**//Select an element**

**var** membersDiv = document.querySelector("#members");

then you can use-

membersDiv.innerHTML

or

membersDiv.value

e.t.c

**//WebService call**

**// API for getting info about an artist/band by name**

**var** url = encodeURI("https://wasabi.i3s.unice.fr/api/v1/artist/name/Metallica");

fetch(url)

.then(**function**(response) {

// response is a json string,

// convert it to a pure JavaScript object

**return** response.json();

})

.then(**function**(band) {

membersDiv.innerHTML += "<h2>Current and old members of " +band.name + "</h2>"

displayMembers(band.members);

})

.**catch**(**function**(error) {

console.log('Error during fetch: ' + error.message);

membersDiv.innerHTML += "<h2>No Results</h2>"

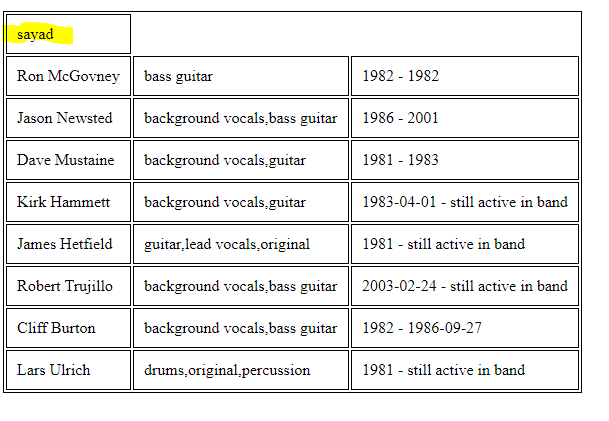
});

**//Create table insert Row and add cell and put value in it**

**var** table = document.createElement("table");

**var** row = table.insertRow();

**var** memberNameCell = row.insertCell();

memberNameCell.innerHTML = "sayad";

//Javascript function call-

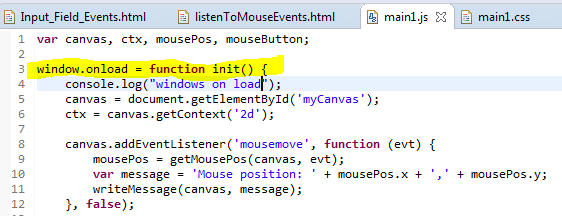
1. **On body load –**

<body onload="init();">

1. **On button click –**

<button onclick="addImageIntoBackground();">Add background image</button>

1. **On window.load inside javascript – this function gets call as soon as browser window loaded.**



//add key listeners to the window element

window.addEventListener('keydown', handleKeydown, **false**);

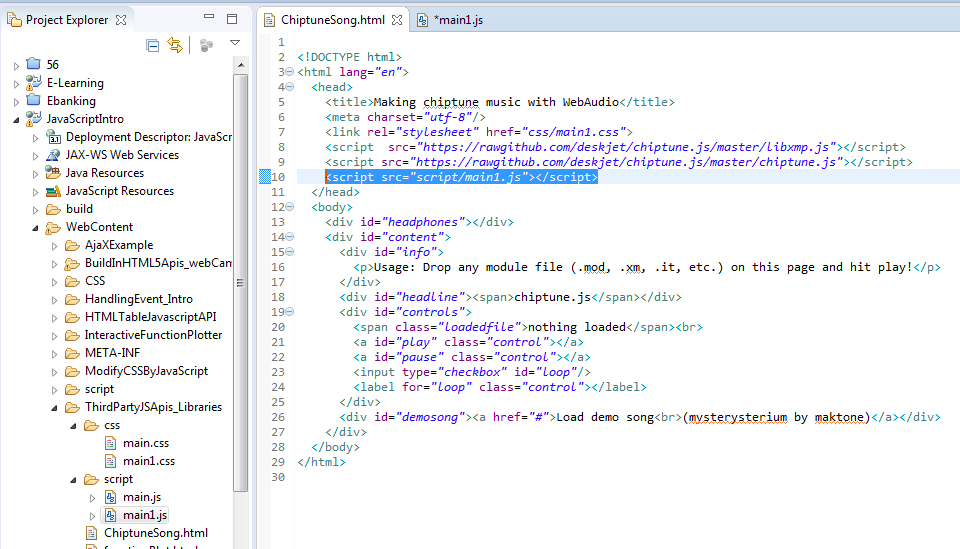
window.addEventListener('keyup', handleKeyup, **false**);

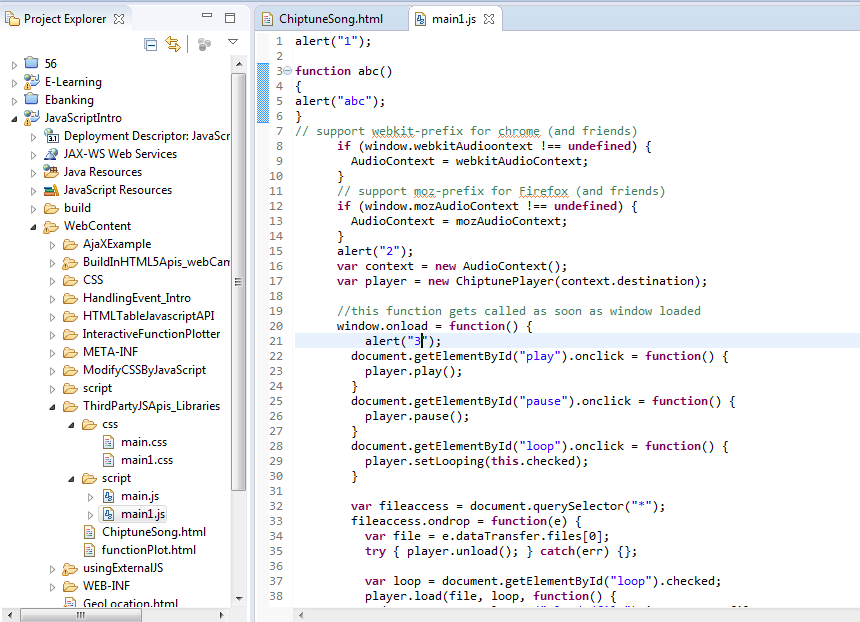
#### // listen to mouse events in an HTML5 canvas

The HTML5 canvas is useful for drawing and animating at 60 frames/second. 

//---------------------------Start ------------------------------------------

//Important things to know for including external script in your html-

Below example - 

We have included a script file(main1.css)- 

**When you include a script file , then whole script file is loaded sequentially .**

**When a script is loaded then it execute sequentially except a function, function only compiled when script was getting loaded. Function get executed only when it is called from an event or by calling et.c.**

**For example in above example we put alert to see the execution flow of main1.js script**

**First when its loaded in html head element then –**

**Alert 1**

**Alert 2**

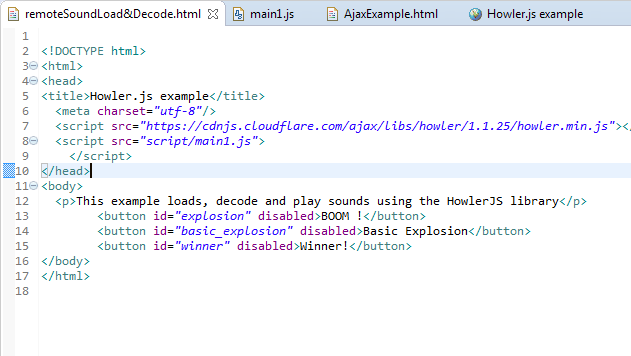
**Then alert 3 but only when window.onLoad event happened.**

**Alert abc never happened because function ‘abc’ never called from somewhere.**

**//--------------------------END-------------------------------------**

Another Example of external javascript –

HTML-



Javascript file- 

Function soundLoaded() called from inside an javascript object sound on its onLoad event.

**//**

**Depending on the context, generally strict equal (or strict not equal) is preferred.**

**Best practice for beginners: always use === or !== for comparisons.**

#### What is the difference between ****==**** and === in JavaScript?

##### Equal (==)

Returns true if the operands are strictly equal **with type conversion**.

##### Strict equal (===)

Returns true if the operands are strictly equal **with no type conversion**.

The triple-equals operator never does type coercion. It returns true if both operands reference the same object, or in the case of value types, have the same value.

Some examples :

1. 1 == 1 ;
2. //true
3. 1 == 2 ;
4. //false
5. /\* Here, the interpreter will try to convert the string ‘1’
6. into a number before doing the comparison \*/
8. 1 == '1';
9. //true :

12. //with strict equal, no conversion:
14. 1 === 1;
15. //true
16. 1 === '1';
17. //false

**Undefined and null values**

**Undefined**

undefined is returned when a variable has not been assigned:

1. var foo;
2. > foo
3. undefined
4. > typeof foo;
5. 'undefined'
6. >if (foo === undefined) {
7. console.log('The variable foo has no value and is undefined');
8. }
9. 'The variable foo has no value and is undefined'

The above example shows how we can test whether a variable has a value (*line 8* uses a conditional statement).

The keyword "**undefined**" is part of the JavaScript language, so you can assign the undefined value to a variable:

1. > var foo = undefined; // equivalent to var foo; without giving any value
2. undefined
4. > foo;
5. undefined

var foo; and var foo = undefined; are equivalent but we recommend that you use the first version to declare the variable (it is shorter, and that reduces the code).

If you try to access a variable that has not been declared before, a ReferenceError will be raised. But the typeof operator will return "undefined":

1. > bar;
2. ReferenceError
3. > typeof bar;
4. 'undefined'

#### Implicit conversions of non boolean values in expressions

Used with logical operators or within statements, **non-boolean values are implicitly converted to Booleans**

1. var b = "one";
2. !b;   false // implicit conversion of "one" to a boolean value
4. var b = "one"; // implicit conversion of "one" to a boolean value
5. !!b; //true

* **All the following values are evaluated as false :**
  + **false**
  + **undefined**
  + **null**
  + **0**
  + **NaN**
  + **the empty string ''**

**Everything else is evaluated as true!**

1. var boo = 'hello' && 'world';

boo equals 'world' because 'hello' is a string value that is evaluated as true.

1. var boo2 = (0/0) || 43.2 ;

 boo2 equals 43.2 because the expression 0/0 equals NaN, which is evaluated as false.

//

* **Specific case of NaN**

As we have already seen, JavaScript has some special values. One of them is NaN: “Not-a-Number”.

NaN has this special property :

1. NaN == NaN;
2. // false
4. NaN === NaN;
5. // false

**Nan is equal to nothing - not even to itself!**  But you do have a function to check the NaN value:  isNaN(expr)

* isNaN: returns true if the argument coerces to NaN, and otherwise returns false.

1. isNaN(NaN);
2. // true
4. isNaN(0/0);
5. // true
7. isNaN(12);
8. // false
10. isNaN('foo');
11. // true

//

* **Lazy evaluation or short-circuit evaluation:**

Logical expressions are evaluated from left to right. JavaScript uses a mechanism known as "short-circuit evaluation" to prevent the second, third, and nth conditions from being tested in certain cases:

* + false && something (an expression)  is always false, and the part to the right of && operator is not tested.
  + true || something (an expression) is evaluated to true, and the part to the right of the || operator is not tested.
* A statement closes with a semicolon, but we will see later that missing semicolons are automatically inserted (for readability reasons, we highly recommend systematically adding a semicolon at the end of all statements).

#### The if-then-else ternary operator

This ternary operator is a shortcut version of if...then...else.

Let's look at this code example:

1. var max;
2. var min = 2;
4. if (min < 10) {
5. max = min + 10;
6. } else {
7. max = min;
8. }

Explanation: You can replace this "if-then-else" statement with the ternary operator that uses a syntax with "?" and ":"

1. var max;
2. var min;
3. max = (min < 10)? min+10 : min;

Line 3 can be read as if (min < 10) then max = min+10, else max = min. The "then" part is after the "?" and the "else" part is after the ":" part.

This "short" version is not recommended except for very simple statements that involve a very obvious block of instructions for the "then" and the "else". Usually this syntax is much harder to read for beginners.

//

The **do-while statement executes the content of the loop once before checking the condition of the while**, whereas a **while statement will check the condition first before executing the content**.

//

##### The continue statement

The continue statement is used to stop the execution of a block and start the next iteration of the loop. The difference from the "break" statement is that the loop continues.

Break exit from the loop, it "breaks" the loop. Loop don’t continue after the break , it exit/break the loop.

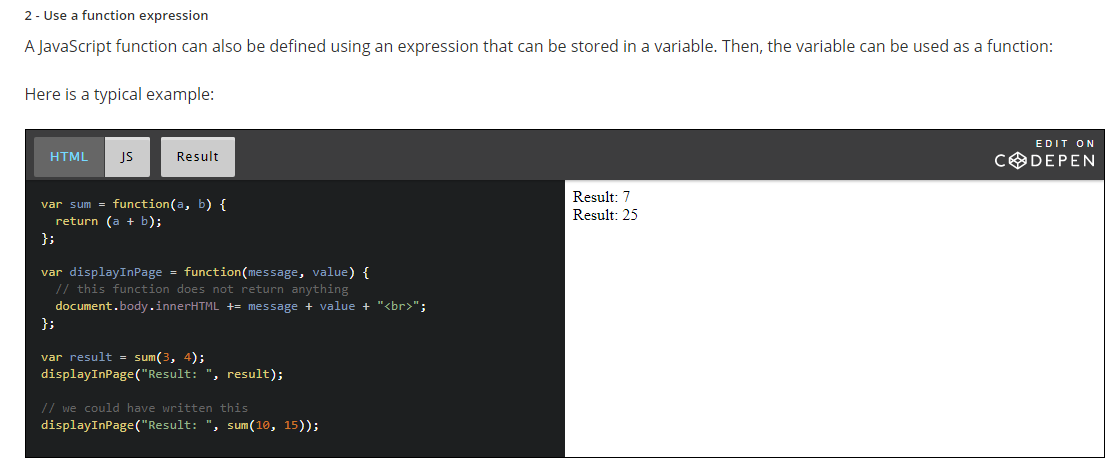
##### The break statement

The break statement is used to stop an iteration, a switch or a labelled statement.

//

JavaScript function Declaration –

1.Standard function declaration : 



We say that functions are "first class objects" which can be manipulated like any other object/value in JavaScript.

This means that functions can also be used as parameters to other functions. In this case they are called "callbacks".

**Callbacks**

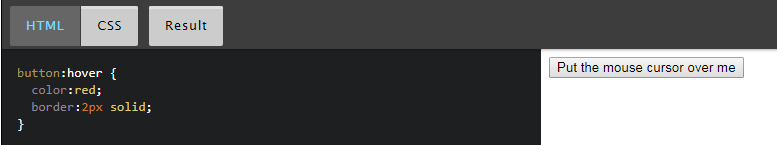
Indeed, as functions are first-class objects, we can pass a function as an argument, as a parameter to another function and later execute that passed-in function or even return it to be executed later. When we do this, we talk about *callback functions* in JavaScript: a function passed to another function, and executed inside the function we called.



//

Handling Event introduction-

Adding interactivity to a Web application can only be achieved with CSS, using the :hover pseudo CSS class, for instance. For example:



However, firing a specific action when the button is clicked, knowing which mouse button has been used, computing the (x, y) mouse pointer position in the button system coordinate, or executing more complex tasks can only be done through JavaScript.

With JavaScript, a button click, a move of the mouse, a resized window, and many other interactions create what are called "events".  The timing and order of events cannot be predicted in advance. We say that "event processing" is asynchronous. Web browsers detect events as they occur, and may pass them to JavaScript code. They do this by allowing you to register functions as event listeners, also called handlers or callbacks for specific events.

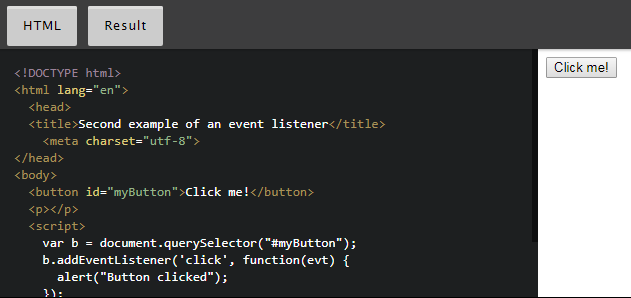
Each time an event occurs, the browser puts it in a "queue of events".

Then the browser looks at a list of "Event Listeners" and calls the ones that correspond to the type of event "they listen to".

**Adding an event listener to specific HTML elements**

Instead of listening to event on the whole document (using addEventListener is the same as using window.addEventListener), we can listen to specific DOM elements.

For example, here is how we can listen to clicks on a specific button (whereas clicks on the rest of the document will be ignored).



Every DOM object has an addEventListener method.

**Removing event listeners – removeEventListener**

**//**

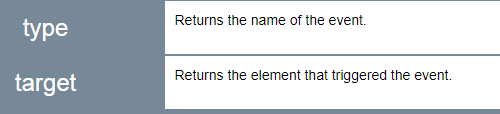
**The event object**

#### The event object is the only parameter passed to event listeners

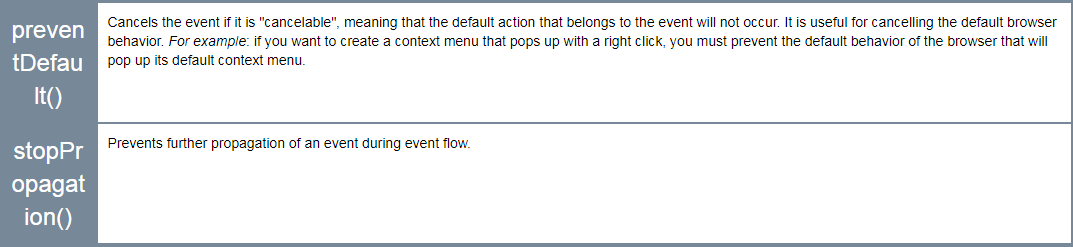
Each event listener has a single parameter that is  a "DOM event object". It has various properties and methods that can be very useful.

#### Reference table

The most useful common properties are:



The most useful common methods are:



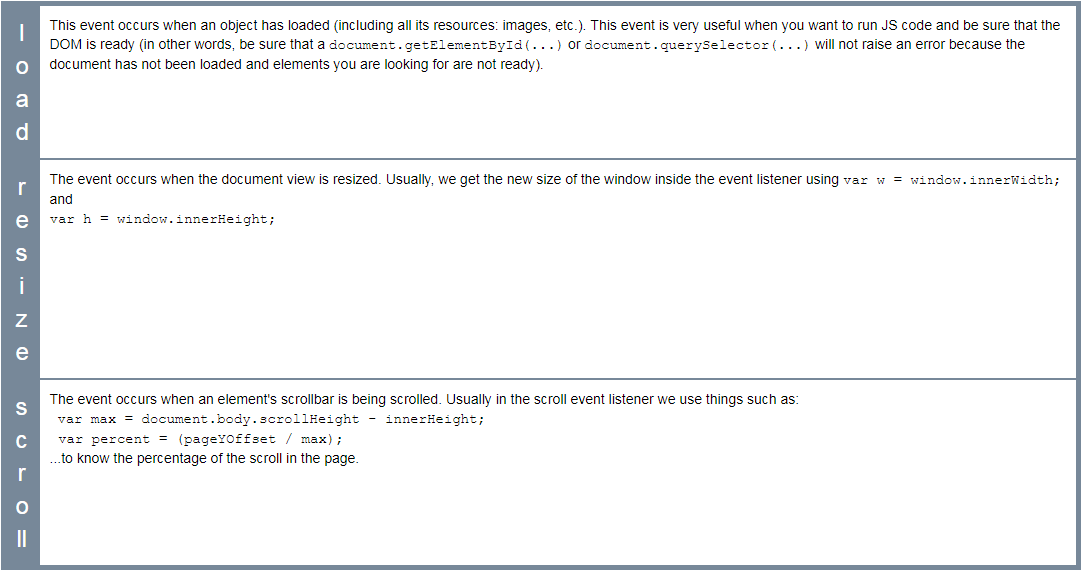
//

**Page lifecycle events**

These events detect when the page is loaded and when the DOM is ready.

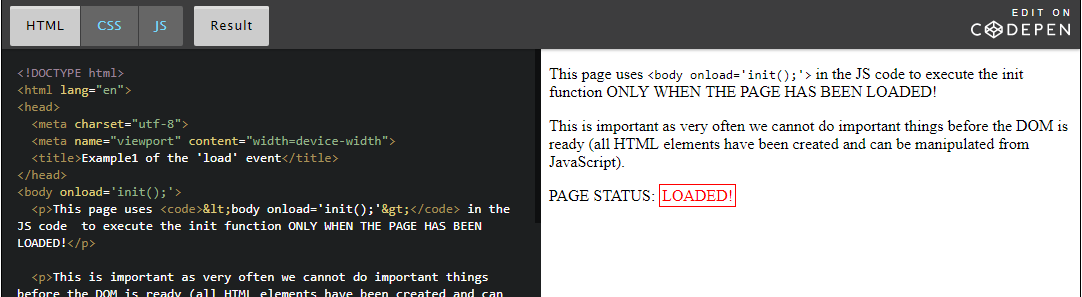
**Events related to the page lifecycle**

There are many other events related to the page life cycle. The most useful ones for an introduction course are shown below:



#### Example 1: wait until the page is loaded (when the DOM is ready) before doing something

This first variant that uses <body onload="init();">

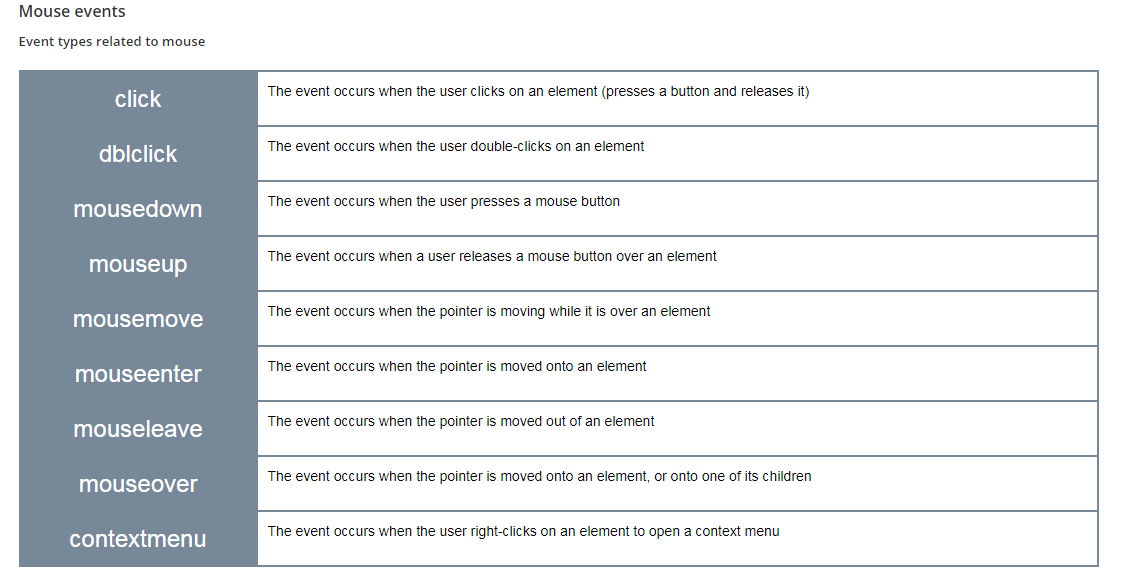


This second variant: using window.onload = init; in the JavaScript code...

This way is better than first as you can separate javascript from HTML 

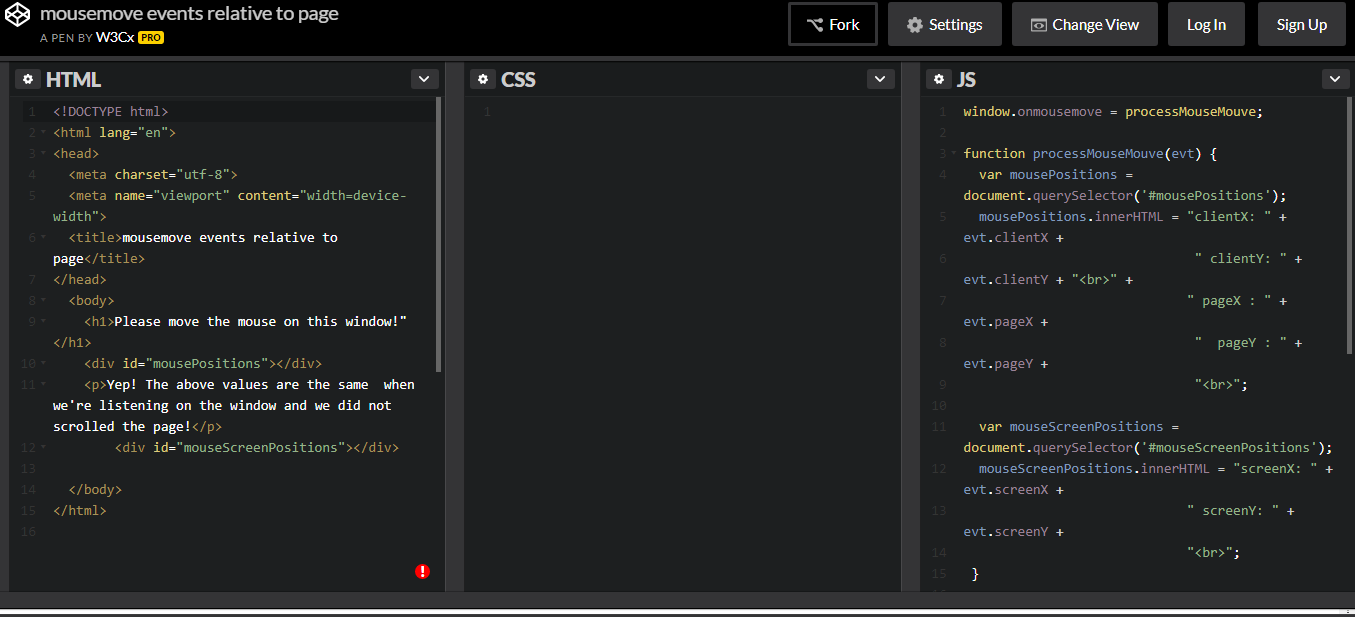
//

Mouse Events



##### MouseEvent properties

//

Some Mouse event and their properties - 

## Form and input field events

#### Forms

##### Events related to forms



**Reference tables**

### Quick summary of event management in JavaScript

#### HTML5 EVENTS

There is no input or output in JavaScript. We treat events caused by user actions as inputs, and we manipulate the DOM structure as output. Usually in a JavaScript application, we will get info such as the key strokes, the mouse button clicks and the mouse position, and we will refer to these variables when determining what action to perform.

In any case, the events are called DOM events, and we use the DOM APIs to create event handlers.

#### HOW TO LISTEN TO EVENTS

There are three ways to manage events in the DOM structure. You could attach an event inline in your HTML code like this:

##### **Method 1: declare an event handler in the HTML code**

1. <div id="someDiv" **onclick**="alert('clicked!')"> content of the div </div>

This method is very easy to use, but it is not the recommended way to handle events. Indeed, although it currently works, it is deprecated (will probably be abandoned in the future). Mixing 'visual layer' (HTML) and 'logic layer' (JavaScript) in one place is really bad practice and causes a host of problems during development.

##### **Method 2: attach an event handler to an HTML element in JavaScript**

1. document.getElementById('someDiv').**onclick**= function() {
2. alert('clicked!');
3. }

This method is fine, but  you will not be able to attach multiple listener functions. If you need to do this, use the version shown below.

##### **Method 3: register a callback to the event listener with the addEventListener method (preferred  method)**

1. document.getElementById('someDiv').**addEventListener**('click', function() {
2. alert('clicked!');
3. }, false);

Note that the third parameter describes whether the callback has to be called during the captured phase. This is not important for now, just set it to false or ignore it (you can even pass only two parameters to the addEventListener function call and do not set this boolean parameter at all).

#### Details of the DOM event are passed to the event listener function

When you create an event listener and attach it to an element, the listener will create an event object to describe what happened. This object is provided as a parameter of the callback function:

1. element.addEventListener('click', function(**event**) {
2. **// now you can use event object inside the callback**
3. }, false);

Depending on the type of event you are listening to, you will consult different properties from the event object in order to obtain useful information such as: "which keys are pressed down?", "what is the location of the mouse cursor?", "which mouse button has been clicked?", etc.

In the following lessons, we will remind you how to deal with the keyboard and the mouse.

### Further reading

In Method 1 (above), we mentioned that "mixing 'visual layer' (HTML) and 'logic layer' (JavaScript) ... is bad practice", and this is similarly reflected in many style features being deprecated in HTML5 and moved into CSS3. The management philosophy at play here is called "the separation of concerns" and applies in several ways to software development - at the code level, through to the management of staff. It's not part of the course, but professionals may find the following references useful:

* [Separation of concerns - Wikipedia, the free encyclopedia](https://en.wikipedia.org/wiki/Separation_of_concerns)
* [Chapter 5. Separation of Concerns](http://chimera.labs.oreilly.com/books/1234000000262/ch05.html) from Programming JavaScript Applications, by Eric Elliott, O'Reilly, 2013.
* [The Art of Separation of Concerns](http://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/) by derekgreer, January 3, 2008

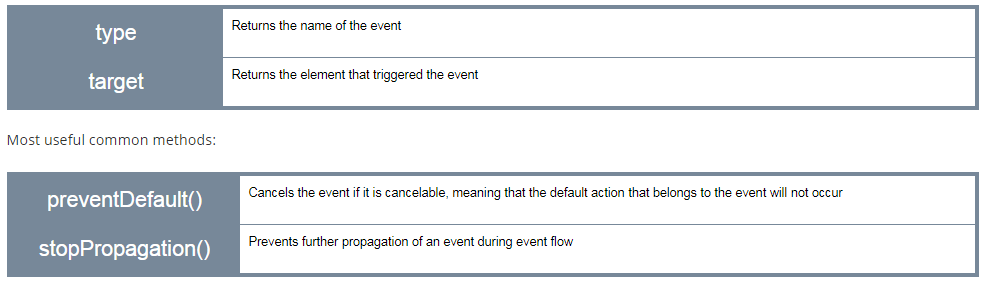
### Reference tables for events and properties/methods

These tables are provided as a reference. They are a compilation of the most common event types sorted by domain (key, mouse, forms, etc.). For each domain you will see the most useful event types and their properties.

In the following sections, we will show  examples that use most of the events displayed in these tables.

#### Event object

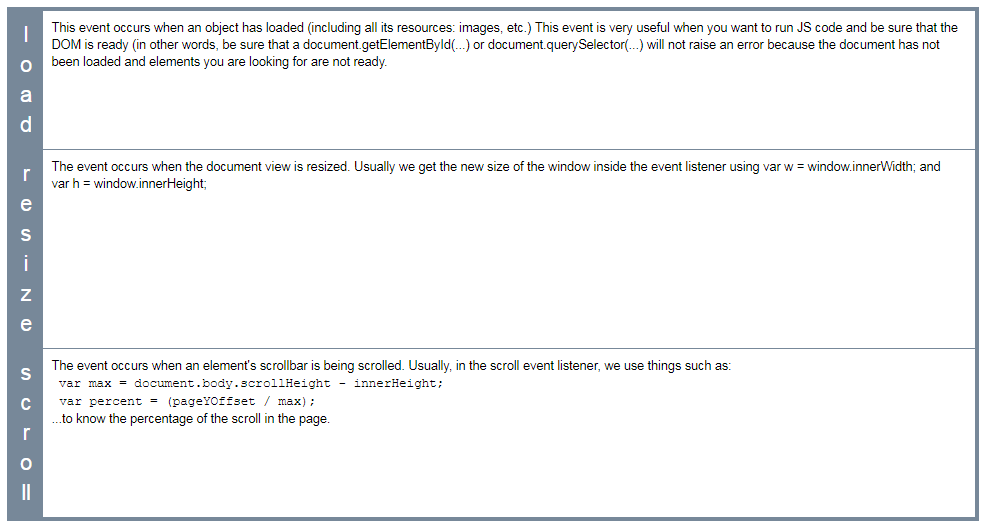
Most useful common properties:



#### Page

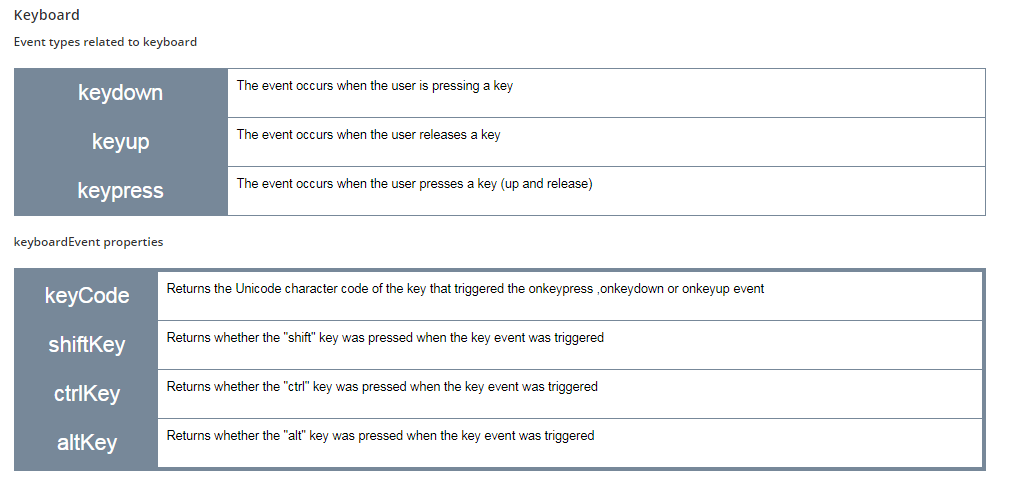
##### Events related to the page lifecycle

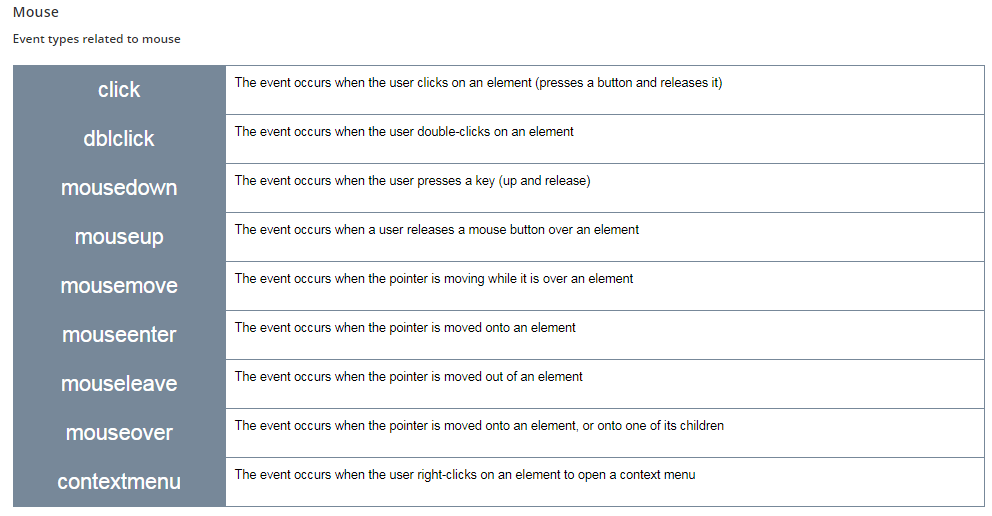
There are many other events related to the page life cycle. Below are the most useful ones for an intro course:

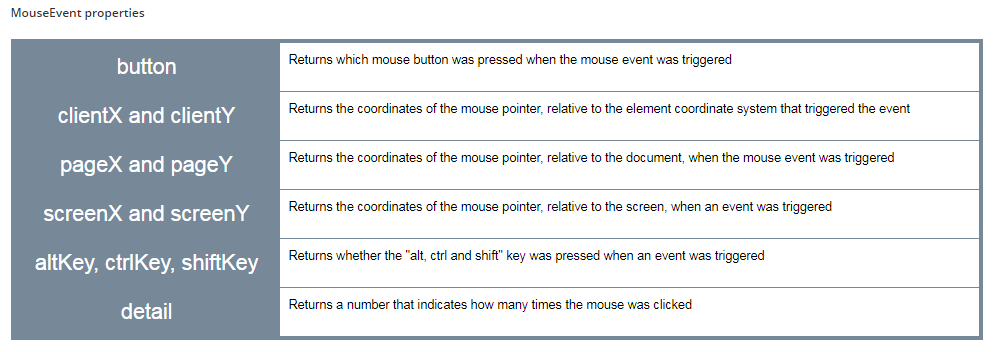


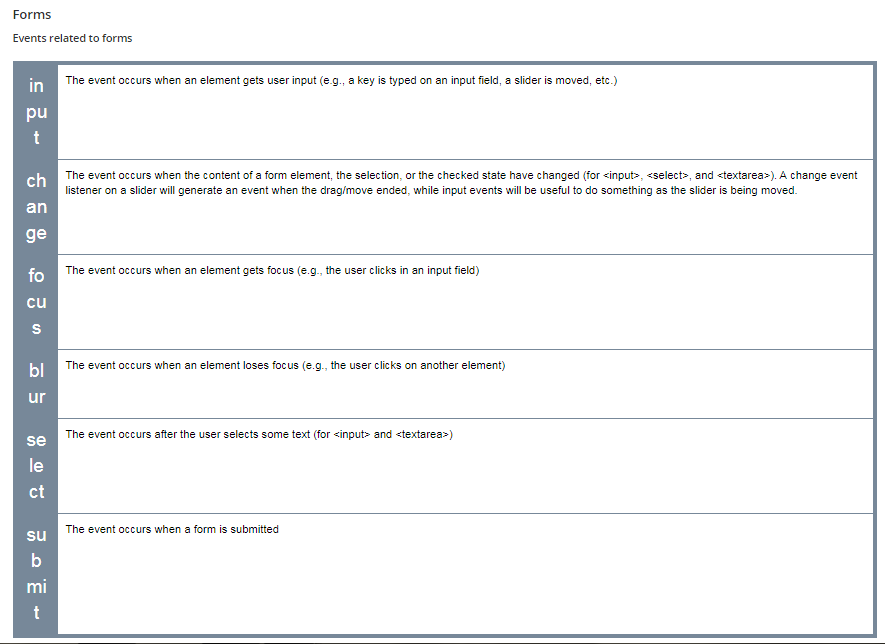
##### Page event properties

There are no particular properties that need to be mentioned here. Usually, the load event listener corresponds to a JavaScript function that can be seen as "the main" function of your Web Application. It is good practice to start everything after the page has been completely loaded. In the resize listener, you get the new size of the window, or the new size of some HTML elements in the page (as they might have been resized too when the window was resized) and then you do something (redraw a graphic in an HTML canvas that takes into account the new canvas size, for example).



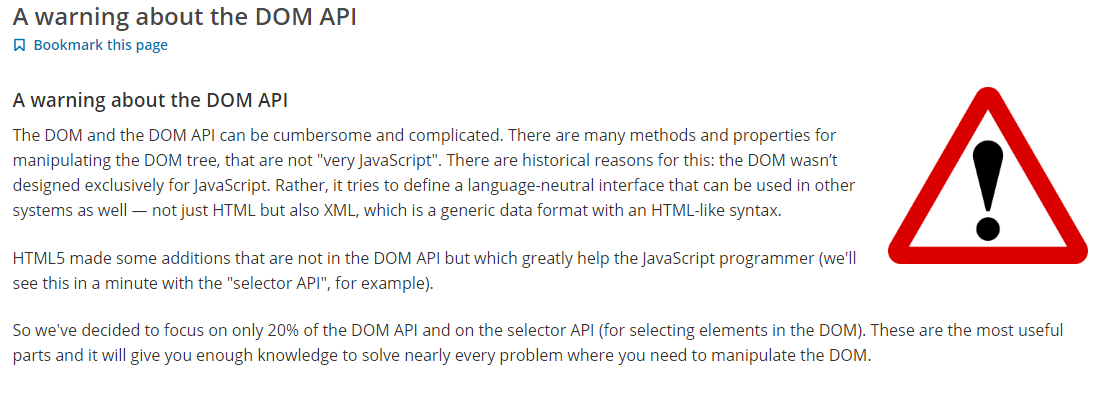






##### FormEvent properties

There are no particular properties that need to be mentioned here. Usually, on a form event listener, we check the content of the different input fields, using their value property.



### Accessing HTML elements

#### 1 - With the selector API (recommended)

Extract from [HTML5 selectors API – It’s like a Swiss Army Knife for the DOM](http://www.webdirections.org/blog/html5-selectors-api-its-like-a-swiss-army-knife-for-the-dom/) : "One of the many reasons for the success of JavaScript libraries like[*jQuery*](https://jquery.org/) and [*Prototype*](http://prototypejs.org/), on top of their easing the pain of cross-browser development was how they made working with the DOM far less painful than it had previously been, and indeed how it was with the standard DOM. Being able to use arbitrary CSS selector notation to get matching elements from a document made the standard DOM methods seem antiquated, or at the very least, far too much like hard work.

Luckily, the standards and browser developers took notice. The W3C developed the Selectors API, a way of easily accessing elements in the DOM using standard CSS selector concepts, and browser developers have baked these into all modern browsers, way back to IE8."

##### The querySelector(CSSSelector) and querySelectorAll(CSSSelector) methods

Ah... these methods owe a lot to [jQuery](https://jquery.org/" \t "_blank)! They introduce a way to use CSS selectors (including CSS3 selectors) for requesting the DOM, like jQuery introduced ages ago.

Any CSS  selector can be passed as a parameter for these methods.

* + While **querySelector(selector)** will return the first element in the DOM that matches the selector (and you will be able to work with it directly),
  + **querySelectorAll(selector) returns a collection of HTML elements corresponding to all elements matching the selector**. To process the results, it will be necessary to loop over each of the elements in the collection.

Use forEach – it requires a call back function

1. // **select all images**
2. var listImages =**document.querySelectorAll("img");**
4. // change all their width to 100px
5. listImages.forEach(function(img) {
6. img.width = 100;
7. }

CSS Selector example-

1. // first li of class="red" in a ul
2. var elm =**document.querySelector("ul li.red");**
3. // get all li directly in a ul of class nav
4. var list =**document.querySelectorAll("ul.nav > li");**

JavaScript code: we select all elements of type input that have an attribute checked equal to true, and located inside an element whose id is "fruits".

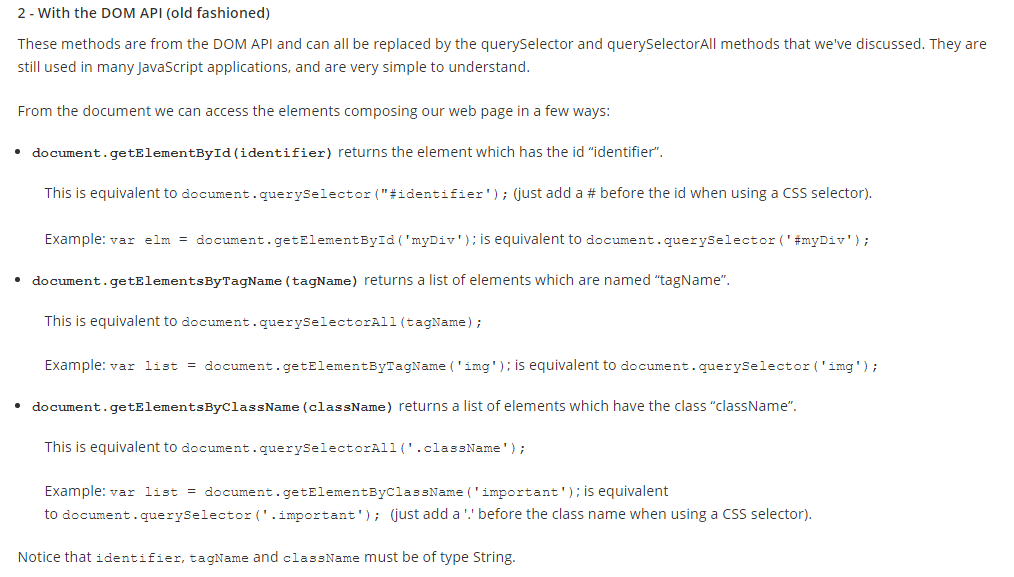
var list = document.querySelectorAll("#fruits input:checked");

we build a CSS selector using the id passed as a parameter

So, the variable CSS selector at line 2 in the JavaScript code below will have a value equal to "#firstDiv p", that means: select all <p> under an element whose id is "firstDiv".

var paragraphs = document.querySelectorAll("#" + id + " p");





#### Using the ClassList interface to change more than one CSS property simultaneously

External resources:

* + [The W3C specification about the classList DOM interface](http://www.w3.org/TR/dom/#dom-element-classlist)
  + [An article from the Mozilla Developer's web site](https://hacks.mozilla.org/2010/01/classlist-in-firefox-3-6/)

Until now, to manipulate CSS classes of an HTML element was a bit complex, both for verifying the presence of a class name in an element, and for adding or removing classes associated with a given element.

The ClassList interface simplifies it all by acting as a container object and by providing a set of methods to manipulate its content.

The classList property applies to an HTML element, and returns a collection of class names:

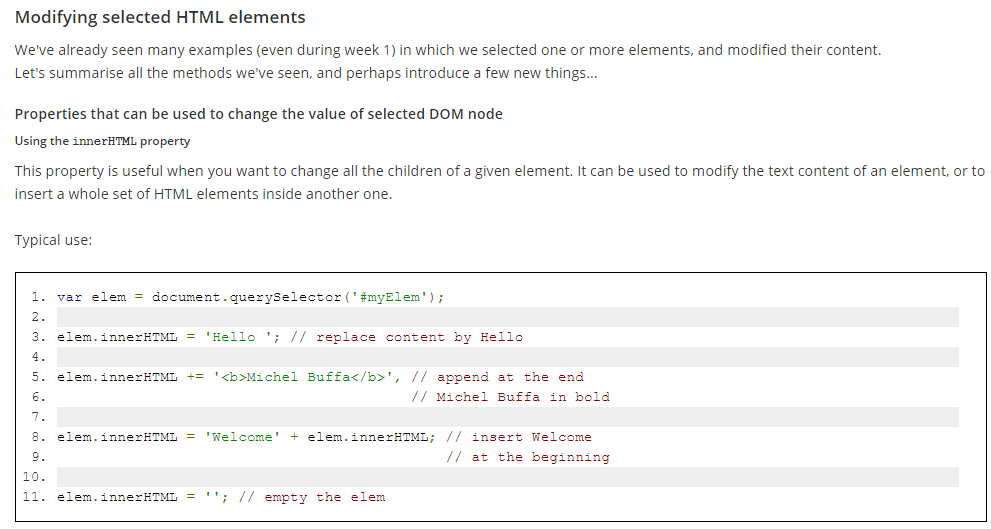
1. var elem= document.querySelector("#id1");
3. var allClasses = elem.**classList**;

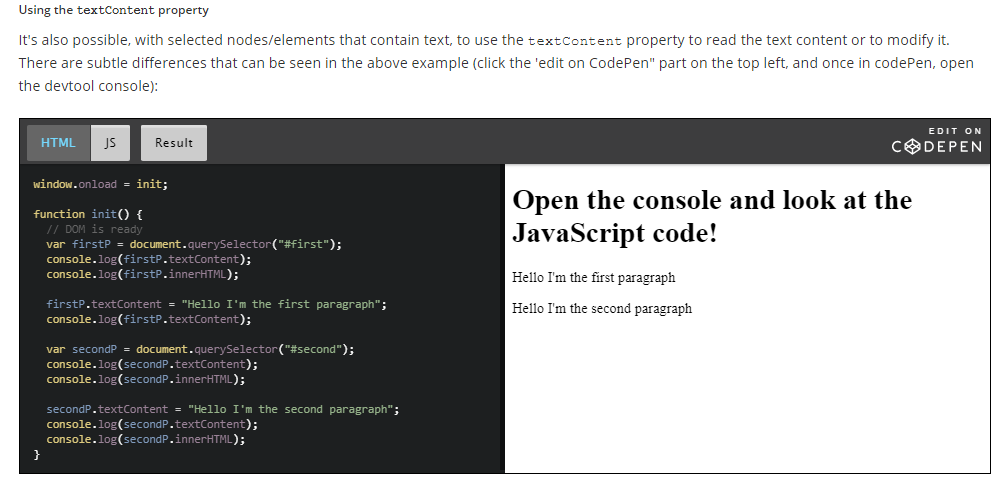
##### The classList API

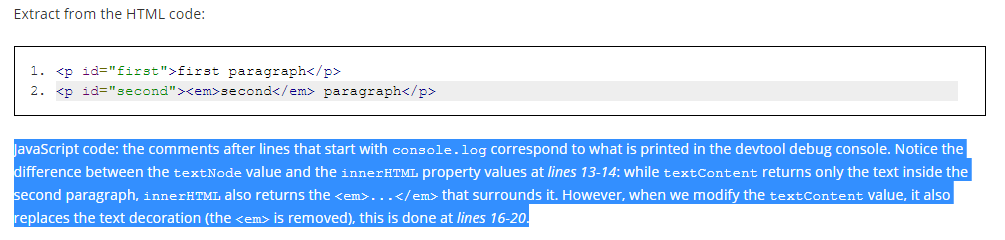
The list of methods usable on a classList object are add(), remove(), toggle() and contains().

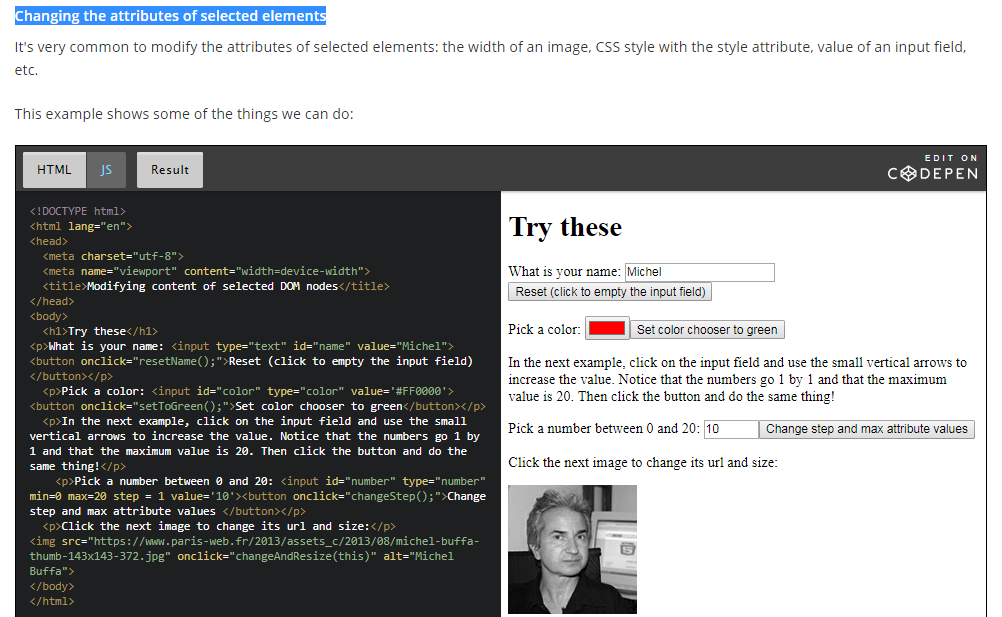
1. // By default, start without a class in the div: <div class=""/>
2. // Set "foo" as the class by adding it to the classList
3. div.classList.add('foo'); // now <div class="foo"/>
4. // Check that the classList contains the class "foo"
5. div.classList.contains('foo'); // returns true
6. // Remove the class "foo" from the list
7. div.classList.remove('foo'); // now <div class=""/>
8. // Check if classList contains the class "foo"
9. div.classList.contains('foo'); // returns false: "foo" is gone
10. // Check if class contains the class "foo",
11. // If it does, "foo" is removed, if it doesn't, it's added
12. div.classList.toggle('foo'); // class set to <div class="foo"/>
13. div.classList.toggle('foo'); // class set to <div class=""/>

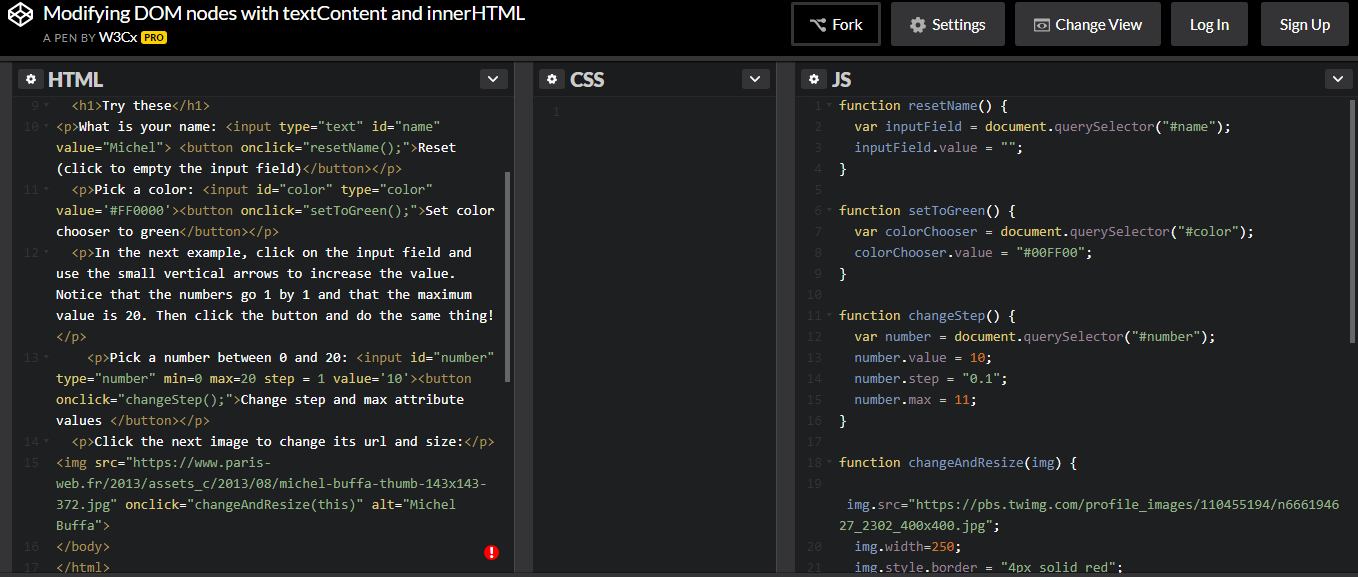
## Changing the content of selected HTML elements











### Adding new elements to the DOM

The DOM API comes with a set of methods you can use on DOM elements.

In general, to add new nodes to the DOM we follow these steps:

* 1. Create a new element by calling the createElement() method, using a syntax like:
     1. var elm = document.createElement(name\_of\_the\_element).

Examples:

* + 1. var li = document.createElement('li');
    2. var img = document.createElement('img'); etc.
  1. Set some attributes / values  / styles for this element.   
       
     Examples:   
       
     1. li.innerHTML = '<b>This is a new list item in bold!</b>'; // can add HTML in it
     2. li.textContent = 'Another new list item';
     3. li.style.color = 'green'; // green text
     4. img.src = "http://..../myImage.jpg"; // url of the image
     5. img.width = 200;

* 1. Add the newly created element to another element in the DOM, using append(), appendChild(), insertBefore() or the innerHTMLproperty  
       
     Examples:
     1. var ul = document.querySelector('#myList');
     2. ul.append(li); // insert at the end, appendChild() could also be used (old)
     3. ul.prepend(li); // insert at the beginning
     4. ul.insertBefore(li, another\_element\_child\_of\_ul);// insert in the middle
     5. document.body.append(img); // adds the image at the end of the document

#### A warning about append vs appendChild, prepend, etc...

**If the examples do not run on your browser (Edge, IE, old browsers), read this!**

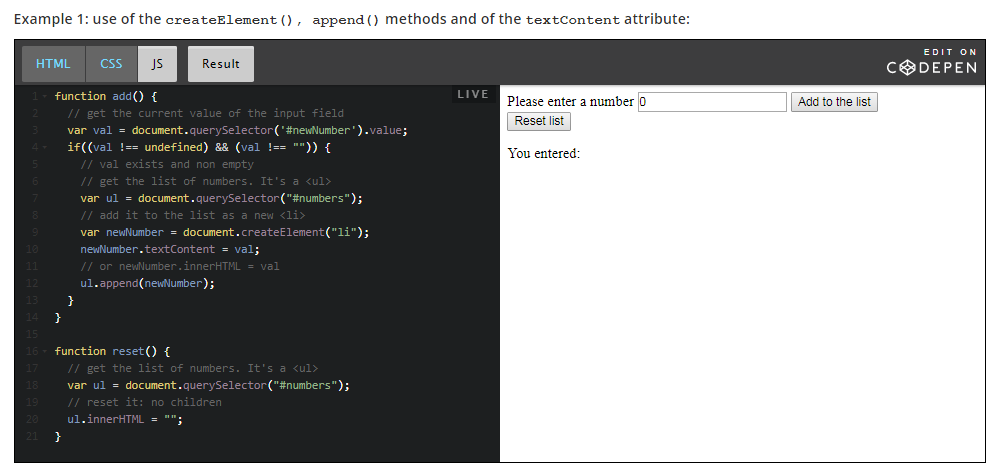
The DOM specification recently added some jQuery like methods that are similar to the ones proposed by the [jQuery](https://jquery.org/" \t "_blank) library (that was designed a long time ago to simplify DOM manipulations).

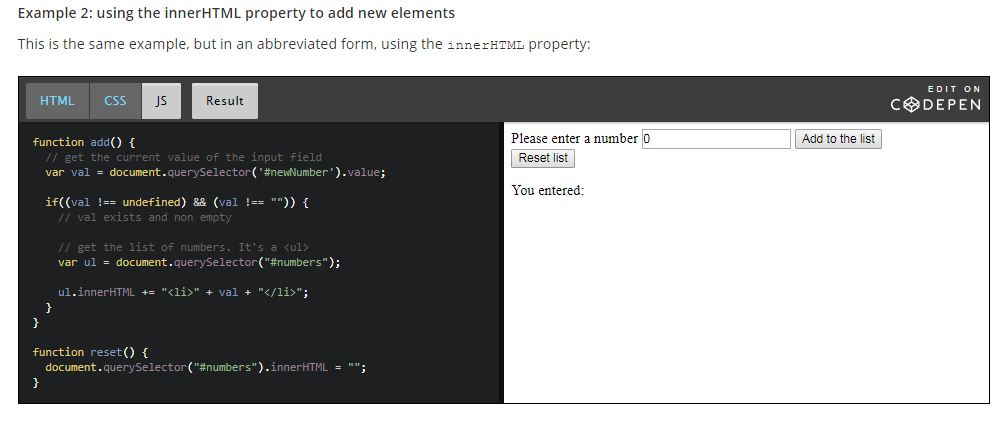
For a long time, developers used document.appendChild to add an element to the DOM. Now you can use document.append, which is shorter, along with some other methods such as document.prepend etc. All these methods are supported by recent browsers except Microsoft Edge (but support is coming soon). See [this table for compatibility](http://caniuse.com/#feat=dom-manip-convenience).

**In the course, we recommend that you use document.append, but if you are looking for maximum compatibility, you can either use document.appendChild instead of document.append (just search and replace all occurrences of document.appendwith document.appendChild in the examples), or add to your JavaScript code**[**this polyfill**](https://developer.mozilla.org/fr/docs/Web/API/ParentNode/append#Polyfill)**.**

Just copy and paste the 10 lines of JavaScript from the polyfill to your code and append will work on browsers that do not yet implement it.

Here is [an example in JsBin that uses all these new methods, and that includes the polyfill](http://jsbin.com/sodeheh/edit?html,js,output) for append and prepend at the end of the JavaScript code.





### Moving DOM elements within a document

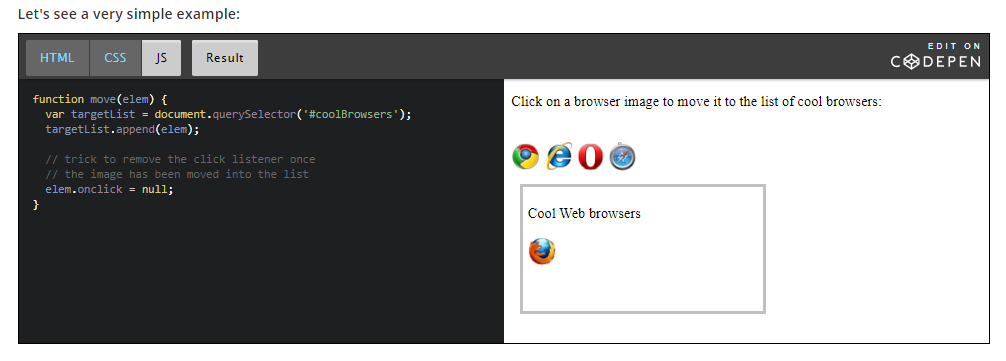
The append(), appendChild() methods normally adds  a new element to an existing one, as shown in this example:

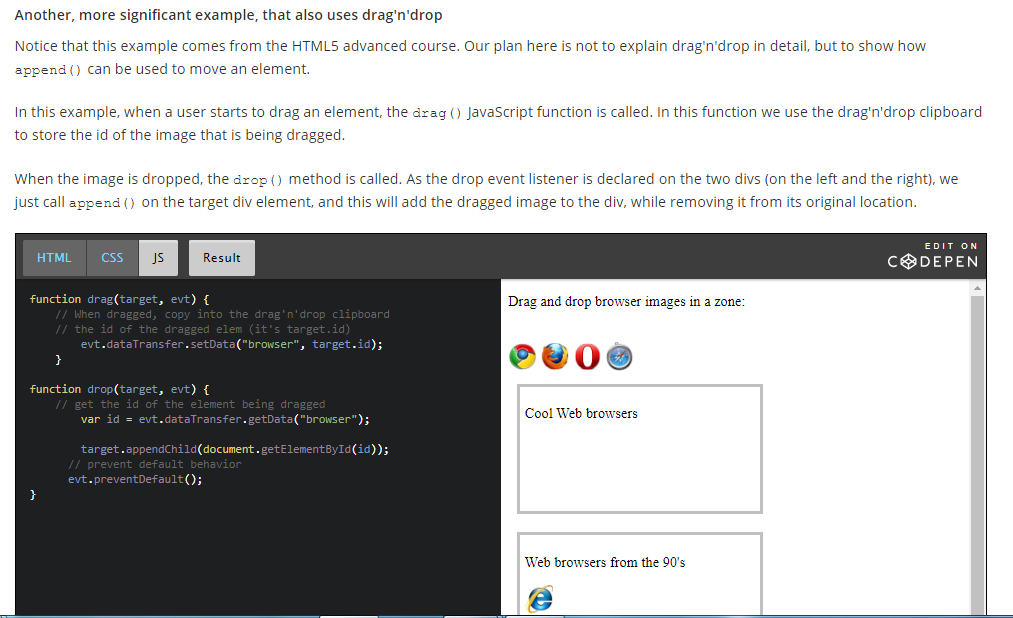
1. var li = createElement('li');
2. ul.append(li); // adds the new li to the ul element

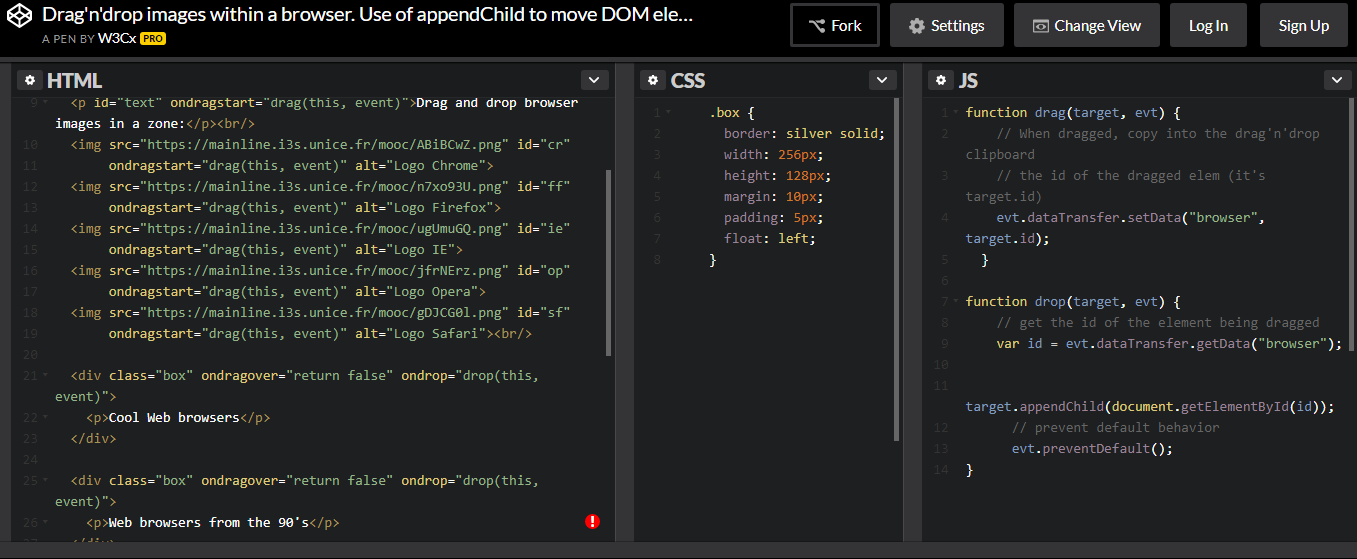
One interesting thing to know is that if we do not create the new element, but rather get it from somewhere else in the document, it is then removed from its parents and added to the new parent.

In other words: it moves from its original location to become a child of the targetElem.

Look below Javascript code – the elem is the current DOM element that was clicked, so if we append an element which is already a part of DOM then it removes from previous parent and added to new parent.







### Removing elements from the DOM

#### Removing elements using the removeChild() method