**Variables in JavaScript:**

JavaScript is a dynamically typed language which means you can change the variable during runtime. variable is a container in JS that stores the value, the value of the variables in JS can be changed during runtime.

Var a = 7 – here **a** is called identifier, and 7 is called literal and this process is called Declaring variable.

**Rules for choosing a variable**: 1)Letters, Digits, Underscore and $ sign allowed

2) Can not start with a number and follow case sensitivity

3) The reserved keywords in JS can not be used as a variable

**Let, var and Const :**

Var is globally scoped where let and const are block scoped. Block meaning inside a curly braces.

Var can be updated and re-declared within its scope but let can be updated but could not be redeclared. Const can neither be updated and re-declared. We will use const only when we know that the value of the variable will not change for the entire program. Var variables are initialised with undefined where let and const are not. Const must be initialised during declaration unline let and var.

Re-declared: Let a =5

Let a = 7

This is called redeclaring.

Update: Let a = 5

a = 7

This is called updating the value of the variable.

**Primitive data types and Object in JS:**

There are 7 primitive datatypes in JS. Non-primitive data types are called object.

7 primitive data types: Null, Number, Symbol, String, Boolean, Bigint, Undefined

To find datatype we can do, console.log(typeof a)

**Objcets:** Object in Js are key and value pair. Key meaning data type-

EG: const item = {

“harry” = true,

“Ram” = False,

“Subh” = 35

}

We can do look up of object and do mapping using key-value pairs.

**Practice set question- Basic:**

1. Create a variable of type string and try to add a number to it. (Concatenation concept)
2. Use type of function to find out the datatype of a variable.
3. Create a const object in Js and can you change it to hold a number later- No, it can not be done.
4. Try to add a new key to the const object in problem 3, were you able to do it ?
5. WAP to create a word meaning dictionary of 5 words.

**JavaScript Operators and Expressions:**

A fragment of code that produces a value called an expression. Every value written in JS is literally an expression. Suppose, Harry or “+”

**Operators**: Addition(+), Multiplication(\*), Division(/), Subtraction(-), Exponentials(\*\*), incremental(++), decrement(--), modulus(%)

**Comparison Operators:** equal to(==), not equal(!=), equal value and type of the variable both(===), not equal value and type of the variable both(!==), greater than(>), less than(<). Greater equal(>=), less equal(<=)

**Logical Operators:** It generally works on Boolean, &&(logical and), logical OR (||), logical NOT(!)

**Conditional Expressions:**

Sometimes, we might have to execute a block of code based on some condition. In Js we have three types of if-else statements.

1. If 2) If-else 3) if-else if -else statement

**Prompt() and alert() function in JS:**

prompt() instructs the browser to display a dialog with an optional message prompting the user to input some text, and to wait until the user either submits the text or cancels the dialog.

Let a= prompt(“what is your age”)

Alert(): It display a message.

**Typecasting string into number**- Number.ParseInt(a)

1)IF : It enables the condition inside a bracket if the condition is evaluated true then, the code executes, otherwise it goes under the else bracket.

If the condition is true then it goes under if and if the condition is false then it goes under else.

**Ternary operator:**

*console.log*("you can do", (a>5 ? "bad" : "good"))

**Loop and Function in JS:**

For loop, While loop, Do-While loop, For-IN loop, Foor-Of loop

Usage: We use loop to perform a repeated action.

**Type of Loops:**

1. For Loop: Loop a block of code, no of times
2. For In Loop: Loop through the keys of an object
3. For Of Loop: Loop through the values of an object
4. While Loop: Loop a block based on a specific condition
5. Do-While Loop: While Loop variant which runs at least one.
6. For Loop: for(statement1; statement2; statement3)

Statement1-> it executes only one time

Statement2-> It is a condition, if condition is true then only loop body executes

Statement3-> It executes everytime when loop body is executed

**while and do-while loop:**

In while loop first the condition is checked then the block runs. But in do-while loop first the block runs after that the condition got checked. Do-while loop at least runs once before the condition got checked.

**Function in JS:**

Function is a block of code assigned to perform a particular task.

When we use a repeated logic again and again and use that in our code, for the betterment of code we can separate the logic and put that in a function so that from the next time onwards we just call it and it work.

Function invocation:- function(arg1,arg2) ;

Function invocation is a way to use the function inside a code.

**let a=parseInt(prompt("take first number"));**

**let b=parseInt(prompt("take second number"));**

**function sum(a,b){**

**return 1+((a+b)/2)**

**}**

**console.log("the value is", sum(a,b))**

**Arrow function:**

**c**onst hello=()=>{

Console.log()

}

**c**onst sum=(a,b)=>{

return a+b

}

Practice set:

Q1) create and access an object of marks and use for loop

const *marks*={

*Riya*:44,

*Simi*:67,

*Jinia*:8,

*Ammna*:5

}

//*create for loop*

*for*(leti=0; i<*Object.keys*(marks)*.*length; i++)

{

*console.log*("The marks of " + *Object.keys*(marks)[i] + "is" + marks[*Object.keys*(marks)[i]])

}

**String in JS:**

Strings are used to store and manipulate text. It is collection of character. It can be created using the following syntax-

Let name = “Deblina”

**Template Literal:**

It use backticks instead of quote to define a function. We can use $ sign to refer variable. With Template literal it is possible to use both single quoted and double quoted inside a string.

We can insert variable directly in Template Literal, this is call **string interpolation**. **${variable}**

leta1="Deblina"

leta2="Indranil"

letsentence=`${a1} *is a friend of* ${a2}`

*console.log*(sentence)

**Escape sequence character:**

We can use single quote escape sequence character inside our string.

Let fruit = r-carriage return, t-tab, n-new line

letfruit='App\'le'

*console.log*(fruit)

letfruit1='App\nle'

*console.log*(fruit1)

letfruit2='App\tle'

*console.log*(fruit2)

**Methods in String:**

String is immutable, it can never be changed but new string can be made out of it. Difference between property and a function.

Name.length ->property

Name.toUpperCase() - > method

Name.toLowerCase() -> method

Escape sequence character consider as one charecter.(eg: back slash n)

Name.slice(2,4)-> print 2nd and 3rd position

Name.slice(2) ->print 2nd position to last

Name.replace(“Deb”,”Rim”) ->the deb position in string will be replaced by Rim

Name.concat(“I am a very nice”, Name, “ok done”)

Name.trim() -> it will remove all the blank space from string

**Use .includes in string**: It return **true** or **false** value

//*use .includes*

letsentence=`*Deblina is a very good girl*`

letword="very"

*console.log*(*sentence.includes*(word))

*console.log*(`the word ${word} ${*sentence.includes*(word) ? "is" : "is not"} there`)

**Array in Js:**

1. An array can hold many values under a single name, and you can access the values by referring to an index number.
2. Arrays are variable which can hold more than one value.
3. Syntax: let arr= square bracket
4. Array will return undefined if we try to access an index which doesn’t come in array list.
5. he value of array can also be change by using index number.
6. We use const variable to define an array mostly. But It does NOT define a constant array. It defines a constant reference to an array. Because of this, we can still change the elements of a constant array.
7. Arrays are typeof object.

**Methods in Array:**

toString() -> convert an array into a string of comma-separated value

join() -> join all the array elements using a separator

pop() -> remove the last element from the array and return the popped element

push() -> add new elements at the end of the array and return the length of the array

shift() -> removes the first array element and "shifts" all other elements to a lower index. The shift() method returns the value that was "shifted out"

unshift() -> the unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements. The unshift() method returns the new array length.

delete -> it is not a method but a operator. After deleting also the length of the array stays same. It doesn’t change.

letb= *cars.push*("Skoda")

*console.log*(cars, b)

delete cars[4]

*console.log*(cars)

*console.log*(cars[4])

(6) ['BMW', 'BENZ', 'TOYOTA', 'Audi', 'jaguar', 'Skoda'] 6

practice 401.js:14

(6) ['BMW', 'BENZ', 'TOYOTA', 'Audi', …, 'Skoda']

practice 401.js:16

undefined

So, it means the value just get deleted but the memory space still remains and it reutn a undefined value.

concat() ->use to join array from a given array-. It doesn’t change existing array but create anew array.

new\_arry = oldarray.concat(concatenated\_array)

sort() -> it sort an array alphabetically. It changes the original array.

**Numeric Sort:** By default, the sort() function sorts values as strings.

This works well for strings ("Apple" comes before "Banana"). However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".Because of this, the sort() method will produce incorrect result when sorting numbers.

You can fix this by providing a compare function:

const points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return a - b});

**The Compare Function:** the purpose of the compare function is to define an alternative sort order. The compare function should return a negative, zero, or positive value, depending on the arguments:

***function(a, b){return a - b}***

When the sort() function compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.

If the result is negative, a is sorted before b. If the result is positive, b is sorted before a. If the result is 0, no changes are done with the sort order of the two values.

**Example:** the compare function compares all the values in the array, two values at a time (a, b). When comparing 40 and 100, the sort() method calls the compare function(40, 100).

The function calculates 40 - 100 (a - b), and since the result is negative (-60),  the sort function will sort 40 as a value lower than 100.

***NOTE:*** *The values in the num array are all numbers, so when the sort() method compares them, it knows that it is dealing with numbers and performs the comparison accordingly. Therefore, there is no need to explicitly define the type of a and b in the sorting function*

**reverse()** - > reverse method reverses the elements in an array.

**Splice()** -> It can be used to add new items in array and also deleted existing items from the array depending on the index position

**Slice()** - > it will remove the items from an array and make a new array

**Loops with Array**

forEach Loop: *{*

*num.forEach*((parameter)=>

{*console.log*(parameter+1)})

It call s function for each array element. forEach method will only works on array. It takes one argument and do the required work on that.

forEach can take three arguments also. forEach(value,index,array)

**Array.from** -> use to create an array from any another object. Maybe when we want a string to change an array we will use this.

Array from string:

letname="Deblina Karmakar"

*console.log*(*Array.from*(name))

O/P: 16) ['D', 'e', 'b', 'l', 'i', 'n', 'a', ' ', 'K', 'a', 'r', 'm', 'a', 'k', 'a', 'r']

**Map, Filter and reduce in JS:**

This is higher order array methods.

**map()** -> creates a new array by performing some operation on each array element. We used map when we needed to make a new array. Return an array.

letarr=[1,3,9,4,6]

letp=*arr.map*((*element*,*index*)=>{

*console.log*(element,index,arr)

*return* (element+1)

})

*console.log*(p)

**Filter() ->** Filters an array and form a new array which passes a given test. Return an array.

letarr2=[1,3,9,4,6,98,100]

lett=*arr2.filter*((*item*)=>{

*return* item>10

})

*console.log*(t)

**reduce() ->** itreturns a value and perform an operation.

letarr3=[1,3,9,4,6,98,100]

letk=*arr3.reduce*((*n1*,*n2*,*n3*)=>{

*return* n1+n2\*n3

})

*console.log*(k)

1. Note: The **console.log** statement is placed after the **return** statement, which means it will never be executed. The **return** statement ends the function execution and returns a value to the caller. Any code after the **return** statement is unreachable.

**Javascript in Browser:**

Form JS was initially created to make webpages alive. Js was written right in the webpages`s HTML to make it look attractive. The browser has an embedded engine calked JavaScript engine or JS runtime. JS ability in the browser is very limited to protect the user’s safety.

HTML is the skeleton of web pages. JS is the logic of webpages. CSS is the styling in webpages.

Developer’s Tool: every browser have some developers tool which make a developer life little bit easy. There are three important tools that are elements, console and network.

Element: all HTML elements

Console: Where all the error are logged.

Network: All return request

**JS console Object**

The console object has several methods and log is one of them. Some of them are as follows:

Assert() –

Clear() – clear the console

Log()- output a message to console

Table()- displays table data

Warn()- use for warning

Error()- use for error

Info()- use for special information

Time()- shows time

**Alert, Prompt, Confirm in JS:**

alert() – used to invoke a mini window with a message. Eg- alert(“how are you”)

prompt() – it takes an input and store it, it take user input as a string always

prompt(variable, “No”) – No is a default value here, it can store default values also.

document.write(varibale)

confirm () – shows a message and waits for the user to press it either ok or cancel it. Return true for ok and false for cancel.

**Window object, BOM,DOM:**

Shows we have the following when Javascript runs in a browser. Window object represents browser window and provides methods to control them. It is also called global object in JS.

JS CORE

BOM

DOM

**BOM:** The **Browser Object Model** (BOM) is used to interact with the browser. It represents the additional objects provided by the browser for working with everything except documents.

The default object of browser is window means you can call all the functions of window by specifying window or directly. For example:

1. window.alert("hello javatpoint");

is same as:

alert("hello javatpoint");

You can use a lot of properties (other objects) defined underneath the window object like document, history, screen, navigator, location, innerHeight, innerWidth

**DOM:** Document object model represents the page content as HTML.

document.body – page body as JS object

document.body.style.backgroud = “green” – change the background colour to green

**DOM Manupulation:**

DOM tree refers to the HTML page where all the nodes are object. There can be three main type of nodes.

* Text Nodes
* Element Nodes
* Object Nodes

In HTML page, <html> is at the root and <head> and <body> are the children.

A text node is always a leaf node of the tree.

**Autocorrect:** If an erroneous HTML code is encountered by the browser, then the browser tends to correct it by its own, that is called auto correct.

**document.body()** – It can sometimes return null if the script is written before the body tag.

**Accessing children, parent and sibling of an element:**

**Children of an element**: Direct as well as deeply nested elements of an element are called its children.

**Child Nodes**: Elements that are direct children. Suppose, body and head tags are children of HTML tag.

**Descendant Nodes**: All nested elements, children, their children and so on.

Element.firstchild() – first child element

Element.lastchild() – last child element

Element.childNodes() – all child nodes

**$0-** To give the currently selected element

**$1**-to give previously selected element

Child Nodes: It looks like an array but not an array

**DOM Collection:** they are read-only, they are live.

Head and body are sibling of each other, sibling meaning both have the same parent.

<body> is said to be the next or right sibling of <head> and <head> is called the the previous or left sibling of <body>.

**document.documentElement.parentNode()** – return document

**document.documentElement.parentElement**() – return null, if the parent is an element then only return value, otherwise return null.

Document.body – document.body.firstChild

document.body.firstElementChild – gives only elements

**Table Links:**

Table elements support the following properties:

Table.rows –

table.caption –

table.tHead

Table.tFoot-

Table.tBodys

Tbody.rows

Tr.cells

Tr.SectionRowIndex

Tr.RowIndex

**document.getElementById(a)**-It will give the element which id is declared as a.

**matches(),closest() and contains() methods:**

element.matches() – checks if the element matches the given CSS selector or not.

Eg; let t=document.getElementById(id1)

Id1.matches(“.class”) – return true or false

element.closest() – check for the nearest ancestor that matches the css selector.

sp1.closest(“.class”)

eg: **console.log(id1.contains(sp1))**

element1.contains(element2) – return true if element2 is inside element1(or decentdant of element1) or when element1 ==element2

**IMP NOTE:** if tbody tag is not used while making a table then they will not be seen in the view page as well as it only shows the raw content whatever is written in the program but in browser if you open it, in console it will be seen, tbody tag will automatically inserted there.

**innerHTML and outerHTML:**

console.log() – shows the element DOM tree

console.dir() – shows element as an object with its properties

tagName – only exists for element node

elementName -exist for any node(text node, element node )

**innerHTML()** – allows to get the HTML inside the element as a string, only applicable for element nodes.

**outerHTML() –** contains full element, inner HTML + element itself

**NOTE**: innerHTML is valid only foe the element nodes, for other node type we can use nodevalue or data.

EG:- document.body-firstChild.data or document.body-firstChild.nodevalue

document.body.textContains -

**Different Attributes :**

Document.getAttribute():- used to get the value of an attribute

document.setAttribute():- used to set the value of an attribute

document.hasAttribute():- check the existence of an C

document.removeAttribute()

**data- attributes**- So, we can always create customize attributes but the attributes started with “data-“ are reserved for the programmer use. They can be access through dataset property.If an element has an attribute called data-one that is available in data.one.

EG: <div id=”first”data-game=”POP”>

Console.log(first.dataset) – will return POP or Console.log(first.dataset.game) – will return POP

**DOM Insertion methods :**

Insert elements in DOM using below methods:

let a=document.getElementsByTagName("div")[0]

a.innerHTML= a.innerHTML + "<h1>Good Day</h1>"

let div=document.createElement("div")

div.innerHTML="<p>How are you doing?<p/>"

a.appendChild(div)

a.append(div)

a.prepend(div)

a.before(div)

a.after(div)

a.replaceWith(div)

**className and classList in JS:**

If we assign something to the element.className, it will replaces the whole string of class.

Elem.classList.add/remove(“class”)

Elem.classList.toggle(“class”) – add the class if it not present and remove if it was there

Elem.classList.contains(“class”)

**setTimeout and setInterver:**

**setTimeout(function,lacking time, arguments)**

**clearTimeout() –** stop setTimeout

**Browser Events in JS:**

# Introduction to browser events:

An event is a signal that something has happened. All DOM nodes generate such signals (but events are not limited to DOM).

Here’s a list of the most useful DOM events, just to take a look at:

**Mouse events:**

* click – when the mouse clicks on an element (touchscreen devices generate it on a tap).
* contextmenu – when the mouse right-clicks on an element.
* mouseover / mouseout – when the mouse cursor comes over / leaves an element.
* mousedown / mouseup – when the mouse button is pressed / released over an element.
* mousemove – when the mouse is moved.

**Keyboard events:**

* keydown and keyup – when a keyboard key is pressed and released.

**Form element events:**

* submit – when the visitor submits a <form>.
* focus – when the visitor focuses on an element, e.g. on an <input>.

**Document events:**

* DOMContentLoaded – when the HTML is loaded and processed, DOM is fully built.

**CSS events:**

* transitionend – when a CSS-animation finishes.

There are many other events. We’ll get into more details of particular events in upcoming chapters.

## [Event handlers](https://javascript.info/introduction-browser-events" \l "event-handlers)

To react on events we can assign a handler – a function that runs in case of an event.

Handlers are a way to run JavaScript code in case of user actions.

There are several ways to assign a handler. Let’s see them, starting from the simplest one.

### [HTML-attribute](https://javascript.info/introduction-browser-events" \l "html-attribute)

A handler can be set in HTML with an attribute named on<event>.

For instance, to assign a click handler for an input, we can use onclick, like here:

<input value="Click me" onclick="alert('Click!')" type="button">

On mouse click, the code inside onclick runs.

Please note that inside onclick we use single quotes, because the attribute itself is in double quotes. If we forget that the code is inside the attribute and use double quotes inside, like this: onclick="alert("Click!")", then it won’t work right.

An HTML-attribute is not a convenient place to write a lot of code, so we’d better create a JavaScript function and call it there.

Here a click runs the function countRabbits():

<script>

function countRabbits() {

for(let i=1; i<=3; i++) {

alert("Rabbit number " + i);

}

}

</script>

<input type="button" onclick="countRabbits()" value="Count rabbits!">

As we know, HTML attribute names are not case-sensitive, so ONCLICK works as well as onClick and onCLICK… But usually attributes are lowercased: onclick.

### [DOM property](https://javascript.info/introduction-browser-events" \l "dom-property)

We can assign a handler using a DOM property on<event>.

For instance, elem.onclick:

<input id="elem" type="button" value="Click me">

<script>

elem.onclick = function() {

alert('Thank you');

};

</script>

If the handler is assigned using an HTML-attribute then the browser reads it, creates a new function from the attribute content and writes it to the DOM property.

So this way is actually the same as the previous one.

These two code pieces work the same:

1. Only HTML:

<input type="button" onclick="alert('Click!')" value="Button">

1. HTML + JS:
2. <input type="button" id="button" value="Button">
3. <script>
4. button.onclick = function() {
5. alert('Click!');
6. };

</script>

In the first example, the HTML attribute is used to initialize the button.onclick, while in the second example – the script, that’s all the difference.

**As there’s only one onclick property, we can’t assign more than one event handler.**

In the example below adding a handler with JavaScript overwrites the existing handler:

<input type="button" id="elem" onclick="alert('Before')" value="Click me">

<script>

elem.onclick = function() { // overwrites the existing handler

alert('After'); // only this will be shown

};

</script>

To remove a handler – assign elem.onclick = null.

## [Accessing the element: this](https://javascript.info/introduction-browser-events" \l "accessing-the-element-this)

The value of this inside a handler is the element. The one which has the handler on it.

In the code below button shows its contents using this.innerHTML:

<button onclick="alert(this.innerHTML)">Click me</button>

## [Possible mistakes](https://javascript.info/introduction-browser-events" \l "possible-mistakes)

If you’re starting to work with events – please note some subtleties.

We can set an existing function as a handler:

function sayThanks() {

alert('Thanks!');

}

elem.onclick = sayThanks;

But be careful: the function should be assigned as sayThanks, not sayThanks().

// right

button.onclick = sayThanks;

// wrong

button.onclick = sayThanks();

If we add parentheses, then sayThanks() becomes a function call. So the last line actually takes the result of the function execution, that is undefined (as the function returns nothing), and assigns it to onclick. That doesn’t work.

…On the other hand, in the markup we do need the parentheses:

<input type="button" id="button" onclick="sayThanks()">

The difference is easy to explain. When the browser reads the attribute, it creates a handler function with body from the attribute content.

So the markup generates this property:

button.onclick = function() {

sayThanks(); // <-- the attribute content goes here

};

**Don’t use setAttribute for handlers.**

Such a call won’t work:

// a click on <body> will generate errors,

// because attributes are always strings, function becomes a string

document.body.setAttribute('onclick', function() { alert(1) });

**DOM-property case matters.**

Assign a handler to elem.onclick, not elem.ONCLICK, because DOM properties are case-sensitive.

## [addEventListener](https://javascript.info/introduction-browser-events" \l "addeventlistener)

The fundamental problem of the aforementioned ways to assign handlers is that we can’t assign multiple handlers to one event.

Let’s say, one part of our code wants to highlight a button on click, and another one wants to show a message on the same click.

We’d like to assign two event handlers for that. But a new DOM property will overwrite the existing one:

input.onclick = function() { alert(1); }

// ...

input.onclick = function() { alert(2); } // replaces the previous handler

Developers of web standards understood that long ago and suggested an alternative way of managing handlers using the special methods addEventListener and removeEventListener which aren’t bound by such constraint.

The syntax to add a handler:

element.addEventListener(event, handler, [options]);

**event**

Event name, e.g. "click".

**handler**

The handler function.

**options**

An additional optional object with properties:

* once: if true, then the listener is automatically removed after it triggers.
* capture: the phase where to handle the event, to be covered later in the chapter [Bubbling and capturing](https://javascript.info/bubbling-and-capturing). For historical reasons, options can also be false/true, that’s the same as {capture: false/true}.
* passive: if true, then the handler will not call preventDefault(), we’ll explain that later in [Browser default actions](https://javascript.info/default-browser-action).

To remove the handler, use removeEventListener:

element.removeEventListener(event, handler, [options]);

**Removal requires the same function**

To remove a handler we should pass exactly the same function as was assigned.

This doesn’t work:

elem.addEventListener( "click" , () => alert('Thanks!'));

// ....

elem.removeEventListener( "click", () => alert('Thanks!'));

The handler won’t be removed, because removeEventListener gets another function – with the same code, but that doesn’t matter, as it’s a different function object.

Here’s the right way:

function handler() {

alert( 'Thanks!' );

}

input.addEventListener("click", handler);

// ....

input.removeEventListener("click", handler);

Please note – if we don’t store the function in a variable, then we can’t remove it. There’s no way to “read back” handlers assigned by addEventListener.

Multiple calls to addEventListener allow it to add multiple handlers, like this:

<input id="elem" type="button" value="Click me"/>

<script>

function handler1() {

alert('Thanks!');

};

function handler2() {

alert('Thanks again!');

}

elem.onclick = () => alert("Hello");

elem.addEventListener("click", handler1); // Thanks!

elem.addEventListener("click", handler2); // Thanks again!

</script>

As we can see in the example above, we can set handlers both using a DOM-property and addEventListener. But generally we use only one of these ways.

**For some events, handlers only work with addEventListener**

There exist events that can’t be assigned via a DOM-property. Only with addEventListener.

For instance, the DOMContentLoaded event, that triggers when the document is loaded and the DOM has been built.

// will never run

document.onDOMContentLoaded = function() {

alert("DOM built");

};

// this way it works

document.addEventListener("DOMContentLoaded", function() {

alert("DOM built");

});

So addEventListener is more universal. Although, such events are an exception rather than the rule.

## [Event object](https://javascript.info/introduction-browser-events" \l "event-object)

To properly handle an event we’d want to know more about what’s happened. Not just a “click” or a “keydown”, but what were the pointer coordinates? Which key was pressed? And so on.

When an event happens, the browser creates an event object, puts details into it and passes it as an argument to the handler.

Here’s an example of getting pointer coordinates from the event object:

<input type="button" value="Click me" id="elem">

<script>

elem.onclick = function(event) {

// show event type, element and coordinates of the click

alert(event.type + " at " + event.currentTarget);

alert("Coordinates: " + event.clientX + ":" + event.clientY);

};

</script>

Some properties of event object:

**event.type**

Event type, here it’s "click".

**event.currentTarget**

Element that handled the event. That’s exactly the same as this, unless the handler is an arrow function, or its this is bound to something else, then we can get the element from event.currentTarget.

**event.clientX / event.clientY**

Window-relative coordinates of the cursor, for pointer events.

There are more properties. Many of them depend on the event type: keyboard events have one set of properties, pointer events – another one, we’ll study them later when as we move on to the details of different events.

**The event object is also available in HTML handlers**

If we assign a handler in HTML, we can also use the event object, like this:

<input type="button" onclick="alert(event.type)" value="Event type">

That’s possible because when the browser reads the attribute, it creates a handler like this: function(event) { alert(event.type) }. That is: its first argument is called "event", and the body is taken from the attribute.

## [Object handlers: handleEvent](https://javascript.info/introduction-browser-events" \l "object-handlers-handleevent)

We can assign not just a function, but an object as an event handler using addEventListener. When an event occurs, its handleEvent method is called.

For instance:

<button id="elem">Click me</button>

<script>

let obj = {

handleEvent(event) {

alert(event.type + " at " + event.currentTarget);

}

};

elem.addEventListener('click', obj);

</script>

As we can see, when addEventListener receives an object as the handler, it calls obj.handleEvent(event) in case of an event.

We could also use objects of a custom class, like this:

<button id="elem">Click me</button>

<script>

class Menu {

handleEvent(event) {

switch(event.type) {

case 'mousedown':

elem.innerHTML = "Mouse button pressed";

break;

case 'mouseup':

elem.innerHTML += "...and released.";

break;

}

}

}

let menu = new Menu();

elem.addEventListener('mousedown', menu);

elem.addEventListener('mouseup', menu);

</script>

Here the same object handles both events. Please note that we need to explicitly setup the events to listen using addEventListener. The menu object only gets mousedown and mouseup here, not any other types of events.

The method handleEvent does not have to do all the job by itself. It can call other event-specific methods instead, like this:

<button id="elem">Click me</button>

<script>

class Menu {

handleEvent(event) {

// mousedown -> onMousedown

let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);

this[method](event);

}

onMousedown() {

elem.innerHTML = "Mouse button pressed";

}

onMouseup() {

elem.innerHTML += "...and released.";

}

}

let menu = new Menu();

elem.addEventListener('mousedown', menu);

elem.addEventListener('mouseup', menu);

</script>

Now event handlers are clearly separated, that may be easier to support.

## [Summary](https://javascript.info/introduction-browser-events" \l "summary)

There are 3 ways to assign event handlers:

1. HTML attribute: onclick="...".
2. DOM property: elem.onclick = function.
3. Methods: elem.addEventListener(event, handler[, phase]) to add, removeEventListener to remove.

HTML attributes are used sparingly, because JavaScript in the middle of an HTML tag looks a little bit odd and alien. Also can’t write lots of code in there.

DOM properties are ok to use, but we can’t assign more than one handler of the particular event. In many cases that limitation is not pressing.

The last way is the most flexible, but it is also the longest to write. There are few events that only work with it, for instance transitionend and DOMContentLoaded (to be covered). Also addEventListener supports objects as event handlers. In that case the method handleEvent is called in case of the event.

No matter how you assign the handler – it gets an event object as the first argument. That object contains the details about what’s happened.

We’ll learn more about events in general and about different types of events in the next chapters.

## [Tasks](https://javascript.info/introduction-browser-events#tasks)

### [Hide on click](https://javascript.info/introduction-browser-events" \l "hide-on-click)

importance: 5

Add JavaScript to the button to make <div id="text"> disappear when we click it.

The demo:

[Open a sandbox for the task.](https://plnkr.co/edit/6vWtOzQrn7qGGuey?p=preview)

solution

### [Hide self](https://javascript.info/introduction-browser-events" \l "hide-self)

importance: 5

Create a button that hides itself on click.

Like this:

solution

### [Which handlers run?](https://javascript.info/introduction-browser-events" \l "which-handlers-run)

importance: 5

There’s a button in the variable. There are no handlers on it.

Which handlers run on click after the following code? Which alerts show up?

button.addEventListener("click", () => alert("1"));

button.removeEventListener("click", () => alert("1"));

button.onclick = () => alert(2);

solution

### [Move the ball across the field](https://javascript.info/introduction-browser-events" \l "move-the-ball-across-the-field)

importance: 5

Move the ball across the field to a click. Like this:

Requirements:

* The ball center should come exactly under the pointer on click (if possible without crossing the field edge).
* CSS-animation is welcome.
* The ball must not cross field boundaries.
* When the page is scrolled, nothing should break.

Notes:

**Callbacks in JS:**

Script.onload = function(){}

Script.onerror = function(){}

//promise resolve

let p = new Promise((resolve,reject) => {

console.log("I am a promise")

setTimeout(()=>{

console.log("I am fullfiled")

resolve(true)

}, 5000)

console.log("I am pending")

})

console.log(p)

//promise reject

let p1 = new Promise((resolve,reject) => {

console.log("I am a promise")

setTimeout(()=>{

console.log("I am rejected")

rejecet(new Error("I am an error"))

}, 5000)

console.log("I am pending")

})

console.log(p1)

**Promise Chaining:**

**onst loadscript = (src) => {**

**return new Promise((resolve, reject) => {**

**let script = document.createElement("script");**

**script.type = "text/javascript";**

**script.src = src;**

**document.body.appendChild(script);**

**script.onload = () => {**

**resolve("script loaded successfully");**

**};**

**script.onerror = () => {**

**reject("Script has failed to load");**

**};**

**});**

**};**

**let p1 = loadscript(**

**"https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/js/bootstrap.min.js");**

**p1.then((value) => { console.log(value);**

**return loadscript("https://cdn.jsdelivr.net/npm/popper.js@1.14.7/dist/umd/popper.min.j")**

**}).catch((error) => {**

**console.log("there has been some difficulty loading your script");**

**console.log(error);**

**});**

**PROMISE API:**

letp\_all= *Promise.all*([p1,p2,p3])

*p\_all.then*((value)=>{

*console.log*(value)

})