The **Document Object Model (DOM)** is the data representation of the objects that comprise the structure and content of a document on the web. The DOM represents an HTML or XML document in memory. You send data to APIs endpoints to create web content and applications.

[HTTPS://DEVELOPER.MOZILLA.ORG/EN-US/DOCS/WEB/API/DOCUMENT\\_OBJECT\\_MODEL/INTRODUCTION](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction)

# DOM (Document Object Model)

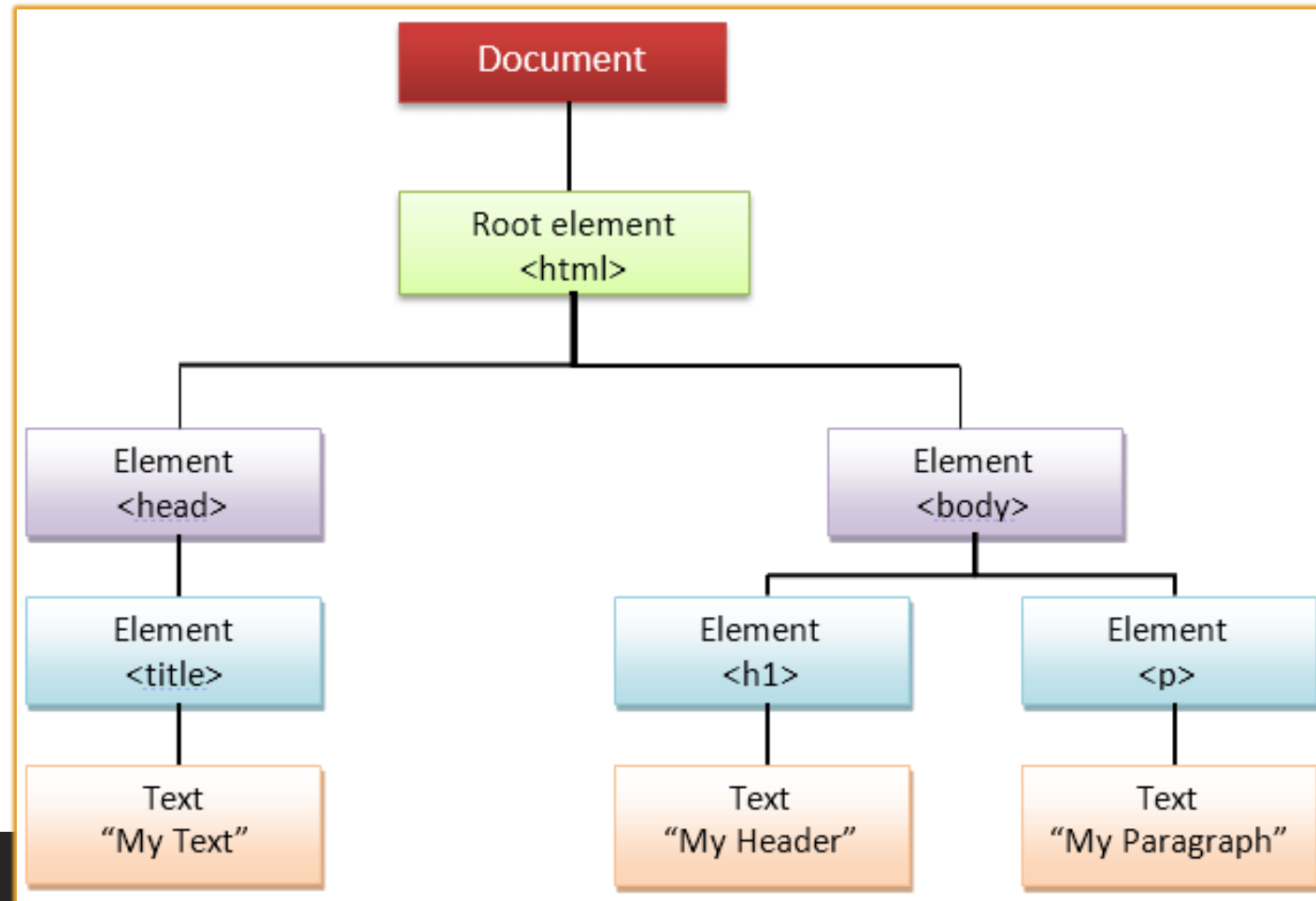
[https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model/Introduction](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction)

The *Document Object Model (DOM)* is a programming interface for HTML and XML documents. It represents the page (as nodes and objects) so that programs can change the document structure, style, and content.

A Web page is one document that can be

- displayed in the browser window,
- viewed as HTML, and
- represented by the DOM.

Only the DOM can be manipulated by scripting languages like JavaScript.



# DOM in action

<https://javascript.info/>

---

All the properties, methods, and events available for manipulating and creating web pages are organized into objects. The document object represents the document itself and the `<table>` object implements the *HTMLTableElement* DOM interface for accessing HTML tables.

*getElementsByTagName()* must return a list of all the `<p>` elements in the document.

```
1  const paragraphs = document.getElementsByTagName("p");
2  // paragraphs[0] is the first <p> element
3  // paragraphs[1] is the second <p> element, etc.
4  alert(paragraphs[0].nodeName);
```

# Dom – How to Access the DOM

---

Within the `<head>` or your .HTML file or at the bottom of the `<body>`, include a `<script>` tag which contains the `.js` file you want to use for the `.HTML` page. You can then access the `document` in the `.js` file with something like `document.getElementById("#IdName")`.

```
<head>
    // other tags, etc....
</head>
<body>
    //...
    <script type="text/javascript" src="jsfile.js"></script>
</body>
```

You can add JavaScript code directly in the .HTML in a `<script>` tag.

# DOM – Selectors

<https://blog.bitsrc.io/dom-selectors-explained-70260049aaf0>

---

DOM Selectors are used to select HTML elements within a document using JavaScript. There are 5 selectors.

Selector Name	Purpose
<code>document.getElementsByTagName("tagName")</code>	Returns a collection (array) of Items matching the tag name.
<code>document.getElementsByClassName(".className")</code>	Returns a collection (array) of Items matching the class name.
<code>document.getElementById("#idName")</code>	Returns the <u>first</u> matched id name. Id's are supposed to be unique in the .HTML file.
<code>document.querySelector("li")</code>	Returns the <u>first</u> element that matches the specified CSS selector.
<code>document.querySelectorAll("ol")</code>	Returns <u>all</u> the elements that match the specified CSS selector

# DOM – Events

<https://developer.mozilla.org/en-US/docs/Web/Events>

<https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onclick>

---

DOM *Events* are sent when things happen on the HTML page, such as when a *button* is clicked. Each event is represented by an *object* which is based on the *Event* interface and may have additional custom fields and/or functions used to get additional information about what happened.

The two most prevalent events are mouse clicks and form submissions.



# Event Listeners and Event Handlers

[https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First\\_steps/A\\_first\\_splash](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/A_first_splash)

---

The “construct” that listens for a event to happen is called an *event listener*. The block of code that runs when the event fires is called an *event handler*.

```
var guessSubmit = document.getElementById("#button");  
guessSubmit.addEventListener('click', checkGuess);
```

*guessSubmit* holds all the data from an element.

It uses a built-in JS helper function called *addEventListener()* which takes two arguments.

1. The type of event we are listening for (*click*), and
2. A callback to the code we want to run when the event occurs. Because *checkGuess* is a callback, you don't need to use the ( ).



# Bubbling and Capture

[https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model/Examples#Example\\_5:\\_Event\\_Propagation](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Examples#Example_5:_Event_Propagation)  
[https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building\\_blocks/Events](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events)

---

Event ***bubbling*** and event ***capture*** are two mechanisms that describe what happens when two ***handlers*** of the same ***event type*** are activated on one ***element***.

When an event is fired (a '***click***') on an element that has parent elements, modern browsers run two different phases — the ***capturing*** phase and the ***bubbling*** phase.

Capture Phase	Bubble Phase
The browser checks to see if the element's <u>outer-most</u> ancestor (<html>) has an 'onclick' event handler registered on it and runs it if so. Then it moves on to the next element inside <html> and does the same thing, until it reaches the element that was actually clicked.	The browser checks to see if the element that was actually clicked on has an 'onclick' event handler registered on it for the bubbling phase and runs it if so. Then it moves on to the next immediate ancestor element and does the same thing until it reaches the <html> element.

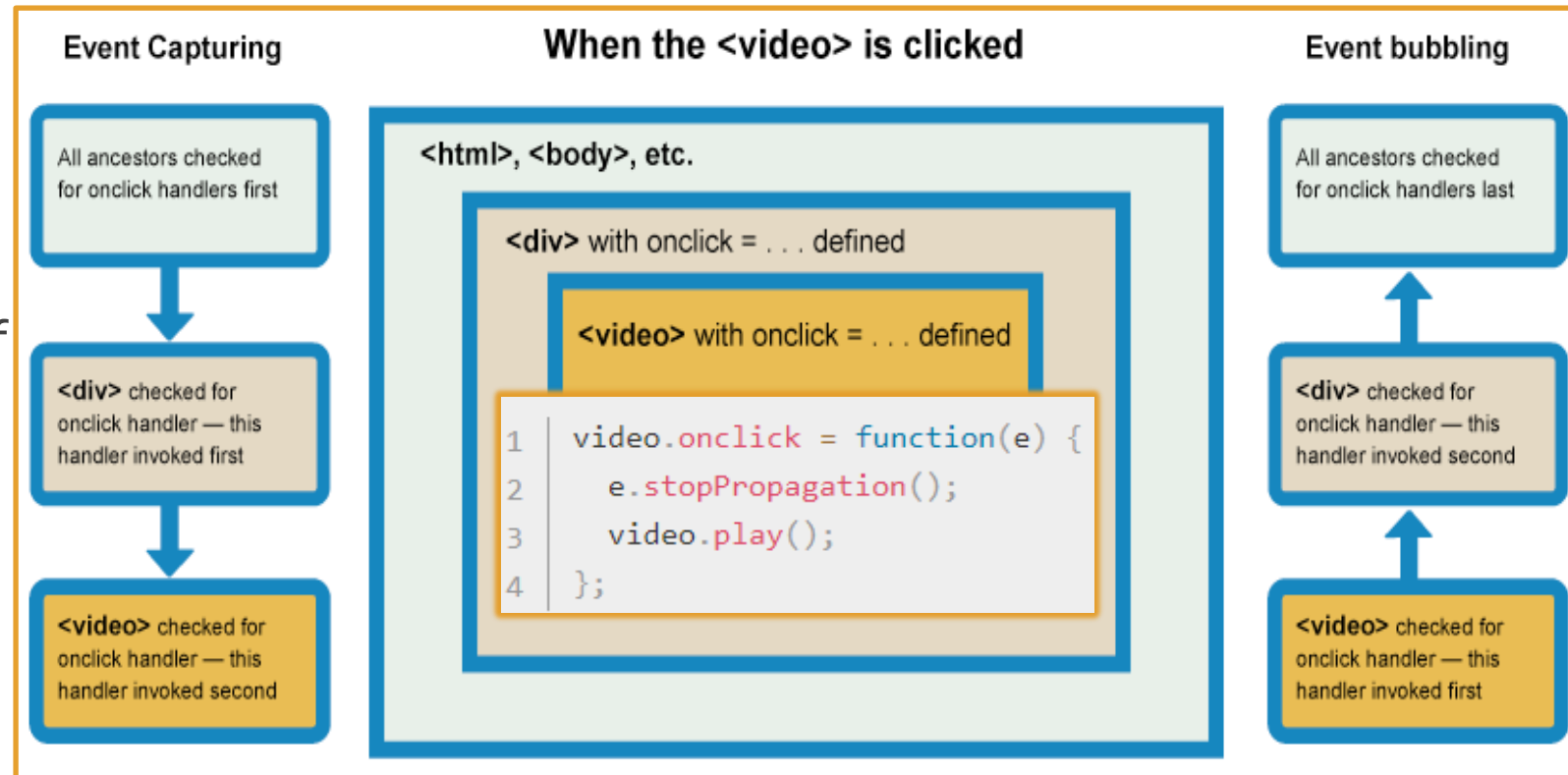
# Bubbling and Capture

[https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model/Examples#Example\\_5:\\_Event\\_Propagation](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Examples#Example_5:_Event_Propagation)  
[https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building\\_blocks/Events](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events)

Browsers automatically register event handlers for the *bubbling* phase.

When you click a video, the click event bubbles from the **<video>** element outwards to its parent **<div>** element, then to the **<html>** element. Along the way, if any of these elements has an **‘on-click’** event handler, they will fire, too.

**.stopPropagation()** is provided to stop further propagation.



# Bubbling and Capture

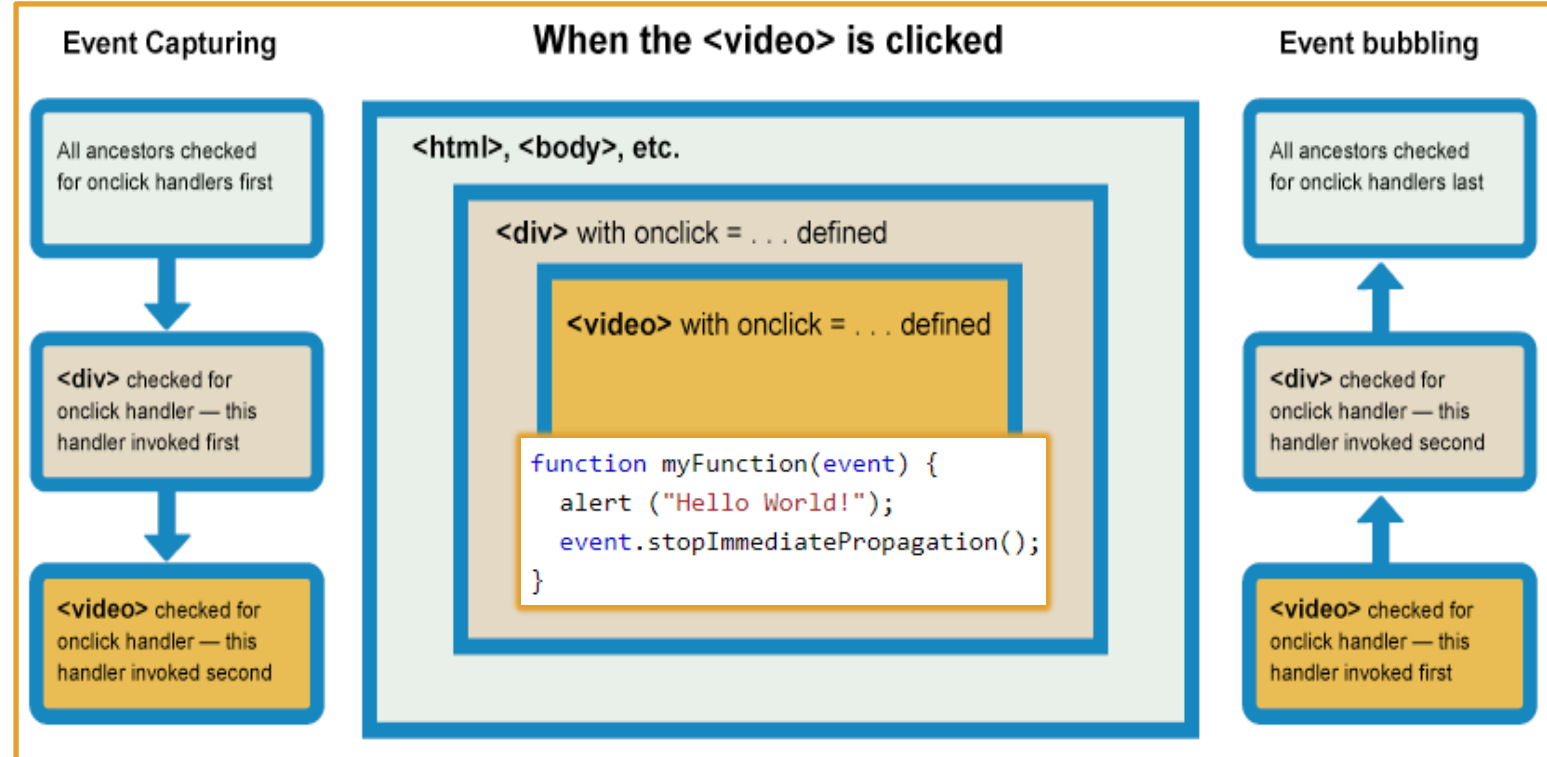
[https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model/Examples#Example\\_5:\\_Event\\_Propagation](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Examples#Example_5:_Event_Propagation)  
[https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building\\_blocks/Events](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events)

Browsers automatically register event handlers for the *bubbling* phase.

The ***stopImmediatePropagation()*** method prevents other listeners of the same event from being called.

If several listeners are attached to the same element for the same event type, they are called in the order in which they were added.

If ***stopImmediatePropagation()*** is invoked, no remaining listeners will be called.



# Walking the DOM - Basics

<https://javascript.info/dom-navigation>

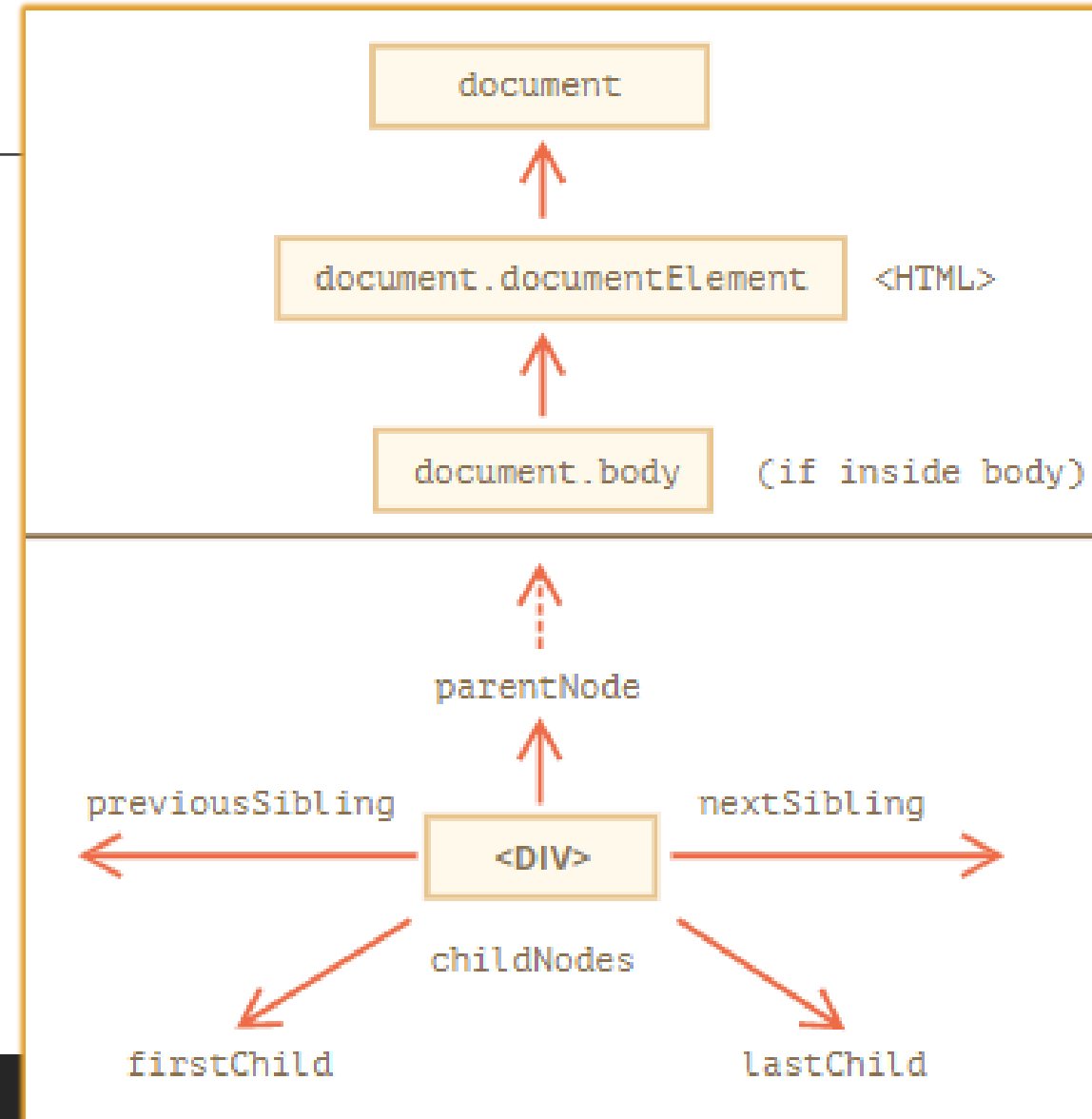
We can do anything with elements and their contents after reaching the correct DOM object.

The topmost tree nodes are available directly as document properties:

`<html> = document.documentElement`

`<body> = document.body`

`<head> = document.head`



# Walking the DOM - Basics

<https://javascript.info/dom-navigation>

---

Child nodes – Elements that are nested in the given element.  
<head> and <body> are children of <html> element.

Siblings - nodes that are children of the same parent.

Descendants – all elements that are nested in the given one, including children, their children, etc.

Here <body> has children <div> and <ul>.

<div> and <ul> are siblings

Descendants of <body> are not only direct children <div>, <ul> but also more deeply nested elements, such as <li> (a child of <ul>) and <b> (a child of <li>) – the entire subtree.

```
1  <html>
2  <body>
3    <div>Begin</div>
4
5    <ul>
6      <li>
7        <b>Information</b>
8      </li>
9    </ul>
10 </body>
11 </html>
```

# Walking the nodes of the DOM

<https://javascript.info/dom-navigation>

---

Method	Explanation/Example
<code>document.body.childNodes[i]</code>	<code>.childNodes</code> lists all child nodes in a (read-only) <u>collection</u> , including text nodes.
<code>.firstChild</code>	Gives access to the first child. <code>elem.firstChild</code>
<code>.lastChild</code>	Gives access to the last child. <code>elem.lastChild</code>
<code>.nextSibling</code>	Access the following or “right” sibling going down the page.
<code>.previousSibling</code>	Access the prior or “left” sibling going up the page.
<code>.parentNode</code>	Access the parent of the current node.

# Walking the elements of the DOM

<https://javascript.info/dom-navigation>

---

Method	Explanation/Example
.firstElementChild	Gives access to the first child.
.lastElementChild	Gives access to the last child.
.nextElementSibling	Access the following or “right” sibling element going down the page.
.previousElementSibling	Access the prior or “left” sibling element going up the page.
.parentElement	Access the parent of the current node if it’s an element. Returns <i>null</i> if not an element
.children	Returns all children elements.



# IIFE

## Immediately Invoked Function Expression

<https://developer.mozilla.org/en-US/docs/Glossary/IIFE>

An **IIFE** (*Immediately Invoked Function Expression*) (pronounced “iffy”) is a JavaScript function that runs as soon as it is defined. It’s also known as a Self-Executing Anonymous Function

IIFE functions contain two major parts:

- The first is the anonymous function with lexical scope enclosed within the Grouping Operator (). This prevents accessing variables within the IIFE idiom as well as polluting the global scope.
- The second part creates the immediately invoked function expression () through which the JavaScript engine will directly interpret the function.

```
1  (function () {  
2      statements  
3  })();
```

```
1  (function() {  
2      alert("I am not an IIFE yet!");  
3  });
```

```
1  // Variation 1  
2  (function() {  
3      alert("I am an IIFE!");  
4  })();
```

# IIFE

## Immediately Invoked Function Expression

<https://developer.mozilla.org/en-US/docs/Glossary/IIFE>

---

```
1  (function () {  
2      var aName = "Barry";  
3  })();  
4  // Variable aName is not accessible from the outside scope  
5  aName // throws "Uncaught ReferenceError: aName is not defined"
```

Any variable declared within the expression can not be accessed from outside it.

Assigning an *IIFE* to a variable stores the function's return value, not the function definition itself.

```
1  var result = (function () {  
2      var name = "Barry";  
3      return name;  
4  })();  
5  // Immediately creates the output:  
6  result; // "Barry"
```

# DOM Events Order

[https://developer.mozilla.org/en-US/docs/Web/API/Document/DOMContentLoaded\\_event](https://developer.mozilla.org/en-US/docs/Web/API/Document/DOMContentLoaded_event)

---

The ***DOMContentLoaded*** event fires when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading.

A different event, `load`, should be used only to detect a fully-loaded page. It is a common mistake to use `load` where ***DOMContentLoaded*** would be more appropriate.

Synchronous JavaScript pauses parsing of the DOM. To parse the DOM as fast as possible after the user has requested the page, make your JavaScript asynchronous to optimize the loading of stylesheets.

If loaded as usual, stylesheets slow down ***DOM*** parsing as they're loaded in parallel, "stealing" traffic from the main HTML document.

```
1 <script>
2   document.addEventListener('DOMContentLoaded', (event) => {
3     console.log('DOM fully loaded and parsed');
4   });
5
6   for( let i = 0; i < 1000000000; i++)
7   {} // This synchronous script is going to delay parsing of the DOM,
8     // so the DOMContentLoaded event is going to launch later.
9 </script>
```

# DOM Events Order

[https://developer.mozilla.org/en-US/docs/Web/API/Document/DOMContentLoaded\\_event](https://developer.mozilla.org/en-US/docs/Web/API/Document/DOMContentLoaded_event)

***DOMContentLoaded*** may fire before your script has a chance to run, so it is wise to check before adding a listener.

## HTML

```
1 <div class="controls">
2   <button id="reload" type="button">Reload</button>
3 </div>
4
5 <div class="event-log">
6   <label>Event log:</label>
7   <textarea readonly class="event-log-contents" rows="8" cols="30"></textarea>
8 </div>
```

## JS

```
1 const log = document.querySelector('.event-log-contents');
2 const reload = document.querySelector('#reload');
3
4 reload.addEventListener('click', () => {
5   log.textContent = '';
6   window.setTimeout(() => {
7     window.location.reload(true);
8   }, 200);
9 });
10
11 window.addEventListener('load', (event) => {
12   log.textContent = log.textContent + 'load\n';
13 });
14
15 document.addEventListener('readystatechange', (event) => {
16   log.textContent = log.textContent + `readystate: ${document.readyState}\n`;
17 });
18
19 document.addEventListener('DOMContentLoaded', (event) => {
20   log.textContent = log.textContent + `DOMContentLoaded\n`;
21 });
```

## Result of the above

Reload

Event log:

```
readystate: interactive
DOMContentLoaded
readystate: complete
load
```

# GuessingGame Tutorial

[https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First\\_steps/A\\_first\\_splash](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/A_first_splash)

---

1. Complete the guessingGame Tutorial.
2. Change guessingGame from using *events* to using a *form* to get the number.
3. Use <https://javascript.info/ui> for independent study.