-----------------------------------------------------------------------------------------------------------------------------------------------------------------------

**JS Best Practices**

|  |  |  |
| --- | --- | --- |
| Global Variables | Global variables and functions can be overwritten by other scripts => Avoid |  |
| Local Variables | Always declare with the **var** keyword or the **let** keyword |  |
| Declaration | Should be at the top of each script or function. | // Declare at the beginning let firstName, lastName, price, discount, fullPrice;  // Use later firstName = "John"; lastName = "Doe"; |
| Initialize Variables | Should be initialized when you declaring | // Declare and initiate at the beginning let firstName = "", let lastName = "", |
| Declare Objects | Declaring with **const** to prevent any accidential change of type | const car = {type:"Fiat", model:"500", color:"white"}; car = "Fiat";      // Not possible |
| Declare Arrays | Declaring with **const** to prevent any accidential change of type | const cars = ["Saab", "Volvo", "BMW"]; cars = 3;    // Not possible |
| new Object() | Don't Use new Object()  - Use "" instead of new String()  - Use 0 instead of new Number()  - Use false instead of new Boolean()  - Use {} instead of new Object()  - Use [] instead of new Array()  - Use /()/ instead of new RegExp()  - Use function (){} instead of new Function() | let x1 = "";             // new primitive string let x2 = 0;              // new primitive number let x3 = false;          // new primitive boolean const x4 = {};           // new object const x5 = [];           // new array object const x6 = /()/;         // new regexp object const x7 = function(){}; // new function object |
| Automatic Type Conversions | Beware of Automatic Type Conversions:  A variable can change its data type | let x = "Hello";     // typeof x is a string x = 5;               // changes typeof x to a number  let x = 5 + "7";     // typeof x is a string  "Hello" - "Dolly"    // returns NaN |
| Comparison | Using === operator forces comparison of values and type | 0 == "";        // true 0 === "";       // false |
| Parameter Defaults | Missing argument, the value is set to undefined.  => It's a good practice to assign default values to arguments. | function myFunction(x, y) {   if (y === undefined) {     y = 0;   } } |
| Switches with Defaults | Always end switch statements with a default.  Even if you think there is no need for it. | switch (new Date().getDay()) {   case 0:     day = "Sunday";     break;  // some code    case 6:     day = "Saturday";     break;   default:     day = "Unknown"; } |
| Primative Values  or Objects | Not treat numbers, strings, or booleans as objects.  slow down execution speed, and produces nasty side effects: | let x = "John";              let y = new String("John"); (x === y) // is false because x is a string and y is an object. |
|  |  | let x = new String("John");              let y = new String("John"); (x == y) // is false because you cannot compare objects. |
| eval() | it should not be necessary to use it. Because it allows arbitrary code to be run, it also represents a security problem. |  |

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------

**JS Common Mistakes**

|  |  |  |
| --- | --- | --- |
| Assignment Operator | assignment operator (=), instead of a comparison operator (==) in an if statement. | let x = 0; if (x = 10) // return true  let x = 0; if (x = 0) // return false because 0 is false |
| Loose or Strict Comparison | switch statements use strict comparison: | let x = 10; switch(x) {   case "10": alert("Hello"); } // not work |
| Addition & Concatenation | **Addition** is about adding **numbers**.  **Concatenation** is about adding **strings**. | let y = 10; y += "5";   // Now y is "105" |
| Floats | **Problem**: all programming languages have difficulties with precise floating point values: | let x = 0.1; let y = 0.2; let z = x + y  // the result in z will not be 0.3 |
|  | **Solution**: it helps to multiply and divide: | let z = (x \* 10 + y \* 10) / 10;  // z will be 0.3 |
| Breaking a JS String | Breaking a statement in the middle of a string will not work: | let x = "Hello World!"; // not work  // using \ instead  let x = "Hello \ World!"; |
| Semicolon | Misplacing Semicolon can make the code runing without checking | let x = 5;  if (x == 19);  {  document.getElementById("demo").innerHTML = "Hello";  } |
| Return Statement | By default, JS close a statement automatically at the end of a line =>  semicolon is optional in JavaScript. | function myFunction(a) {   let // not a complete statement => JS check the next line;   power = 10;   return a \* power; } // return: 550 |
|  | **Solution**: Never break a return statement. | function myFunction(a) {   let // not a complete statement => JS check the next line;   power = 10;    return // a complete statement => the same as return;    a \* power; } // return: undefined |
| Accessing Arrays  with Named Indexes | In many languages, arrays use named indexes:  In JS, arrays use numbered indexes: | const person = []; person[0] = "John"; person[1] = "Doe"; person[2] = 46; person.length;       // person.length will return 3 person[0];           // person[0] will return "John" |
|  | In JS, objects use named indexes. | const person = []; person["firstName"] = "John"; person["lastName"] = "Doe"; person["age"] = 46; person.length;      // person.length will return 0 person[0];          // person[0] will return undefined |
| Ending Definitions  with a Comma | Trailing commas in object and array is legal  But Internet Explorer 8 will crash => not neccessary | person = {"firstName":"John", "lastName":"Doe", "age":46,}  points = [40, 100, 1, 5, 25, 10,]; |
| Undefined  is Not Null | JS objects, variables, properties, methods can be undefined.  Empty JS objects can have the value null.  => difficult to test if an object is empty. |  |
|  | **Best solution**: test for not undefined before you can test for not null: | if (typeof myObj !== "undefined" && myObj !== null) |

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------

**JS Performance**

|  |  |  |
| --- | --- | --- |
| General | How to speed up your JavaScript code. |  |
| Reduce Activity  in Loops | Statements or assignments should be placed outside the loop to make the loop run faster. | // bad  for (let i = 0; i < arr.length; i++) {  // better  let l = arr.length; for (let i = 0; i < l; i++) { |
| Reduce DOM Access | **Problem**: accessing the HTML DOM is very slow, compared to other JS statements. |  |
|  | **Solution**: if you expect to access a DOM element several times, access it once, and use it as a local variable: | const obj = document.getElementById("demo");  obj.innerHTML = " Hello"; |
| Reduce DOM Size | Keep the number of elements in the HTML DOM small. |  |
| Avoid Unnecessary Variables | Don't create new variables if you don't plan to save values. | // bad  let fullName = firstName + " " + lastName; document.getElementById("demo").innerHTML = fullName;  // better  document.getElementById("demo").innerHTML = firstName + " " + lastName; |
| Delay JS Loading | 1. Putting your scripts at the bottom of the page body lets the browser load the page first. |  |
|  | 2. Use **defer="true"** in the script tag.  (it specifies that the script should be executed after the page has finished parsing, but it only works for external scripts.) |  |
|  | 3. add your script to the page by code, after the page has loaded: | <script> window.onload = function() {   const element = document.createElement("script");   element.src = "myScript.js";   document.body.appendChild(element); }; </script> |
| Avoid Using **with** | **with** keyword is not allowed in strict mode.  (It has a negative effect on speed. It also clutters up JS scopes) |  |

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**JS HTML DOM**

**General**

|  |  |  |
| --- | --- | --- |
| HTML DOM | **Document Object Model** is a standard for how to get, change, add, or delete HTML elements.  **DOM** defines:  - The HTML elements as objects  - The properties of all HTML elements  - The methods to access all HTML elements  - The events for all HTML elements | The HTML DOM Tree of Objects  DOM HTML tree |

**DOM Methods**

|  |  |  |
| --- | --- | --- |
| General | DOM methods are **actions** you can perform on HTML Elements. (like add or deleting an HTML element).  DOM properties are **values** (of HTML Elements) you can set, or change (like changing the content of an HTML element). | // changes the content (the innerHTML) of the <p> element  <p id="demo"></p>  <script> document.getElementById("demo").innerHTML = "Hello World!"; </script>  // getElementById is a **method**, while innerHTML is a **property**. |
| DOM Programming Interface | the properties and methods of each object. |  |
| getElementById() | The most common way to access an HTML element is to use the **ID** |  |
| innerHTML Property | useful for getting or replacing the content of HTML elements. |  |

**DOM Document**

|  |  |
| --- | --- |
| DOM Document Object | If you want to access any element in an HTML page, you always start with accessing the document object. |
| Finding Elements | **Methods**:  document.**getElementById**(id): Find an element by element id  document.**getElementsByTagName**(name): Find elements by tag name  document.**getElementsByClassName**(name): Find elements by class name |
| Changing HTML Elements | **Property**  element.**innerHTML** = new html content: Change the inner HTML of an element  element.**attribute** = new value: Change the attribute value of an HTML element  element.**style**.property = new style: Change the style of an HTML element  **Method**  element.**setAttribute**(attribute, value): Change the attribute value of an HTML element |
| Adding and Deleting Elements | **Method**  document.**createElement**(element) Create an HTML element  document.**removeChild**(element) Remove an HTML element  document.**appendChild**(element) Add an HTML element  document.**replaceChild**(new, old) Replace an HTML element  document.**write**(text) Write into the HTML output stream |
| Adding Events Handlers | **Method**  document.**getElementById**(id).onclick = function(){code} Adding event handler code to an onclick event |
| Finding HTML Objects | **Property**  document.anchors Returns all <a> elements that have a name attribute  document.baseURI Returns the absolute base URI of the document  document.body Returns the <body> element  document.cookie Returns the document's cookie  document.doctype Returns the document's doctype  document.documentElement Returns the <html> element  document.documentMode Returns the mode used by the browser  document.documentURI Returns the URI of the document  document.domain Returns the domain name of the document server  document.embeds Returns all <embed> elements  document.forms Returns all <form> elements  document.head Returns the <head> element  document.images Returns all <img> elements  document.implementation Returns the DOM implementation  document.inputEncoding Returns the document's encoding (character set)  document.lastModified Returns the date and time the document was updated  document.links Returns all <area> and <a> elements that have a href attribute  document.readyState Returns the (loading) status of the document  document.referrer Returns the URI of the referrer (the linking document)  document.scripts Returns all <script> elements  document.strictErrorChecking Returns if error checking is enforced  document.title Returns the <title> element  document.URL Returns the complete URL of the document |

**DOM Elements**

|  |  |  |
| --- | --- | --- |
| Finding Elements | | |
| by Id | getElementById() (the easiest way) | const element = document.**getElementById**("intro"); |
| by Tag Name |  | // finds all <p> elements:  const element = document.**getElementsByTagName**("p");  // finds all <p> elements inside "main":  const x = document.getElementById("main"); const y = x.getElementsByTagName("p");  // finds the first <p> elements  const x = document.getElementsByTagName("p")[0].innerHTML |
| by Class Name | getElementsByClassName() | const x = document.getElementsByClassName("intro"); |
| by CSS Selectors | querySelectorAll()  (id, class names, types, attributes, values of attributes, etc) | const x = document.querySelectorAll("p.intro"); |
| by HTML Object Collections | document.forms  document.anchors  document.body  document.documentElement  document.embeds  document.forms  document.head  document.images  document.links  document.scripts  document.title | // finds the form element with id="frm1", in the forms collection  const x = **document**.**forms**["frm1"]; let text = ""; for (let i = 0; i < x.length; i++) {   text += x.elements[i].value + "<br>"; } document.getElementById("demo").innerHTML = text; |

**DOM HTML -** change the content of elements.

|  |  |  |
| --- | --- | --- |
| Change Content | **innerHTML** property - the easiest way  document.getElementById(*id*).innerHTML =*new HTML* | <p id="p1">Hello World!</p>  <script>  document.getElementById("p1").**innerHTML** = "New text!";  // the same as:  **const** **element** = document.getElementById("p1");  **element**.**innerHTML** = "New text!";  </script> |
| Change Value  of an Attribute | document.getElementById(*id*).*attribute = new value* | <img id="myImage" **src**="smiley.gif">  <script> document.getElementById("myImage").**src** = "landscape.jpg"; </script> |
| Dynamic Content | JS can create dynamic HTML content | document.getElementById("demo").innerHTML = "Date : " + **Date**(); |
|  | **document**.**write**() - write directly to the HTML output stream  (Never use document.write() after the document is loaded. It will overwrite the document) | document.write(Date()); |

**DOM Forms**

|  |  |  |
| --- | --- | --- |
| Form Validation | form validation can be done by JS. | function validateForm() {   let x = document.forms["myForm"]["fname"].value;   if (x == "") {     alert("Name must be filled out");     return false;   } } |
|  | JS is often used to validate numeric input | function myFunction() {  // Get the value of the input field with id="numb"  let x = document.getElementById("numb").value;  // If x is Not a Number or less than one or greater than 10  let text;  if (isNaN(x) || x < 1 || x > 10) {  text = "Input not valid";  } else {  text = "Input OK";  }  document.getElementById("demo").innerHTML = text;  } |
| Automatic | form validation can be performed automatically by the browser | <form action="/action\_page.php" method="post">  <input type="text" name="fname" **required**>  <input type="submit" value="Submit">  </form> |
| Data Validation | ensure that user input is clean, correct, and useful.  Typical validation tasks are:  - has the user filled in all **required** fields?  - has the user entered a **valid** date?  - has the user entered text in a numeric field? | **Server side validation** is performed by a web server, after input has been sent to the server.  **Client side validation** is performed by a web browser, before input is sent to a web server. |
| Constraint Validation | Constraint validation is based on:  - Constraint validation HTML Input Attributes  - Constraint validation CSS Pseudo Selectors  - Constraint validation DOM Properties and Metho |  |

**DOM CSS**

|  |  |  |
| --- | --- | --- |
| Changing Style | document.getElementById(id).style.**property** = new style | <p id="p2">Hello World!</p>  <script> document.getElementById("p2").style.**color** = "blue"; </script> |
| Using Events | Changing style ofen goes with Event | <h1 id="id1">My Heading 1</h1>  <button type="button" onclick="document.getElementById('id1').style.**color** = 'red'"> Click Me!</button> |

**DOM Animation**

|  |  |  |
| --- | --- | --- |
| Basic Web Page |  | <h1>My First JavaScript Animation</h1> <div id="animation">My animation will go here</div> |
| Animation Container | All animations should be relative to a container element. | <div id ="container">   <div id ="animate">My animation will go here</div> </div> |
| Style the Elements | container element should have style = "**position**: **relative**". | #container {   width: 400px;   height: 400px;   position: relative;   background: yellow; } |
|  | animation element should have style = "**position**: **absolute**". | #animate {   width: 50px;   height: 50px;   position: absolute;   background: red; } |
| Full Animation Code |  | function myMove() {   let id = null;   const elem = document.getElementById("animate");   let pos = 0;   clearInterval(id);   id = setInterval(frame, 5);   function frame() {     if (pos == 350) {       clearInterval(id);     } else {       pos++;       elem.style.top = pos + 'px';       elem.style.left = pos + 'px';     }   } } |

**DOM Events**

|  |  |  |
| --- | --- | --- |
| Reacting to Events | typeofevent=Javascript | <h1 **onclick**="this.innerHTML = 'Ooops!'">Click on this text!</h1>  // the same as:  <h1 **onclick**="changeText(this)">Click on this text!</h1>  <script>  function **changeText**(id) {  id.innerHTML = "Ooops!";  }  </script> |
| Event Attributes | We can use event attributes to assign events to HTML elements | // Assign an onclick event to a button element  <button **onclick**="displayDate()">Try it</button> |
| HTML DOM | We can use HTML DOM to assign events to HTML elements | // Assign an onclick event to a button element  <button id="myBtn">Try it</button>  <script> document.getElementById("myBtn").onclick = displayDate; </script> |
| onload, onunload | triggered when the user enters or leaves the page. | // deal with cookies  <body **onload**="checkCookies()">  <p id="demo"></p>  <script>  function checkCookies() {  var text = "";  if (navigator.cookieEnabled == true) {  text = "Cookies are enabled.";  } else {  text = "Cookies are not enabled.";  }  document.getElementById("demo").innerHTML = text;  } |
| onchange | used in combination with validation of input fields | // convert to upperCase when users input  <input type="text" id="fname" **onchange**="upperCase()">  <script>  function upperCase() {  const x = document.getElementById("fname");  x.value = x.value.toUpperCase();  }  </script> |
| onmouseover, onmouseout | when the user mouses over, or out of, an HTML element | <div **onmouseover**="mOver(this)" **onmouseout**="mOut(this)"  style="background-color:#D94A38;width:120px;height:20px;padding:40px;">  Mouse Over Me</div>  <script>  function mOver(obj) {  obj.innerHTML = "Thank You"  }  function mOut(obj) {  obj.innerHTML = "Mouse Over Me"  }  </script> |
| onmousedown, onmouseup, onclick | when a mouse-button is clicked, released, mouse-click is completed | <div **onmousedown**="mDown(this)" **onmouseup**="mUp(this)"  style="background-color:#D94A38;width:90px;height:20px;padding:40px;">  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> |

**DOM Event Listener**

|  |  |  |
| --- | --- | --- |
| addEventListener() | attach an event handler to the specified element, without overwriting existing event handlers.  *element*.**addEventListener**(*event, function, useCapture*);  (1st parameter: type of the event like "click" or "mousedown"  2nd parameter: function to call when the event occurs.  3rd parameter: specifying whether to use event bubbling or event capturing - optional) | *element*.addEventListener("click", function(){ alert("Hello World!"); });  // the same as:  *element*.addEventListener("click", myFunction);  function myFunction() {   alert ("Hello World!"); } |
| Many Event Handlers to the Same Element | It's possible to add many events to the same element, without overwriting existing events: | *element*.addEventListener("click", myFunction); *element*.addEventListener("click", mySecondFunction); |
| Event Handler to the window Object | It's possible to add event listeners on any HTML DOM object | **window**.addEventListener("resize", function(){   document.getElementById("demo").innerHTML = *sometext*; }); |
| Passing Parameters | use an "anonymous function" that calls the specified function with the parameters: | *element*.addEventListener("click", function(){ myFunction(p1, p2); }); |
| Event Propagation | a way of defining the element order when an event occurs:  - Bubbling: inner to outer (**false** - default)  -Capturing: outer to inner (**true**) | document.getElementById("myP").addEventListener("click", myFunction); // false => bubbling  document.getElementById("myP").addEventListener("click", myFunction, **true**); |
| removeEventListener() | removes event handlers that have been attached with the addEventListener() method | *element*.**removeEventListener**("mousemove", myFunction); |

**DOM Navigation**

|  |  |  |
| --- | --- | --- |
| DOM Nodes | everything in an HTML document is a node  all nodes in the node tree can be accessed by JS. |  |
| Node Relationships | root, parent, child/children, siblings |  |
| Navigating Between Nodes | parentNode, childNodes[nodenumber]  firstChild, lastChild, nextSibling, previousSibling |  |
| Child Nodes and Node Values | the value of the **text node** can be accessed by the node's **innerHTML** property: | <title id="demo">**DOM Tutorial**</title>  myTitle = document.getElementById("demo").**innerHTML**; |
|  | accessing the **innerHTML** property is the same as accessing the **nodeValue** of the first child: | myTitle = document.getElementById("demo").**firstChild**.**nodeValue**; |
|  | Accessing the first child can also be done like this: | myTitle = document.getElementById("demo").**childNodes**[**0**].**nodeValue**; |
| DOM Root Nodes | two special properties that allow access to the full document:  **document**.**body** - The body of the document  **document**.**documentElement** - The full document | document.getElementById("demo").innerHTML = **document**.**body.**innerHTML; |
| nodeName Property | specify the name of a node.  (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.) | <h1 id="id01">My First Page</h1>  <p id="id02"></p>  <script> document.getElementById("id02").innerHTML = document.getElementById("id01").**nodeName**; </script> // result: H1 |
| nodeValue Property | specify the value of a node.  - nodeValue for element nodes is null  - nodeValue for text nodes is the text itself  - nodeValue for attribute nodes is the attribute value |  |
| nodeType Property | return the type of a node  - element node: 1  - attribute node: 2  - text node: 3  - comment node: 8  - document node: 9  - document type node: 10 | document.getElementById("id02").innerHTML = document.getElementById("id01").nodeType; // return 1 |

**DOM Nodes**

|  |  |  |
| --- | --- | --- |
| Creating New Elements | | |
| appendChild() | Basic Code | <div id="div1">   <p id="p1">This is a paragraph.</p>   <p id="p2">This is another paragraph.</p> </div>  <script>  // new <p> element const para = document.createElement("p");  // create a text node const node = document.createTextNode("This is new.");  // append the text node to the <p> element  para.appendChild(node); |
|  | appended the new element as the last child of the parent | // finds an existing element  const element = document.getElementById("div1");  // append the new element to an existing element.  element.**appendChild**(para); </script> |
| insertBefore() | appended the new element as the first child of the parent. | // finds an existing element  const element = document.getElementById("div1");  // finds an existing first child  const child = document.getElementById("p1");  // append the new element to an existing element.  element.**insertBefore**(para, child); |
| Removing Existing Elements | | |
| remove() method | basic code | <div>   <p id="p1">This is a paragraph.</p>   <p id="p2">This is another paragraph.</p> </div> |
|  | Removing the element | <script>  // Find the element you want to remove:  const elmnt = document.getElementById("p1");  // execute the remove() method on that element:  elmnt.remove(); </script> |
| removeChild() | For browsers that does not support the **remove**() method, you have to find the parent node to remove an element: | <script>  // Find the parent element  const parent = document.getElementById("div1");  // Find the child element  const child = document.getElementById("p1");  // Remove the child element  parent.removeChild(child); </script> |
|  |  | // the same as this common workaround:  const child = document.getElementById("p1"); child.parentNode.removeChild(child); |
| Replacing Elements | | |
| replaceChild() | replace an element to the HTML DOM | <script>  // Find the parent element  const parent = document.getElementById("div1");  // Find the child element  const child = document.getElementById("p1");  // Create a new <p>  const para = document.createElement("p");  const node = document.createTextNode("This is new.");  para.appendChild(node);  // remove  parent.replaceChild(para,child);  </script> |

**DOM Collections**

|  |  |  |
| --- | --- | --- |
| HTMLCollection Object | an array-like list (collection) of HTML elements.  **getElementsByTagName**() method returns an HTMLCollection object. | // collect all <p> elements  const myCollection = document.getElementsByTagName("p");  // The elements can be accessed by an index number.  myCollection[0] |
| length property | define the number of elements in an HTMLCollection | myCollection.length |
|  | useful when you want to loop through the elements | // change the text color of all <p> elements  const myCollection = document.getElementsByTagName("p");  for (let i = 0; i < myCollection.length; i++) {  myCollection[i].style.color = "red";  } |

**DOM NodeLists**

|  |  |  |
| --- | --- | --- |
| NodeList Object | a list (collection) of nodes extracted from a document.  (almost the same as an HTMLCollection object)  - some browsers: return a NodeList for methods like **getElementsByClassName**().)  - most browsers return a NodeList object for the method **querySelectorAll**().  - all browsers: return a NodeList object for the property **childNodes**. | // collect all <p> notes  const myNodeList = document.querySelectorAll("p");  // The elements can be accessed by an index number.  myNodeList[0] |
| length property | define the number of notes in a NoteList | myNodelist.length |
|  | useful when you want to loop through the nodes in a node list | // change the color of all <p> elements in a node list:  const myNodelist = document.querySelectorAll("p"); for (let i = 0; i < myNodelist.length; i++) {   myNodelist[i].style.color = "red"; } |
| HTMLCollection  or NodeList | **similarity**:  Both is an array-like list (collection) of objects.  Both have a length property | **difference**:  - HTMLCollection items can be accessed by their name, id, or index number.  - NodeList items can only be accessed by their index number.  - Only the NodeList object can contain attribute nodes and text nodes. |

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**JS Browser BOM**

**JS Window**

|  |  |  |
| --- | --- | --- |
| BOM | BOM (Browser Object Model) allows JS to "talk to" the browser. |  |
| window | this object represents the browser's window.  Global **variables** are **properties** of the window object.  Global **functions** are **methods** of the window object. | // Even the document object is a property of the window object:  window.document.getElementById("header");  // the same as:  document.getElementById("header"); |
| Window Size | **window**.**innerHeight** property:  return the inner height of the browser window (in pixels)  **window**.**innerWidth** property:  return the inner width of the browser window (in pixels)  (The browser window (viewport) NOT including toolbars and scrollbars) | <script>  document.getElementById("demo").innerHTML =  "Browser inner window width: " + window.innerWidth + "px<br>" +  "Browser inner window height: " + window.innerHeight + "px";  </script> |
| Other Methods | **window**.**open**() - open a new window  **window**.**close**() - close the current window  **window**.**moveTo**() - move the current window  **window**.**resizeTo**() - resize the current window |  |

**JS Screen**

|  |  |  |
| --- | --- | --- |
| window.screen | this object contains information about the user's screen.  (can be written without the window prefix: window) | |
| Properties | | |
| screen.width | returns the width of the visitor's screen in pixels. | document.getElementById("demo").innerHTML = "Screen Width: " + screen.width; // result: Screen Width: 1920 |
| screen.height | returns the height of the visitor's screen in pixels. | document.getElementById("demo").innerHTML = "Screen Height: " + screen.height; // result: Screen Height: 1080 |
| screen.availWidth | returns the width of the visitor's screen, in pixels, minus interface features like the Windows Taskbar. | document.getElementById("demo").innerHTML = "Available Screen Width: " + screen.availWidth;  // result: Available Screen Width: 1920 |
| screen.availHeight | returns the height of the visitor's screen, in pixels, minus interface features like the Windows Taskbar. | document.getElementById("demo").innerHTML = "Available Screen Height: " + screen.availHeight;  // result: Available Screen Height: 1032 |
| screen.colorDepth | returns the number of **bits** used to display one color. | document.getElementById("demo").innerHTML = "Screen Color Depth: " + screen.colorDepth;  // result: Screen Color Depth: 24 |
| screen.pixelDepth | returns the pixel depth of the screen.  (For modern computers, Color Depth and Pixel Depth are equal.) | document.getElementById("demo").innerHTML = "Screen Pixel Depth: " + screen.pixelDepth;  // result: Screen Pixel Depth: 24 |

**JS Location**

|  |  |  |
| --- | --- | --- |
| window.location | this object can be used to get the current page address (URL) and to redirect the browser to a new page.  (can be written without the window prefix: window) | |
| Properties | | |
| location.href | returns the width of the visitor's screen in pixels. | document.getElementById("demo").innerHTML = "Page location is " + window.location.href;  // result: Page location is https://www.w3schools.com/js/js\_window\_location.asp |
| location.hostname | returns the domain name of the web host | document.getElementById("demo").innerHTML = "Page hostname is " + location.hostname;  // result: |
| location.pathname | returns the path and filename of the current page | document.getElementById("demo").innerHTML = "Page path is " + location.pathname;  // result: |
| location.protocol | returns the web protocol used (http: or https:) | document.getElementById("demo").innerHTML = "Page protocol is " + location.protocol;  // result: |
| location.port | returns the number of the internet host port (of the current page) | document.getElementById("demo").innerHTML = "Port number is " + location.port;  // result: |
| location.assign() | loads a new document | </script>  function newDoc() {   window.location.assign("https://www.w3schools.com") } </script>  // result: go to the the new page |

**JS History**

|  |  |  |
| --- | --- | --- |
| window.history | this object contains the browsers history. (can be written without the window prefix: window) | |
| Methods | | |
| history.back() | loads the previous URL in the history list. | <script> function goBack() {   window.history.back() } </script> |
| history.forward() | loads the next URL in the history list. | function goForward() {   window.history.forward() } </script> |

**JS Navigator**

|  |  |  |
| --- | --- | --- |
| window.navigator | This object contains information about the visitor's browser. | |
| Properties | | |
| cookieEnabled | Return true if cookies are enabled, otherwise false: | document.getElementById("demo").innerHTML = "cookiesEnabled is " + navigator.cookieEnabled; |
| appName | Return the application name of the browser: | document.getElementById("demo").innerHTML = "navigator.appName is " + navigator.appName; |
| appCodeName | Return the application code name of the browser: | document.getElementById("demo").innerHTML = "navigator.appCodeName is " + navigator.appCodeName; |
| product | Return the product name of the browser engine: | document.getElementById("demo").innerHTML = "navigator.product is " + navigator.product; |
| appVersion | Return version information about the browser: | document.getElementById("demo").innerHTML = navigator.appVersion; |
| userAgent | Return the user-agent header sent by the browser to the server: | document.getElementById("demo").innerHTML = navigator.userAgent; |
| platform | Return the browser platform (operating system): | document.getElementById("demo").innerHTML = navigator.platform; |
| language | Return the browser's language: | document.getElementById("demo").innerHTML = navigator.language; |
| onLine | returns true if the browser is online: | document.getElementById("demo").innerHTML = navigator.onLine; |
| Methods | | |
| javaEnabled() | returns true if Java is enabled: | document.getElementById("demo").innerHTML = navigator.javaEnabled(); |

**JS Popup Alert**

|  |  |  |
| --- | --- | --- |
| Methods | | |
| alert() | often used if to make sure information comes through to the user.  **alert**("*sometext*");  (can be written without the window prefix: window)  (the user click OK to proceed) | alert("I am an alert box!"); |
| confirm() | often used if you want the user to verify or accept something.  **confirm**("*sometext*");  (can be written without the window prefix: window)  (the user click OK to Cancel proceed) | confirm("Press a button!"); |
| prompt() | often used if you want the user to input a value before entering a page.  **prompt**("*sometext*","*defaultText*");  (can be written without the window prefix: window)  (the user click OK to Cancel proceed) | prompt("Please enter your name", "Harry Potter"); |
| Line Breaks | To display line breaks inside a popup box, use a **back**-**slash** followed by the character **n**. | alert("Hello**\n**How are you?");  // result: Hello  How are you |

**JS Timing**

|  |  |  |
| --- | --- | --- |
| Methods | | |
| setTimeout() | execute a function, after a specified number of milliseconds.  **setTimeout**(*function*,*milliseconds*);  - 1st parameter: afunction to be executed.  - 2nd parameter: milliseconds before execution.  (can be written without the window prefix: window) | setTimeout(myFunction, 3000) |
| setInterval() | execute a given function at every given time-interval.  **setInterval**(*function*,*milliseconds*);  (can be written without the window prefix: window) | setInterval(myTimer, 1000); // like a digital watch |
| clearTimeout() | stops the execution of the function specified in **setTimeout**().  uses the variable returned from **setTimeout**()  **clearTimeout**(*timeoutVariable*)  (can be written without the window prefix: window) | myVar = setTimeout(*function*,*milliseconds*);  clearTimeout(myVar); |
| clearInterval() | stops the execution of the function specified in **setInternal**().  uses the variable returned from setInterval()  **clearInterval**(*timerVariable*)  (can be written without the window prefix: window) | let myVar = setInterval(*function*,*milliseconds*); clearInterval(myVar); |

**JS Cookies**

|  |  |  |
| --- | --- | --- |
| Cookies | Cookies are data, stored in small text files, on your computer.  solve the problem "how to remember information about the user": | - When a user visits a web, his name can be stored in a cookie.  - Next time, the cookie "remembers" his/her name. |
|  | saved in **name-value** pairs like: username = John Doe |  |
| Creating | **document**.**cookie** property | // By default, deleted when the browser is closed, belongs to the current page.  document.cookie = "username=John Doe";  // with an expiry date (in UTC time)  document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC";  // With a path parameter to tell the browser what path the cookie belongs to.  document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/"; |
| Reading | **document**.**cookie** property | let x = document.cookie;  // return in one string much like: cookie1=value; cookie2=value; |
| Changing | the same way as you create it, the old cookie is overwritten.  **document**.**cookie** property | document.cookie = "username=John Smith; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/"; |
| Deleting | Just set the expires parameter to a past date:  (Some browsers will not delete if you don't specify the path.) | document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;"; |
| Example | | |
| Set a Cookie | a function that stores the name of the visitor in a cookie variable: | function setCookie(cname, cvalue, exdays) {   const d = new Date();   d.setTime(d.getTime() + (exdays\*24\*60\*60\*1000));   let expires = "expires="+ d.toUTCString();   document.cookie = cname + "=" + cvalue + ";" + expires + ";path=/"; } |
| Get a Cookie | a function that returns the value of a specified cookie: | function getCookie(cname) {   let name = cname + "=";   let decodedCookie = decodeURIComponent(document.cookie);   let ca = decodedCookie.split(';');   for(let i = 0; i <ca.length; i++) {     let c = ca[i];     while (c.charAt(0) == ' ') {       c = c.substring(1);     }     if (c.indexOf(name) == 0) {       return c.substring(name.length, c.length);     }   }   return ""; } |
| Check a Cookie | a function that checks if a cookie is set. | function checkCookie() {   let username = getCookie("username");   if (username != "") {    alert("Welcome again " + username);   } else {     username = prompt("Please enter your name:", "");     if (username != "" && username != null) {       setCookie("username", username, 365);     }   } } |

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**JS Web APIs**

**Web API Intro**

|  |  |  |
| --- | --- | --- |
| API | **A**pplication **P**rogramming **I**nterface for the Web  - can extend the functionality of a web browser.  - can extend the functionality of a web server. |  |
| Browser APIs | built-in Web APIs to support complex operations, and to help accessing data. | // Get the latitude and longitude of the user's position:  const myElement = document.getElementById("demo");  function getLocation() {   if (navigator.geolocation) {     navigator.geolocation.getCurrentPosition(showPosition);   } else {     myElement.innerHTML = "Geolocation is not supported by this browser.";   } }  function showPosition(position) {   myElement.innerHTML = "Latitude: " + position.coords.latitude +   "<br>Longitude: " + position.coords.longitude; } |
| Third Party APIs | not built into your browser, must download from the web | YouTube API - to display videos on a web site.  Twitter API - display Tweets on a web site.  Facebook API - display Facebook info on a web site. |

**Web Validation API**

|  |  |  |
| --- | --- | --- |
| Constraint Validation DOM Methods | | |
| checkValidity() | Returns true if an input element contains valid data. | // basic code  <input id="id1" type="number" min="100" max="300" required>  <button onclick="myFunction()">OK</button>  <p id="demo"></p>  // check input  <script>  function myFunction() {  const **inpObj** = document.getElementById("id1");  if (!inpObj.**checkValidity**()) {  document.getElementById("demo").innerHTML = inpObj.validationMessage;  } else {  document.getElementById("demo").innerHTML = "Input OK";  }  } |
| setCustomValidity() | Sets the validationMessage property of an input element. |  |
| Constraint Validation DOM Properties | | |
| validity | boolean properties related to the validity of an input element. |  |
| validationMessage | the message a browser will display when the validity is false. |  |
| willValidate | Indicates if an input element will be validated. |  |
| Validity Properties: contains a number of properties related to the validity of data: | | |
| customError | Set to true, if a custom validity message is set. | // greater than 100, display a message:  <input id="id1" type="number" max="100"> <button onclick="myFunction()">OK</button>  <p id="demo"></p>  <script> function myFunction() {   let text = "Value OK";   if (document.getElementById("id1").**validity**.**rangeOverflow**) {     text = "Value too large";   } } </script>  // input 1000 => result: Value too large. |
| patternMismatch | Set to true, if an element's value does not match its pattern attribute. |
| rangeOverflow | Set to true, if an element's value is greater than its max attribute. |
| rangeUnderflow | Set to true, if an element's value is less than its min attribute. |
| stepMismatch | Set to true, if an element's value is invalid per its step attribute. |
| tooLong | Set to true, if an element's value exceeds its maxLength attribute. |
| typeMismatch | Set to true, if an element's value is invalid per its type attribute. |
| valueMissing | Set to true, if an element (with a required attribute) has no value. |
| valid | Set to true, if an element's value is valid. |

**Web History API**

|  |  |  |
| --- | --- | --- |
| General | provide easy methods to access the **windows**.**history** object.  **window**.**history** object contains the URLs visited by the user. |  |
| forward() | Loads the next URL in the history list | <button onclick="myFunction()">Go Back</button>  <script> function myFunction() {   window.history.**forward**(); } </script> |
| back() Method | loads the previous URL in the windows.history list.  (the same as clicking the "back arrow") | <button onclick="myFunction()">Go Back</button>  <script> function myFunction() {   window.history.**back**(); } </script> |
| go() Method | loads a specific URL from the history list: | <button onclick="myFunction()">Go Back 2 Pages</button>  <script> function myFunction() {   window.history.**go**(-2); // back two pages } </script> |
| length property | Returns the number of URLs in the history list | var x = history.length;  document.getElementById("demo").innerHTML = x; |

**Web Storage API**

|  |  |  |
| --- | --- | --- |
| General | a simple syntax for storing and retrieving data in the browser. | localStorage.setItem("name", "John Doe"); localStorage.getItem("name"); |
| localStorage Object | provides access to a local storage for a particular Web Site  (the data is stored with no expiration date, and will not be deleted when the browser is closed.) |  |
| sessionStorage Object | identical to the localStorage object, but stxores data for one session. (the data is deleted when the browser is closed) |  |
| Methods | | |
| setItem() | stores a data item in a storage.  localStorage.**setItem**(*name*,*value*);  sessionStorage.**setItem**(*name*,*value*); | localStorage.setItem("name", "John Doe");  sessionStorage.setItem("name", "John Doe"); |
| getItem() | retrieves a data item from the storage.  localStorage.**getItem**(*name*);  sessionStorage.**getItem**(*name*); | localStorage.getItem("name");  sessionStorage.getItem("name"); |
| key() | returns name of the key with the specified index.  localStorage.**key**(*index*)  sessionStorage.**key**(*index*) | var x = localStorage.key(0); |
| removeItem() | removes the specified Storage Object item.  localStorage.**removeItem**(*keyname*)  sessionStorage.**removeItem**(*keyname*) | localStorage.removeItem("mytime"); |
| clear() Method | removes all the Storage Object item for this domain.  localStorage.**clear**()  sessionStorage.**clear**() | sessionStorage.clear(); |
| Properties | | |
| length Property | returns the number of items stored in the Storage Object  localStorage.**length**;  sessionStorage.**length**; | var x = localStorage.length; |

**Web Woker API**

|  |  |  |
| --- | --- | --- |
| Web Worker | a JavaScript that runs in the background, independently of other scripts, without affecting the performance of the page. |  |
| Checking | check whether the user's browser supports it: | if (typeof(Worker) !== "undefined") {   // Yes! Web worker support!   // *Some code.....* } else {   // Sorry! No Web Worker support.. } |
| Creating a Web Worker File |  | // create a script that counts.  let i = 0;  function timedCount() {   i ++;   **postMessage**(i); // the most important part   setTimeout("timedCount()",500); }  timedCount(); |
| Create a Web Worker Object | checks if the worker already exists,  if not - it creates a new web worker object  and runs the code in "demo\_workers.js": | if (typeof(w) == "undefined") {   w = new Worker("demo\_workers.js"); } |
|  | add an "onmessage" event listener to the web worker. | w.onmessage = function(event){   document.getElementById("result").innerHTML = event.data; }; |
| Terminate a Web Worker | use the **terminate**() method: | w.terminate(); |
| Reuse the Web Worker | If you set the worker variable to undefined, after it has been terminated, you can reuse the code: | w = undefined; |
| Full Code |  | <!DOCTYPE html> <html> <body>  <p>Count numbers: <output id="result"></output></p> <button onclick="startWorker()">Start Worker</button> <button onclick="stopWorker()">Stop Worker</button>  <script> let w;  function startWorker() {   if (typeof(w) == "undefined") {     w = new Worker("demo\_workers.js");   }   w.onmessage = function(event) {     document.getElementById("result").innerHTML = event.data;   }; }  function stopWorker() {   w.terminate();   w = undefined; } </script>  </body> </html> |

**Web Fetch API**

|  |  |  |
| --- | --- | --- |
| General | allows web browser to make HTTP requests to web servers. | fetch(file) .then(x => x.text()) .then(y => myDisplay(y)); |
|  | Since Fetch is based on async and await, the example above might be easier to understand like this: | async function getText(file) {   let x = await fetch(file);   let y = await x.text();   myDisplay(y); } |
|  | Or even bettter: Use understandable names instead of x and y: | async function getText(file) {   let myObject = await fetch(file);   let myText = await myObject.text();   myDisplay(myText); } |

**Web Geolocation API**

|  |  |  |
| --- | --- | --- |
| Geolocation API | used to get the geographical position of a user. |  |
| getCurrentPosition() | used to return the user's position. | // returns the latitude and longitude of the user's position:  const x = document.getElementById("demo"); function getLocation() { // Check if Geolocation is supported  // supported, run the getCurrentPosition() method   if (navigator.geolocation) {     navigator.geolocation.getCurrentPosition(showPosition);  // not supported, display a message to the user    } else {     x.innerHTML = "Geolocation is not supported by this browser.";   } } // outputs the Latitude and Longitude function showPosition(position) {   x.innerHTML = "Latitude: " + position.coords.latitude +   "<br>Longitude: " + position.coords.longitude; } |
| Errors and Rejections | the second parameter of the **getCurrentPosition**() method is used to handle errors. It specifies a function to run if it fails to get the user's location: | function showError(error) {   switch(error.code) {     case error.PERMISSION\_DENIED:       x.innerHTML = "User denied the request for Geolocation."       break;     case error.POSITION\_UNAVAILABLE:       x.innerHTML = "Location information is unavailable."       break;     case error.TIMEOUT:       x.innerHTML = "The request to get user location timed out."       break;     case error.UNKNOWN\_ERROR:       x.innerHTML = "An unknown error occurred."       break;   } } |
| Displaying the Result in a Map |  | function showPosition(position) {   let latlon = position.coords.latitude + "," + position.coords.longitude;    let img\_url = "https://maps.googleapis.com/maps/api/staticmap?center=   "+latlon+"&zoom=14&size=400x300&sensor=false&key=YOUR\_KEY";    document.getElementById("mapholder").innerHTML = "<img src='"+img\_url+"'>"; } |
| watchPosition() | Returns the current position of the user and continues to return updated position as the user moves (like the GPS in a car). | <script> const x = document.getElementById("demo"); function getLocation() {   if (navigator.geolocation) {     navigator.geolocation.watchPosition(showPosition);   } else {     x.innerHTML = "Geolocation is not supported by this browser.";   } } function showPosition(position) {   x.innerHTML = "Latitude: " + position.coords.latitude +   "<br>Longitude: " + position.coords.longitude; } </script> |
| clearWatch() | Stops the watchPosition() method. |  |

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**JS AJAX**

**AJAX Intro**

|  |  |  |
| --- | --- | --- |
| General | **A**synchronous **J**avaScript **A**nd **X**ML: a combination of  - a browser built-in **XMLHttpRequest** object (to request data from a web server)  - JS and HTML DOM (to display or use the data) | AJAX is a misleading name. AJAX applications might use XML to transport data, but it is equally common to transport data as plain text or JSON text. |
| How AJAX Works | 1. An event occurs in a web page (the page is loaded, a button is clicked)  2. An XMLHttpRequest object is created by JS  3. The XMLHttpRequest object sends a request to a web server  4. The server processes the request  5. The server sends a response back to the web page  6. The response is read by JS  7. Proper action (like page update) is performed by JS |  |
| Fetch API | Modern Browsers can use Fetch API instead of the XMLHttpRequest Object.  - The Fetch API interface allows web browser to make HTTP requests to web servers.  - If you use the XMLHttpRequest Object, Fetch can do the same in a simpler way. |  |

**AJAX XMLHttpRequest Object**

|  |  |  |
| --- | --- | --- |
| General | used to exchange data with a web server behind the scenes. | => possible to update parts of a web page, without reloading the whole page. |
| Creating | *variable*= new XMLHttpRequest();W  (All modern browsers have a built-in XMLHttpRequest object) |  |
| Define a Callback Function | A callback function is a function passed as a parameter to another function.   the callback function should contain the code to execute when the response is ready. | // Define a callback function xhttp.onload = function() {   // Here you can use the Data } |
| Send a Request to the Server | use the open() and send() methods of the XMLHttpRequest object | // Send a request xhttp.open("GET", "ajax\_info.txt"); xhttp.send(); |
| Access Across Domains | - For security reasons, modern browsers do not allow access across domains. | => both the web page and the XML file it tries to load, must be located on the same server. |
| XMLHttpRequest Object Methods | | |
| new XMLHttpRequest() | create a new XMLHttpRequest object |  |
| abort() | cancel the current request |  |
| getAllResponseHeaders() | return header information |  |
| getResponseHeader() | Returns specific header information |  |
| open(method, url, async, user, psw) | specify the request  url: the file location  async: true (asynchronous) or false (synchronous)  user: optional user name  psw: optional password |  |
| send() | Sends the request to the server  Used for GET requests |  |
| send(string) | Sends the request to the server.  Used for POST requests |  |
| setRequestHeader() | add a label/value pair to the header to be sent |  |
| XMLHttpRequest Object Properties | | |
| onload | Defines a function to be called when the request is recived (loaded) |  |
| onreadystatechange | Defines a function to be called when the readyState property changes |  |
| readyState | Holds the status of the XMLHttpRequest.  0: request not initialized  1: server connection established  2: request received  3: processing request  4: request finished and response is ready |  |
| responseText | Returns the response data as a string |  |
| responseXML | Returns the response data as XML data |  |
| status | Returns the status-number of a request  200: "OK"  403: "Forbidden"  404: "Not Found"  https://www.w3schools.com/tags/ref\_httpmessages.asp |  |
| statusText | Returns the status-text (e.g. "OK" or "Not Found") |  |

**AJAX Request**

|  |  |  |
| --- | --- | --- |
| General | XMLHttpRequest object is used to request data from a server. |  |
| Send a Request | **open**(*method, url, async*) - specify the type of request  *method*: the type of request: GET or POST *url*: the server (file) location *async*: true (asynchronous) or false (synchronous) | xhttp.open("GET", "ajax\_info.txt", true); |
|  | **send**() - send the request to the server (used for GET) | xhttp.send(); |
|  | **send**(*string*) - send the request to the server (used for POST) | xhttp.send("fname=Henry&lname=Ford"); |

**AJAX Response**

|  |  |  |
| --- | --- | --- |
| Sever Response Properties | | |
| responseText | returns the server response as a JS string, and you can use it accordingly: | document.getElementById("demo").innerHTML = xhttp.**responseText**; |
| responseXML | returns the server response as an XML DOM object.  (Using this property you can parse the response as an XML DOM object:) | // Request the file [cd\_catalog.xml](https://www.w3schools.com/js/cd_catalog.xml) and parse the response:  const xmlDoc = xhttp.responseXML; const x = xmlDoc.getElementsByTagName("ARTIST");  let txt = ""; for (let i = 0; i < x.length; i++) {   txt += x[i].childNodes[0].nodeValue + "<br>"; } document.getElementById("demo").innerHTML = txt;  xhttp.open("GET", "cd\_catalog.xml"); xhttp.send(); |
| Server Response Methods | | |
| getResponseHeader() | return specific header information from the server resource | const xhttp = new XMLHttpRequest(); xhttp.onload = function() {     document.getElementById("demo").innerHTML =     this.getResponseHeader("Last-Modified"); } xhttp.open("GET", "ajax\_info.txt"); xhttp.send(); |
| getAllResponseHeaders() | return all the header information from the server resource | const xhttp = new XMLHttpRequest(); xhttp.onload = function() {     document.getElementById("demo").innerHTML =     this.getAllResponseHeaders(); } xhttp.open("GET", "ajax\_info.txt"); xhttp.send(); |

**AJAX XML File**

|  |  |
| --- | --- |
| General |  |

**AJAX Example**

|  |  |  |
| --- | --- | --- |
| XML file | AJAX can be used for interactive communication with an XML file. |  |
| PHP | AJAX is used to create more interactive applications. |  |
| ASP | AJAX is used to create more interactive applications. |  |
| ASP | AJAX can be used for interactive communication with a database. |  |

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**JS and JQuery**

|  |  |  |
| --- | --- | --- |
| later |  |  |

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**JS References**

|  |  |  |
| --- | --- | --- |
| later |  |  |

**JS Bitwise Operations**

|  |  |  |
| --- | --- | --- |
| later |  |  |