

The JavaScript Language - JS for Web

What you need to know...

- Client-Side JavaScript
 - JavaScript in web browsers
 - The window object
 - Scripting documents (DOM)
 - Scripting CSS
 - Handling events
 - Client-Side Storage
 - Scripted Media and Graphics
 - HTML5 APIs

The Window object

- The Window object is the main entry point to all client- side JavaScript features and APIs
- It represents a web browser window or frame, and you can refer to it with the identifier “window”
- The Window object defines properties like location (URL currently displayed in the window) and methods like alert(), which displays a message in a dialog box or setTimeout(), which registers a function to be invoked after a specified amount of time
- The Window object is also the global object
- its properties and methods are global variables and global functions

```
// Set the location property to navigate to a new web page
window.location = "http://www.google.it/";
// Wait 2 seconds and then say hello
setTimeout(function() { alert("hello world"); }, 2000);
```

The Document object

- One of the most important properties of the Window object is document: it refers to a Document object that represents the content displayed in the window
- The Document object has important methods such as getElementById()
- It returns a single document element (representing an open/close pair of HTML tags and all of the content between them) based on the value of its id attribute

```
// Find the element with id="timestamp"
let timestamp = document.getElementById("timestamp");
```

The Document object

- The Element object returned by `getElementById()` has other important properties and methods that allow scripts to get its content, set the value of its attributes, and so on

```
// If the element is empty, then insert the current date
// and time into it
if (timestamp.firstChild == null)
timestamp.appendChild(document.createTextNode`Node` (new Date().toString()));
```

- Each Element object has `style` and `className` properties that allow scripts to specify CSS styles for a document element or to alter the CSS class names that apply to the element

```
// Explicitly alter the presentation of the heading element
timestamp.style.backgroundColor = "yellow";
// Or just change the class and let the stylesheet specify the details
timestamp.className = "highlight";
```

Event handler properties

- Important properties on Window, Document, and Element objects
- Allow scripts to specify functions that should be invoked asynchronously when certain events occur
- Allow JavaScript code to alter the behavior of windows, documents, and elements
- Event handler properties have names that begin with the word “on”

```
// Update the content of the timestamp element when the user
// clicks on it
timestamp.onclick = function() {
this.innerHTML = new Date().toString();
}
```

Example

```
<!DOCTYPE html>
<html>
<head>
<style>
/*CSS styles for this page*/
.reveal *{ display: none; } /* Children of class="reveal" are not shown */
.reveal*.handle { display: block;} /*Except for the class="handle" child*/
</style>
<script>
// Don't do anything until the entire document has loaded
```

```
window.onload = function() {  
  // Find all container elements with class "reveal"  
  let elements = document.getElementsByClassName("reveal");  
  for(let i = 0; i < elements.length; i++) { // For each one...  
    let elt = elements[i];  
    // Find the "handle" element with the container  
    let title = elt.getElementsByClassName("handle")[0];  
    // When that element is clicked, reveal the rest of the content  
    title.onclick = function() {  
      if (elt.className == "reveal") elt.className = "revealed";  
      else if (elt.className == "revealed") elt.className = "reveal";  
    }  
  }  
};  
</script>  
</head>
```

Example

```
<body>  
<div class="reveal">  
<h1 class="handle">Click Here to Reveal Hidden Text</h1>  
<p>This paragraph is hidden. It appears when you click on the title.</p>  
</div>  
</body>  
</html>
```

- JavaScript program can traverse and manipulate document content through the Document object and the Element objects it contains
- It can alter the presentation of that content by scripting CSS styles and classes
- It can define the behavior of document elements by registering appropriate event handlers
- The combination of scriptable content, presentation, and behavior is called Dynamic HTML

Embedding JavaScript in HTML

Client-side JavaScript code is embedded within HTML documents in four ways:

- Inline, between a pair of `<script>` and `</script>` tags
- From an external file specified by the `src` attribute of a `<script>` tag
- In an HTML event handler attribute, such as `onclick` or `onmouseover`
- In a URL that uses the special `javascript:` protocol.
- A programming philosophy known as unobtrusive JavaScript argues that content (HTML) and behavior (JavaScript code) should as much as possible be kept separate
- JavaScript is best embedded in HTML documents using `<script>` elements with `src` attributes

Inline JavaScript code

- Between `<script>` and `</script>` tags
- Example: simple digital clock

```
<!DOCTYPE html> <!-- This is an HTML5 file -->
<html> <!-- The root element -->
<head> <!-- Title, scripts & styles go here -->
<title>Digital Clock</title>
<script>
// A script of js code
// Define a function to display the current time
function displayTime() {
let elt = document.getElementById("clock"); // Find element with id="clock"
let now = new Date(); // Get current time
elt.innerHTML = now.toLocaleTimeString();
// Make elt display it
setTimeout(displayTime, 1000);
// Run again in 1 second
}
window.onload = displayTime;
// Start displaying the time when document loads.
</script>
```

Inline JavaScript code

```
<style>
/*A CSS stylesheet for the clock*/
/*Style apply to element with id="clock" */
# clock {
/*Use a big bold font*/
font: bold 24pt sans;
/*On a light bluish-gray background */
background: #ddf;
/*Surround it with some space*/
padding: 10px;
/*Round the corners (where supported)*/
border: solid black 2px;
/*And a solid black border */
border-radius: 10px;
}
</style>
```

```
</head>
<body> <!-- The body is the displayed parts of the doc. -->
<h1>Digital Clock</h1> <!-- Display a title -->
<span id="clock"></span> <!-- The time gets inserted here -->
</body>
</html>
```

External JavaScript code

- Scripts can be placed in external files
- Useful when the same code is used in many different web pages
- Can be called in or
- JavaScript files: extension .js
- A JavaScript file contains pure JavaScript, without `<script>` tags or any other HTML

```
<!DOCTYPE html>
<html>
<body>
<script src="myScript.js"></script>
</body>
</html>
```

Event handlers in HTML

- JavaScript code in a script is executed once: when the HTML file that contains it is loaded into the web browser
- To be interactive, a JavaScript program must define event handlers
- JavaScript code can register an event handler by assigning a function to a property (such as "onclick" or "onmouseover") of an Element object
- Typically an HTML event handler attribute consists of a simple assignment or a simple invocation of a function defined elsewhere

```
<input type='button' value='Change color' id='theSubmit'
onclick="document.bgColor=changeColor()" />
<input type='reset' value='White' id='theReset'
onclick="document.bgColor='white'" />
```

Basic interaction methods

- Alert dialog box
- OK to confirm

- Mostly used to give a warning message to the users

```
<head>
<script type="text/javascript">
<!--
alert("Warning Message");
//-->
</script>
</head>
```

Basic interaction methods

- Confirmation dialog box
- OK, cancel
- True if user clicks on OK
- Mostly used to take user's consent on any option

```
<script type="text/javascript">
let retVal = confirm("Do you want to continue ?");
if( retVal == true ){
alert("User wants to continue!");
}else{
alert("User does not want to continue!");
}
</script>
```

Basic interaction methods

- Prompt dialog box
- Returns a string with the text written by the user
- Returns null if user clicks on Cancel
- Used to get user input

```
<script type="text/javascript">
<!--
let retVal = prompt("Enter your name : ",
"your name here");
alert("Hello " + retVal );
//-->
</script>
```

JavaScript display possibilities

- JavaScript can “display” data in different ways
 - Writing into an alert box: `window.alert()`
 - Writing into the HTML output: `document.write()`
 - Writing into an HTML element: `innerHTML`
 - Writing into the browser console: `console.log()`
 - http://www.w3schools.com/js/js_output.asp
-

JavaScript display possibilities

- Using `document.write()` after an HTML document is fully loaded deletes all existing HTML
- `document.write()` is useful only for testing purposes

```
<!DOCTYPE html>
<html>
<body>
<h1>Esempio</h1>
<p>Quanto fa 5 + 6 ?</p>
<button type="button"
onclick="document.write(5 + 6) ">Prova</button>
</body>
</html>
```

JavaScript display possibilities

- To access an HTML element, JavaScript can use the `document.getElementById(id)` method
- The `id` attribute defines the HTML element
- The `innerHTML` property defines the HTML content

```
<!DOCTYPE html>
<html>
<body>
<h1>Esempio</h1>
<p>Quanto fa 5 + 6 ?</p>
<p id="demo"></p>
<button type="button"
onclick="document.getElementById('demo').innerHTML
= 5 + 6;">Prova</button>
</body>
</html>
```

JavaScript display possibilities

- Example of innerHTML property

```
<!DOCTYPE html>
<html>
<body>
<h1>Esempio</h1>
<button type="button" onclick=
"document.getElementById('demo').innerHTML = Date()">
Premi qui per sapere data e ora</button>
<p id="demo"></p>
</body>
</html>
```

JavaScript display possibilities

- In your browser, you can use the console.log() method to display data

```
<!DOCTYPE html>
<html>
<body>
<h1>My First Web Page</h1>
<p>My first paragraph.</p>
<p>
Activate debugging in your browser (Chrome, IE, Firefox) with F12, and
select "Console" in the debugger menu.
</p>
<script>
console.log(5 + 6);
</script>
</body>
</html>
```

Scripting documents

- Client-side JavaScript exists to turn static HTML documents into interactive web applications
- Scripting the content of web pages is the central purpose of JavaScript
- Overview
- Basic architecture of the DOM
- How to query or select individual elements from a document
- How to traverse a document as a tree of **nodes**: how to find the ancestors, siblings, and descendants of any document element
- How to query and set the attributes of document elements
- How to query, set, and modify the content of a document
- How to modify the structure of a document by creating, inserting, and deleting **nodes**
- How to work with HTML forms

Document Object Model (DOM)

- The Document Object Model, or DOM, is the fundamental API for representing and manipulating the content of HTML (and XML) documents
 - HTML or XML document are represented in the DOM as a tree of objects
-

Tree structure

- HTML documents are trees

The DOM structure

- The entire document is a Document **node**
 - If you want to access objects in an HTML page, you always start with accessing the document object
 - Every HTML tag is an Element **node**
 - The texts contained in the HTML elements are Text **nodes**
 - Every HTML attribute is an Attribute **node**
 - Comments are Comment **nodes**
 - **Nodes** have a hierarchical relationship to each other
-

The **Node** object

- Hierarchy of the subclasses of **Node**
-

Selecting document elements

- Most client-side JavaScript programs work by manipulating one or more document elements
 - They must obtain or select the Element objects that refer to those document elements
 - The DOM defines a number of ways to select elements: you can query a document for an element (or elements)
 - with a specified id attribute
 - with a specified name attribute
 - with the specified tag name
 - with the specified CSS class or classes
 - matching the specified CSS selector
-

Selecting elements by id

- Any HTML element can have an id attribute
- The value of this attribute must be unique within the document

- You can select an element based on this unique id with the `getElementById()` method of the Document object
- The simplest and most commonly used way to select elements

```
let section1 = document.getElementById("section1");
/*
 * This function expects any number of string arguments. It treats each
 * argument as an element id and calls document.`getElementById()` for each.
 */
function getElements(/*ids...*/) {
  let elements = {}; // Start with an empty map
  for(let i = 0; i < arguments.length; i++) {
    // For each argument
    let id = arguments[i];
    // Argument is an element id
    let elt = document.getElementById(id);
    // Look up the Element
    if (elt == null) // If not defined,
      throw new Error("No element with id: " + id);
    // throw an error
    elements[id] = elt; // Map id to element
  }
  return elements; // Return id to element map
}
```

Selecting elements by name

- The HTML name attribute was originally intended to assign names to form elements, and the value of this attribute is used when form data is submitted to a server
- Name assigns a name to an element, but unlike id the value of a name attribute does not have to be unique
- Common in the case of radio buttons and checkboxes in forms
- Unlike id, the name attribute is only valid on a handful of HTML elements, including forms, form elements, `<iframe>`, and `` elements
- You can select an element based on its name attribute with the `getElementsByName()` method
- `let radiobuttons = document.getElementsByName("favorite_color");`

Selecting elements by tag name

- You can select all HTML elements of a specified type (or tag name) using the `getElementsByTagName()` method
- Like `getElementsByName()`, `getElementsByTagName()` returns a `NodeList` object
- The elements of the returned `NodeList` are in document order
- `NodeList` objects are read-only array-like objects
- They have length properties and can be indexed (for reading but not writing) like true arrays: you can iterate the contents of a `NodeList` or HTML collections with a standard loop

- Properties like `document.images` and `document.forms` are **HTMLCollection** objects

```
let spans = document.getElementsByTagName("span");
let firstpara = document.getElementsByTagName("p")[0];
let firstParaSpans = firstpara.getElementsByTagName("span");
for(let i = 0; i < document.images.length; i++) // Loop through all images
document.images[i].style.display = "none"; // ...and hide them.
```

Selecting elements by CSS class

- The class attribute of an HTML element is a space-separated list of zero or more identifiers
- It describes a way to define sets of related document elements: any elements that have the same identifier in their class attribute are part of the same set
- JavaScript uses the `className` property to hold the value of the HTML class attribute
- You can select all HTML elements of a specified CSS class using the `getElementsByClassName()` method
- `getElementsByClassName()` returns a live **NodeList**
- `getElementsByClassName()` takes a single string argument, but the string may specify multiple space-separated identifiers

```
// Find all elements that have "warning" in their class attribute
let warnings = document.getElementsByClassName("warning");
// Find all descendants of the element named "log" that have the class
// "error" and the class "fatal"
let log = document.getElementById("log");
let fatal = log.getElementsByClassName("fatal error");
```

Selecting elements with CSS Selectors

- CSS stylesheets have a very powerful syntax, known as selectors, for describing elements or sets of elements within a document
- The Document method `querySelectorAll()` takes a single string argument containing a CSS selector and returns a **NodeList** that represents all elements in the document that match the selector
- The Document object also defines `querySelector()`, that returns only the first (in document order) matching element

```
// Returns a list of all div elements within the document with a class
// of either "note" or "alert"
let matches = document.querySelectorAll("div.note, div.alert");
// Returns the first element <input> with the attribute name equal to
// "login" within a <div> of class="user-panel main"
let el = document.querySelector("div.user-panel.main input[name=login]");
```

Document structure and traversal

- Once you have selected an Element from a Document, you sometimes need to find structurally related portions (parent, siblings, children) of the document
 - A Document can be conceptualized as a tree of `Node` objects
-

DOM and `node` 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
-

The `Node` object properties

Navigation among `nodes`

- `Node` properties useful to navigate between `nodes`
- `parentNode`
- `childNodes[nodenum]`
- `firstChild`
- `lastChild`
- `nextSibling`
- `previousSibling`

Other `node` properties

- `nodeName`
- `nodeValue`
- `nodeType`

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

Document traversal

- Example

```
<html>
<head>
<title>Test</title>
</head>
<body>
Hello World!
```

```
</body>
</html>
// The second child of the first child (it is the <body> element
document.childNodes[0].childNodes[1]
// The same
document.firstChild.firstChild.nextSibling
```

Example

- Collects the `node` value of an

element and copies it into a
element

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

HTML attributes as Element properties

- The `HTMLElement` objects that represent the elements of an HTML document define read/write properties that mirror the HTML attributes of the elements
- Note: HTML attributes are not case sensitive, but JavaScript property names are
- The `Element` type also defines `getAttribute()` and `setAttribute()` methods to query and set nonstandard HTML attributes

```
let image = document.getElementById("myimage");
let imgurl = image.src;
// The src attribute is the URL of the image
```

```
let f = document.forms[0];
// First <form> in the document
f.action = "http://www.example.com/submit.php";
// Set URL to submit it to.
f.method = "POST";
// HTTP request type
let image = document.images[0];
let width = parseInt(image.getAttribute("WIDTH"));
image.setAttribute("class", "thumbnail");
```

Element content

- Reading the innerHTML property of an Element returns the content of that element as a string of markup
- Setting this property on an element invokes the web browser's parser and replaces the element's current content with a parsed representation of the new string

```
<!DOCTYPE html>
<html>
<body>
<h1 id="header">Old Header</h1>
<script>
let element = document.getElementById("header");
element.innerHTML = "New Header";
</script>
<p>"Old Header" was changed to "New Header"</p>
</body>
</html>
```

Creating, inserting, and deleting nodes

- It is also possible to alter a document at the level of individual **nodes**
- The Document type defines methods for creating Element and Text objects
- The **Node** type defines methods for inserting, deleting, and replacing **nodes** in the tree

```
// Asynchronously load and execute a script from a specified URL
function loadasync(url) {
// Find document <head>
let head = document.getElementsByTagName("head")[0];
// Create a <script> element
let s = document.createElement("script");
// Set its src attribute
s.src = url;
// Insert the <script> into head
head.appendChild(s);
}
```

Creating nodes

- 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
- The createElement() method of the Document object creates new Element **nodes**
- The createTextNode() method of the Document object creates new Text **nodes**
- The createComment() method of the Document object creates new Comment **nodes**

```
let s = document.createElement("script"); let newnode =  
document.createTextNode("text node content");
```

Inserting nodes

- Once you have a new **node**, you can insert it into the document with the **Node** methods appendChild() or insertBefore()
- appendChild() is invoked on the Element **node** that you want to insert into, and it inserts the specified **node** so that it becomes the lastChild of that **node**
- insertBefore() takes two arguments: the first argument is the **node** to be inserted; the second argument is the **node** before which that **node** is to be inserted. This method is invoked on the **node** that will be the parent of the new **node**
- If you pass null as that second argument of insertBefore(), it behaves like appendChild()

```
// Insert the child `node` into parent so that it becomes child `node` n  
function insertAt(parent, child, n) {  
  if (n < 0 || n > parent.childNodes.length)  
    throw new Error("invalid index");  
  else if (n == parent.childNodes.length) parent.appendChild(child);  
  else parent.insertBefore(child, parent.childNodes[n]);  
}
```

Removing and replacing nodes

- The removeChild() method removes a **node** from the document tree
 - Invoke the method on the parent **node** and pass the child **node** that is to be removed as the method argument
 - Example: remove the **node** n from the document n.parentNode.removeChild(n);
 - n.parentNode.removeChild(n);
 - The replaceChild() method removes one child **node** and replaces it with a new one
 - Invoke this method on the parent **node**, passing the new **node** as the first argument and the **node** to be replaced as the second argument
-

Replacing nodes

- Example

```
<!DOCTYPE html>
<html>
<body>
<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");
let para = document.createElement("p");
let `node` = document.createTextNode("This is new.");
para.appendChild(`node`);
parent.replaceChild(para,child);
</script>
</body>
</html>
```

HTML forms

- Tool to gather input from the user
- Server-side programs are based on form submissions
- They process data in form-sized chunks
- Client-side programs are event based
- They can respond to events on individual form elements
- E.g.: a client-side program might validate the user's input as she types it, or it might respond to a click on a checkbox by enabling a set of options that are only meaningful when that box is checked

HTML form elements and handlers

Selecting forms and form elements

- Standard methods like `getElementById()` and `getElementsByTagName`, or `querySelectorAll()`
- Through the **HTMLCollection** document.forms
- Allows form elements to be selected by numerical order, by id, or by name

```
let fields =
document.getElementById("address").getElementsByTagName("input");
// All radio buttons in the form with id "shipping"
document.querySelectorAll('#shipping input[type="radio"]');
// All radio buttons with name "method" in form with id "shipping"
document.querySelectorAll('#shipping input[type="radio"][name="method"]');
document.forms.address
// Explicit access to a form with name or id
```



```
document.forms[n]
// n is the form's numerical position
document.forms.address[0]
document.forms.address.street
document.address.street // only for name="address", not id="address"
```

Form and element event handlers

- Examples

```
// Ask the user to confirm the reset
<form... onreset="return confirm('Really erase ALL input and start
over?')">
...
<button type="reset">Clear and Start Over</button>
</form>
Enter some text: <input type="text" name="txt" value="Hello"
onchange="myFunction(this.value)">
<script>
function myFunction(val) {
alert("The input value has changed. The new value is: " + val);
}
</script>
// Create a new Option object
let zaire = new Option("Zaire", // The text property
"zaire", // The value property
false, // The defaultSelected property
false); // The selected property
// Display it in a Select element by appending it to the options array:
let countries = document.address.country; // Get the Select object
countries.options[countries.options.length] = zaire;
```

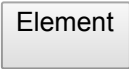
Scripting CSS style

- Through the style property of the Element `node`
- `style.property=new style`
- Examples

```
e.style.position = "absolute";
e.style.fontFamily = "sans-serif";
e.style.backgroundColor = "#ffffff";
e.style.left = "300px";
// Set the style attribute of e to the string s with either
// of these lines:
e.setAttribute("style", s);
e.style.cssText = s;
// Query the inline style string of the element e with
```

```
either of these:  
s = e.getAttribute("style");  
s = e.style.cssText;
```

Handling events

- Client-side JavaScript programs use an asynchronous event-driven programming model
- Events are simply occurrences that a web browser will notify your program about
- Events are not JavaScript objects and have no manifestation in the source code of your program
- Event type
 - A string that specifies what kind of event occurred
 - E.g., the type "mousemove" means that the user moved the mouse, the type "load" means that a document (or some other resource) has finished loading from the network
- Event target
 - The object on which the event occurred or with which the event is associated
 - E.g., a load event on a Window, a click event on a 
- Event handler or event listener
 - A function that handles or responds to an event
- Event object
 - An object that is associated with a particular event and contains details about that event
 - Passed as an argument to the event handler function
 - Each event type defines a set of properties for its associated event object
 - E.g., the object associated with a mouse event includes the coordinates of the mouse pointer, the object associated with a keyboard event contains details about the key that was pressed

Handling events

- Event propagation
 - The process by which the browser decides which objects to trigger event handlers on
 - For events that are specific to a single object (such as the load event on the Window object), no propagation is required
 - When certain kinds of events occur on document elements, however, they propagate or "bubble" up the document tree
 - Some events have default actions associated with them

- E.g., when a click event occurs on a hyperlink, for example, the default action is for the browser to follow the link and load a new page
- Event handlers can prevent default actions

Handling events

- JavaScript code can be executed when an event occurs
- Simplest way: assign event attributes to HTML elements you can use (e.g onclick, onload, onchange, ...)
- event=JavaScript code
- Javascript code can be an expression or a function call
- Alternative: write event handlers
- `addEventListener()` method
- First advantage: you can add many event handlers of the same type to one element, i.e two "click" events.
- Second advantage: JavaScript is separated from the HTML markup for better readability
- Third advantage: allows you to add event listeners even when you do not control the HTML markup

Handling events

- Example with JavaScript expression

```
<!DOCTYPE html>
<html>
<body>
<p id="p1">
The HTML DOM allows you to execute code when an event occurs.
</p>
<input type="button" value="Hide text"
onclick="document.getElementById('p1').style.visibility='hidden'">
<input type="button" value="Show text"
onclick="document.getElementById('p1').style.visibility='visible'">
</body>
</html>
```

Handling events

- Example with function call

```
<!DOCTYPE html>
<html>
<body>
<div onmousedown="mDown(this)" onmouseup="mUp(this)"
style="background-color:#D94A38;width:120px;height:20px;
padding:40px;color:white;font-family:Arial;font-weight:bold;">
```

```
Click Me</div>
<script>
function mDown(obj) {
obj.style.backgroundColor = "#1ec5e5";
obj.innerHTML = "Release Me";
}
function mUp(obj) {
obj.style.backgroundColor="#D94A38";
obj.innerHTML="Thank You";
}
</script>
</body>
</html>
```

addEventListener()

- The `addEventListener()` method attaches an event handler to the specified element
- `element.addEventListener(event, function, useCapture)`
- The first parameter is the type of the event (like "click" or "mousedown") * note: "click", not "onclick"
- The second parameter is the function to be called when the event occurs
- The third parameter (optional) is a boolean value specifying whether to use event bubbling or event capturing
- You can add more than one event handler to the same element

addEventListener()

- Advantages
- The `addEventListener()` method attaches an event handler to an element without overwriting existing event handlers
- You can add many event handlers of the same type to one element, i.e two "click" events
- You can add event listeners to any DOM object not only HTML elements, i.e the window object
- The `addEventListener()` method makes it easier to control how the event reacts to bubbling
- When using the `addEventListener()` method, the JavaScript is separated from the HTML markup, for better readability and allows you to add event listeners even when you do not control the HTML markup
- You can easily remove an event listener by using the `removeEventListener()` method

addEventListener()

- Example: single event handler

```
<!DOCTYPE html>
<html>
<body>
<p>This example uses the `addEventListener()` method to attach
a click event to a button.</p>
```

```

<button id="myBtn">Try it</button>
<script>
document.getElementById("myBtn").addEventListener("click", function()
{
alert("Hello World!");
}
);
</script>
</body>
</html>

```

addEventListener()

- Example: multiple event handlers

```

<p>This example uses the `addEventListener()` method to add many events on
the same
button.</p>
<button id="myBtn">Try it</button>
<p id="demo"></p>
<script>
let x = document.getElementById("myBtn");
x.addEventListener("mouseover", myFunction);
x.addEventListener("click", mySecondFunction);
x.addEventListener("mouseout", myThirdFunction);
function myFunction() {
document.getElementById("demo").innerHTML += "Moused over!<br>"; }
function mySecondFunction() {
document.getElementById("demo").innerHTML += "Clicked!<br>"; }
function myThirdFunction() {
document.getElementById("demo").innerHTML += "Moused out!<br>"; }
</script>

```

Event propagation

- Two ways of event propagation in the HTML DOM: bubbling and capturing
- If you have a
 - element inside a
 - element, and the user clicks on the
 - element, which element's "click" event should be handled first?
- Bubbling (default, useCapture=false): the inner most element's event is handled first and then the outer one
- Capturing (useCapture=true): the outer most element's event is handled first and then the inner one
- Example

References

- [ECMAScript® 2015 Language Specification](#)
- [Mozilla Developer Network](#)
- [JSbooks](#)
- [Slides embedded references](#)

License

- This work is licensed under the Creative Commons “Attribution- NonCommercial-ShareAlike Unported (CC BY-NC-SA 3,0)” License.
- You are free:
- to Share - to copy, distribute and transmit the work
- to Remix - to adapt the work
- Under the following conditions:
- Attribution - You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Noncommercial - You may not use this work for commercial purposes.
- Share Alike - If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.
- To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/>