**WEB**

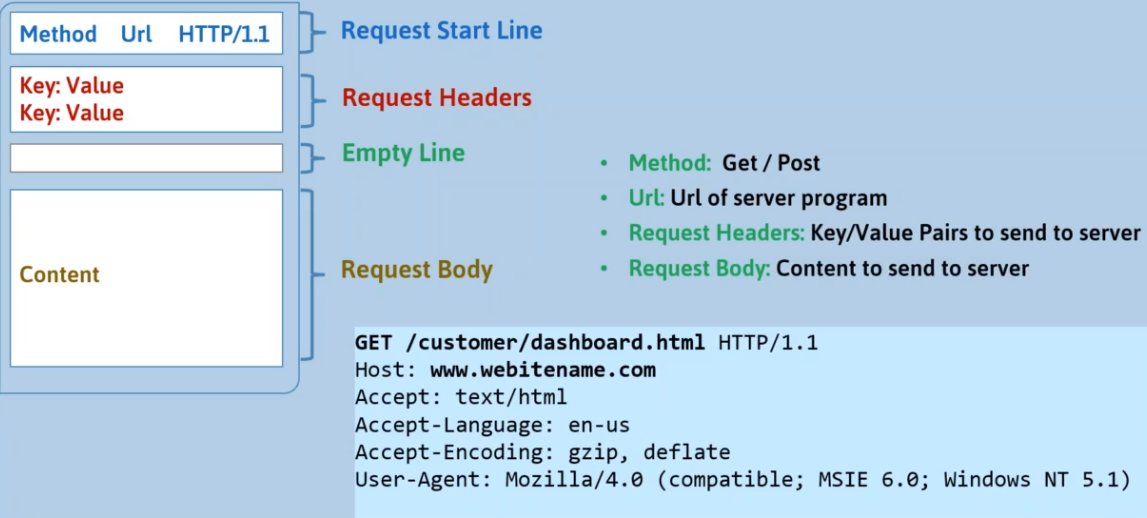
Last Update: May 30 2021.

Most services on the web use HTTP(port 80) or HTTPS(port 443) for communication.

• Request and Response: First, browser sends a request to the server and then server sends a response back to the browser. If no response is received in a set time then it times out.

• HTTP: Hyper Text Transfer Protocol, used primarily for webpages (html documents).

First HTTP request message is sent by the browser, it has this format.



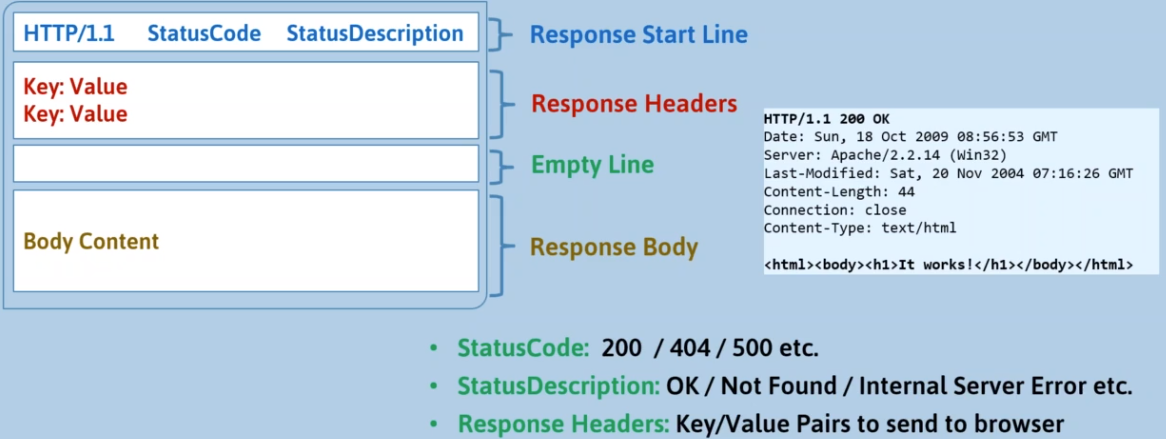
This request message is sent in either GET/POST method.

Get: Any data is sent as query parameters in the url itself. Format: <url>?<key>=<value>&<key2>=<value2> and so on. 2048 chars limit, only ASCII.

Post: Data is sent in a request body, no size limit.

Along with url, http request headers are also sent by the browser. The same is returned with the response as well. They specify what the browser is, time, cookies, mime types, content length etc.

HTTP response message is then received by the browser, it has this format.



**HTML**

• Hyper Text Markup Language

Document to describe webpages.

• <tagname> content </tagname> basic syntax

•<!DOCTYPE html> means the document is an HTML file.

• Meta: inside head, metadata information about the doc. Self-closing tag. It defines important metadata about the html document.

•Basic Tags: Consists of Block (starts on new line) and inline elements (no new line),

HTML: Defines the starting and ending point of an html document.

Head: head, this is used to describe the webpage.

Title: inside head.

Body: main body. There can only be 1 body in an HTML doc.

Hx: header, big heading. Subsequent headers decrease in size.

P: inside hx , para tag.

A href=”<link>” target=”<options>”: Mark a text as hyperlink to link. Target is set to \_self by default, meaning doc opens in the same frame, use \_blank to open it in a new tab.

Img src=”<link>” alt=<text to show if image fails to load> usemap=”#<mapname>”: Self-closing tag. Maps are optional but if provided provide clickable areas on the image.

<map name=<mapname>: Map tag, it provides custom clickable areas on images.

<area shape=<shapename> coords=<x,y points depending on the chosen shape> alt=<alt text> href=<link>> : Area is a specified shape or area defined to be clickable in a map. Self-closing tag.

Picture: Picture tag allows the browser to choose from multiple images and display any one that it supports and./or is allowed to be shown. We can even use img tag inside these.

Source: This tag is used inside picture and has media attribute to define some property that needs to be satisfied for the image to be shown and srcset attribute which contains image location.

B, Strong: Bolds the text.

Br: line break.

Hr: Paragraph break. Self-closing tag.

Pre: Preformatted text tag, meaning whatever is inside these will not be formatted by the browser and so spaces, indents etc. will remain as is.

Em, i: italic.

Small: smaller text

Mark: highlights text

Q: quotes text

Iframe src=”source” title=”title”: This tag is used to provide a box which holds another html page.

Blockquote: Indents a para and centers it to say it is quoted.

Cite: Changes text to italic and is used to cite stuff.

Address: Used to say text is address

Bdo dir=””: Bi-directional override, ltr and rtl are 2 of those.

Abbr title=””: Abbreviates text, given in the title with text given between the start and end tag of it.

Ul: unordered list. Uses bullets for list items.

Li: list item and is inside ul.

Ol: ordered list, uses numbers for list items.

Dl: description list. Doesn’t use numbers or bullets and has description about each list item.

Dt: define terms, inside dl.

Dd: describe terms, inside dl after each dt.

Table: adds a table in the document. Use border=”1” to show a thin border.

Tr: table row, inside table. This hold tds.

Td:table data, inside tr. Each td item is another column in the table.

Div style=”” id=””: Adds style to the text by inserting rectangle row block in the doc , a valid style is: “background-color:black; color:white;”. Block element.

Span: Just like div but it doesn’t insert a rectangle row block, this is an inline element. It can also be used by css to provide a cursor attribute which changes cursor on being hovered.

Section: Creates a column.

Style: Give properties to id, each element with that id will then use the given style. There are 3 type of selectors any of which is needed, #<idname> for ids, .<classname> for classes and <tagname> for tags. We can also add classes to different types of tags and then apply only to a set of them using <tagname>.<classname> or <tagname>.<idname>. Multiselectors work, so <tagname>,<tagname> work.  
For ex:

<!DOCTYPE html>

<html>

<head>

    <title>

        Yo lets hit web dev up!

    </title>

</head>

<body>

<style>

    #main-n{

        color: aqua;

    }

</style>

<div id="main-n">

    <h1>

        Booboo

    </h1>

</div>

<div id="main-n">

    <h2>

        Booki

    </h2>

</div>

</body>

</html>

Here, 2 divs have the same id so they have the same styling.

Class attribute: Give tags a class using class=””, each class has the same style. Class attribute can have multiple classes if they are space separated, and multiple styles can be applied to the element, if two classes apply style for same attribute then specificity is followed.

For example:

Class=”one two”; are 2 classes with one and two.

Id must be unique per tag but class can be same.

Script: This tag holds various javascript functions which can be called in attributes such as onPressed by simply “<function name>()”.

defer: This attribute is used with script tag which has src. It tells browser to load the script in parallel and not wait for it to finish. Normally, in an html doc everything is loaded as it appears from top to bottom, even script tags. That means if any script tag takes too long to load it may cause slowdown on the loading of the webpage. Using defer means that it will load the script in the background and it will be finished after the document is loaded, and before the microtask queue has began and before the DOMContentLoaded Event is fired. However, these scripts keep their relative order, meaning if 1 deferred script depends upon the other then that script should be placed above in the HTML doc, it might finish loading up later but it will be executed in the order it appeared in the doc. Basically they are executed ‘appearance first’.

Async: This attribute is just like defer except browser doesn’t wait for it even after the document is loaded, these scripts are independent and shouldn’t rely on other scripts. The DOMContentLoaded event is not dependent on the loading of these scripts. Basically they are executed ‘load first’, i.e., first to load is first to be executed.

If we insert a script using DOM then it is async by default. If we set async to false then it is deferred.

Form action=”function” method=”get or post”: Creates a form. The action attribute is invoked when submit button is pressed.

Input type=”” name=”” placeholder=””: inside form, used to create a input box with name as its name, placeholder as the dimmed text and type that specifies the type of input. Text type creates a textbox and Submit creates a submit button. An input can have a form attribute, which should have a value that is an id of a form. This way any input tag can be an input of any form. Some input types such as reset and submit have special meaning to the form tag, meaning if they are pressed they affect the form, however if a submit input type has a formaction attribute specified, that action is done over the action defined in the form tag. Self-closing tag.

Label for=”name”: Label is used to provide names to input types, for attribute is supposed to have the same id as its corresponding input type.

Svg: svg tag allows custom svg to be drawn, we can specify width and height of the box that holds the svg.

Circle: Used to create circle svg.

Video: video tag can be used to insert video in an html doc. It is good practise to define width and height.

The source tag with src attribute and type attribute is used inside it to define a video, we can specify multiple videos and browser will play first one it supports.

Audio: Same as video tag, it provides controls and player from the browser.

Textarea: inside form, creates a big box for text.

Select: inside form, Creates a dropdown list.

Option value=””: inside select, elements for each items in the list.

For textarea, always access the value with textarea.value not textarea.innerHTML in DOM.

Meta: This self-closing tag holds tags which define metadata. It is only available to be used under head tag.

Some meta tags are,

Viewport: <meta name=”viewport” content=”width=device-width, initial-scale=1.0”>

Keywords:<meta name=”keywords” content=”HTML,CSS”>

Comment: <!-- <content> -->

Basic Layout:



Code: This tag is used to write code.

Kbd: Keyboard input styling.

Samp: Tag for computer output formatting to text.

Var: Tag for variable names in text.

• Entities: HTML entities are characters. Many characters aren’t available on keyboard or are reserved like the < character. To use these we can use &entityname or &#entity\_number. For UTF-8 charset which is default and can be changed or defined in meta charset, the entity numbers also cover emojis.

• URLs can only contain ASCII characters (but not space), however ascii doesn’t cover a lot of characters so browser automatically replaces such characters with %<utf8 entity number> and then at the server the %<number> is decoded for the character it was. Max URL size is 2048 characters.

**CSS**

Used to apply styles to html pages, in html most tags have a style property, css is used in the style property. CSS properties are in a key:value pair, i.e for each css property the key will be the name and value will be any applicable value. It can be implemented in 3 ways,

Inline: by specifying style in each tag in html document.

Internal: by using a <style> tag in <head>.

External: by creating a .css file and then including it in an html document by specifying it in the <link> tag.

• Basic Syntax:  
Inline: <tagname style=”property:value”>

Internal: <style>

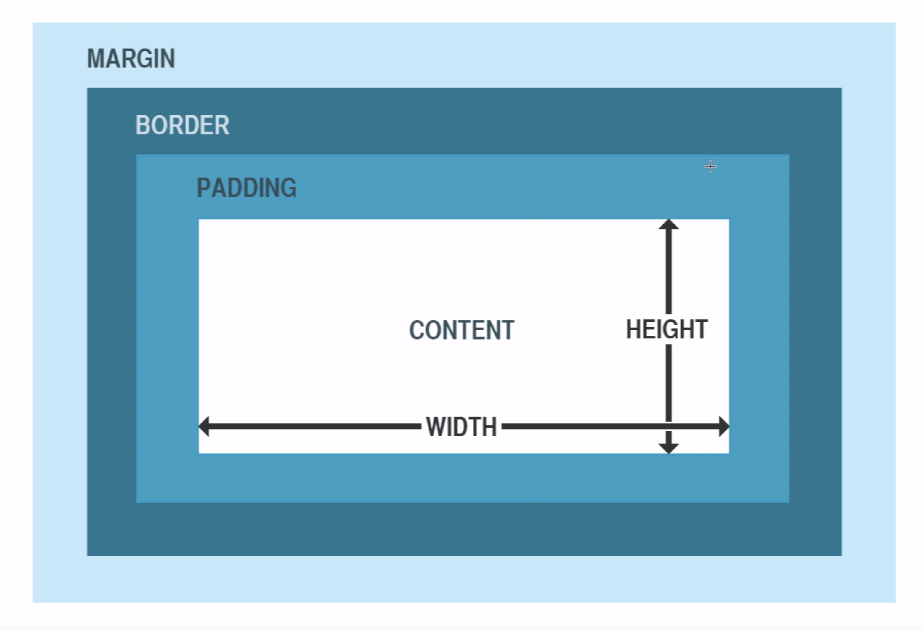
<tagname> {<properties>}

</style>

External: In another file,

<tagname> {<properties>},

•Comment: /\* <content> \*/

• CSS Box Model: Just like in flutter.

• Attribute selector: [attribute] These selectors are used to select multiple tags with ‘attribute’ attributes.  
They are:

<tagname/classname/id>[<attribute>]: Like p[HTML]{} selects all p which have the HTML attribute.

< tagname/classname/id >[<attribute>=<value>]: Like p[style=”color:blue;”]{} selects all p with the given attribute.

< tagname/classname/id >[<attribute>~=<value>]: Used to select all tags with attributes containing the specified value.

< tagname/classname/id >[<attribute>|=<value>]: Selects all tags with attribute that begins with the <value> word.

< tagname/classname/id >[<attribute>^=<value>]: Same as | but the value can be anything including a word.

< tagname/classname/id >[<attribute>$=<value>]: Selects all the tags with attributes that ends with <value> word.

< tagname/classname/id >[<attribute>\*=<value>]: Selects all the tags with attributes that contain <value> word.

\*{}: Applies the attribute to all tags.

• Combinator: These are used to make more complex selectors, like

Space selector: div p {}: Selects all p that are a descendant to div.

Child selector: div > p{}: Selects all p that are a direct child of div.

Adjacent Sibling Selector: div + p{}: Selects all p that are directly next to a div and have the same parent.

General Sibling Selector: div ~ p{}: Same as adjacent sibling except p doesn’t need to be directly next to a div.

Multi-selector: div, p{}: Selects both/more of tags.

• Pseudo-class: Selectors can have a pseudo class, i.e class with more properties. These can be applied to any tag. Like  
a:hover : We can specify a style for a button while it is being hovered.

A:visited: We can specify a style for a button if it is visited. This way we can give the same tag different styles based on its state. (However, the pseudo classes for ‘a’ have a certain order in which they must be specified.)

These are, hover, visited, link, active, first-child, nth-child(),lang etc.

• We can apply same style to multiple tags separating them with ‘,’.

•nth-child(): This selector can be applied to anything that has multiple items, and applies the css to only the numbers for which nth-child() returns true. Like

Tr:nth-child(even){…} applies css to even table rows.

• Pseudo-elements: Defined after ‘::’ these are selectors for single elements such as first line in a block element.

These are first-line, first-letter, before, after, marker, selection etc.

**•** Tags:

Link rel=”” type=”” Href=”” specifies the location: links a doc to another, rel is relation of the linking document with the current one. Stylesheet is an option. Type specifies the file type, for ex. text/css. We use link instead of style tag to keep the css in a diff file.

Background-image attribute: Takes color or image, image can be passed using url(“<(on)/(off)line location>”);

Background attribute: accepts url, color and repeat at once or using shorthand notation, space indented.

Border: Many borders to choose from and to customize as well. Border-style, width, color, sides, radius.

Margin: Margins can be extended or contracted, we can specify each side’s margin width or give auto (to horizontally center the child) or inherit (to inherit parent tag’s margins). Sometimes if a top and bottom margin is provided for 2 tags aligned vertically then the smaller margin is collapsed automatically.

Padding: Same as margin but inside the box for any tag. Width and padding add up to create padding, to avoid that we use border-box value for box-sizing attribute.

Height and width: Used to specify the same for any tag. We can set max-width too, this means the element will have this as max width and even if the window size is smaller the box will extend only the height, otherwise the browser cuts the box and provides horizontal scroll bar.

Outline: Outlines are outside border. Offset can be given to give space between child element and outline itself.

Text: Has a lot of attributes like text-align, text-transform, color, letter-spacing, text-decoration etc.

Font: While text applies to entire bodies of fonts, font applies to each character. Family, size , style all can be changed. Preferred way is to first set font-size of body to 100% then use em (1 em=16px), browsers will resize font if the user wants to change font size using browser menu if they are in em and not if they are in px. Like 2em. Px and em will resize automatically with zoom level, however to avoid that we use vw, viewport width. 2vw means 2% of max vw. To add additional fonts we can use google fonts by specifying the font css using link rel=”stylesheet”. It is preferrable to use font pairings, i.e 2 or more fonts that compliment each other and use them both. Supports shorthand.

Icons: HTML doesn’t natively bring any icons so we use scripts and/or external css stylesheets to add icons.

Display: Hides or displays and element. Default is inline. If set to none hides the element. Can make an element inline like a span or a block like a p tag or the mix with inline-block.

Visibility: Hides or displays an element, however unlike display if it is set to hidden element will still take up some space.

Position: Specifies what type of position an element will use. Absolute, relative, fixed, static etc. Default is static, meaning it sticks to the same position no matter the window or viewport.

z-index: This attribute specifies the position of an element on the z axis, meaning if there are 2 overlapping elements, the one with the higher index will be on the top. Default is that the later an element is on an HTML document the higher its z index is.

Overflow: Defines what happens with elements that exceed a set height and width.

Float: Float defines position of an element relative to its parent.

Opacity: defines opacity.

Transition: This attribute can specify transition animations that run whenever the style is built.

Counter: Like variables these can be used to hold values. Counter-reset must be defined first and a name is given to a counter, then to increment it we define counter-increment attribute and to use the value in the counter we specify counter(<variable name>) in stylesheet.

Transform: This attribute can be used to transform an element, like scale, rotate, skew etc.

Animation: This attribute can be specified for a tag and then it will animate to its keyframes using linear interpolation. Animation-name must be specified first for the tag we want to animate. Then @keyframes tag in stylesheet will define the keyframes.

@Keyframes <animation name>{ from: … to:… }: Defines keyframes for animations, meaning that the browser will apply linear interpolation to transition from a set style to a given style. Alternatively we can have more keyframes by using %. Like

@keyframe <animation name> {0% {…} 50%{…} 100%{…}}. After the animation is executed the initial state is restored for it.

Webkit-box-reflect: This attribute can be used to make reflection of an image. Defined as an attribute for the image.

Column-count: Column attributes divide and specify division properties for elements. Like column-count divides a tag into given equal parts.

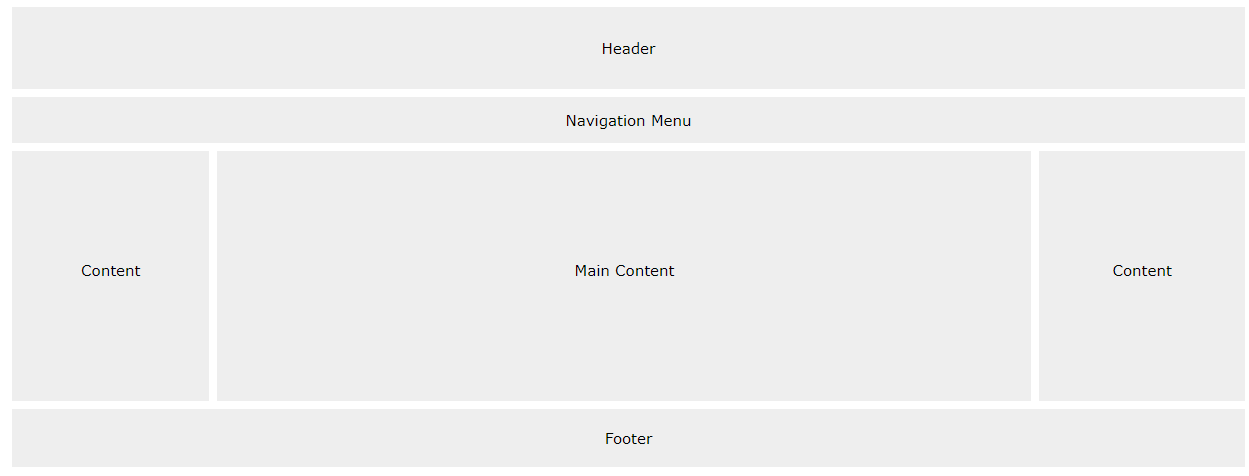
Flex: Display attribute has a value called flex, this allows us to use flexible and responsive layout.

After specifying flex value for display we can use flex attributes to specify properties for containers that wrap these elements.

Grid: Very useful value to display, grid or inline-grid are the same thing, allows us to use grids on webpage, which basically divides the webpage in many columns and rows. After specifying grid value we can use many attributes that grid brings, like grid-column, grid-area, grid-template-areas etc.

contentEditable: Used to make text content of an element editable.

• CSS Web Layout:



•Specificity: In other words, hierarchy of styles. The way to calculate specificity is a bit tedious but generally inline styles have highest priority then the styles for the given ids and then lastly multi-targeting styles such as classes, pseudo classes etc. The highest priority style is the only style that is applied to an element. However if !important is specified after an attribute then that is applied regardless of specificity, if important is specified multiple times then specificity is used back again.

• var(): We can also use var() to use variables, to declare them we do –variable-name: value, these will be available to the entire scope. We can make them global by putting them in the :root {} attribute specified at the start of the css doc. We use the value by var(variable name, optional fallback value) wherever we were to use the normal value. Variables can be overridden for and inside local scope by using the same name as they had in the global scope.

• Grid-View: Divide the html page into vertical columns.

Use box-sizing: border-box; to include padding and border in width and height, grid view needs this.

• MediaQuery: Used to set different css styles if expression evaluates to true.

Syntax: @media not|only mediatype and|not|only (expression) {…} applies styles to given elements if expression evaluates to true.

For example:

@media only screen and(min-width: 750px){} uses this style only min-width of viewport is >750px. Here screen is mediatype meaning expression is evaluated for the screen mediatype.

**Script**

This tag is the most important tag in an html document since it allows the loading of scripts, which run JavaScript to enhance the webpage and even create new ones.

Basic Syntax:

<script>   
JS code  
</script>

Can be placed in head or in body.

We can provide src attribute to allow external js file to be loaded. However if src is defined then any js code between the tags is ignored, Multiple script tags can be used so multiple files can be included.

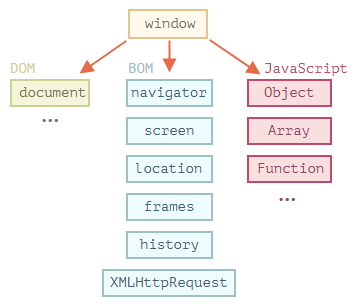
**Browser**

Browser based environments in JS have a lot of browser specific operations. It extends over the vanilla JS.

• alert(<string>): Displays string as a notification on the webpage.

•prompt(<string>, <optional default value>): Notification bar that asks for 1 string and returns the input string.

•confirm(<string>). Provides notification with ok and cancel buttons, returns true for ok press and false for cancel press.



• In a browser, the window global Object can access the DOM, BOM and JS.

• BOM: Browser Object Model, it is used to get browser specific properties (as Objects). Like OS, browser version, current URL etc.

• DOM: Document Object Model, represents all HTML/XML page content as an Object.

From an HTML document, every tag is an Object and every text string is also visible in the DOM (as #text ). Texts inside main tags are usually the leaf nodes, meaning there are no tags after them. But text can be between tags too, such as spaces and newlines which are available in the DOM. Spaces and newlines before head are ignored and if they are after /body then they are automatically pushed inside.

HTML comments are also visible with #comment tags.

The rule is that every HTML element should be in the DOM, even comments and the !Doctype.

For example:

document.body.style.background=”red”;

There’s also CSSOM, CSS object model for modifying css rules but its rarely used.

There are 12 types of DOM nodes, the main 4 are:

document   
element nodes  
text nodes  
comments

• Document: In the ‘document’ Object are all the nodes of an HTML document, the main ones are document.documentElement (the content of <HTML> tag itself>, document.<tagName> (the content of this given tag) and document.head (the content of <Head>). Since the browser goes from top to the bottom of the document, if a ‘script’ tag is in the head then it can’t read body (document.body == null), i.e document.body will be null. There are a few fixes for it like using a zero-delay timeout or modules or putting the script tag at the bottom of the HTML document.

• Node Navigation Properties/Methods: For document.<tag> there are methods that return first child, last child, all children ( all direct children tags), all descendants ( all children tags) and some other combinations. Document.<tag>.<tag> is obviously an option but there’s also this.

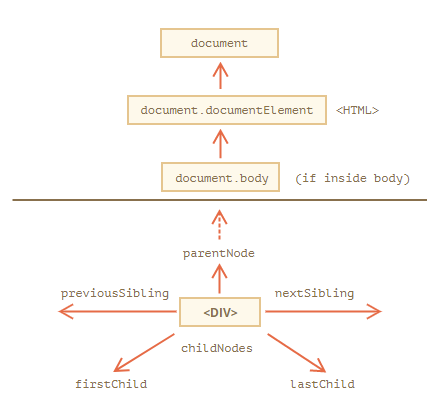
Types:  
first: document.<tag>.firstChild

Last: document.<tag>.lastChild

All children: document.<tag>.childNodes (returns a ‘collection’ which is just an iterable with all direct child nodes, use for..of).

Document.<tag>.previousSibling: used to get the previous sibling node of a tag. So if tag is body and before it is head tag under the same tag (html) then they are siblings and head is the previous Sibling of body. null if there’s no such node.

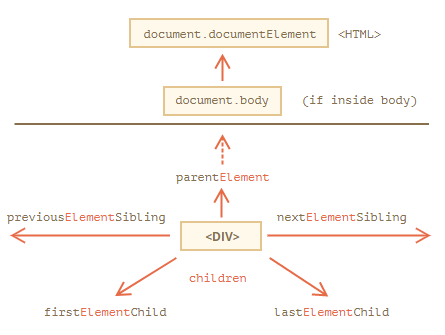
Document.<tag>.nextSibling: same as previous Sibling but for the next one. Next is the node/tag below in the tree of tags, previous is the previous node.



Almost all DOM children reference the current state of the HTML document so modifying a property in a child affects the actual live page.

• Element Navigation Properties/Methods: Elements are the tags that represent the html document, nodes are all the elements, including text and comments but elements are all the tags except these 2.

The methods are basically the same and so is the structure,



parentElement == parentNode for all tags except <HTML>. ParentNode for HTML is document Object itself, and null for parentElement. (document.documentElement.parentNode and document.documentElement.parentElement)

• Sub-properties of elements: Some elements provide helpful sub-properties on the tags.

For example:

Table

Document.table.rows: Collection of <tr> elements.

Document.table.caption: <Caption> element of a table.

Document.table.thead: <thead>

Document.table.tfoot: <tfoot>

Document.table.tBodies: Collection of <tBody> elements, will be atleast one.

<tbody>.rows: Collection of <tr> inside a <tBody>

<tr>.cells: Collection of <th> and <td> inside a <tr>

<tr>.sectionRowIndex: Returns index of the tr inside the table.

<tr>.rowIndex: Returns no. of rows in the table.

<td>.cellIndex: Returns the no. of cell inside a <tr>

• Global id: The unique IDs of tags can also be used to access the elements.   
<table id=”<x>”>

Then

<x>.tagName; //works in the script given <x> doesn’t exist in the JS global scope as another variable, and that the script is executed after the <table> tag definition.

• document.getElementById(<id>): We can assign HTML tags an ID and get a reference to them using this method. This method is only available in the document Object. It is preferred to have unique id’s since the behavior can be unpredictable otherwise.

If there’s a local variable with the same name it is prioritized.

It is advised to use this instead of directly accessing elements using document.<tag> since it could be hard to understand otherwise.

Document.getElementsByName(<name>), uses “name” property to collect nodes.

• document.querySelector(<query>): This method can return a collection of tags by running a css selector on them, which is the argument here.

• document.elementFromPoint(x,y): Returns the most nested element at this coordinate. X and Y are absolute coordinates and hence scrolling the window affects the element that will be selected. If X and Y are outside the visible area (current window) then null is returned.

• document.querySelectorAll(<query>): Returns a static collection of nodes matching query.

• <elem>.querySelector(<query>): Same as the one above but only returns the first matching tag.

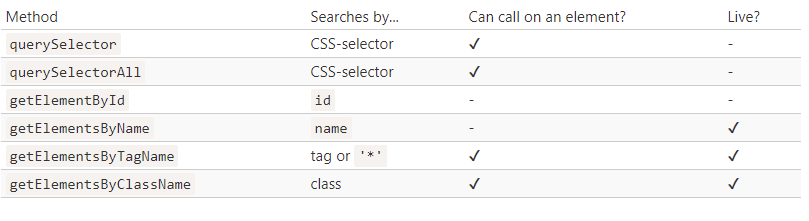
• <elem>.matches(<query>): Returns true if an element passes the query selector check.

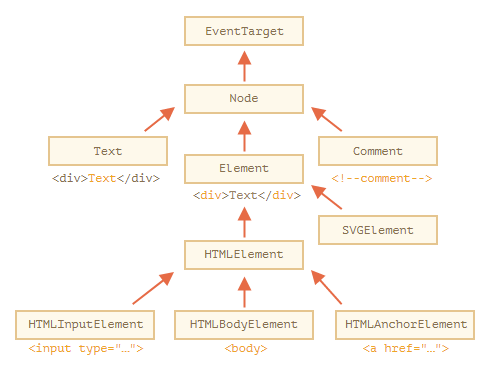
•<elem>.closest(<query>): Returns the first ancestor node that satisfies the given query. Ancestor is the node above in the tree (the current element is included in the search).

There’s also <elem>.getElementsByTagName(<tag>) . Tag can be \* for all tags.

<elem>.getElementsByClassName(<class>): Looks for tags with a given class string. Class “X” would also return “A X” in the collection.

All getElementsBy\* return live collection references, meaning the collection itself gets modified whenever the document gets modified.



• DOM Class Hierarchy:   


EventTarget inherits from Object and provides many abstract methods related to Events. Node provides many node based methods like nextSibling etc.

• console.dir(<elem>): Used to output a DOM Object into the console, has a lot more properties visible than console.log(<string>);

• We can interact with an element using the console and inspect element. If we select an element in Elements tab of chrome, a grayed out ‘$0’ is visible after it, this is to say $0 is a variable with the elements reference in it. Using the console tab, we can call methods on this element object. If we select any other element, it will be $0 whilst the older selected element loses the grayed out $0 but is automatically available in $1 , and so on for further selections.

• IDL: Interface Description Language is the language used in the spec to define DOM classes and their properties and not JS.

• nodeType: Each <elem> has a nodeType property which is an old property used to return an int for the type of node.



• nodeName and tagName: Both are strings which return the node name if the node is an element, but for nodes that aren’t element the returned string by nodeName is the `#nodeType` and undefined for tagName.

• <elem>.innerHTML: Returns the text inside a tag as a string, since this is a getter/setter we can write to it too and it will display it on the webpage. We can insert HTML tags into the string as well, except script.

<elem>.innerHTML+=”something”; does not append to the document but copy it, append the string and then writes it all back which can be pretty slow/ affect UX.

• <elem>.outerHTML: Returns the innerHTML of a tag + the tag itself. If we write to it, it replaces the tag as well. However, unlike innerHTML, the variable itself is not modified. Meaning, if we write to outerHTML then the variable remains the same as if it had been not modified. The DOM Object does get modified so it’s better to query the DOM for an <elem>’s outerHTML than reading it from any variable holding it.

• <node>.nodeValue/data: For non-element nodes like comments and text nodes, these methods can be used. These methods are almost the same in functionality.

• <elem>.textContent: Used to return and write ‘pure text’, unlike innerHTML it does not parse tags and leaves them as is. This is useful for returning user operated strings.

• <elem>.hidden: Useful to hide an element. Takes a bool.

There are a lot more properties like type, name and value which can be defined by the tag. However, them being retrieved by DOM is up-to the standard, i.e elements that extend HTMLInputElement have a DOM Object that does have type but elements/nodes that don’t like body which extends HTMLBodyElement don’t have a DOM Object that has a type property.

Any HTML tag can have any attribute, if it is in the standard, i.e if it is supported then that attribute is applied however these non-standard attributes cannot be parsed by browser or the DOM.

To get/set any attribute, we use:

<elem>.hasAttribute(<attr>)

<elem>.getAttribute(<attr>)

<elem>.setAttribute(<attr>,<val>)

<elem>.removeAttribute(<attr>)

<elem>.attributes: returns a collection of attributes which extend the Attr class with name and value properties.

<attr> name is case-insensitive in these methods and value is always string.

Any css attribute with ‘–‘ like ‘background-color’ have is available in DOM as <elem>.BackgroundColor

• Property-attribute synchronization: Usually properties sync between property data and attribute, like ‘id attribute. But there are some attributes that are only updated only by using setAttribute, like ‘value’ attribute.

For example:

<input id=”lol” value=”no”>

Let inp= document.getElementById(“lol”);

Inp.setAttribute(‘id’,’yas’);

Inp.id //’yas’

Inp.id=’nas’

Inp.id //nas

Inp.setAttribute(‘value’,’naa’);

Inp.value //naa

Inp.value=’k’

Inp.value //naa.

Inp.getAttribute(‘value’); //’naa’

• DOM properties are typed, i.e not all attributes return a string. Attributes retrieved using <elem>.<attribute> return their value as it should be. However, <elem>.getAttribute(<attr>) expects to return a string attribute and may not work correctly sometimes.

• Standard vs non Standard attributes: Standard attributes keep getting added to HTML, if we wish to include our own attribute we can use “data-<attr>”. The “data-” is reserved for programmers to use as they see fit. To access such an attribute, we use <elem>.<dataset>.<attr>. These attributes are also visible to the DOM.

• Element creation: We can create an element and then insert it into the document using DOM.

Syntax:

Let <varname>=document.createElement(‘<tagname>’); //creates a element.

Let<varname>=document.createTextNode(‘<string>’);// creates a text node.

Then after modifying this variable, we can insert it into the Document using

<node>.append(<node or string>): Inside the node but at the end.

<node>.prepend(<node or string>): Inside and at beginning.

<node>.before(<node or string>): Outside and before.

<node>.after(<node or string>): Outside and after.

<node>.replaceWith(<node or string>); Replaces this node.

These methods accept string which they use as pure text and don’t parse them. They also accept multiple nodes to insert.

To insert a string and allow browser to parse it, we use <node>.insertAdjacentHTML(<where>,<string / node>);

Where ‘where’ can be:

“beforebegin”: Outside and before

“afterbegin”: Inside and at start

“beforeend”: Inside and at the end

“afterend”: Outside and at the end.

The above methods can insert outside a node as well as inside, using old DOM methods we can insert only inside and after or before a certain child element.

<node>.appendChild(<node>): Just like <node>.append.

<node>.insertBefore(<node>,<child node>): Insert a node before a sibling.

<node>.replaceChild(<node>,<child node>)

<node>.removeChild(<node>,<child node>)

The last 3 methods only work if the child node is a direct child.

These methods can also add multiple nodes if the nodes are in a DocumentFragment Object (is used to add multiple nodes to). A better option is to simply use an array and unpack it if it needs to be given to these methods.

• <node>.remove(): Removes the node from the Document and the DOM.

• <node>.cloneNode(<bool>): If true deep copies an element into a variable, else copies the element and it’s properties but not it’s child elements.

• document.write(<string>): Inserts and parses a string directly into webpage right where it is called. If the page is still loading, and the script is executed inside it then it inserts the dynamically generated content right there. If it is in the microtask queue then this method overwrites the entire document.

• <elem>.className and <elem>.classList: ClassName can return class name and write to it, but as class attribute in css can have multiple classes (separated by space), we can use classList which returns a collection with all class names. To add/remove classes we use:

<elem>.classList.add(<classname>);

<elem>.classList.remove(<classname>);

<elem>.classList.toggle(<classname>); // adds a class if it doesn’t exist else removes it.

There are also other methods like contains.

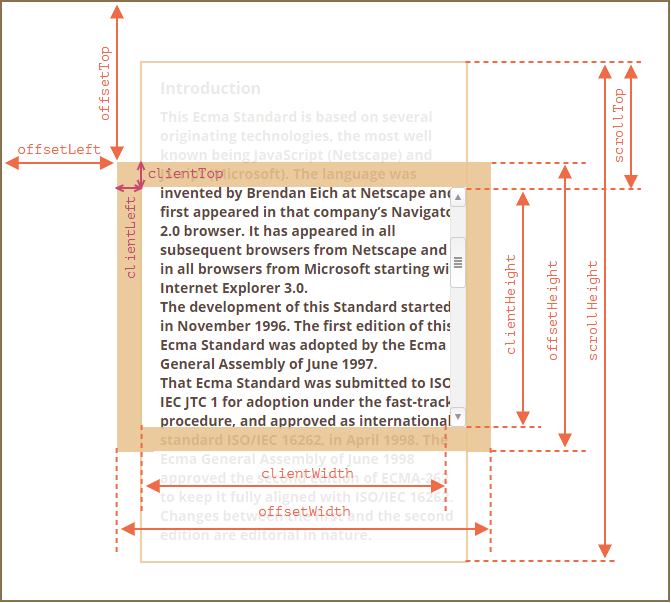
• <elem>.style: <elem>.style.cssText=<string>; to overwrite entire style for the element. We can use setAttribute for the style as well. We can’t change the style object as it is read only.

<elem>.style can’t read the css styles. It can only write to them.

getComputedStyle(<elem>,<optional pseudo element>); returns the resolved (not computed value) of the css style of an element. Computed css styles are the final values after all styles are applied whereas resolved css styles are the finally applied styles which use only absolute values.

For security reasons no JS method can modify or read the ‘:visited’ pseudo class.

• Scroll and other sizes: There are many properties that help with positioning elements and modifying how they look with scrolls.



The offsets are returned in absolute units, returns null for <body>,<html>, display:none and position:fixed elements.

<elem>.offsetParent : Returns the node that is directly above the <elem> in positioning, i.e. the closest node which affects the position of this <elem>.

<elem>.offsetLeft and top give the respective offsets.

And so on.

Document.documentElement.clientWidth/height to get the max available size for the html document for a client. For safari, document.body.clientWidth/height is better.

We can also use innerHeight and innerWidth, however they return the full document’s/scroll’s height and width. clientWidth and clientHeight return the available space minus full scroll width or height.

However, even clientWidth/Height may not return full window width or height due to inconsistencies in the browser, so we take the max between clientWidth/height, scrollWidth/height and offsetWidth/height.

These properties may work incorrectly if the HTML document doesn’t have a DOCTYPE tag at the start.

• Scroll State: window.scrollX / window.pageXOffset return the current horizontal axis scroll position while window.scrollY / window.pageYOffset return the vertical axis scroll position.

• Scroll Manipulation: Methods to let JS scroll a page/document.

<elem>.scrollTop/scrollLeft can be used to scroll an element.

Window.scrollBy(x,y): From the current position add x and y and scroll to that position.

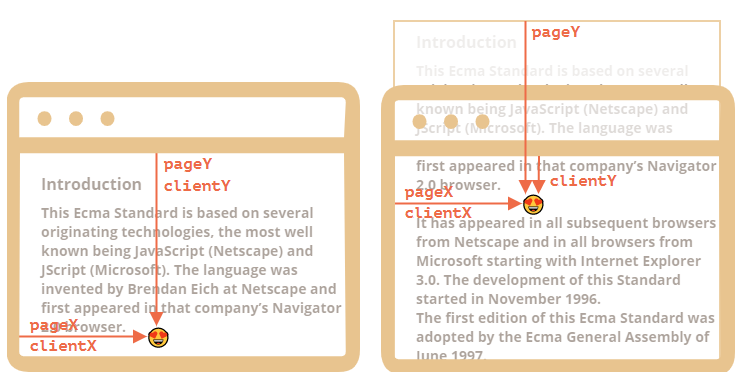
Window.scrollTo(x,y): Scroll to x,y absolute position.

<elem>.scrollIntoView(top=true): Scrolls the page to the given element, true and none align the top of the element to the top of the window, false aligns it to the bottom.

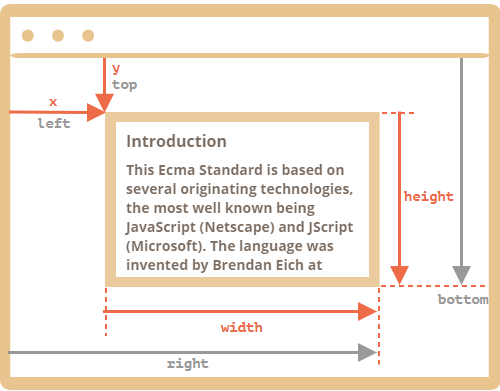
<elem>.style.overflow=”hidden”: freezes the scroll at current position, null or otherwise allows it. However, ‘hidden’ removes the scrollbar entirely so that space is used by the document, use padding to avoid this issue.

It is better to use these to get various sizes and lengths than using getComputedStyle(), as the computed style may not take scrollHeight into account, or may return ‘auto’ for sizes which isn’t what we need or due to a different box-sizing the returned size may be all too different than the actual size.

• Coordinates: There are 2 types of coordinate system followed in js. Relative to window (assume clientX/Y) which is like ‘position: fixed’ in css, and relative to document (pageX/Y) which is like ‘position: absolute’ in css.



We can use <elem>.getBoundingRectClient() to get an element’s relative and absolute coordinates. This method returns an Object of DOMRect class.



Top/bottom may not always be equal to x,y since the rectangle’s starting point could be in the bottom right and then it grows (negative coordinate).

• events: In HTML, almost all nodes generate an event for one thing or the other. Event handlers are the functions that are called when the event is triggered. These handlers are assigned to attributes that are called automatically when their corresponding event triggers. Some event handlers are:

<elem>.onclick=<func>; // for onclick on an element. (without the () since we aren’t calling it here)

Syntax in html: <htmltag <eventHandlerAttribute>=”<func>”>

The eventHandlerAttribute is used to assign handler to an attribute which will call the method when the event is triggered, like ‘onclick’ is an attribute which will call it’s handlers when ‘click’ event is triggered.

We can call ‘this’ in an event handler if we pass only a function body. ‘this’ is the reference to the <elem>.

We can’t use <elem>.setAttribute() on handlers directly (like providing function(){…} in it) since arguments are converted to strings in setAttribute(). We can however pass function names (like sayHi()).

Essentially there are these ways of providing handlers for events:

1. <elem <eventHandlerAttribute>=”<function body>”>> in the html tag itself.
2. <elem>.<eventHandlerAttribute>=<func>; in script
3. <elem>.addEventListener(“<eventname>”,<handler>,<optional options>)
4. <elem>.setAttribute(“<eventHandlerAttribute >”,”<func>()”);
5. <elem>.setAttribute(“<eventHandlerAttribute >”,”<funcbody>”); works

<elem>.<eventHandlerAttribute>= <funcbody> in script doesn’t work because browsers assign a “function(event){<funcbody>}” to the handlers with only <funcbody> in html, however using setters that optimization is never done.

• <elem>.addEventListener(“<eventname>”,<handler>,<optional options>)/<elem>.removeEventListener(“<eventname>”,<handler>,<optional options>): We can add/remove multiple event handlers to/from events of an element using these methods. All handlers are called when the event triggers and it is preferred to use these methods. To remove a handler the exact same function reference must be passed (not anonymous functions).

We can pass Objects/classes as handlers as well. In that case, the “handleEvent(event){…}” method defined in these Objects are called.

Optional option takes an Object with these properties:

Once: <bool> , if true removes the listener after 1 event trigger.

Capture: <bool>, if true allows the event handler to run in the capture phase. False, which is default allows event handler in bubbling phase.

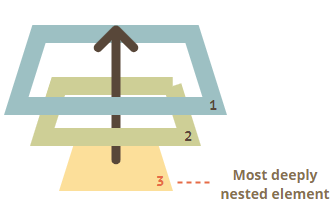
Passive: <bool>, if true tells the browser that the handler may remove default browser behaviour. Useful because if this is true browser will simply not worry about a handler stopping default behaviour and hence doesn’t need to check the handlers to stop the flow, if it’s false it has to check the handlers.

If an element has multiple listeners for the same event then they are triggered in the order they were inserted.

• event Object: An event object is always passed to handlers. It has methods like type (click/keypress etc.), currentTarget (this==<elem>) and clientX/clientY for coordinates of the event trigger.

• Bubbling: Almost all events in a document ‘bubble’ (except for events like focus event), i.e if an event is triggered on an element, then the same event is triggered for all the parents in a sequential order which is nested element->parent element->grand parent element and so on till there is no parent (usually they reach till html tag but sometimes window too).

However, for elements don’t bubble to their sibling elements. Meaning, if 1 element shadows another element and is a sibling to another then the shadowed element will never have it’s events reach it.

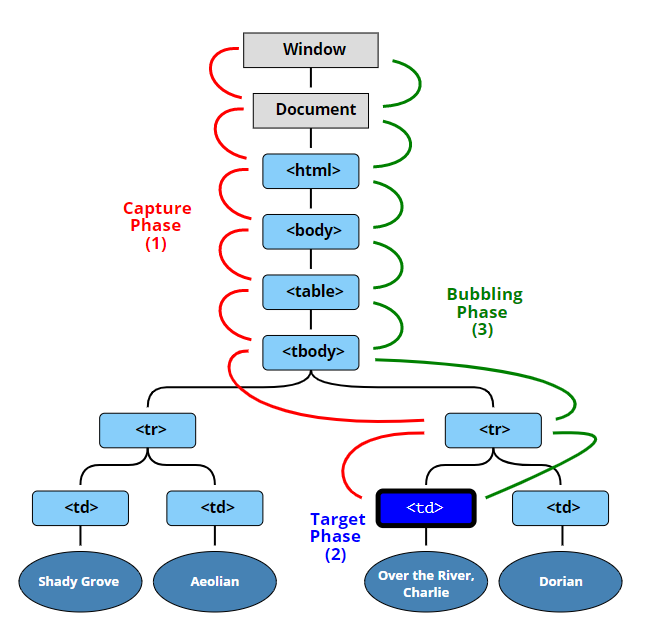


<event>.target: returns the <elem> at which the original event occurred. currentTarget will always be the <elem> that has the running handler which target will always be the <elem> that had it’s handler triggered.

To stop bubbling at any point, <event>.stopPropagation(); is used. However, the event handlers on the current elem would still work. To prevent even that we can use <event>.stopImmediatePropagation();

• Capturing:

There are 3 phases for an event, first is capture phase in which an event goes down the element tree looking for the target element which has to receive it, after finding the target element it executes the handlers there in target phase and lastly the event is propagated back up in the bubbling phase.



<elem>.addEventListener(“<eventname>”,<handler>,true); or <elem>.addEventListener(“<eventname>”,<handler>,{capture: true});  
are the same thing. Allows the event handler to be executed in the capture phase. To remove such a listener, we need to pass the same options.

• Event delegation: Through the use of capturing and bubbling we can do event delegation, i.e., assigning a single handler for multiple elements.

For example:

script

Class SomeStuff{

Constructor(elem){

This.\_elem=elem;

Elem.onclick=this.onClick.bind(this); //binds the ‘this’ to the onclick and returns that function to elem onclick.

}

<some Attribute>(){…}

onClick(event){

if(event.type==’click’ && event.target.<some attribute>==<some value>)

{

This[event.target.<some attribute>]();

}

}

};

New SomeStuff(<elem>);

/script

This method can simply be attached to a parent element and through the use of event.target we will change the handler’s behaviour.

• Default Browser Events: By default, browser adds events handlers to elements like <a>, i.e to open new window and a link etc. We can prevent default behaviour by returning false or calling <event>.preventDefault(); Some events automatically ‘flow’ into other events, i.e., they trigger them as well. If any event handler cancels default action anywhere, the chain is broken there.

• <event>.preventDefault() vs <event>.stopPropagation(): It is wasteful to stop bubbling, as it stops the calling of any and all events up in the tree. If we have a handler than manages something else in the tree then that never gets executed, but we can use preventDefault along with <event>.defaultPrevented. If preventDefault is called then defaultPrevented is true, this way we can atleast allow the handlers to be called in bubbling. This is akin to state management where we are using preventDefault as a toggle.

• Custom Events: We can create custom events using the Event class.

Syntax:

Let <varname>=new Event(<type>,<optional options>);

This is the base Event type, there are many other classes that extend Event and allow better properties like MouseEvent, UIEvent etc.

Type: event type, like click/mouse press etc. or custom event

Options: Object with 2 properties,   
 bubbles: <bool>, if true event bubbles.  
 cancellable: <bool>, if true then default action may be prevented.

To dispatch a custom event (i.e. manually invoke an event)we use, <elem>.dispatchEvent(<event>);

<event>.isTrusted; is a bool that returns true for all events that are generated by browser, false for events that are script-generated.

Custom events can only have handlers inserted through addEventListener.

For example:

Script

Let elem=document.getElementById(someID);

Elem.addEventListener(“yo”,function(event) {…});

If(someCondition)

{

Elem.dispatchEvent(new Event(“yo”));

}

/script

Thus, we have created our own event and a trigger for it. We also dispatch it manually.

The CustomEvent(<type>,<optional options>); is preferred over Event since it conveys that we are creating a custom event and it also allows a ‘detail’ property in the options. ‘detail’ takes an Object and then passes it to the handlers.

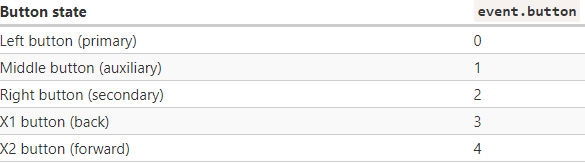
Custom Events don’t have default actions however if cancellable is true and if any subsequent event listeners call <event>.preventDefault(); then <elem>.dispatch(<event>) returns false.

Normal user-generated events are synchronous and queued, however custom events dispatched manually are executed then and there. However, if a setTimeout is used then the custom event is dispatched after the synchronous events.

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

A single interaction may send out multiple events, browsers process (call their target event handlers) events in the order they were executed.

• Mouse Button: All mouse button events pass a button property with an int value to the event in the handlers.



<event>.which == <int> is a deprecated way of checking button id.

Mouse events are useful for many things, like ondblclick which is a handler for double click event, or mousedown for mouse button press. And if they return false then those events don’t occur, preventing a user from selecting text.

mouseover/mouseOut are attributes for handling the events that are triggered when mouse hovers over an element. Both of these events have <event>.target and <event>.relatedTarget, for mouseover event, target is the current element mouse is on and relatedTarget is the last element that the mouse was on. For mouseover, target is the old element and relatedTarget is the element mouse left to. relatedTarget can be null since mouse can come from outside window. The trigger isn’t instantaneous, browsers have an interval between triggers so if mouse is too fast then it may not trigger the event at all. mouseover will always lead to a mouseout event, and they are triggered per element and not per element tree. So, a mouseover event for an element will trigger mouseout if the mouse leaves, even if it is to a child element. These events also bubble.

Mouseenter/mouseleave are just like mouseover/mouseout but they are triggered per element tree and do not bubble.

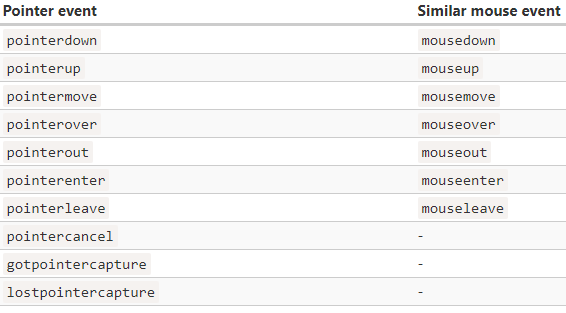
• Oncopy is an event handler than handles copy events, i.e., if we copy something from an element then this handler is called. If it returns false then that behaviour is prevented.

• Drag ’n’ Drop: We can implement our own Drag ‘n’ drop behaviour, by default it takes the url of buttons/images etc. and allows us to move it around. Ondragstart and ondragend are the 2 eventHandlerAttributes that define this behaviour.

To create a custom drag implementation, we can provide handlers for mousedown, mouseup and mousemove events. That the element has the highest z index, is in the body directly, follows the mouse from mousedown on it using the x and y of the mouse provided by <event> (of document) till mouseup is triggered when it stops following.

• Pointer events: Mouse events are useful only with mouses, but for touch/pen etc. we need to use these.

Most pointer events are named similar to mouse events.



Pointer events also bring a few more properties along with the properties available on mouse events.

pointerType: is a string with “mouse”, ”pen” or “touch”.

pointerId: UID for any pointer. This is how we can differentiate between multiple pointers if multi-touch is performed.

isPrimary: bool, true if the current event handler has the pointer that started triggered the event.

Width: width of area contacted by a pointer, unsupported devices have it always at 1.

Height: height at which the pointer is. Unsupported=1.

Pressure: Pressure of pointer touch. Unsupported: 0.5 for pressed and 0 for not pressed.

tangentialPressure: Normalized tangential pressure.

tiltX, tiltY, twist: Pen specific properties.

We can use pointerdown/etc. instead of mousedown/etc. wherever we want and it functions similar to mousedown but also supports mouse events.

pointercancel: This event may be triggered by the browser when there’s a change in screen orientation or some other change in the window. When it is called, the pointer events are all stopped.   
So, for drag ‘n’ drop this would mean if we are dragging an image and we click it, browser ‘hijacks’ our drag behaviour for an element in favour of it’s own image dragging behaviour by firing a pointercancel event. To prevent this we use touch-action= none attribute for our element.

• Pointer capturing: We can capture pointers, i.e., assign a pointerId to an element and whenever this pointerId fires an event anywhere, this element’s handler will handle the pointer event.

Syntax: <elem>.setPointerCapture(<pointerId>);

After this, if an event with this pointerId every triggers, this elem will receive the <event> and handle it. This binding is removed automatically on pointerup or pointercancel event is triggered, when <elem> is removed from the document or if <elem>.releasePointerCapture(<pointerId>); is called on a captured pointerId.

setPointerCapture fires getpointercapture event and lostpointercapture is fired on releasePointerCapture/ when the binding is automatically removed.

• Keyboard Events: Keyboard events are usually named what they do. <event>.<keyName>; is a bool that returns true for whatever key is requested.

Usually these are modifiers to mouse events, that is they are pressed along with a mouse click (as they will be triggered on mouse click since the event handler is attached to onclick).

For example:

<event>.shiftKey; will return true if shift + was pressed on an element.

There’s altKey, ctrlKey for ctrl windows and metaKey for ctrl in mac.

There’s <event>.key which returns which key was pressed. (like pressing ‘z’ puts ‘z’ in it while shift+’z’ puts ‘Z’);

<event>.code returns keycode, which is same for same keys despite any modifiers like shift, ‘KeyZ’ is for z and ‘Z’ and so on.

Letter keys: “Key<letter>”

Digits: “Digit<digit>”

Special Keys: “<name>” like “Enter”, “Backspace” etc. For right shift it’s “ShiftRight” and “Shiftleft” for left shift.

Event.code has the same code for multiple layouts, it’s Key<key in qwerty>. Even in qwertz/ etc. layouts, the key code will be what it is in qwerty at that physical location on the keyboard. Event.key however has the key that was pressed and hence changes with layouts.

• Auto-Repeat: If a key is held for a long time, it will auto-repeat. It will fire keydown events in small intervals and then one keyup. For repeating events such as these, <event>.repeat is set to true.

• Scroll Event: scroll event is triggered on scroll. Onscroll is the eventHandlerAttribute.

• Forms: All forms in a document are available under document.forms, we can access any single form through name or index as this is a “named collection” Object, i.e., it is named and ordered at the same time.

For example:  
<form name =”on”>

<input name= “yo” value=”1”>

</form>

We can get this form using document.forms.on; or document.forms[0];

We can then call <form>.elements.<element name such as “yo” in this case>; to get elements from the form. If there are multiple elements with the same name they are grouped and form.elements.<elem name>; returns a collection, we can access elements using <elem>[<index>];

All elements inside form, regardless of nesting level are available in form.elements.<elem name>. Shorter notation: form.<elem name> or form[<elem name>];

All form elements can access the form with <elem>.form;

We use, input, textarea, select and option with forms usually.

<Input elem>.value; returns the value in the input box

<Input elem>.checked; returns the value in the input box in bool (iff. if it has a checkbox)

<textarea elem>.value; returns the value in the textarea box.

<select>.options: The collection of <option> sub elements.

<select>.value: Currently selected <option>’s value.

<select>.selectedIndex: Currently selected <option>’s index.

Just like we can use these 2 methods to get the value of the select box, we can put value in them to select that option.

If the multiple attribute of the select tag is defined then multiple values can be selected.

To insert an <option> in a <select>, we use:

Let <varname>= new Option(<text>,<value>,<defaultSelected>,<selected>);

Text: option’s text

Value: option’s value

defaultSelected: bool, if true then <selected> HTML attribute is created. We can getAttribute(‘selected’) on the <option> to get its attribute.

selected: bool, if true then it is selected by default

Then we simply add it using <elem>.append/etc.().

• Focus and blur: These are 2 events that are triggered on elements like <input>, focus is triggered when it is clicked/tabbed or focused on while blur is an event triggered as soon as focus leaves the element. We can use onfocus and onblur on fields of a form for any task. These events do not bubble but do run in capturing phase.

<elem>.focus(); will focus the given element.

<elem>.blur() will remove the focus from a given element.

These events and eventHandlerAttributes are only supported on elements a user can interact with.

Tab-based switching: All elements can be focusable with mouse or pointer if they have tabIndex attribute set to a value. If we press tab or if tab brings focus onto an element with tabIndex, it has this switching order:

1. Elements with tabIndex>=1
2. Elements with tabIndex null but are focusable
3. Elements with tabIndex=0

Elements which have same tabIndex or don’t have a tabIndex but are focusable get focused by tab in the order they are there in the html tree.

If we assign ‘-1’ tabIndex to an element then that element is focusable by mouse and through script only, tab never switches to an element with negative index.

FocusIn/FocusOut are just like focus and blur events but they bubble and they must be inserted through addEventListener().

• Change: change is an event triggered when an element is finished being modified, for <input> it is when a user leaves the element. For select it is when an option is selected. Onchange is the eventHandlerAttribute for it.

• input: This is an event triggered when an element is modified, unlike change it doesn’t trigger after the modification is complete but it gets triggered as the element is modified. Oninput eventHandlerAttribute.

• cut, copy and paste: Events are used to access the clipboard, by default they work for every element and allows the 3 events as per the element but if we provide our own handler then we can modify that behavior. We can only modify the behavior but not see the data itself on the clipboard, if the handler for any of the events returns false or has preventDefault then that action doesn’t occur. We cannot dispatchEvent on these events either. Oncut, oncopy and onpaste are the eventHandlerAttributes.

• submit: This event is triggered when the last option of a form receives Enter or the submit option is selected. Onsubmit EventHandlerAttribute. Submit event also triggers click event for the submit option. We can manually call submit using <form>.submit();

• DOMContentLoaded, load, unload/beforeunload: Events which are triggered when an html page is loaded.

DOMContentLoaded event is triggered when HTML doc is loaded. This means, it is triggered after all the scripts have been initialized/executed including document.write. However, this event occurs before the microtask queue has been started, and also before document.createElement() methods. Normally this event isn’t dependent on stylesheets, but if there is a script tag after the link to a stylesheet or style itself then that holds the event from being fired until that stylesheet is loaded. Autofills on elements that browsers implement wait until this event is fired. We use document.addEventListeners() for this event.

Load event is triggered when HTML doc + assets such as images etc. are loaded.

beforeunload: This event is triggered when the document has requested unloading, i.e., browser is closing the page. If the handler for it returns false or a non-empty string then an alert is thrown on the page if user requests to close it.

Unload: this event is triggered when the browser has initiated unloading. Navigator.sendBeacon(<url>,<dataObject>) can used with it, this method sends an Object with <=64kb size to a url in the background even after the page has unloaded, useful for analytics.

Error: This event is triggered if a script faced an issue in loading. Although it doesn’t store the error code. There are 2 types of error events, one for script and the other is for loading error. For loading error, i.e., if the browser failed to load the script on an element that has src tag we can define <elem>.onerror=function(){…}. The event doesn’t store http error code or any detail, just that it failed.

If the loading is successful but there is an error in the retrieved script then that error could be tracked using window.onerror=function(message,url,line,col,errorObj){…};

If the script comes from the same domain, CORS isn’t enforced and the onerror works normally but for scripts from other domains CORS is enforced and we need to manually request access using crossorigin attribute on the element that has the src=”<external domain script>” attribute. The crossorigin attribute must have “anonymous” or “use-credentials” value, to which the server (where the script is coming from) must respond with “Access-Control-Allow-Origin” header. And “Access-Control-Allow-Credentials: true” as well for “use-credentials”.

For load, unload and beforeunload we can use window.onload/onunload/onbeforeunload. For onerror the syntax is

Load and error events are triggered by any element that has an ‘src’ attribute.

Error event wi

• readyState: document.readyState stores the current state of the document, it has 3 possible values:

“loading”: doc is loading

“interactive”: doc is loaded and now it’s parsing scripts etc.

“complete”: doc is loaded, scripts are parsed, images loaded etc.

readystatechange is the event for it, this event is fired for whenever readyState changes. We give it a listener using document.addEventListener().

• Mutation Observer: This is an observer on any DOM node, if any change occurs on the node then this observer runs a callback function and passes it the data that was changed.

Syntax:

Let <varname>=new MutationObserver(<function>);

<observer>.observe(<node>,<Config Object>);

Now if a change is observed in the node, the callback will execute.

The config has properties for defining what changes to observe, what data to gather etc.

childList: <bool>, observe changes in the direct children of the <node>.

subtree: <bool>, observe changes in all the descendants of <node>.

attributes: <bool>, observe changes in the attributes of <node>.

attributeFilter: <<array of attribute names>, observe changes for these attributes of the <node>.

attributeOldValue: <bool>, if attributes property is true then pass both old and new attribute data to callback

characterData: <bool>, observe changes in <node>.data (the text data).

characterOldValue: <bool>, if characterData property is true then pass both old and new character data to callback.

The callback function accepts 2 parameters, list of MutationRecord objects and the <observer> object itself.

MutationRecord Object has these properties:

Type: could be “attributes”, ”characterData”, “childList”. Specifies which change is caught in the observer.

Target: where the change occurred, <elem> for attributes and childList and <text node> for characterData.

addedNodes/removedNodes: If <node>s were removed or added then they are here.

previousSibling/nextSibling: If <nodes>s were removed or added then their previous/next sibling node.

attributeName/attributeNamespace: name/namespace of changed attribute.

oldValue: if either of oldValue bools were true in the config object then their old value.

<observer>.disconnect(): to disconnect, however, if observer couldn’t execute callbacks on all the mutations then we can call <observer>.takeRecords(); before this method to safely handle all the MutationRecords.

• Range: Using this method we can retrieve any range of text from a document (DOM basically).

Syntax:

Let <varname>= new Range();

<range>.setStart(<node>,<offset>); Inclusive of the first element/character.

<range>.setEnd(<node>,<offset>); Exclusive of the last element/character.

If node is a text node then offset is the index offset from the start of the string.

If node is an element then offset is the child element offset from the start of the children list.

For example:

<div id="lmao">

    truck

    <i>

      potato

    </i>

    <b>

      tomato

      <i>

        aloo

      </i>

    </b>

    <i>

      car

    </i>

  </div>

  <script>

    let range = new Range();

    range.setStart(document.getElementById("lmao"), 0)

    range.setEnd(document.getElementById("lmao"), 3)

    console.log(range.toString());//prints truck potato

  </script>

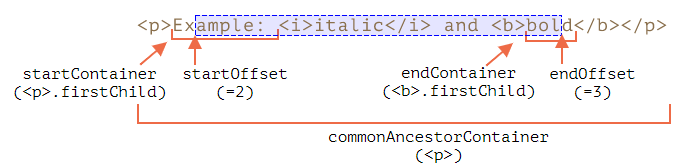
toString() returns the text elements (strings) inside the range The format is preserved as it appears in the html document, meaning the spaces, indents and format are the same.

In this case, the div element, the I element and the b element are captured.

If we selected lmao.firstChild (gets the first node, which in this case is just “” i.e., the first child in this case has no text) we would get the text node.

The end element/node must be the same type as the start element/node but it could be any element/node as long as it comes after the start element/node in the document.

Properties and their values of range object:



There’s also collapsed:<bool>, true if the range has no content inside it.

Range Selection methods: Apart for the ones given above and a few more like those, (like setStartBefore/After), we have:

selectNode(<node>): set start and end on whole <node>.

selectNodeContents(<node>): set start and end on whole contents of <node>

<range>.cloneRange(): returns a range with the same start and end applied.

Range Editing Methods: We can use these methods to modify the document itself.

<range>.deleteContents(); removes the content from the document.

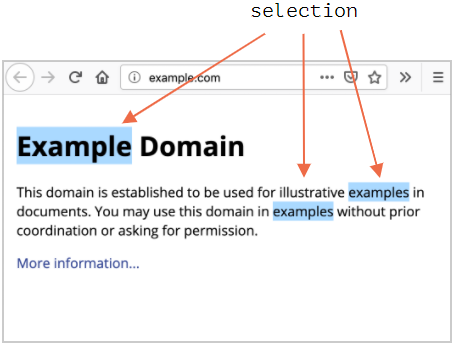
<range>.extractContents(); returns DocumentFragment object with the contents and removes it from the document.

<range>.cloneContents(); same as extractContents but doesn’t remove it.

<range>.insertNode(<node>); insert node at beginning.

<range>.surroundContents(<node>); wraps the <node> around the range, given the range has all the elements with a proper beginning and ending tag.

• Selection: Ranges select data from an html document manually and don’t show anything on the page itself, whereas selection can be made by the user and is visible.



We can get the user selection using, window/document.getSelection()

In firefox, we can select multiple words/elements using ctrl+click but not in other browsers where only 1 selection is possible.

Properties of a selection Object:

<selection>.anchorNode: <node> where the selection starts.

<selection>.anchorOffset: offset in the anchorNode from where selection starts.

<selection>.focusNode: <node> where selection ends.

<selection>.focusOffset

<selection>.isCollapsed: <bool>, true if nothing selected or selection isn’t made.

<selection>.rangeCount: count of all ranges.

It is not necessary for anchorNode to be before focusNode in selection.

<elem>.onselectstart: EventHandlerAttribute for an event that triggers when selection starts. For handlers that return false or have <elem>.preventDefault() selection doesn’t start. Though it can come from outside the element anyway.

Document.onselectionchange: EventHandlerAttribute for an event that triggers whenever selection is started or changes.

Selection methods:

<selection>.getRangeAt(<index>): returns a range object, where index is the number of the selection that was made, in firefox it can be more than 0 (index, the later the index the later it was selected). Everywhere else since we can only select 1 element, 0 is the index we use. We can also use this method to get a range for the selection, then clone it. Preserving the element in memory even if it is removed from DOM.

<selection>.addRange(<range>); adds given range to the selection. Although only firefox can take more than one range in the selection. Yes, it selects a given range on the document and more than 1 range if it is firefox.

<selection>.removeRange(<range>); The opposite.

<selection>.removeAllRanges(<range>);

<selection>.empty(); same as above.

<selection>.collapse(<node>,<offset>); sets the selection to be at the given node’s start (with <offset> offset) and it’s end.

<selection>.setPosition(<node>,<offset>); same as above.

<selection>.collapseToStart(): replace selection with an empty range with both start and end of range as 0.

<selection>.collapseToEnd(): The opposite.

<selection>.extend(<node>,<offset>); extends

<selection>.setBaseAndExtent(<anchorNode>,<anchorOffset>,<focusNode>,<focusOffset>): replace this selection with the given selection properties.

<selection>.selectAllChildren(<node>): selects all children of this <node>.

<selection>.deleteFromDocument(); deletes

<selection>.containsNode(<node>,<allowPartialContainment=false>)

Selection in forms: <input> and <textarea> provide their own methods for selection.

<input>.selectionStart: Writable position of selection start.

<input>.selectionEnd

<input>.selectionDirection: “forward”, “backward” or “none” (if selection is made by a double click)

<input>.onselect: eventHandlerAttribute

<input>.select(): selects everything in the text.

<input>.setSelectionRange(<start>,<end>,<optional direction>): change selection to this given selection range.

<input>.setRangeText(<replacement string>,<optional start>, <optional end>, <optional selection mode>): replace text of selection. Overrides start/end position if start/end are provided.

selectionMode: is a string that defines what will be selected after the change

“select”: new text is selected

“start”: cursor is placed at the start of new text

“end”: cursor is placed at the end of new text

“preserve”: default, tries to preserve selection of pre-change.

If we set the <textarea/input>’s selectionStart and selectionEnd to the same value then cursor is placed there.