****

**Javascript Introduction**

**- Introduction to Javascript –**

**edX – W3CX(Microsoft)**

1. **Module 5 Outline**

.

### **What you will learn in Module 5**

* **We continue the study of object-oriented JavaScript programming with "predefined classes"**: Object, Array, String, Date, Math
* **We present a very useful API for building HTML5 tables dynamically**.
* **We'll also see how to use HTML5 forms from JavaScript**: intercept their submission, retrieve input field values, and so on.
* **We will study the JavaScript Object Notation (JSON) format**, used extensively when manipulating data from remote servers (via REST WebServices) or for persistence (saving data client or server-side). We will see several examples that consume such data and display them in a dynamic HTML5 table.
* **Finally, we will develop together a mini contact manager** using ES6 classes, a dynamic table, a form and load/save its data using the JSON format. It will be your job to improve it by adding additional features.

1. **Objects (part 4): objects and references, built-in JS classes**
   1. **References and objects**

**[ -- Live coding video: references and objects -- ]**

### **References and objects**

First of all, we have to define "reference". Unlike a pointer variable, which contains the actual address of an object within the memory, a reference variable is an alias to a variable. This means that when you modify a reference variable, the original variable is modified too. This is because the two variables reference (i.e. point to) the same object.

When you define a variable (such as var x = 10; or let name = "Michel"; or let courseAuthor = {firstName:'Michel', lastName:'Buffa'), this is what happens:

* If its value is a primitive value (number, string, or boolean), the variable contains this value directly.
* If its value is an object, the variable contains the memory address of the object. We say that this variable "points to" an object (or references this object). Accessing the variable will automatically resolve the reference, meaning that the value of the variable is the referenced object.

Examples:

1. // Defining two variables
2. var x = 2; // the variable x contains the primitive datum 2
3. var y = { a: 2 } // The variable y references the object {a: 2}
4. // "Copying" two variables
5. var x2 = x;
6. var y2 = y;
7. var y3 = y;
8. // Modifying copied variables
9. x2 = 3;
10. y2 = { a: 3 };
11. // Check
12. x; // 2 <- x is not modified because it contains a primitive value
13. y; // { a: 2 } <- y is not modified because y2 does not point to same object
14. y3.a = 4;
15. y; // { a: 4 } <- The object referenced by "y" and "y3" is modified

Of course, these rules also apply to the properties of objects.

Example:

1. var driver = {
2. name: 'Jean'
3. };
4. var car = {
5. color: 'red',
6. driver: driver
7. };
8. driver.name = 'Albert';
9. car.driver.name; // 'Albert'

JavaScript is a "pass by value" language, unlike some other languages, which are "pass by reference" languages. This means that when you pass a variable to a function as argument, the value of the variable is copied into the argument.

Example:

1. var x = 2;
2. function sum(a, b) {
3. a = a + b;
4. return a;
5. }
6. sum(x, 3); // returns 5
7. x; // 2 <- but x equals 2

When working with objects, the reference of the object is copied into the argument. That means you can modify the referenced object. But if you change the reference (for example by assigning a new object), the original variable (which now points to another object) will not be modified.

Example 1:

1. var obj = { x: 2 }
2. function add(a, b) {
3. a.x += b;
4. }
5. add(obj, 3);
6. obj.x; // 5 <- The referenced object is modified

Example 2:

1. var obj = { x: 2 };
2. function addAndSet(a, b) {
3. var addition = a.x + b;
4. a = { x: addition };
5. };
6. addAndSet(obj, 3);
7. obj.x; /\* 2 <- The referenced object is not modified
8. because at the end of the function the variable "obj"
9. and the variable "a" are not referencing the same object.\*/

Other examples:

1. > var originalObject = {name:'Michel'};
2. undefined
4. > var copy = originalObject;
5. undefined
7. > copy.name;
8. "Michel"
10. > copy.name = 'Dark Vador';
11. "Dark Vador"
13. > originalObject.name
14. "Dark Vador"
16. // They are the same. originalObject and copy are two "references" of the same object in memory
17. // If we change the name, we change the value in memory, but copy and originalObject "point to" the
18. // same place, to the same object. They are just "pointers" or "reference" to the same object
    1. **Comparing two objects**

Comparing two objects will only return true if they point to the same object (i.e., if they have the same reference).

Two objects of the same type, with the same property value, that look identical, will not be equal one to another if they don’t have the same reference (if they point to different places in memory).

1. > var originalObject = {name:'Michel'};
2. undefined
4. > var copy = originalObject;
5. undefined
7. > copy === originalObject
8. true
10. > var anotherObject = {name:'Michel'};
11. undefined
13. > copy === anotherObject
14. false
    1. **The “global” window object**

It is time to tell you the truth: the JavaScript code is executed by an “environment" (usually a Web browser, but there are some HTTP Web servers that use JavaScript for coding the server side of Web sites of applications, such as the NodeJS HTTP server).

This environment defines a “global object”.

**When this environment is a Web server  
(and this is the case for all examples we have seen in this course),**  
**this global object is named window.**

**The “global variables” defined with the keyword var are properties of this window object,  
and we can say the same of predefined functions like prompt, alert, etc.**

**However, at the top level of programs and functions,  
let, unlike var, does not create a property on the global window object.**

**TIP: if you have global variables/objects declared with let,  
just declare them with var instead, and you will be able to inspect them  
easily from the devtool console.  
You can switch back to using let, later.**

Let's see some examples:

1. > var a = 1;
2. undefined
4. > a;
5. 1
7. > window.a;
8. 1
10. > window['a'];
11. 1
12. > **let z = 1;** // LET DOES NOT DEFINE properties of the window object
13. undefined
14. > **window.z**
15. **undefined**

a and window.a are the same variable.  
navigator and window.navigator are the same, document and window.document are the same thing.

1. > document === window.document
2. true
4. > navigator === window.navigator
5. true
6. Predefined functions are methods from the global object window:
7. > parseInt('10 little children');
8. 10
10. > window.parseInt('10 little children');
11. 10
13. > alert === window.alert
14. true
16. > prompt === window.prompt
17. true
19. > window.addEventListener === addEventListener
20. true
    1. **Built-in JS class: Object**

**[ -- Live coding video: predefined class - Object -- ]**

### **Built-in JavaScript classes: Object**

#### The father of all objects: Object

All objects will inherit the properties and methods from the special class named Object.

These two lines are equivalent:

1. > var o = {}; // creation of an empty object
2. undefined
4. > var o = new Object(); // same thing as in line 1
5. undefined

#### The toString method inherited from Object by all objects

1. > o.toString();
2. "[object Object]"
4. > o.name = 'Michel';
5. "Michel"
7. > o.toString();
8. "[object Object]"
10. > var t = [1, 2, 3];
11. undefined
13. > t.toString();
14. "1,2,3"

toString() in JavaScript is rather similar to the Object.toString() method we find in the Java programming language: **when we try to "display" an object, it is transformed into a string by calling toString() implicitly**.

1. > alert(**t**);
3. > alert(**t.toString()**); **// same as previous line of code**
5. > "An object into a string : " **+ t**// same as **t.toString()**
6. "The object as a String : 1, 2, 3"

Line 5: using the + operator with a string as the left argument will force the other arguments to convert to string by implicitly calling their toString() method.

#### The valueOf method inherited from Object by all objects

The ValueOf method returns the value of an object:

1. > var t = [1, 2, 3];
2. undefined
4. > t.valueOf()
5. [1, 2, 3]
7. > t.toString();
8. "1,2,3"
   1. **Built-in JS class: Array**

**[ -- Live coding video: predefined object - Array -- ]**

### **Built-in JavaScript objects: Array**

#### The Array class can be used for creating arrays (however, we recommend that you use the other methods presented instead):

1. > **var a = new Array();** // **same as a = []; use this instead!**
2. undefined
4. > **var b = new Array(1, 2, 3);**
5. undefined
7. > b;
8. [1, 2, 3]

Attention: if only one element, this corresponds to the initial size of the array.

1. > var myArray = new Array(3);
2. undefined
4. > myArray;
5. [undefined × 3]

#### Arrays are objects, but they are “special” objects

* Their property names are numerical indexes that start from 0
* They have a special length property that represents their length/number of elements
* They have other built-in properties in addition to the ones inherited from Object (toString, valueOf)

1. > var a = [], o = {};
2. undefined
4. > a.length; // a is an array
5. 0
7. > o.length; // o is a simple literal object
8. undefined

Some horrible things we can do with arrays (TO AVOID!):

1. > var a = [1, 2];
2. undefined
4. > typeof a
5. "object"
7. > a.push(3);
8. 3
10. > a
11. [1, 2, 3]
13. > a.length
14. 3
16. **// Now let’s add a name property to the array. Yes, we can do that!**
18. **> a.name = "I'm an array named a!";**
19. **"I'm an array named a!"**
21. **> a.length;**
22. **3**
24. **> a;**
25. **[1, 2, 3, name: "I'm an array named a!"]**

**With arrays, only properties with a numerical index are taken into account by the length property!**

#### ****The length property can be modified: reducing or increasing the size of an array****

If you give to the length property a value bigger than the number of elements in an array, it adds undefined elements to it:

1. > var a = [1, 2];
2. undefined
4. > a.length = 5;
5. 5
7. > a;
8. [1, 2, undefined × 3]

If you give to the length property a value less than the array’s number of elements, it reduces the size of the array:

1. > var a = [1, 2, 3];
2. undefined
4. > a.length = 2;
5. 2
7. > a;
8. [1, 2]
   1. **The most useful methods of the class Array**

#### Most useful methods you can use on arrays: sort(), join(), slice(), splice(), push()and pop()

* sort: sort the elements in the array. Either alphabetically if they are strings, or in ascending order if they are numbers. There is also a way to sort the elements using other criteria, which is explained a bit further on in the course. With a call to var b = a.sort(), a is also sorted. The sort method sorts the array + returns it.
* join: adds a string between each element and returns the result as a string
* slice: returns a sub-array without modifying the original array
* splice: modifies the array, it removes “a slice” and it also adds new elements
* push: appends an element at the end of the array and returns the new length
* pop: removes the last element and returns it

#### Typical uses of push, pop, sort, join:

1. > var a = [3, 5, 1, 7, 'test'];
2. undefined
4. > a.push('new') // appends at the end and returns the new length
5. 6
7. > a;
8. [3, 5, 1, 7, "test", "new"]
10. > a.pop(); // removes the last element and returns it
11. "new"
13. > a;
14. [3, 5, 1, 7, "test"]
16. > var b = a.sort();
17. undefined
19. > b;
20. [1, 3, 5, 7, "test"]
22. > a;
23. [1, 3, 5, 7, "test"]
25. // a is also sorted. The sort method sorts the array + returns it
26. undefined
28. > a.join(' and ');
29. "1 and 3 and 5 and 7 and test"

#### The slice() method returns a sub-array without modifying the original array:

The slice() method returns a shallow copy of a portion of an array into a new array object selected from begin to end (**end not included**). The original array will not be modified.

Possible syntaxes:

* arr.slice()
* arr.slice(begin)
* arr.slice(begin, end) // ELEMENT AT INDEX=end will not be included in the slice!

1. > a;
2. [1, 3, 5, 7, "test"]
4. > b = a.slice(1, 3); // elements of indexes = 1 and 2
5. [3, 5]
7. > b = a.slice(0, 1); // element of index = 0
8. [1]
10. > b = a.slice(0, 2); // elements o indexes = 0 and 1
11. [1, 3]
13. > a;
14. [1, 3, 5, 7, "test"]
16. // a is unchanged by calls to a.slice(...)

#### The splice() method modifies the array: it removes “a slice” and also adds new elements

The first two parameters are start and the number of elements to delete, the other parameters are the elements to add to the array to replace the slice that will be removed.

Possible syntaxes:

* array.splice(start)
* array.splice(start, deleteCount)
* array.splice(start, deleteCount, item1, item2, ...)

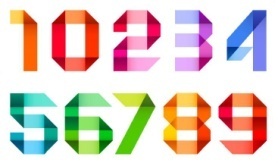
start: index at which to start changing the array (with origin 0)

deleteCount: An integer indicating the number of old array elements to remove.

item1, item2, ...: these are optional. They are the elements to add to the array, beginning at the start index. If you don't specify any elements, splice() will only remove elements from the array.

Examples:

1. > a;
2. [1, 3, 5, 7, "test"]
4. > b = a.splice(1, 2, 100, 101, 102);
5. [3, 5]
7. > a;
8. [1, 100, 101, 102, 7, "test"]
10. > a.splice(1, 3);
11. [100, 101, 102]
13. > a;
14. [1, 7, "test"]
    1. **Built-in JS class: Number**

The Number class can be used to transform strings into numbers, but it is recommended that you use parseInt or parseFloat instead.

1. > var n = Number('3.1416');
2. undefined
4. > n;
5. 3.1416
7. > typeof n;
8. "number"
10. > var n = parseInt('3.1416'); // convert a string to an integer number
11. undefined
13. > n;
14. 3
16. > var n = parseFloat('3.1416'); // convert a string to a float number
17. undefined
19. > n;
20. 3.1416

#### Number has useful non-modifiable properties (constants): MAX\_VALUE and MIN\_VALUE

1. > Number.MAX\_VALUE;
2. 1.7976931348623157e+308
4. > Number.MIN\_VALUE;
5. 5e-324

#### Methods useful for converting numbers: toFixed(), toExponential(), toString()

* toFixed: sets the number of digits for the decimal part of a number.  
  There is also another method, named [toPrecision](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toPrecision), that has a very close behavior, and can also return numbers in scientific notation.
* toExponential: force a number to use a scientific notation. For example var a=1000; a.toExponential(); console.log(a); will give 1e+3

1. > var n = 123.456;
2. 123.456
4. > n.toFixed(1); // sets the number of digits for the decimal part of the number
5. "123.5"
7. > n = new Number(123.456); // same as n = 123.456
8. Number {[[PrimitiveValue]]: 123.456} // well, not exactly, but when you use n, it is equivalent
10. > n.toFixed(1);
11. "123.5"
13. > n.toExponential();
14. "1.23456e+2"

17. > var n = 255;
18. undefined
20. > n.toString();
21. "255"
23. > n.toString(10);
24. "255"
26. > n.toString(16);
27. "ff"
29. > (3).toString(2);
30. "11"
32. > (3).toString(10);
33. "3"
    1. **Built-in JS class: String**

**[ -- Live coding video: predefined class - String -- ]**

### **Built-in JavaScript class: String**

#### The String class can be used to build new strings, but it’s preferable to use the standard syntax

1. >**var name = 'Michel'; // use this rather than using new String(...)**
2. undefined
4. > typeof name;
5. "string"
7. > var name = new String('Michel');
8. undefined
10. > typeof name;
11. "string"

#### Some reminders about strings:

1. > var name = 'Michel';
2. undefined
4. > name.length;
5. 6
7. > name[0];
8. "M"
10. > name[0] = 'Z';
11. "Z"
13. > name; **// we cannot modify a string using s[index] = value;**
14. "Michel"
16. > 'Michel'.length;
17. 6
19. > 'Michel'[0];
20. "M"

Explanations:

* Line 10: in JavaScript, and in many other programming languages,**a string is not modifiable at all**.  
  **When we do var s = s + "hello", in fact, we are building a new string somewhere in memory, and we assign this new value to the variable s.**  
  We never "modify" the characters of the string s, we just give to s another address in memory to point to.

#### Useful methods: toUpperCase, toLowerCase, indexOf, charAt

These methods are all inherited from the String class:

* toUpperCase: returns the string in upper case. Do not change the original string.
* toLowerCase: returns the string in lower case. Do not change the original string.
* indexOf: returns the index of the string value passed as parameter, -1 if the string value is not found in the original string.
* charAt: returns the char at the index passed as parameter. Returns an empty string if the index is out of bounds (less than 0 or greater than the length of the string).

1. > var s = "I'm the Walrus";
2. undefined
4. > var s1 = s.toUpperCase();
5. undefined
7. > s1;
8. "I'M THE WALRUS"
9. > var s2 = s1.toLowerCase();
10. undefined
12. > s2;
13. "i'm the walrus"
15. > s; // s is unchanged
16. "I'm the Walrus"
18. > s.indexOf('w'); // no ‘w’ in s
19. -1
21. > s2.indexOf('w');
22. 8
24. > s2[8]; // char at index 8
25. "w"
27. s2.charAt(8); // same as s2[8]
28. "w"

#### Other useful methods: lastIndexOf, chaining methods

* lastIndexOf: returns the last index of the string value passed as parameter
* indexOf can also be used with two parameters, the second one being the starting index when looking for the string value passed as parameter

1. > s = 'wow wow wow!';
2. "wow wow wow!"
4. > s.lastIndexOf('w');
5. 10
7. > s.indexOf('w', 1); // start looking at s at index=1, s[0] is ignored
8. 2
10. > var s1 = s.toUpperCase();
11. undefined
13. > s1;
14. "WOW WOW WOW!"
16. > s1.toLowerCase().lastIndexOf('w'); // we can chain method calls using ‘.’
17. 10
    1. **The most useful methods of the class String**

### **The most useful methods of the String class: slice, substring, split, join**

#### The slice and substring methods

**Both these methods can be used to extract a substring from a string.**

They take two parameters: the start and end index of the slice (**element at end index will NOT be included in the slice**): “please cut from this index, to this one, not included!”.

These two methods are very similar.

Examples:

1. > var s = "My name is Bond! James Bond!";
2. undefined
4. > s;
5. "My name is Bond! James Bond!"
7. > s.slice(11, 16);
8. "Bond!"
10. > s; **// s is unchanged**
11. "My name is Bond! James Bond!"
13. s.substring(11, 16);
14. "Bond!"
16. > s; **// s is still unchanged**
17. "My name is Bond! James Bond!"
18. > s = s.substring(11, 16);
19. "Bond!"
21. > s; **// this time s has changed, because we did s = s.substring(...), the same**
22. **// could have been done with s = s .slice(...)**
23. "Bond!"

**[advanced] There is a difference between slice and substring, when the second parameter is negative:**

If you are a beginner, we recommend that you use substring for most common cases (as it will behave the same as slice) and that you stay away from negative parameters, where slice and substring show small differences.

Beginners: do not read what follows about slice and substring! There will be no quiz questions at the end of this chapter about this part!

1. > var s = "My name is Bond! James Bond!";
2. undefined
4. > s.slice(11, -1); // start from index = 11 to length-1, extract the end of the string from 11th element
5. "Bond! James Bond"
7. > s.substring(11, -1); // the reverse, extract from 0 until 11-1, get the first 10 chars
8. "My name is "
10. > s.substring(1, -1); // extract from 0 to 1-1 = 0, get the first char
11. "M"

Actually, here is a summary of the common behaviors and the differences between slice and substring.

**[advanced] slice(start, stop) works like substring(start, stop) with a few different behaviors:**

**What they have in common:**

* If start equals stop: returns an empty string
* If stop is omitted: extracts characters to the end of the string
* If either argument is greater than the string's length, the string's length will be used instead.

**Distinctions of substring():**

* If start > stop, then substring will swap those two arguments.
* If either argument is negative or is NaN, it is treated as if it were 0.

**Distinctions of slice():**

* If start > stop, slice() will NOT swap the two arguments.
* If start is negative: sets char from the end of string.
* If stop is negative: sets stop to: string.length – Math.abs(stop.

#### The split(), join() and concat() methods

The split method returns an array of strings, the parameter is a separator. The join method builds a string from an array of strings.

1. > var s = "My name is Bond! James Bond!";
2. undefined
4. > s.split(" ");
5. ["My", "name", "is", "Bond!", "James", "Bond!"]
7. > s;
8. "My name is Bond! James Bond!"
10. > s.split(' ').join('-#-');
11. "My-#-name-#-is-#-Bond!-#-James-#-Bond!"
13. > s.split(' ').join('.......');
14. "My.......name.......is.......Bond!.......James.......Bond!"
16. > s.split('Bond!').join('.......');
17. "My name is ....... James ......."
19. > s.split('Bond!').join(' ');
20. "My name is James "
22. > s; // s is unchanged
23. "My name is Bond! James Bond!"
25. > s.concat("And I've made a lot of movies!");
26. "My name is Bond! James Bond! And I've made a lot of movies!"
28. > s; // s is also unchanged by concat
29. "My name is Bond! James Bond!"
31. > s = s + "and I've made a lot of movies!"; // this changes s
32. "My name is Bond! James Bond! And I've made a lot of movies!"
34. > s += " Action films!" // this too, most common syntax for concatenating strings
35. "My name is Bond! James Bond! And I've made a lot of movies! Action films!"
37. > s; // s changed too
38. "My name is Bond! James Bond! And I've made a lot of movies! Action films!"
    1. **Built-in JavaScript class: Math**

#### It’s not possible to do var m = new Math();

1. > var m = new Math();
2. VM5777:1 **Uncaught TypeError: Math is not a constructor**
3. at <anonymous>:1:9
4. (anonymous) @ VM5777:1

But the Math class has a lot of properties and methods that are useful for arithmetic expressions. **They are all class methods and properties, so you will need to use the name of the class followed by the dot operator to access them.**

Here are some examples:

1. > **Math.PI;**
2. 3.141592653589793
4. > **Math.SQRT2;**
5. 1.4142135623730951
7. > **Math.E; // Euler constant**
8. 2.718281828459045
10. > **Math.LN2; // Neperian log of 2**
11. 0.6931471805599453
13. > **Math.LN10; // Neperian log of 10**
14. 2.302585092994046

#### Random numbers between 0 and 1 with Math.random()

Math.random() returns a float value between 0 and 1.

Examples:

1. > **Math.random();**
2. 0.6033316111663034
4. > **100 \* Math.random(); // between 0 and 100**
5. 11.780563288516422

#### To get a number between a min and a max value, use this formula: val = ((max - min) \* Math.random()) + min

And here is an utility function:

1. function getRandomValue(min, max) {
2. return ((max - min) \* Math.random()) + min;
3. }
5. > getRandomValue(5, 10);
6. 5.064160540161435

#### Math and rounding methods round(), ceil(), floor()

round: to get the closest integer value.

For example Math.round(Math.random()); will return 0 or 1.

* Indeed, if Math.random() returns a value above 0.5, Math.round of this value will return 1, if the value is below 0.5, Math.round will return 0:

1. > Math.round(Math.random());
2. 1
4. > Math.round(Math.random());
5. 0
7. > Math.round(Math.random());
8. 1
10. > Math.round(Math.random());
11. 1

#### Get the min and the max of two values with Math.min(a, b) and Math.max(a, b)

1. > Math.min(12, 4);
2. 4
4. > Math.max(12, 4);
5. 12

A useful function that restricts a value between  min and  max bounds:

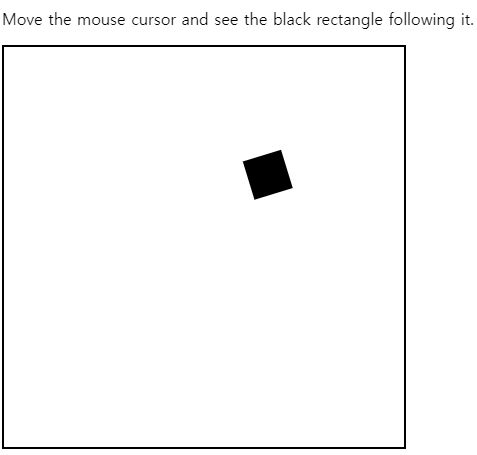
1. **function restrictValue(value, min, max) {**
2. **return Math.min(Math.max(1, value), max);**
3. **}**
5. > restrictValue(40, 1, 20);
6. 20
8. > restrictValue(-10, 1, 20);
9. 1
11. > restrictValue(10, 1, 20);
12. 10

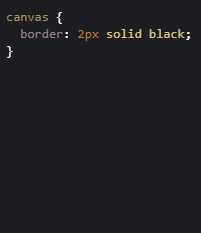
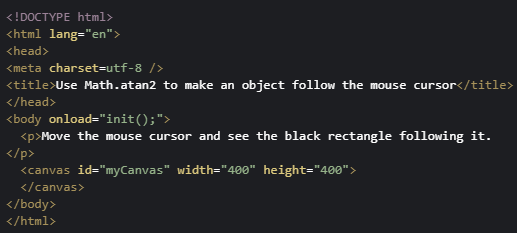
#### Math functions for arithmetical computations sin(), cos(), tan(), atan(), atan2(), pow(), sqrt()

1. > Math.pow(2, 8); // 2^8
2. 256
4. > Math.sqrt(9);
5. 3
7. > Math.sin(Math.PI/2);
8. 1
10. > Math.cos(Math.PI/2);
11. 6.123233995736766e-17

#### Math.atan2(dy, dx) is useful for getting an angle between a point in a canvas and the mouse cursor

Here is a typical example of the use of Math.atan2 in a video game, in order to make an object follow the mouse cursor by moving towards it. Look at the code in the mainloop function.



****

|  |
| --- |
| <script>    var canvas, ctx, width, height;    var rect = { x: 40, y: 40, radius: 30, width: 40, height: 40, v: 3 };    var mousepos = { x: 0, y: 0 };    function init() {      canvas = document.querySelector("#myCanvas");      ctx = canvas.getContext('2d');      width = canvas.width;      height = canvas.height;      canvas.addEventListener('mousemove', function (evt) {        mousepos = getMousePos(canvas, evt);      }, false);      mainloop();    }    function mainloop() {      // 1) clear screen      ctx.clearRect(0, 0, canvas.width, canvas.height);      // 2) move object      var dx = rect.x - mousepos.x;      var dy = rect.y - mousepos.y;      var angle = Math.atan2(dy, dx);      rect.x -= rect.v \* Math.cos(angle);      rect.y -= rect.v \* Math.sin(angle);      // 3) draw object      drawRectangle(angle);      // 4) request new frame      window.requestAnimationFrame(mainloop);    }    function drawRectangle(angle) {      ctx.save();      // These two lines move the coordinate system      ctx.translate(rect.x, rect.y);      ctx.rotate(angle);      // recenter the coordinate system in the middle      // the rectangle. Like that it will rotate around      // this point instead of top left corner      ctx.translate(-rect.width / 2, -rect.height / 2);      ctx.fillRect(0, 0, rect.width, rect.height);      ctx.restore();    }    function getMousePos(canvas, evt) {      // necessary to take into account CSS boudaries      var rect = canvas.getBoundingClientRect();      return {        x: evt.clientX - rect.left,        y: evt.clientY - rect.top      };    }  </script> |

* 1. **Built-in JS class: Date**

#### Getting a date by calling the Date constructor

**Without any argument, a call to new Date() returns the current date.**

Note: The return value is actually a Date object, which is displayed by calling toString() on this object.

1. > var date = new Date();
2. undefined
4. > date;
5. Wed Apr 12 2017 11:10:28 GMT+0200 (CEST)
6. > date.toString(); // same thing!
7. Wed Apr 12 2017 11:10:28 GMT+0200 (CEST)

**We can also pass it an argument that can be:**

* A string that encodes a date
* A set of numeric values separated by a comma for month, day, hour, and so on
* A Unix "timestamp"  (number of milliseconds elapsed since 1970)

... in this case it returns a date object that corresponds to the encoded date passed as argument.

Examples:

1. > new Date('2017 04 28');
2. Fri Apr 28 2017 00:00:00 GMT+0200 (CEST)
4. > new Date('2017 1 2');
5. Mon Jan 02 2017 00:00:00 GMT+0100 (CET)
7. > new Date('2017 1 2 8:30');
8. Mon Jan 02 2017 08:30:00 GMT+0100 (CET)

Numerical parameters can also be passed in this order: year, month (0-11), day (1-31), time (0-23), minutes (0-59), seconds , milliseconds (0-999). We do not have to pass everything but it should always be in this order.

Examples:

1. > new Date(2017, 3, 16, 14, 43, 10, 120);
2. Sun Apr 16 2017 14:43:10 GMT+0200 (CEST)
4. > new Date(2017, 0, 10, 14);
5. Tue Jan 10 2017 14:00:00 GMT+0100 (CET)
7. > new Date(2017, 1, 28) // 1 is February! Month indexes start at 0!
8. Tue Feb 28 2017 00:00:00 GMT+0100 (CET)
10. > new Date(2008, 1, 29);
11. Fri Feb 29 2008 00:00:00 GMT+0100 (CET)
13. > new Date(2017, 1, 29); // No February 29th in 2017! Gives 1st of March
14. Wed Mar 01 2017 00:00:00 GMT+0100 (CET)
16. > new Date(2017, 11, 31); // Happy new year!
17. Sun Dec 31 2017 00:00:00 GMT+0100 (CET)
19. > new Date(2017, 11, 32) // 32 Dec -> 1st of January!
20. Mon Jan 01 2018 00:00:00 GMT+0100 (CET)

One can build the date with a Unix timestamp (number of milliseconds since 1970):

1. > new Date(1199885822900);
2. Wed Jan 09 2008 14:37:02 GMT+0100 (CET)

Calling Date() without "new" returns the current date as a string. It does not matter if we pass parameters:

1. > Date();
2. "Sun Apr 16 2017 14:51:47 GMT+0200 (CEST)"

#### Useful methods

1. > var d = new Date();
2. undefined
4. > d.toString();
5. "Sun Apr 16 2017 14:52:52 GMT+0200 (CEST)"
7. > d.setMonth(2); // Change for month with index=2
8. 1489672372092
10. > d.toString();
11. "Thu Mar 16 2017 14:52:52 GMT+0100 (CET)"
13. > d.getMonth(); // get current month index
14. 2

#### Let's play with my birthday!

1. > var d = new Date(1965, 3, 16); // Michel Buffa's birthday
2. undefined
4. > d.getDay(); // Sunday is 0
5. 5
7. > d; // let's verify
8. **Fri** Apr 16 1965 00:00:00 GMT+0200 (CEST)
10. **> // Great, it was a Friday :-)**

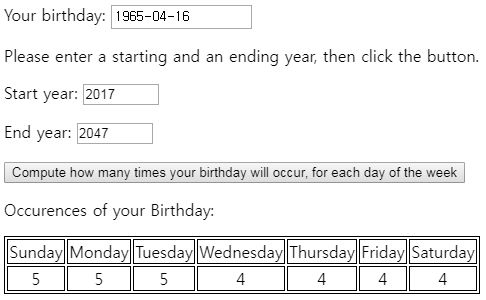
Let's write a small piece of code that will guess which days of the week Michel Buffa's birthday will occur, between 2017 and 2047:

1. > var dayOfTheWeek = [0,0,0,0,0,0,0];
3. for (var year = 2017; year <= 2047; year++) {
4. dayOfTheWeek[new Date(year, 4, 16).getDay()]++;
5. }
7. > dayOfTheWeek
8. [4, 4, 5, 5, 5, 4, 4] // 4 times on a Sunday, Monday, Friday and Saturday,  
                         // 5 times on Tuesday, Wednesday and Thursday

Explanations:

* Line 1 we use an array with each element being the number of times the birthday occurs on a Sunday, Monday, etc.
* Line 3: we iterate using a for loop on every year between 2017 and 2047.
* Line 4: we build a Date object using 16 of April, but change the year, we compute the date of each of Michel Buffa's birthdays between 2017 and 2045, and we get the index of the day (using the getDay() method). This index is used to increment corresponding elements of the array defined in line 1.
* Finally, line 7 displays the content of the array. Remember  that typing a variable name in the devtool console is equivalent to calling the object toString() method.

And here is a full version with input fields and results displayed in an HTML table:

****

****

|  |
| --- |
| <script>    function computeBirthdays() {      // An array. Each element is the number of times my birthday      // will occur. For the moment: 0 times on a Monday, 0 times on Friday      // etc.      var dayOfTheWeek = [0, 0, 0, 0, 0, 0, 0];      var birthday = document.querySelector("#birthday").value;      // birthday is the value of the input field,      // as a string (ex: "1965-4-16")      // Let's turn it into a Date object      var birthdayAsDate = new Date(birthday);      // Get the month and year (ex: 16 April)      var birthdayMonth = birthdayAsDate.getMonth(); // ex: April      var birthdayDate = birthdayAsDate.getDate();   // ex: 16      var startYear = document.querySelector("#start").value;      var endYear = document.querySelector("#end").value;      for (var year = startYear; year <= endYear; year++) {        var dayOfTheWeekMyBirthDayOccurs =          new Date(year, birthdayMonth, birthdayDate).getDay();        console.log('Year : ' + year + " Day of your birthday: " +          getDayName(dayOfTheWeekMyBirthDayOccurs));        // increment the counter for this day        dayOfTheWeek[dayOfTheWeekMyBirthDayOccurs]++;      }      // add a table to the web page, presenting the results      displayResults(dayOfTheWeek);    }    function getDayName(dayIndex) {      var days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday'];      return days[dayIndex];    }    function displayResults(array) {      document.querySelector("#results").innerHTML = "<p>Occurences of your Birthday:</p>";      var table = document.createElement("table");      var firstRow = table.insertRow();      var secondRow = table.insertRow();      array.forEach(function (dayOccurence, index) {        var dayNameCell = firstRow.insertCell(index);        dayNameCell.innerHTML = getDayName(index);        var nbCell = secondRow.insertCell(index);        nbCell.innerHTML = dayOccurence;      });      document.querySelector("#results").appendChild(table);    }  </script> |

* 1. **Discussion topics and projects**

Here is the discussion forum for this part of the course. Please either post your comments/observations/questions or share your creations.

See below for suggested topics of discussion and optional projects.

#### Suggested topics

* Did you notice that the Math class has only class methods and properties: you always use Math.PI, Math.cos(...), etc. Do class properties and methods make sense to you now? It would be nonsense to create two Math objects such as m1 = new Math(); m2 = new Math(); var result = m1.cos(0.5); ...
* For a long time, we've talked about "predefined JavaScript objects", not "classes" when we talked about Math, Date, Array, etc. This is because JavaScript is not a class-based programming language.  
  ES6 introduced classes and the class keyword, but in fact there are no "real classes" in JavaScript, like in class-based languages such as Java or C#. ES6 classes are just constructor functions and prototypes (the thing behind Object Oriented JavaScript) disguised. Did you know that?

1. **HTML5 tables, forms and input fields**
   1. **The HTML table basics**

### **The HTML table basics: tags, attributes and CSS styling**

#### Introduction

The <table> element helps with rendering tables in an HTML document.

Each table row is defined with the <tr> tag (**T**able **R**ow). A table header is defined with the <th> tag (**T**able **H**eader). By default, table headings are bold and centered. A table data/cell is defined with the <td> tag (**T**able **D**ata). In each cell, you can have other HTML elements/tags. You can have only "column table headers" (the first row of the table will be in bold), or you can also have "row headers" (first cell of each row).

**Best practice for making the table accessible:** always add a <caption> tag inside the <table> tag. Data tables very often have brief descriptive text before or after the table that indicates the content of that table. This text should be associated to its respective table using the <caption> element. The <caption> element must be the first thing after the opening <table> tag.

**Second best practice for accessibility:** use a scope attribute with all <th scope = "row or column"> for identifying whether a table header is a column header or a row header.

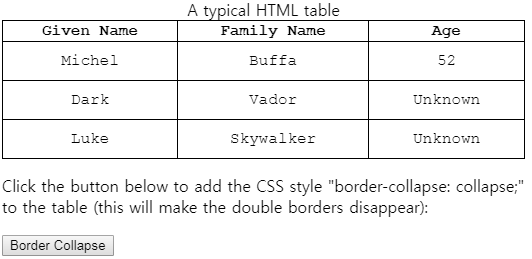
* You can read [these recommendations](https://www.w3.org/WAI/tutorials/tables/) for making accessible tables.

#### Typical example:

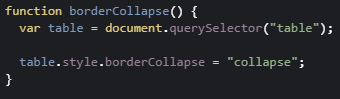
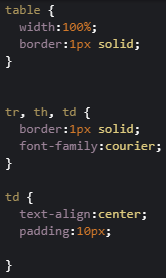
1. <table>
2. **<caption>A typical HTML table</caption>**
3. <tr>
4. <th **scope="col"**>Given Name</th>
5. <th **scope="col"**>Family Name</th>
6. <th **scope="col"**>Age</th>
7. </tr>
8. <tr>
9. <td>Michel</td>
10. <td>Buffa</td>
11. <td>52</td>
12. </tr>
13. <tr>
14. <td>Dark</td>
15. <td>Vador</td>
16. <td>Unknown</td>
17. </tr>
18. <tr>
19. <td>Luke</td>
20. <td>Skywalker</td>
21. <td>Unknown</td>
22. </tr>
23. </table>

Most of the time, we add some CSS rules for rendering cell/row/table borders and for adjusting spacing between the text in the cells and the cell borders. Let's look at another example:

#### HTML table with a very light CSS styling



|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="utf-8">    <title>A typical HTML table with simple styling</title>  </head>  <body>    <table>      <caption>A typical HTML table</caption>      <tr>        <th scope="col">Given Name</th>        <th scope="col">Family Name</th>        <th scope="col">Age</th>      </tr>      <tr>        <td>Michel</td>        <td>Buffa</td>        <td>52</td>      </tr>      <tr>        <td>Dark</td>        <td>Vador</td>        <td>Unknown</td>      </tr>      <tr>        <td>Luke</td>        <td>Skywalker</td>        <td>Unknown</td>      </tr>    </table>    <p>Click the button below to add the CSS style "border-collapse: collapse;"      to the table (this will make the double borders disappear):</p>    <button onclick="borderCollapse();">Border Collapse</button>  </body>  </html> |

****

This is a static table. You can look at the CSS code:

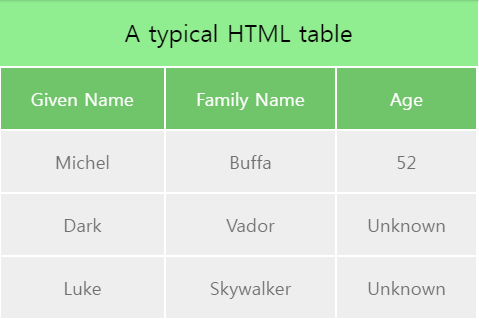
1. table {
2. width:100%;
3. border:1px solid;
4. }

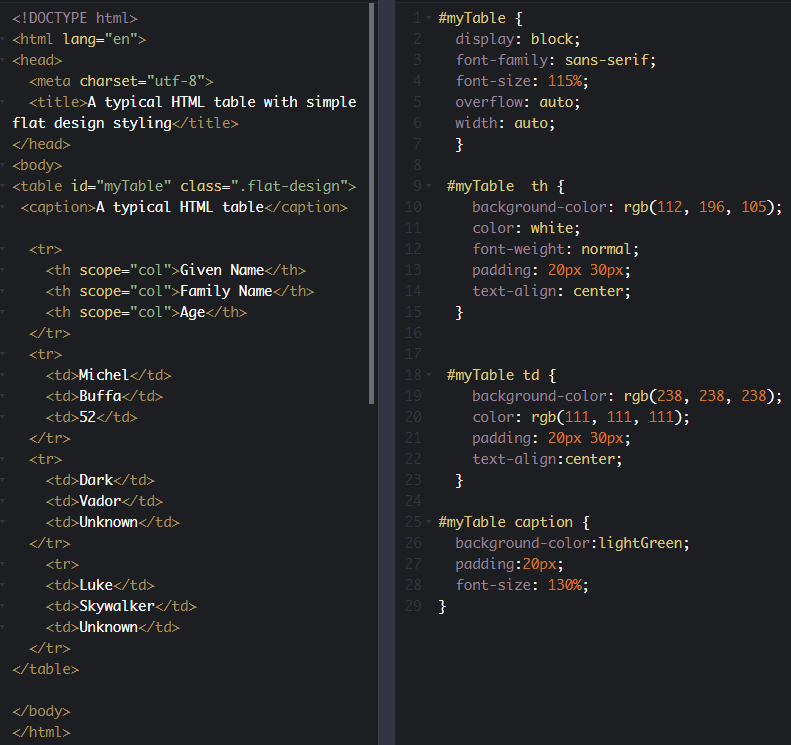
7. tr, th, td {
8. border:1px solid;
9. font-family:courier;
10. }
12. td {
13. text-align:center;
14. padding:10px;
15. }

**Explanations**:

* Line 1: this rule says that the table will occupy the width of the window and will have a black, continuous border that is one pixel wide.
* Line 7: this rule says that table rows, table cells and table headers will also have a border and will use the font family Courier.
* Line 12: this says that all cells will have the text horizontally centered and an internal margin (called padding) of 10px in each direction (top, bottom, left, right).

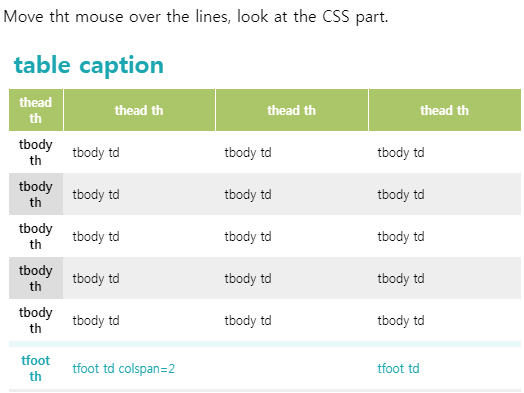
#### Here is another example with more CSS styling (flat design)

****

****

#### And a final example with colored lines, header, footer, legend

Look at the CSS - it's the only part that changed:

****

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="utf-8">    <title>A typical HTML table with colored lines and more CSS styling</title>  </head>   <body>     <p>Move tht mouse over the lines, look at the CSS part.</p>    <table>  <caption>table caption</caption>      <thead>          <tr>              <th scope="col">thead th</th>              <th scope="col">thead th</th>              <th scope="col">thead th</th>              <th scope="col">thead th</th>          </tr>      </thead>      <tbody>          <tr>              <th scope="row">tbody th</th>              <td>tbody td</td>              <td>tbody td</td>              <td>tbody td</td>          </tr>          <tr>              <th scope="row">tbody th</th>              <td>tbody td</td>              <td>tbody td</td>              <td>tbody td</td>          </tr>          <tr>              <th scope="row">tbody th</th>              <td>tbody td</td>              <td>tbody td</td>              <td>tbody td</td>          </tr>      <tr>              <th scope="row">tbody th</th>              <td>tbody td</td>              <td>tbody td</td>              <td>tbody td</td>          </tr>      <tr>              <th scope="row">tbody th</th>              <td>tbody td</td>              <td>tbody td</td>              <td>tbody td</td>          </tr>      </tbody>          <tfoot>          <tr>              <th scope="row">tfoot th</th>              <td colspan="2">tfoot td colspan=2</td>              <td>tfoot td</td>          </tr>      </tfoot>  </table>  </body>  </html> |
| table {    width:98%;    margin:0 auto;    border-bottom:3px #eee solid;    border-collapse:collapse;    font-size:12px  }  th, td {    padding:5px 10px  }  caption {    font-weight:bold;    font-size:24px;    padding:5px;    color:#1BA6B2;    text-align:left;    margin-bottom:5px  }  thead th {    background:#ABC668;    color:#fff;    border-right:1px solid #fff  }  thead th:last-child {    border-right:none  }  tbody th {    width:10%  }  tbody tr:nth-child(even) {    background:#eee  }  tbody tr:nth-child(even) th {    background:#ddd  }  tbody tr:hover {    background:#F3F5BB  }  tbody tr:hover th {    background:#F2F684;    color:#1BA6B2  }  tfoot tr {    border-top:6px solid #E9F7F6;    color:#1BA6B2  } | |

* 1. **The HTML table JavaScript API: dynamic tables!**

**[ -- Live coding video: HTML table JavaScript API -- ]**

### **The HTML table JavaScript API: dynamic tables!**

#### Introduction

There is a JavaScript API associated with the HTML table elements that makes dynamic table management possible, enabling you to add or delete a row, add or delete a cell, modify the content of the cells, etc.

We've already seen some examples in the course, but we have not completely covered the table JavaScript API.

#### The Table object (<table>)

When you look for a table using the DOM API or the selector API, or when you create a table using the DOM API, you get a Table object:

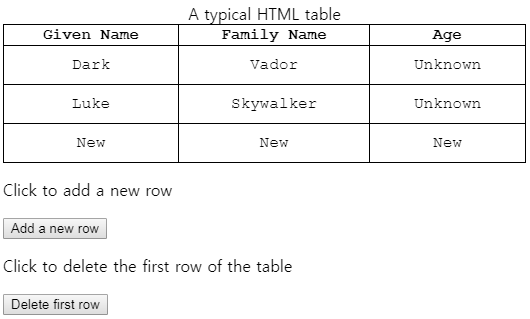
1. var table = document.getElementById("myTable");
3. var table = document.querySelector("#myTable");
5. var table = document.createElement("table"); // creates a new table

Like all objects, an instance of Table will have properties and methods:

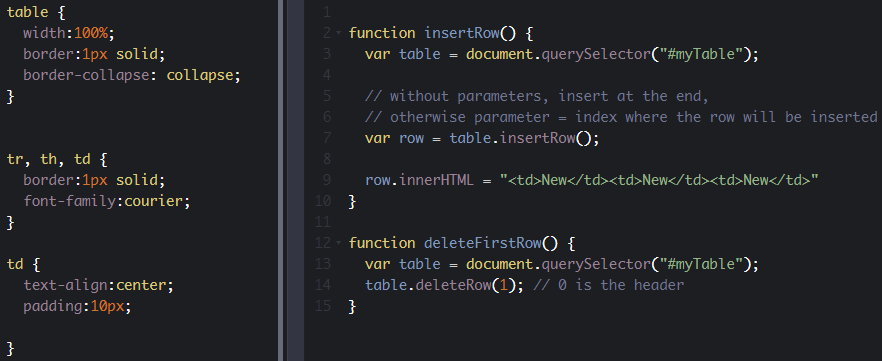
|  |  |
| --- | --- |
| Most useful properties | |
| **rows** | Returns a collection of all **<tr>** elements in a table |
| **caption** | Returns the **<caption>** element of a table |
| **tFoot** | Returns a reference to the **<tfoot>** element of a table |
| **tHead** | Returns a reference to the **<thead>** element of a table |

|  |  |
| --- | --- |
| Most useful methods | |
| **insertRow()** | Creates an empty **<tr>** element and adds it to the table. Example: **var row = table.insertRow();** inserts a new row at the end of the table. **var row = table.insertRow(0);** inserts at index = 0, **var row = table.insertRow(10);** inserts at index = 10, and pushes all the rows after this index. |
| **deleteRow()** | Removes a row (**<tr>**) from the table. Example **table.deleteRow(0)**; deletes the row at index 0. |
| **createCaption()** | Creates an empty **<caption>** element and adds it to the table |
| **deleteCaption()** | Removes the first **<caption>** element from the table |
| **createTHead()** | Creates an empty **<thead>** element and adds it to the table |
| **deleteTHead()** | Removes the **<thead>** element from the table |
| **createTFoot()** | Creates an empty **<tfoot>** element and adds it to the table |
| **deleteTFoot()** | Removes the **<tfoot>** element from the table |

##### Example that adds a new row or removes a row to/from a table using the insertRow() / deleteRow() methods:

****

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="utf-8">    <title>A typical HTML table with simple styling</title>  </head>  <body>    <table id="myTable">      <caption>A typical HTML table</caption>      <tr>        <th scope="col">Given Name</th>        <th scope="col">Family Name</th>        <th scope="col">Age</th>      </tr>      <tr>        <td>Michel</td>        <td>Buffa</td>        <td>52</td>      </tr>      <tr>        <td>Dark</td>        <td>Vador</td>        <td>Unknown</td>      </tr>      <tr>        <td>Luke</td>        <td>Skywalker</td>        <td>Unknown</td>      </tr>    </table>    <p>Click to add a new row</p>    <button onclick="insertRow();">Add a new row</button>    <p>Click to delete the first row of the table</p>    <button onclick="deleteFirstRow();">Delete first row</button>  </body>  </html> |

****

Notice the use of row.innerHTML= here to add some cells to the row. We will soon see another method for doing this.

#### The TableRow object (<tr>)

When you look for a row using the DOM API or the selector API, or when you create a row using the DOM API, you get a Row object:

1. var row1 = document.getElementById("row1");
3. var row1 = document.querySelector("#row1");
5. var newRow = document.createElement("row"); // creates a new row

You can also access a row from the rows property of a table:

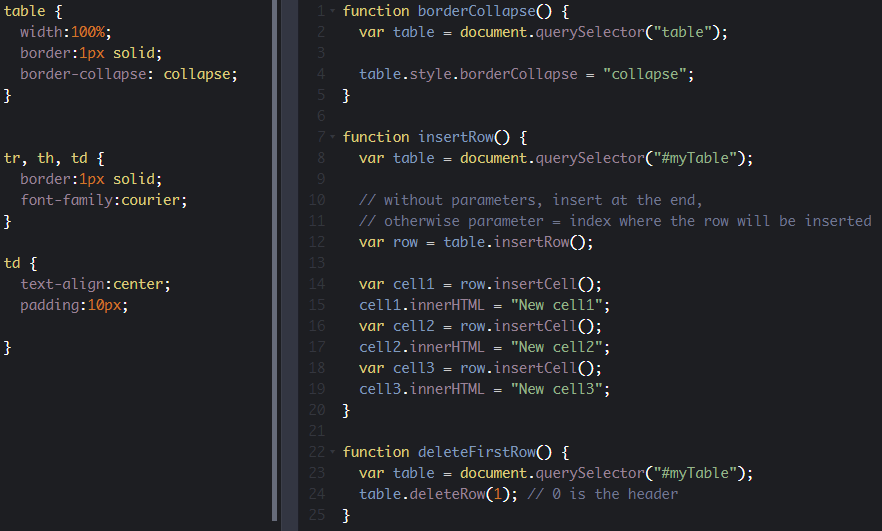
1. > var t = document.createElement("table");
2. undefined
4. > var r1 = t.insertRow(0);
5. undefined
7. > r1.innerHTML="<td>Hello</td>";
8. "<td>Hello</td>"
10. > var r2 = t.insertRow();
11. undefined
13. > r2.innerHTML="<td>Hello 2</td>";
14. "<td>Hello 2</td>"
16. > **var row1 = t.rows[0];**
17. undefined
19. > row1;
20. <tr><td>Hello</td></tr>

Like all objects, a TableRow object has properties and methods. Here are the most useful ones:

|  |  |
| --- | --- |
| Most useful properties | |
| **cells** | Returns a collection of all <**td>** or **<th>** elements in a table row |
| **rowIndex** | Returns the position of a row in the **rows** collection of a table |
| **sectionRowIndex** | Returns the position of a row in the **rows** collection of a **<tbody>**, **<thead>**, or **<tfoot>** |

|  |  |
| --- | --- |
| Most useful methods | |
| **insertCell()** | Inserts a cell into the current table row. Without parameters, appends a cell after the last cell of the row. You can pass the index of the cell as a unique parameter, in which case other cells are "pushed" to the right. The value of 0 results in the new cell being inserted at the first position. The value of -1 can also be used, which results in the new cell being inserted in the last position. |
| **deleteCell()** | Deletes a cell from the current table row. There is one parameter for this method: the index of the cell to remove. The value of 0 results in the deletion of the first cell. The value of -1 can also be used, which results in the deletion of the last cell. |

##### New versions of the previous examples, but instead of using the innerHTML of the TableRow object, we use the insertCell() method.



Notice how we've created the new row cells:

1. function insertRow() {
2. var table = document.querySelector("#myTable");
3. // without parameters, insert at the end,
4. // otherwise parameter = index where the row will be inserted
5. var row = table.insertRow();
6. **var cell1 = row.insertCell();**
7. **cell1.innerHTML = "New cell1";**
8. **var cell2 = row.insertCell();**
9. **cell2.innerHTML = "New cell2";**
10. **var cell3 = row.insertCell();**
11. **cell3.innerHTML = "New cell3";**
12. }

So use insertCell() or just row.innerHTML="<td>...</td>" ? It's up to you: depending on the HTML that you plan to insert into each cell, one version may be more readable than the other.

* 1. **HTML forms: best practices**

#### 1 - Creating accessible forms

Forms are commonly used to enable user interaction within Web sites and Web applications, for example, for login, registering, commenting, and purchasing.

Since HTML5 provides functionalities to assist with accessibility, developers should make a concerted effort to mark up Web based forms. The following two guidelines are to give you a good start to make your forms accessible:

1. For every form field, ensure that a descriptive **label** is provided and use the <label> element to identify each form control.
2. For larger or complex forms, use the <fieldset> and <legend> elements to respectively **group and associate** related form controls.

We will give usage examples for each of these two basic guidelines in the following pages.

##### Further reading

The WAI Web site hosts a [Forms tutorial](http://www.w3.org/WAI/tutorials/forms/) where you will find more guidelines to help make your forms truly accessible: Form Instructions, Validating Input, User Notifications, Multi-Page Forms, and Custom Controls.

#### 2 - Why is this important?

Forms can be visually and cognitively complex and difficult to use. Accessible forms are easier to use for everyone, including people with disabilities.

* **People with cognitive disabilities** can more easily understand the form and how to complete it, as making forms accessible improves the layout structure, instructions, and feedback.
* **People using speech input** can use the labels via voice commands to activate controls and move the focus to the fields that they need to complete.
* **People with limited dexterity** benefit from large clickable areas that include the labels, especially for smaller controls, such as radio buttons and checkboxes.
* **People using screen readers** can identify and understand form controls more easily because they are associated with labels, field sets, and other structural elements.

#### 3 - Labeling controls

##### Labels need to describe the purpose of the form control

Form fields and other form controls usually have visible labels, such as "E-mail Address:" as the label for a text field (see figure below).

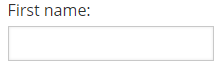
****

When these labels are marked up correctly, people can interact with them using only the keyboard, using voice input, and using screen readers. Also, the label itself becomes clickable, which enables a person who has difficulty clicking on small radio buttons or checkboxes to click anywhere on the label text.

##### Associating labels explicitly

Whenever possible, use the label element to explicitly associate text with form elements. The for attribute of the label must exactly match the id of the form control.

###### Example 1 (click on the label, not on the input field to see the effect)



양식의 맨 위

Source code:

1. 양식의 맨 아래
2. <label for="first\_name">Your First Name</label>
3. <input id="first\_name" type="text" name="fname"/>

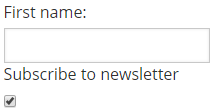
###### Alternative example 1

Note that you can also include the <input> element inside the <label>...</label> element, and also add a <span lang="en"> for example, to indicate the language used in the label. Sometimes, [nesting labels and inputs can also make CSS styling easier and produce better results with screen readers](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Forms/How_to_structure_an_HTML_form).

Source code (with <input> inside the <label>):

1. **<label for="first\_name"><span lang=en">**Your First Name**</span>**
2. <input id="first\_name" type="text" name="fname"/>
3. **</label>**

###### Example 2 (click on the label "Subscribe to newsletter" to see the effect)

****

양식의 맨 위

Source code:

1. 양식의 맨 아래
2. <label for="firstname">First name:</label>
3. <input type="text" name="firstname" id="firstname"><br>
4. <label for="subscribe">Subscribe to newsletter</label>
5. <input type="checkbox" name="subscribe" id="subscribe">

#### 4 - Labeling buttons

The label of a <button> element is set inside the element and can include markup. This allows advanced accessibility hints to be included, such as marking up language change.

Example: <button>Mon <span lang="fr">bouton</span></button>, for a button with a label in French.

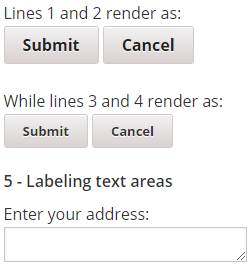
When using the <input> element to create buttons, the label is set in the value attribute of the element.

Example: <input type="submit" value="Please submit">, will be rendered as a button.

Source code for an example of "Submit" and "Cancel" buttons:

1. <button type="submit">Submit</button>
2. <button type="button">Cancel</button>
4. <input type="submit" value="Submit">
5. <input type="button" value="Cancel">

These will produce the same results:

****

Source code:

1. <label for="address">Enter your address:</label>
2. <br>
3. <textarea id="address" name="addresstext"></textarea>

#### 6 - Grouping controls

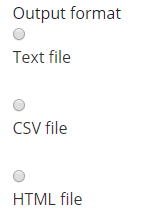
Groupings of form controls, typically groups of related checkboxes and radio buttons, sometimes require a higher level description. Grouping related form controls makes forms more understandable for all users, as related controls are easier to identify.

##### Associating related controls with fieldset

Grouping needs to be carried out visually and in the code, for example, by using the <fieldset> and <legend> elements to associate related form controls. The <fieldset> identifies the entire grouping and <legend> identifies the grouping's descriptive text.

##### Example 1 - Radio buttons

In the example below, there are three radio buttons that allow the user to choose an output format. Radio button groups should always be grouped using <fieldset>.

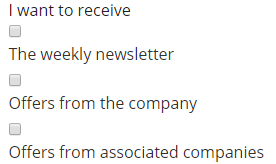
****

Source code:

1. <fieldset>
2. <legend>Output format</legend>
3. <div>
4. <input type="radio" name="format" id="txt" value="txt" checked>
5. <label for="txt">Text file</label>
6. </div>
7. <div>
8. <input type="radio" name="format" id="csv" value="csv">
9. <label for="csv">CSV file</label>
10. </div>
11. […]
12. </fieldset>

##### Example 2 - Checkboxes

In the example below, there are three checkboxes that are all part of an opt-in function for receiving different types of information.

****

Source code:

1. <fieldset>
2. <legend>I want to receive</legend>
3. <div>
4. <input type="checkbox" name="newsletter" id="check\_1">
5. <label for="check\_1">The weekly newsletter</label>
6. </div>
7. […]
8. </fieldset>

#### [ADVANCED] Associating related controls with WAI-ARIA

WAI-ARIA provides a grouping role that functions in a similar way to fieldset and legend. For example, a div element can have role=group to indicate that the contained elements are members of a group.

WAI-ARIA roles are very important in the accessibility world, and we invite you to see an example provided in the [associated WAI tutorial](http://www.w3.org/WAI/tutorials/forms/grouping/). This article by Oscar Cao gives an [Introduction to understanding WAI-ARIA 1.0 roles](http://oscarcao.com/blog/2015/06/08/introduction-to-understanding-wai-aria-1-0-roles/).

* 1. **HTML forms and JavaScript**

**[ -- Live coding video: HTML Forms Best Practices -- ]**

### **HTML forms and JavaScript**

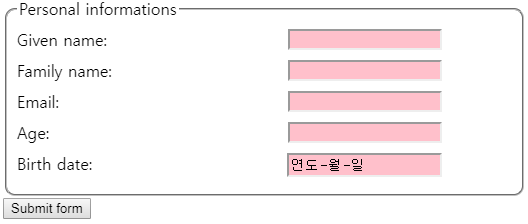
#### Introduction

We highly recommend that you follow the [HTML5 Coding essentials and Best Practices course on W3Cx](https://www.edx.org/course/html5-coding-essentials-w3cx-html5-1x-2) that has an entire module dedicated to HTML5 forms.

Forms are a way to get user input which is either sent to a remote server, or processed locally, or both.

This section of the course will only cover local processing and the client-side part, with a focus on JavaScript processing.

#### Typical example

****

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <meta charset="utf-8">    <title>Form example</title>  </head>  <body>    <form id="myForm">      <fieldset>        <legend>Personal informations</legend>        <label for="givenName">Given name:</label>        <input type="text" id="givenName" required name="givenName">        <br>        <label for="familyName">Family name:</label>        <input type="text" id="familyName" required name="familyName">        <br>        <label for="email">Email:</label>        <input type="email" id="email" required name="email">        <br>        <label for="age">Age:</label>        <input type="number" min=0 max=120 step=5 id="age" required name="age">        <br>        <label for="date">Birth date:</label>        <input type="date" id="date" required name="date">      </fieldset>      <button>Submit form</button>    </form>  </body>  </html> |
| fieldset {    padding:10px;    border-radius:10px;  }  label {    display:inline-block;    margin-bottom:10px;  }  input {    float:right;    margin-right:70px;    width:150px;  }  input:invalid {    background-color:pink;  }  input:valid {    background-color:lightgreen;  } | |

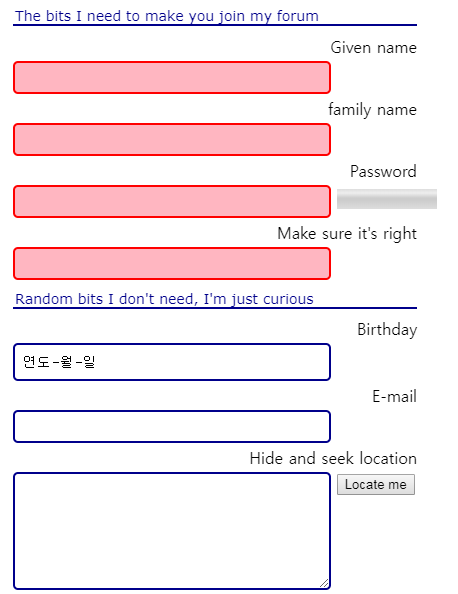
#### HTML form input can be sent to a server without JavaScript

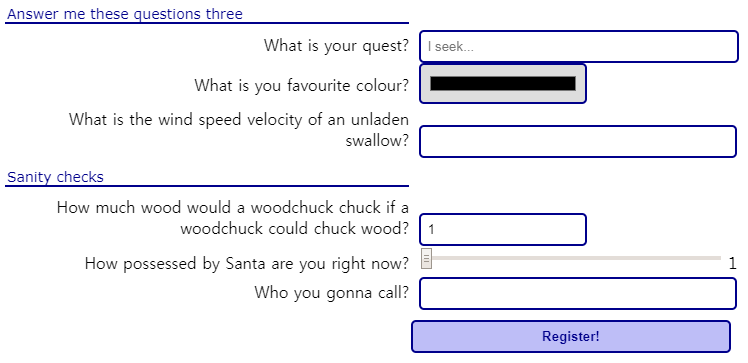
If a form's content is sent to a remote server, on the server side, you may have PHP, Java, C#, Ruby, Python, etc. components. There are several ways to collect server-side data from a form in a Web page: REST Web services, servlets, Microsoft ASP pages, etc.

On the client side, the forms indicate to which server and how the data should be sent,  using the action and method attributes respectively. A <button type="submit"> or an <input type=submit> field is used to submit the form content.

For example: <form action="myServerCode.php" method="POST">...</form>. Here, we set the URL of the server side code (myServerCode.php), and the HTTP method that will be used by the browser for sending the form content (POST).

Example of HTML5 form that will not be sent if invalid input fields are present. Notice that the JavaScript part is only used for giving feedback while entering the password. No JavaScript is used for sending the form data, or for complex, global validation:

****

****

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>    <title>Example of HTML5 form</title>    <meta charset="utf-8">  </head>  <body>    <form action="forum.cfm" method="post">      <fieldset>        <legend>The bits I need to make you join my forum</legend>        <label for="givenName">Given name</label>        <input required type="text" name="givenName" id="givenName" autocomplete="off" autofocus />        <br />        <label for="familyName">family name</label>        <input required type="text" name="familyName" id="familyName" autocomplete="off" />        <br />        <label for="passwordA">Password</label>        <input required type="password" name="passwordA" id="passwordA" oninput="checkpasswords()" />        <meter id="passwordStrength" min="0" max="10" value="0" low="5"></meter>        <br />        <label for="passwordB">Make sure it's right</label>        <input required type="password" name="passwordB" id="passwordB" oninput="checkpasswords()" />        <br />      </fieldset>      <fieldset>        <legend>Random bits I don't need, I'm just curious</legend>        <label for="dob">Birthday</label>        <input type="date" name="dob" id="dob">        <br />        <label for="email">E-mail</label>        <input type="email" name="email" id="email" autocomplete="off" />        <br />        <label for="whereami">Hide and seek location</label>        <textarea name="whereami" id="whereami"></textarea>        <button id="findme" type="button">Locate me</button>        <br />      </fieldset>      <fieldset>        <legend>Answer me these questions three</legend>        <label for="quest">What is your quest?</label>        <input name="quest" id="quest" type="search" placeholder="I seek..." autocomplete="off" />        <br />        <label for="colour">What is you favourite colour?</label>        <input type="color" id="colour" name="colour" oninput="this.style.backgroundColor = this.value);" />        <br />        <label for="swallow">What is the wind speed velocity of an unladen swallow?</label>        <input list="swallows" id="swallow" name="swallow" />        <datalist id="swallows">          <option value="African or European?"></option>        </datalist>        <br />      </fieldset>      <fieldset>        <legend>Sanity checks</legend>        <label for="chuck">How much wood would a woodchuck chuck if a woodchuck could chuck wood?</label>        <input type="number" name="chuck" id="chuck" min="1" max="15" step="1" value="1" />        <br />        <label for="santa">How possessed by Santa are you right now?</label>        <input name="santa" id="santa" type="range" min="1" max="12" step="1" value="1" />        <output name="so" id="santaoutput">1</output>        <br />        <label for="call">Who you gonna call?</label>        <input list="callingoptions" id="call" name="call" pattern="Ghostbusters|[0-9]\*"          oninput="validateCallingOptions();" />        <datalist id="callingoptions">          <option value="Ghostbusters"></option>        </datalist>      </fieldset>      <input type="submit" value="Register!" />    </form>  </body>  </html> |
| input:invalid:required,  input:invalid {    background-color: #FFB6C1;    border: 2px solid #FF0000;  }  form {    margin: 20px;  }  fieldset {    border: none;  }  legend {    border-bottom: 2px solid #00008B;    color: #00008B;    font-family: Verdana;    font-size:14px;    width: 400px;  }  label {    display: inline-block;    padding: 4px;    text-align: right;    width: 400px;  }  input[type=text],  input[type=password],  input[type=date],  input[type=email],  input[type=search],  input[list],  input[type=color],  input[type=number],  input[type=submit],  input[type=button],  textarea {    border: 2px solid #00008B;    border-radius: 5px;    padding: 7px;    width: 300px;  }  textarea {    height: 100px;    vertical-align: top;  }  input[type=search] {    width: 320px;  }  input[type=color] {    cursor: pointer;    width: 150px;  }  input[type=number] {    width: 150px;  }  input[type=range] {    width: 300px;  }  meter {    height: 20px;    width: 100px;  }  input[type=submit] {    background-color: #BEBEF7;    color: #00008B;    cursor:pointer;    font-weight: bold;    margin-left: 420px;    width: 320px;  }  input[type=button] {    background-color: #EFBDB8;    border-color: #8B0000;    color: #8b0000;    cursor:pointer;    font-weight: bold;    margin-left: 420px;    width: 320px;  } |
| function checkpasswords() {    var p1 = document.getElementById("passwordA");    var p2 = document.getElementById("passwordB");    if (p1.value !== p2.value) {      p2.setCustomValidity("Passwords do not match");    } else {      p2.setCustomValidity("");    }    var strength = document.getElementById("passwordStrength");    var multiplier = 0;    if (/[a-z]/.test(p1.value)) { multiplier++; }    if (/[A-Z]/.test(p1.value)) { multiplier++; }    if (/[0-9]/.test(p1.value)) { multiplier++; }    strength.value = p1.value.length \* multiplier;  }  function validateCallingOptions() {    var input = document.getElementById("call");    var validity = input.validity;    if (validity.patternMismatch) {      input.setCustomValidity("Not a phone number");    } else {      input.setCustomValidity("");    }  } |

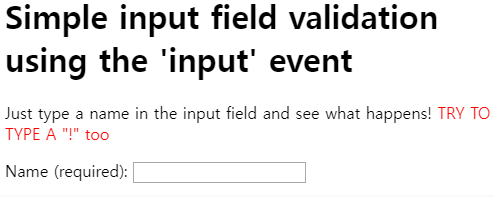
#### HTML form input can be sent using Ajax / JavaScript

Another approach is to use JavaScript for sending the form content with Ajax. This will be covered in the JavaScript advanced course, to be be available on W3Cx.

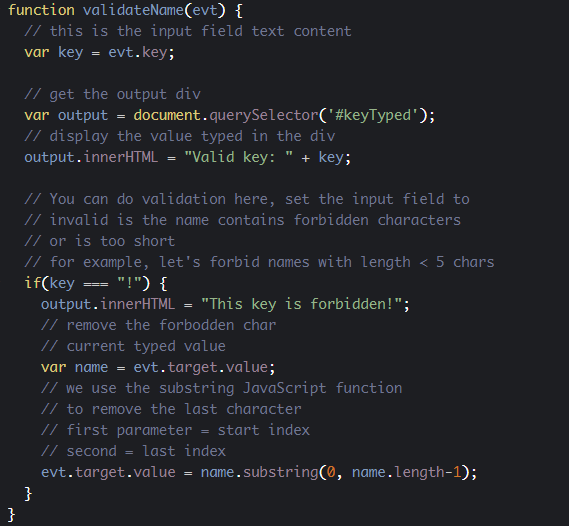
#### JavaScript can be used for validating user input "on the fly"

While one is typing or selecting a color, or moving a slider, JavaScript event listeners can be used to track the user's interactions in real time, and perform some validation steps along with giving visual feedback.

We've already seen how we can track the keys typed in an input field in real time:

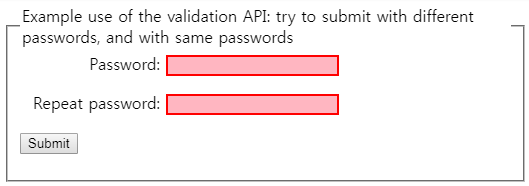
****

****

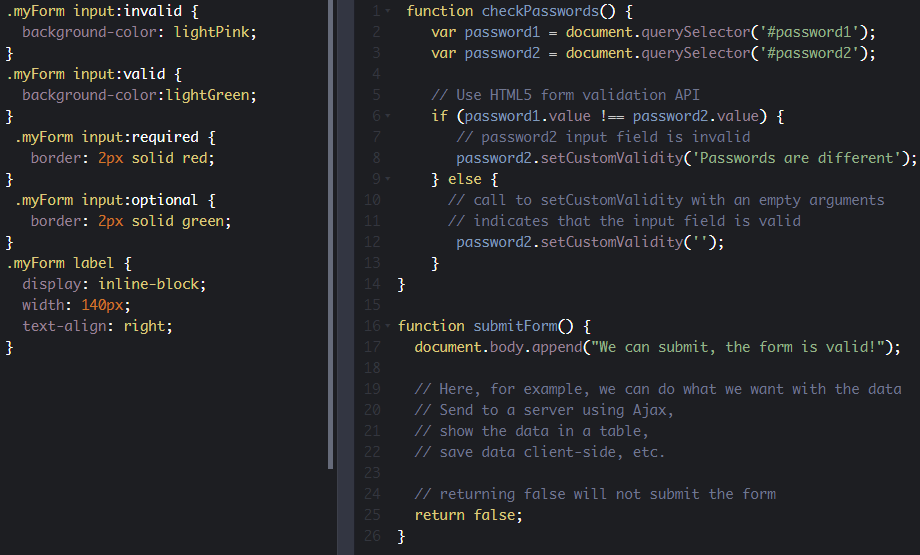
****

#### JavaScript can be used for a more global validation before sending a form to a remote server

Example: checking that a password entered twice is identical in two different input fields, that some values are coherent (e.g. a birthday cannot be in the future), etc.

****

****

****

#### JavaScript can be used to make a WebApp that uses form data locally, perhaps with some client-side persistence API

For example, a contact manager that will work offline, saving data locally, in a database inside the browser. Data will be displayed in a dynamic HTML table, without the need for a remote database.

This is the small project we will build together at the end of the course. :-)

* 1. **Discussion and projects**

Here is the discussion forum for this part of the course. Please either post your comments/observations/questions or share your creations.

See below for suggested topics of discussion and an optional project.

#### Suggested topics

* HTML5 tables are some of the most complicated elements, and there are lots of tricks to make them nice and reactive using CSS3 styles. Please share in the forum some of the best looking tables you find.
* There exist external JavaScript libraries for making "datatables", i.e., HTML tables especially made for displaying structured data. Do you know some of them and can you share your experiences and examples that use them in the forum? What do you think of them?

#### Optional projects

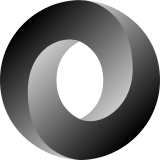
* Add a search input field + a search button to the dynamic table example. Add some more data in the table. Implement a search feature: when you search for "Ian Solo", for example, highlight the table row that contains it. If not found, display a message in the page, next to the search form.
* Please make a big table (with a few hundred rows containing structured data). Tip: use loops and random values. For example, use an array of names, an array of cities, an array of zip codes; use Math.random()  and Math.round() to generate random indexes. Then pick data from the arrays, build an object with random names, cities, zip codes, etc. and add it as a row in the table.
* Ah, but this table looks too long now!!! Try to paginate it by using a previous and a next page button. You will display the table with 15 rows per page, not more!



1. **The JSON notation**
   1. **What is JSON?**

**[ -- Live coding video: JSON notation, working with LocalStorage and remote data -- ]**

### What is JSON?

JSON stands for**J**ava**S**cript **O**bject **N**otation. It's a standard for transforming nearly any object into a string representation that is human readable. It became a standard for exchanging data to/from a remote HTTP server, and is available for many other languages in addition to JavaScript.

A JavaScript object o in JSON looks a lot like what o.toString() returns.

**There are two main methods to know:**

1. Transform any JavaScript object in JSON:                      **var jsonStr = JSON.stringify(obj);**
2. Transform any JSON string into a JavaScript object:    **var jsObj  = JSON.parse(jsonStr);**

#### Let's see some examples:

1. > var x = 3;
2. undefined
4. >**JSON.stringify(x);**
5. "3"
7. > var simpleObject = {x:12, y:30};
8. undefined
10. >**JSON.stringify(simpleObject);**
11. "{"x":12,"y":30}"
13. > var anArray = ['Monday', 'Tuesday', 'Wednesday'];
14. undefined
16. >**JSON.stringify(anArray);**
17. "["Monday","Tuesday","Wednesday"]"
19. > var complexObject = {name:'Metallica',
20. albums:[
21. {name:"Master of Puppets", year:1986},
22. {name:"Black Album", year:1991}
23. ]
24. };
25. undefined
27. >**JSON.stringify(complexObject);**
28. "{"name":"Metallica","albums":[{"name":"Master of Puppets","year":1986},{"name":"Black Album","year":1991}]}"

In the above examples, you can see JSON representations of a simple variable of a predefined type, of an array, of a simple object, of an object that contains an array of objects (Metallica example).

And indeed, it looks like the code you typed to create the objects, with quotes around it and around the property names. This is why it is called JavaScript Object Notation ;-)

#### You cannot use JSON objects as JavaScript objects

The JSON representation of JavaScript objects is a string. JSON has been developed mainly for replacing XML as a format for exchanging data between a client and a remote HTTP server. It has become very popular as the format for exchanging data when a Web Application uses Ajax for its communications with the HTTP server. Ajax is a way to send / receive data in the background, without the need to reload the Web page. Along with the DOM API, the appearance of this technology in 2005 with Internet Explorer, made it possible to make Web pages more dynamic. Google Maps was one of the first popular Ajax-powered Web application: as you moved the map, new parts arrived (downloaded in the background from the Gmap HTTP server), and the page updated to display these new parts.

JSON is also very practical for storing objects where strings are expected. There is a data store in your browser called LocalStorage that can be used as a small database for Web applications, but it stores only pairs of key/values in the string format. If you want to save a JavaScript object, you will have to turn it into JSON, then save it. When you read it from the data store, you will need to turn it back from JSON to JavaScript.

Here is a first example that turns an object into JSON and back into a JavaScript object:

1. > var metallica = {name:'Metallica', albums:[{name:"Master of Puppets", year:1986},  
          {name:"Black Album", year:1991}]};
2. undefined
3. > var metallicaJSON = JSON.stringify(metallica);
4. undefined
5. > metallicaJSON;
6. "{"name":"Metallica","albums":[{"name":"Master of Puppets","year":1986},  
          {"name":"Black Album","year":1991}]}"
7. > metallicaJSON.name; // metallicaJSON is not a JavaScript object
8. undefined
9. > metallica.name; // metallica is an object
10. "Metallica"
11. > var obj = JSON.parse(metallicaJSON); // JSON -> object
12. undefined
13. > obj.name; // this is an object
14. "Metallica"

With the JSON representation of an object you cannot access the original object's properties using the "." operator, nor call its methods. **The JSON format only stores the list of the object properties (name and value) as a string.** Look at line 10: we cannot access the name property of the JSON representation of the metallica object defined at line 1.

When we parse a JSON string using JSON.parse(), we get a real JavaScript object, and we can access its properties (lines 16 and 19).

* 1. **Example: consuming JSON remote data**

**[ -- Live coding Video: Xhr2, remote JSON data, dynamic table -- ]**

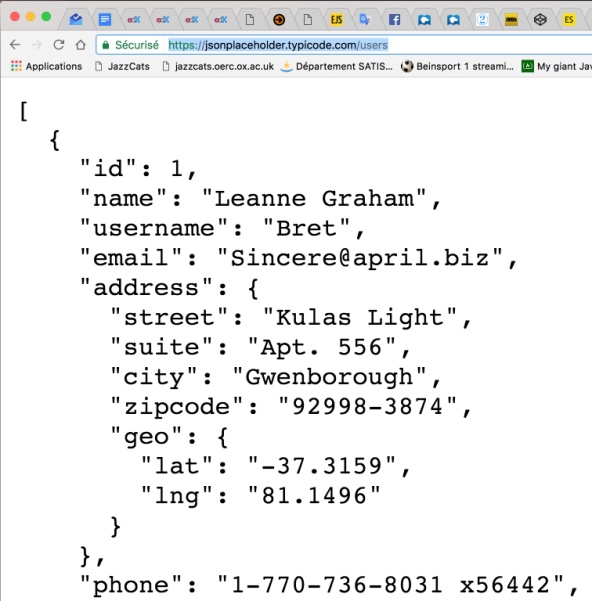
### **Example: consuming remote data**

#### JSON data from a REST WebService

Most "big sites" provide what we call a REST API. This means "they propose to send/receive data to/from programs over HTTP", and most of the time the JSON format is one of the possible transport formats for the data. Google APIs, Facebook and Amazon APIs are like this.

JSONPlaceholder is a free online REST service that you can use whenever you need some fake data in JSON. Faking a server is great for tutorials, and this is exactly what the next example does. It will consume data from [this URL](https://jsonplaceholder.typicode.com/users).

Please click on it - you will see some JSON data coming from the server and being displayed in your browser:



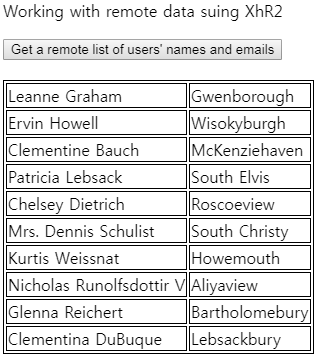
And we would like to use these data in our code, manipulating them as a JavaScript object.

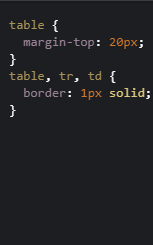
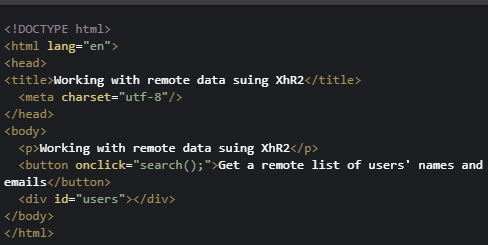
#### First thing: get the remote data as JSON

This course will not cover Ajax and what we call "asynchronous JavaScript". This will be covered in an advanced course to come on W3Cx.

However, we can show you two simple examples that use the Xhr2 API for Ajax requests and the new fetch API that is simplest to use.

#### Downloading JSON data using the Xhr2 API



****

|  |
| --- |
| function search() {    var queryURL = "https://jsonplaceholder.typicode.com/users";    var xhr = new XMLHttpRequest();    xhr.open('GET', queryURL, true);    // called when the response is arrived    xhr.onload = function (e) {      var jsonResponse = xhr.response;      // turn the response into a JavaScript object      var users = JSON.parse(jsonResponse);      displayUsersAsATable(users);    }    // in case of error    xhr.onerror = function (err) {      console.log("Error: " + err);    }    // sends the request    xhr.send();  }  function displayUsersAsATable(users) {    // users is a JavaScript object    // empty the div that contains the results    var usersDiv = document.querySelector("#users");    usersDiv.innerHTML = "";    // creates and populate the table with users    var table = document.createElement("table");    // iterate on the array of users    users.forEach(function (currentUser) {      // creates a row      var row = table.insertRow();      // insert cells in the row      var nameCell = row.insertCell();      nameCell.innerHTML = currentUser.name;      var cityCell = row.insertCell();      cityCell.innerHTML = currentUser.address.city;    });    // adds the table to the div    usersDiv.appendChild(table);  } |

JavaScript source code extract:

1. function search() {
2. var queryURL = "https://jsonplaceholder.typicode.com/users";
3. var xhr = new XMLHttpRequest();
4. xhr.open('GET', queryURL, true);
6. // called when the response has arrived
7. xhr.onload = function(e) {
8. **var jsonResponse = this.response;**
10. **// turn the response into a JavaScript object**
11. **var users = JSON.parse(jsonResponse);**
12. displayUsersAsATable(users);
13. }
14. // in case of error
15. xhr.onerror = function(err) {
16. console.log("Error: " + err);
17. }
18. // sends the request
19. xhr.send();
20. }

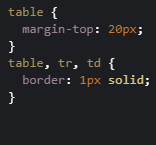
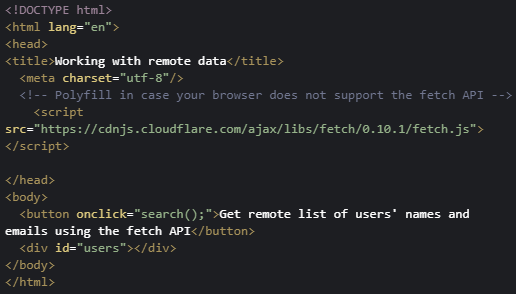
Explanations:

* Lines 4 and 5 build an Ajax request using XhR2.
* Line 22 is executed after: the request is sent in the background (we say "asynchronously").
* Line 8: when the server answers, this callback is executed, and inside it, this.response corresponds to the response from the HTTP server. It's in JSON format (line 9)
* Line 12: we turn the JSON response into a regular JavaScript object we can work with, using JSON.parse().
* Line 13: we pass this list of users, now a JavaScript object, to the displayUsersAsATable method, that will use the HTML table API we saw earlier in the course.

#### [advanced] Downloading JSON data using the fetch API

The [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a global [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/GlobalFetch/fetch) method that provides an easy, logical way to fetch resources asynchronously across the network. You **fetch** data from a URL, **then**, you do something with the response, **then** you do something else. If there is an error you can **catch** this error and display, for example, an error message.

See [this blog post](https://davidwalsh.name/fetch) for a detailed tutorial. Asynchronous JavaScript and JavaScript promises (the fetch...then...then... is based on the concept of "promises"), will be detailed in a next MOOC to appear at W3Cx.

****

|  |
| --- |
| function search() {    var queryURL = "https://jsonplaceholder.typicode.com/users";    fetch(queryURL)      .then(function (response) {        // response.json() returns a json string,        // returning it will convert it        // to a pure JavaScript        // object for the next then's callback        return response.json();      })      .then(function (users) {        // users is a JavaScript object here        displayUsersAsATable(users);      })      .catch(function (error) {        console.log('Error during fetch: ' + error.message);      });  }  function displayUsersAsATable(users) {    // users is a JavaScript object    // empty the div that contains the results    var usersDiv = document.querySelector("#users");    usersDiv.innerHTML = "";    // creates and populate the table with users    var table = document.createElement("table");    // iterate on the array of users    users.forEach(function (currentUser) {      // creates a row      var row = table.insertRow();      // insert cells in the row      var nameCell = row.insertCell();      nameCell.innerHTML = currentUser.name;      var cityCell = row.insertCell();      cityCell.innerHTML = currentUser.address.city;    });    // adds the table to the div    usersDiv.appendChild(table);  } |

JavaScript source code extract:

1. function search() {
2. var queryURL = "https://jsonplaceholder.typicode.com/users";
3. fetch(queryURL)
4. .then(function(response) {
5. **// response is a json string,**
6. **// convert it to a pure JavaScript object**
7. **return response.json();**
8. })
9. .then(function(users) {
10. **// users is a JavaScript object here**
11. displayUsersAsATable(users)
12. })
13. .catch(function(error) {
14. console.log('Error during fetch: ' + error.message);
15. });
16. }

The fetch API will also be covered in an advanced JavaScript course to come. In contrast to XhR2, fetch is based on a concept called "JavaScript promises" (also covered in the advanced course!). You recognize promises when you see ".then..." ".then...".

* 1. **Example of use: the LocalStorage API**

### **Example of use: the LocalStorage API as a client-side database for JavaScript objects**

#### The Web Storage API (localStorage, sessionStorage)

The Web storage API (see the [related W3C specification](http://www.w3.org/TR/webstorage/)) introduces "two related mechanisms, similar to HTTP session cookies, for storing structured data on the client side".

Indeed, Web Storage provides two interfaces - sessionStorage and localStorage - whose main difference is data longevity. This specification defines an API for persistent data storage of key-value pair data in Web clients.

**With localStorage the data will remain until it is deleted, whereas with sessionStorage the data is erased when the tab/browser is closed.**

For convenience, we will mainly illustrate the localStorage object. Just change "local" to "session" and it should work (this time with a session lifetime).

#### key value pairsSimple key-value stores, one per domain (following the [same origin policy](https://en.wikipedia.org/wiki/Same-origin_policy))!

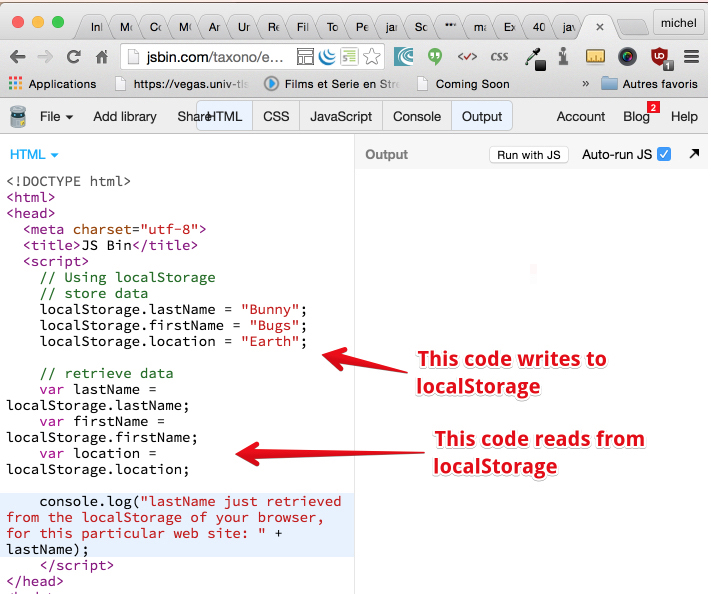
localStorage is a simple key-value store, in which the keys and values are strings. There is only one store per domain. This functionality is exposed through the globally available localStorage object. The same applies to sessionStorage.

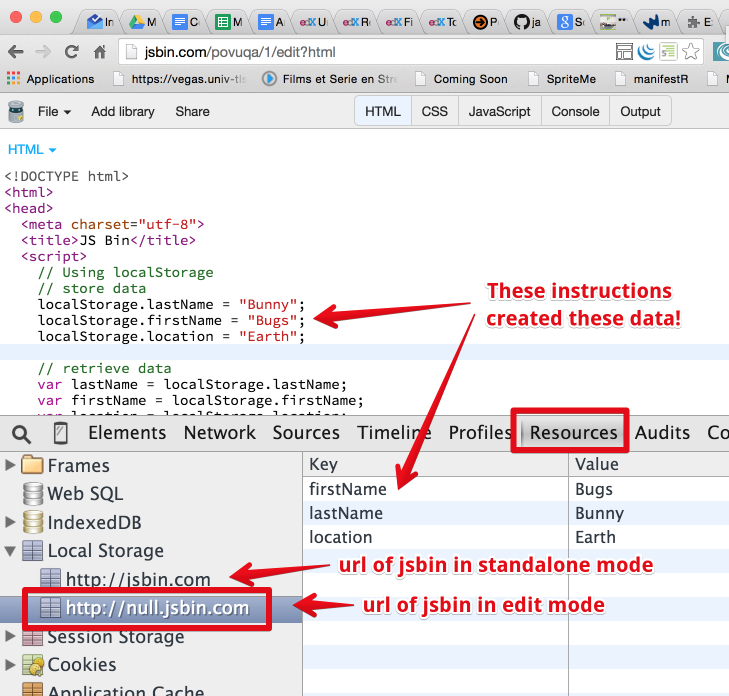
Example:

1. // Using localStorage
2. // store data
3. localStorage.lastName = "Bunny";
4. localStorage.firstName = "Bugs";
5. localStorage.location = "Earth";
6. // retrieve data
7. var lastName = localStorage.lastName;
8. var firstName = localStorage.firstName;
9. var location = localStorage.location;

This data is located in a store attached to the origin of the page. We've created [a JsBin example in which we've included the above code](http://jsbin.com/hebino/1/edit?html,output).

Once opened in your browser, the JavaScript code is executed. With the browser dev. tools, we can check what has been stored in the localStorage for this domain:





#### Differences with cookies?

Cookies are also a popular way to store key-value pairs. Web Storage, however, is a more powerful technique than cookies. The main difference is in size limits: cookies are limited to a few KBytes whereas Web Storage may extend to several MBytes. Also, cookies generate additional HTTP request traffic (whether to request a Web page, an image, a stylesheet, a JavaScript file, etc.).

Objects managed by Web Storage are no longer carried on the network and HTTP, and are easily accessible (read, change and delete) from JavaScript, using the Web Storage API.

#### External resources

* [The W3C Web Storage API recommendation (published on 9 June 2015)](http://www.w3.org/TR/webstorage/)
* [Interesting article on html5rocks that compares the different ways of doing client side persistence with HTML5, including Web Storage.](http://www.html5rocks.com/en/tutorials/offline/storage/)
  1. **Discussion and projects**

Here is the discussion forum for this part of the course. Please either post your comments/observations/questions or share your creations.

See below for a suggested topic of discussion and optional projects.

#### Suggested topic

* Did you know that using XhR2 for getting remote data, you can also monitor the download progression using n xhr.onprogress = event listener and a <progress> HTML element. See [this example from the HTML5 Apps and Games MOOC](https://jsbin.com/nuxanaf/edit?html,output)! This is not yet possible using [the new fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)

#### Optional projects

* Please display the data [from this URL](https://gist.githubusercontent.com/heiswayi/7fde241975ed8a80535a/raw/ff1caaeaf62bd6740ab7cafcd61f1215de173379/datatables-data.json) in a dynamic table.
* Try to add a "hi-score" table to the game developed during this course. The table will be displayed when you lose, and will show only the 10 best scores.

1. **A small application**
   1. **Part 1: a contact manager**

### A small project that uses ES6 classes, local storage, the HTML table JavaScript API and forms

#### Introduction

In this final part of the course, we will build together a minimal contact manager that shows how to use ES6 classes and some JavaScript APIs.

This is a play project that you can easily improve:

* A contact, in this application, is just a person with a name and an email. Feel free to add other properties and methods to the Contact class.
* The contactManager is also an ES6 class with methods for adding, removing, sorting, saving and loading contacts on your hard disk. We will add new functionalities, step by step, in the next sections, but you can improve the examples provided by adding your own new features (build a better HTML table, add new sorting options, etc.)

#### Let's start with a simple skeleton (no GUI), beginning with the Contact class

1. class Contact {
2. constructor(name, email) {
3. this.name = name;
4. this.email = email;
5. }
6. }

As you can see, a contact is just a name and an email. We will use the above class like this:

1. var c1 = new Contact("Jimi Hendrix", "jimi@rip.com");
2. var c2 = new Contact("Robert Fripp", "robert.fripp@kingcrimson.com");

Then you can print the properties of contact c1 or c2 using for example console.log(c1.name), console.log(c2.email), etc.

#### A minimal ContactManager class

1. class ContactManager {
2. constructor() {
3. // when we build the contact manager, it
4. // has an empty list of contacts
5. this.listOfContacts = [];
6. }
7. add(contact) {
8. this.listOfContacts.push(contact);
9. }
10. remove(contact) {
11. // we iterate on the list of contacts until we find the contact
12. // passed as a parameter (we say that they are equal if emails match)
13. for(let i = 0; i < this.listOfContacts.length; i++) {
14. var c = this.listOfContacts[i];
16. if(c.email === contact.email) {
17. // remove the contact at index i
18. this.listOfContacts.splice(i, i);
19. // stop/exit the loop
20. break;
21. }
22. };
23. }
24. printContactsToConsole() {
25. this.listOfContacts.forEach(function(c) {
26. console.log(c.name);
27. });
28. };
29. }

Explanations:

This is a minimal ES6 class for building a contact manager. It has only one property: the list of contacts, and a method for adding a new contact (line 8), one for removing a contact (line 12), that iterate on the list of contacts until the contact passed as a parameter is found (when email properties match), then the contact is removed using the splice method, and we go out from the loop using the break statement (line 22).

It also has a utility method for printing to the console the list of contacts (line 27).

We can use the contact manager like this:

1. var cm = new ContactManager();
2. var c1 = new Contact("Jimi Hendrix", "jimi@rip.com");
3. var c2 = new Contact("Robert Fripp", "robert.fripp@kingcrimson.com");
4. var c3 = new Contact("Angus Young", "angus@acdc.com");
5. var c4 = new Contact("Arnold Schwarzenneger", "T2@terminator.com");
7. console.log("--- Adding 4 contacts ---")
8. cm.add(c1);
9. cm.add(c2);
10. cm.add(c3);
11. cm.add(c4);
13. cm.printContactsToConsole();

As you can see, this is a very minimal version. It's always a good idea to start with very simple structures/classes, and a few methods. Then type the code on CodePen or jsBin and use the devtool console. Check that there are no syntax errors, that everything runs smoothly.

|  |
| --- |
| class Contact {    constructor(name, email) {      this.name = name;      this.email = email;    }  }  class ContactManager {    constructor() {      // when we build the contact manager, it      // has an empty list of contacts      this.listOfContacts = [];    }    add(contact) {      this.listOfContacts.push(contact);    }    remove(contact) {      for (let i = 0; i < this.listOfContacts.length; i++) {        var c = this.listOfContacts[i];        if (c.email === contact.email) {          // remove the contact at index i          this.listOfContacts.splice(i, i);          // stop/exit the loop          break;        }      }    }    printContactsToConsole() {      this.listOfContacts.forEach(function (c) {        console.log(c.name);      });    }  }  // ALWAYS TEST YOUR CODE WITH SIMPLE EXAMPLES, or by typing in the devtool console  var cm = new ContactManager();  var c1 = new Contact("Jimi Hendrix", "jimi@rip.com");  var c2 = new Contact("Robert Fripp", "robert.fripp@kingcrimson.com");  var c3 = new Contact("Angus Young", "angus@acdc.com");  var c4 = new Contact("Arnold Schwarzenneger", "T2@terminator.com");  console.log("--- Adding 4 contacts ---")  cm.add(c1);  cm.add(c2);  cm.add(c3);  cm.add(c4);  cm.printContactsToConsole();  // trying to remove c2  console.log("--- Removing the second one! ---");  cm.remove(c2);  cm.printContactsToConsole(); |

#### Adding a method for sorting the list of contacts by name

Do you remember the sort() method you can use on arrays? We saw it in modules 2 or 3. Since our array contains objects, we must provide a callback for comparing two elements by name. Here is the code for the new sort() method we added to the ContactManager class:

1. sort() {
2. // As our array contains objects, we need to pass as argument
3. // a method that can compare two contacts.
4. // we use a class method, which is similar to the distance(p1, p2)
5. // method we saw in the ES6 Point class in module 4
6. // We always call such methods using the name of the class followed
7. // by the dot operator
8. **this.listOfContacts.sort(ContactManager.compareByName);**
9. }
10. **// class method for comparing two contacts by name**
11. **static compareByName**(c1, c2) {
12. // JavaScript has built in capabilities for comparing strings
13. // in alphabetical order
14. if (c1.name < c2.name)
15. return -1;
16. if (c1.name > c2.name)
17. return 1;
18. return 0; // c1.name = c2.name
19. }

The important thing here is to notice that we declared the compareByName method as a class method (using the static keyword). This is similar to what we did in the Point class example from module 4, when we explained the "class properties and methods". This method compareByName does not depend on any instance of the contact manager, consequently: it's a class method.

|  |
| --- |
| class Contact {    constructor(name, email) {      this.name = name;      this.email = email;    }  }  class ContactManager {    constructor() {      // when we build the contact manager, it      // has an empty list of contacts      this.listOfContacts = [];    }    add(contact) {      this.listOfContacts.push(contact);    }    remove(contact) {      for (let i = 0; i < this.listOfContacts.length; i++) {        var c = this.listOfContacts[i];        if (c.email === contact.email) {          // remove the contact at index i          this.listOfContacts.splice(i, i);          // stop/exit the loop          break;        }      }    }    sort() {      // As our array contains objects, we need to pass as argument      // a method that can compare two contacts.      // we use for that a class method, similar to the distance(p1, p2)      // method we saw in the ES6 Point class in module 4      // We always call such methods using the name of the class followed      // by the dot operator      this.listOfContacts.sort(ContactManager.compareByName);    }    // class method for comparing two contacts by name    static compareByName(c1, c2) {      // JavaScript has builtin capabilities for comparing strings      // in alphabetical order      if (c1.name < c2.name)        return -1;      if (c1.name > c2.name)        return 1;      return 0;    }    printContactsToConsole() {      this.listOfContacts.forEach(function (c) {        console.log(c.name);      });    }  }  // ALWAYS TEST YOUR CODE WITH SIMPLE EXAMPLES, or by typing in the devtool console  var cm = new ContactManager();  var c1 = new Contact("Jimi Hendrix", "jimi@rip.com");  var c2 = new Contact("Robert Fripp", "robert.fripp@kingcrimson.com");  var c3 = new Contact("Angus Young", "angus@acdc.com");  var c4 = new Contact("Arnold Schwarzenneger", "T2@terminator.com");  console.log("--- Adding 4 contacts ---")  cm.add(c1);  cm.add(c2);  cm.add(c3);  cm.add(c4);  cm.printContactsToConsole();  // trying to remove c2  console.log("--- Removing the second one! ---");  cm.remove(c2);  cm.printContactsToConsole();  console.log("--- sorting contacts ---");  cm.sort();  cm.printContactsToConsole(); |

* 1. **Part 2: persistence**

We will use load/save methods is for loading and saving the list of contacts in Local Storage.

#### Our task: load and save methods (persistence)

This time, we will add to the ContactManager class a load() and a save() method for loading/saving from disk (from a key/value pair database located on your hard disk, and associated to the domain of your Web application).

#### Saving the list of contacts in JSON, checking the saved value using the devtools

Here is the code we added to the ES6 class for saving the list of contacts in JSON:

1. class ContactManager {
2. constructor() {
3. // when we build the contact manager, it
4. // has an empty list of contacts
5. this.listOfContacts = [];
6. }
7. ...
9. **save() {**
10. **// We can only save strings in local storage. So, let's convert**
11. **// our array of contacts to JSON**
12. **localStorage.contacts = JSON.stringify(this.listOfContacts);**
13. **}**
14. }

You write data identified by a key in localStorage like this:

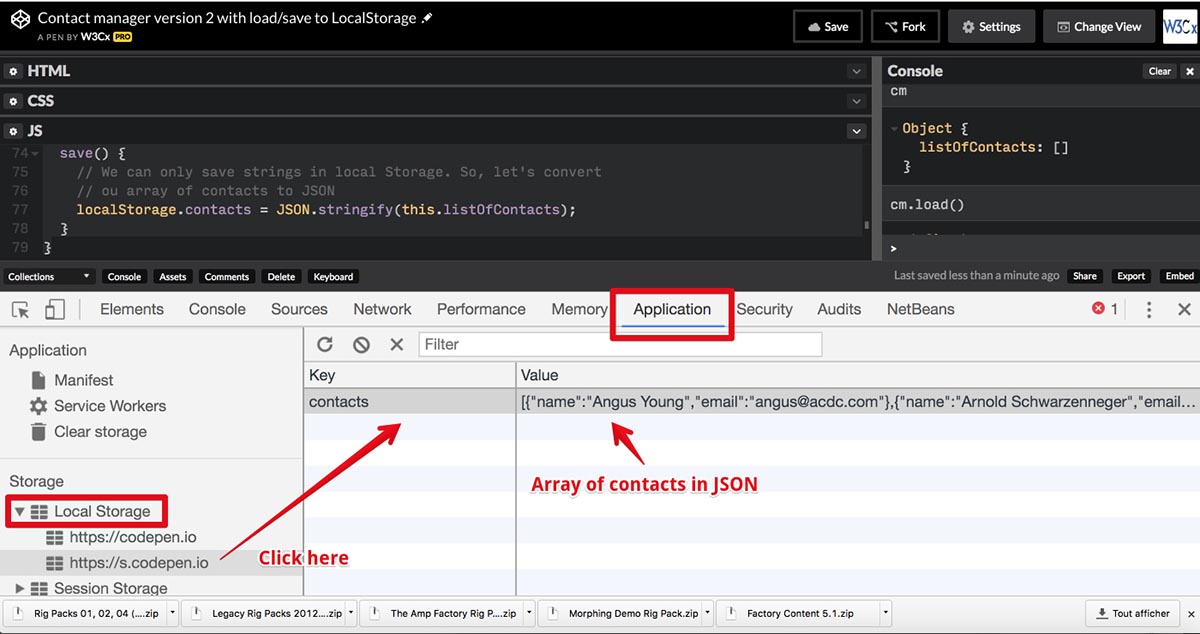
* localStorage.keyName = a string value

In our case, line 13 saves the list of contacts with a key named "contacts" in the local storage. In order to save the list of contacts as a string, we convert it to the JSON format using the  JSON.stringify(...) method (JSON = string based)

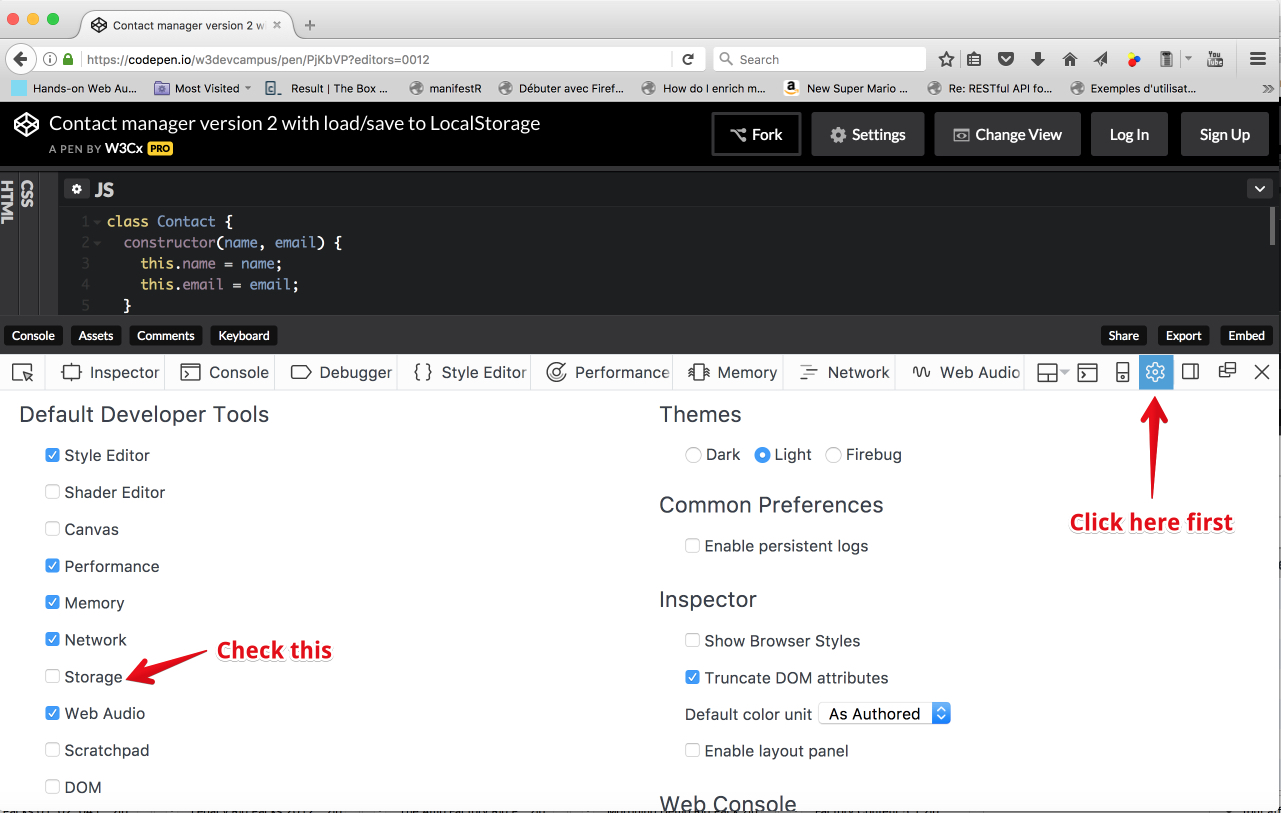
|  |
| --- |
| class Contact {    constructor(name, email) {      this.name = name;      this.email = email;    }  }  class ContactManager {    constructor() {      // when we build the contact manager, it      // has an empty list of contacts      this.listOfContacts = [];    }    // Will erase all contacts    empty() {      this.listOfContacts = [];    }    add(contact) {      this.listOfContacts.push(contact);    }    remove(contact) {      for (let i = 0; i < this.listOfContacts.length; i++) {        var c = this.listOfContacts[i];        if (c.email === contact.email) {          // remove the contact at index i          this.listOfContacts.splice(i, i);          // stop/exit the loop          break;        }      }    }    sort() {      // As our array contains objects, we need to pass as argument      // a method that can compare two contacts.      // we use for that a class method, similar to the distance(p1, p2)      // method we saw in the ES6 Point class in module 4      // We always call such methods using the name of the class followed      // by the dot operator      this.listOfContacts.sort(ContactManager.compareByName);    }    // class method for comparing two contacts by name    static compareByName(c1, c2) {      // JavaScript has builtin capabilities for comparing strings      // in alphabetical order      if (c1.name < c2.name)        return -1;      if (c1.name > c2.name)        return 1;      return 0;    }    printContactsToConsole() {      if (this.listOfContacts.length === 0) {        console.log("EMPTY LIST!");        return;      }      this.listOfContacts.forEach(function (c) {        console.log(c.name);      });    }    load() {      if (localStorage.contacts !== undefined) {        // the array of contacts is saved in JSON, let's convert        // it back to a reak JavaScript object.        this.listOfContacts = JSON.parse(localStorage.contacts);      }    }    save() {      // We can only save strings in local Storage. So, let's convert      // ou array of contacts to JSON      localStorage.contacts = JSON.stringify(this.listOfContacts);    }  }  var cm = new ContactManager();  var c1 = new Contact("Jimi Hendrix", "jimi@rip.com");  var c2 = new Contact("Robert Fripp", "robert.fripp@kingcrimson.com");  var c3 = new Contact("Angus Young", "angus@acdc.com");  var c4 = new Contact("Arnold Schwarzenneger", "T2@terminator.com");  console.log("--- Adding 4 contacts ---")  cm.add(c1);  cm.add(c2);  cm.add(c3);  cm.add(c4);  cm.printContactsToConsole();  // trying to remove c2  console.log("--- Removing the second one! ---");  cm.remove(c2);  cm.printContactsToConsole();  console.log("--- Sorting contacts ---");  cm.sort();  cm.printContactsToConsole();  console.log("--- Saving contacts to local storage ---");  cm.save();  console.log("--- Emptying the list of contacts ---");  cm.empty();  cm.printContactsToConsole();  console.log("--- Loading contacts from local storage ---");  cm.load();  cm.printContactsToConsole();  console.log("Do you notice: contacts have all been restored!"); |

#### Then we can check in the devtools that the list has been saved

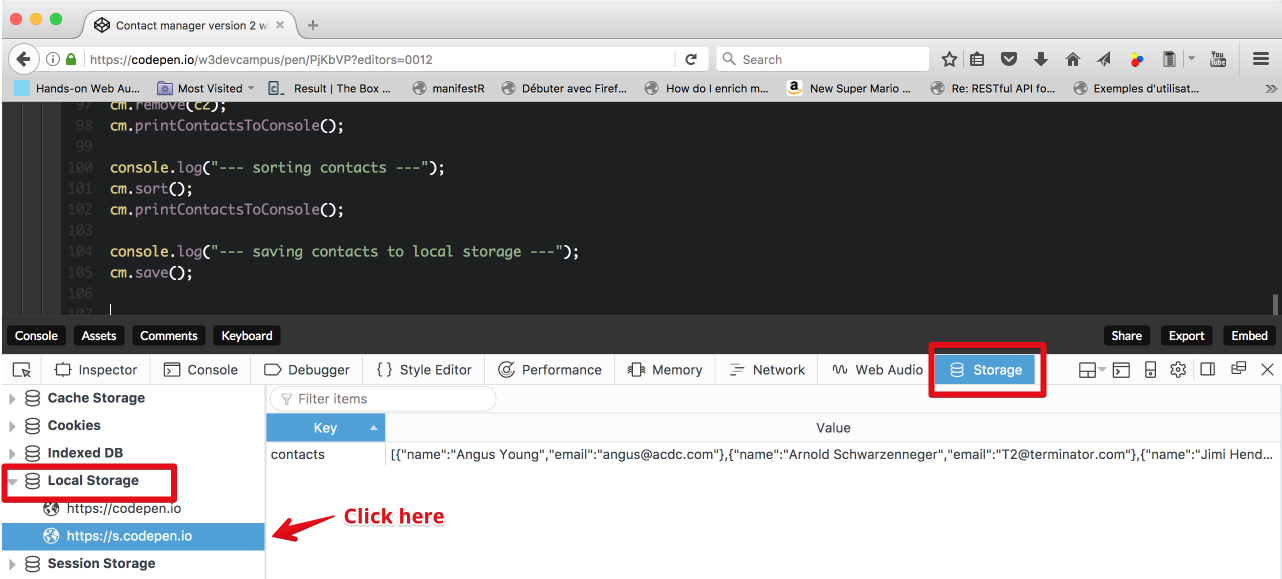
In Google Chrome, click the Application tab, then LocalStorage:



In Firefox, you first need to activate the storage view like this:



You will see the list of contacts when you click on the newly appeared "Storage" tab:



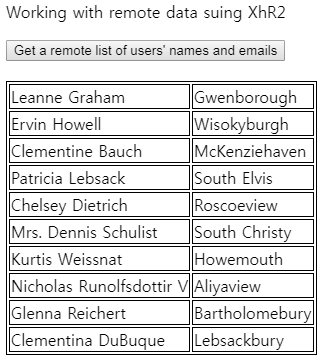
#### Restoring the list of contacts

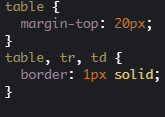
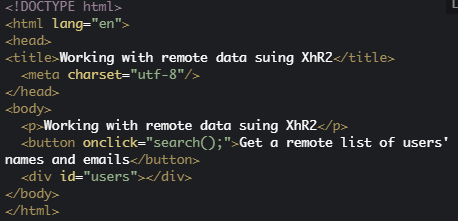
This time, we've added a load() method that will check if a list of contacts has been saved. If this is the case, it will read it from LocalStorage, convert it back from JSON into a JavaScript object. In order to test this, in the following CodePen, we first save the list, then we empty the list in memory (setting the array to an empty array), print the list of contacts (that displays a message "LIST EMPTY!"), then we load the contacts from LocalStorage and print the list again: it has been restored to its previous value.

1. class ContactManager {
2. constructor() {
3. // when we build the contact manager, it
4. // has an empty list of contacts
5. this.listOfContacts = [];
6. }
7. // Will erase all contacts
8. empty() {
9. this.listOfContacts = [];
10. }
12. ...
14. **load() {**
15. **if(localStorage.contacts !== undefined) {**
16. **// the array of contacts is saved in JSON, let's convert**
17. **// it back to a reak JavaScript object.**
18. **this.listOfContacts = JSON.parse(localStorage.contacts);**
19. **}**
20. **}**
21. }
23. ...
25. console.log("--- Saving contacts to local storage ---");
26. cm.save();
28. console.log("--- Emptying the list of contacts ---");
29. cm.empty();
30. cm.printContactsToConsole();
32. console.log("--- Loading contacts from local storage ---");
33. cm.load();
34. cm.printContactsToConsole();
35. console.log("Do you notice: contacts have all been restored!");

Explanations:

* At line 16, we check if a previous version has been saved.
* At line 19, we read the string value associated to the key named "contacts", and use JSON.parse(...) to turn it into a JavaScript object we can work with.
* Lines 26-36 test the load/save/empty functionalities. You can try this yourself live: click on the CodePen label below, on the top right corner, and once in CodePen, open the CodePen console (or the read devtool console) to see the result of the execution of these tests.
  1. **Part 3: display contacts in an HTML5 table**

****

****

|  |
| --- |
| function search() {    var queryURL = "https://jsonplaceholder.typicode.com/users";    var xhr = new XMLHttpRequest();    xhr.open('GET', queryURL, true);    // called when the response is arrived    xhr.onload = function (e) {      var jsonResponse = xhr.response;      // turn the response into a JavaScript object      var users = JSON.parse(jsonResponse);      displayUsersAsATable(users);    }    // in case of error    xhr.onerror = function (err) {      console.log("Error: " + err);    }    // sends the request    xhr.send();  }  function displayUsersAsATable(users) {    // users is a JavaScript object    // empty the div that contains the results    var usersDiv = document.querySelector("#users");    usersDiv.innerHTML = "";    // creates and populate the table with users    var table = document.createElement("table");    // iterate on the array of users    users.forEach(function (currentUser) {      // creates a row      var row = table.insertRow();      // insert cells in the row      var nameCell = row.insertCell();      nameCell.innerHTML = currentUser.name;      var cityCell = row.insertCell();      cityCell.innerHTML = currentUser.address.city;    });    // adds the table to the div    usersDiv.appendChild(table);  } |

This time, we will first add some HTML to the contact manager example (same as in the previous CodePen except that we've renamed "users" as "contacts"):

1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <title>A contact manager, v3</title>
5. <meta charset="utf-8"/>
6. </head>
7. <body>
8. <p>List of contacts</p>
9. **<div id="contacts"></div>**
10. </body>
11. </html>

The div at *line 9* is where we're going to dynamically insert an HTML table with one row for each contact. We will keep the same minimal CSS for displaying table, row and cell borders (we encourage you to improve this):

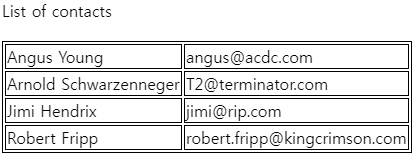
1. table {
2. margin-top: 20px;
3. }
5. table, tr, td {
6. border: 1px solid;
7. }

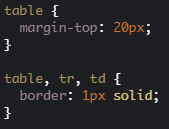
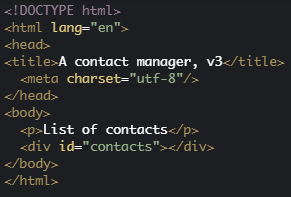
And here is the method we addin our ContactManager class; an adaptation of the function displayUsersAsATable(users) from the previous CodePen:

1. class ContactManager {
2. .....
3. displayContactsAsATable(idOfContainer) {
4. // to empty the container that contains the results
5. let container = document.querySelector("#" + idOfContainer);
6. container.innerHTML = "";
8. if(this.listOfContacts.length === 0) {
9. container.innerHTML = "<p>No contacts to display!</p>";
10. // stops the execution of this method
11. return;
12. }
13. // creates and populates the table with users
14. let table = document.createElement("table");
15. // iterates on the array of users
16. this.listOfContacts.forEach(function(currentContact) {
17. // creates a row
18. let row = table.insertRow();
19. row.innerHTML = "<td>" + currentContact.name + "</td>"
20. + "<td>" + currentContact.email + "</td>"
21. });
22. // adds the table to the div
23. container.appendChild(table);
24. }
25. }

Explanations:

* *Line 3*: the method displayContactsAsATable takes as a parameter the id of the HTML element in which the table will be inserted after being built. This id is used by the querySelector call at *line 5*.
* *Lines 9-13*: if the list of contacts is empty, we just return, but first we display in the HTML container a message: "No contact to display!"
* *Lines 16-25*: we create a table, and for each contact we insert and fill a new row in the table.
* *Line 28*: the table is inserted (appended) in the HTML container.

****

****

|  |
| --- |
| window.onload = init;  // The contact manager as a global variable  let cm;  function init() {    // create an instance of the contact manager    cm = new ContactManager();    cm.addTestData();    cm.printContactsToConsole();    // Display contacts in a table    // Pass the id of the HTML element that will contain the table    cm.displayContactsAsATable("contacts");  }  class Contact {    constructor(name, email) {      this.name = name;      this.email = email;    }  }  class ContactManager {    constructor() {      // when we build the contact manager, it      // has an empty list of contacts      this.listOfContacts = [];    }    addTestData() {      let c1 = new Contact("Jimi Hendrix", "jimi@rip.com");      let c2 = new Contact("Robert Fripp", "robert.fripp@kingcrimson.com");      let c3 = new Contact("Angus Young", "angus@acdc.com");      let c4 = new Contact("Arnold Schwarzenneger", "T2@terminator.com");      this.add(c1);      this.add(c2);      this.add(c3);      this.add(c4);      // Let's sort the list of contacts by Name      this.sort();    }    // Will erase all contacts    empty() {      this.listOfContacts = [];    }    add(contact) {      this.listOfContacts.push(contact);    }    remove(contact) {      for (let i = 0; i < this.listOfContacts.length; i++) {        let c = this.listOfContacts[i];        if (c.email === contact.email) {          // remove the contact at index i          this.listOfContacts.splice(i, i);          // stop/exit the loop          break;        }      }    }    sort() {      // As our array contains objects, we need to pass as argument      // a method that can compare two contacts.      // we use for that a class method, similar to the distance(p1, p2)      // method we saw in the ES6 Point class in module 4      // We always call such methods using the name of the class followed      // by the dot operator      this.listOfContacts.sort(ContactManager.compareByName);    }    // class method for comparing two contacts by name    static compareByName(c1, c2) {      // JavaScript has builtin capabilities for comparing strings      // in alphabetical order      if (c1.name < c2.name)        return -1;      if (c1.name > c2.name)        return 1;      return 0;    }    printContactsToConsole() {      this.listOfContacts.forEach(function (c) {        console.log(c.name);      });    }    load() {      if (localStorage.contacts !== undefined) {        // the array of contacts is savec in JSON, let's convert        // it back to a reak JavaScript object.        this.listOfContacts = JSON.parse(localStorage.contacts);      }    }    save() {      // We can only save strings in local Storage. So, let's convert      // ou array of contacts to JSON      localStorage.contacts = JSON.stringify(this.listOfContacts);    }    displayContactsAsATable(idOfContainer) {      // empty the container that contains the results      let container = document.querySelector("#" + idOfContainer);      container.innerHTML = "";      if (this.listOfContacts.length === 0) {        container.innerHTML = "<p>No contacts to display!</p>";        // stop the execution of this method        return;      }      // creates and populate the table with users      let table = document.createElement("table");      // iterate on the array of users      this.listOfContacts.forEach(function (currentContact) {        // creates a row        let row = table.insertRow();        row.innerHTML = "<td>" + currentContact.name + "</td>"          + "<td>" + currentContact.email + "</td>"      });      // adds the table to the div      container.appendChild(table);    }  } |

Note that we also added a method called addTestData() to the ContactManager class, as this is a way to make testing the class easier. The code in this method is similar to all the code we used in previous examples for testing the class by adding four contacts to it and displaying messages in the devtool console.

* 1. **Part 4: use a form to enter new contacts**

#### Some reminders about HTML forms

In the previous example, we added a form for entering a new contact, and an "add" button.

Here is the HTML code of the form:

1. <form **onsubmit="return formSubmitted();"**>
2. <fieldset>
3. <legend>Personal informations</legend>
4. <label>
5. Name :
6. <input type="text" id="name" required>
7. </label>
8. <label>
9. Email :
10. <input type="email" id="email" required>
11. </label>
12. <br>
13. **<button>Add new Contact</button>**
14. </fieldset>
15. </form>

The button at line 13 will submit the form by default (it's equivalent to an <input type="submit">).

The event listener at line 1:

1. <form **onsubmit="return formSubmitted();"**>

... will call the formSubmitted function when the form is submitted. It is interesting that we use onclick="return formSubmitted();":

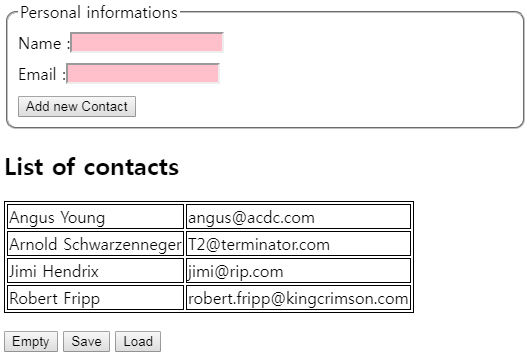
* If the returned value is true, the form will be submitted by your browser (this would reload the HTML page).
* If the returned value is false, the form will not be submitted (this is what we want, so we will return false in the formSubmitted function).

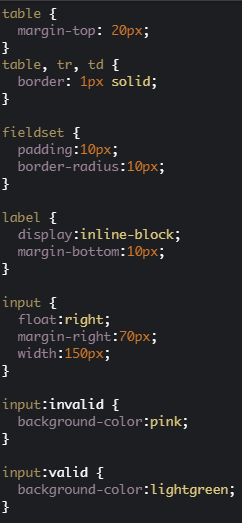
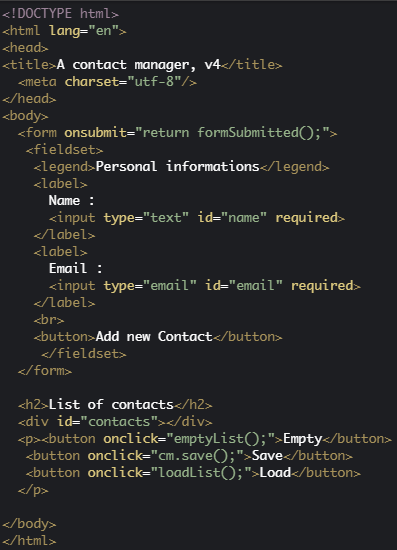
Here is the code of the formSubmitted function:

1. function formSubmitted() {
2. // Get the values from input fields
3. let name = document.querySelector("#name");
4. let email = document.querySelector("#email");
5. **let newContact = new Contact(name.value, email.value);**
6. **cm.add(newContact);**
7. // Empty the input fields
8. name.value = "";
9. email.value = "";
10. **// refresh the table**
11. **cm.displayContactsAsATable("contacts");**
12. // do not let your browser submit the form using HTTP
13. **return false;**
14. }

**Explanations:**

* Lines 2-7: we get the values entered in the form's input fields, build a new contact and add it to the contact list
* Lines 10-11: we reset the content of the input fields (we empty them)
* Line 14: we display the HTML table with the new added contact
* Line 17: we return false so that the form will not be submitted. This will prevent the browser from reloading the HTML page.

****

****

|  |
| --- |
| window.onload = init;  // The contact manager as a global variable  let cm;  function init() {    // create an instance of the contact manager    cm = new ContactManager();    cm.addTestData();    cm.printContactsToConsole();    // Display contacts in a table    // Pass the id of the HTML element that will contain the table    cm.displayContactsAsATable("contacts");  }  function formSubmitted() {    // Get the values from input fields    let name = document.querySelector("#name");    let email = document.querySelector("#email");    let newContact = new Contact(name.value, email.value);    cm.add(newContact);    // Empty the input fields    name.value = "";    email.value = "";    // refresh the html table    cm.displayContactsAsATable("contacts");    // do not let your browser submit the form using HTTP    return false;  }  function emptyList() {    cm.empty();    cm.displayContactsAsATable("contacts");  }  function loadList() {    cm.load();    cm.displayContactsAsATable("contacts");  }  class Contact {    constructor(name, email) {      this.name = name;      this.email = email;    }  }  class ContactManager {    constructor() {      // when we build the contact manager, it      // has an empty list of contacts      this.listOfContacts = [];    }    addTestData() {      var c1 = new Contact("Jimi Hendrix", "jimi@rip.com");      var c2 = new Contact("Robert Fripp", "robert.fripp@kingcrimson.com");      var c3 = new Contact("Angus Young", "angus@acdc.com");      var c4 = new Contact("Arnold Schwarzenneger", "T2@terminator.com");      this.add(c1);      this.add(c2);      this.add(c3);      this.add(c4);      // Let's sort the list of contacts by Name      this.sort();    }    // Will erase all contacts    empty() {      this.listOfContacts = [];    }    add(contact) {      this.listOfContacts.push(contact);    }    remove(contact) {      for (let i = 0; i < this.listOfContacts.length; i++) {        var c = this.listOfContacts[i];        if (c.email === contact.email) {          // remove the contact at index i          this.listOfContacts.splice(i, i);          // stop/exit the loop          break;        }      }    }    sort() {      // As our array contains objects, we need to pass as argument      // a method that can compare two contacts.      // we use for that a class method, similar to the distance(p1, p2)      // method we saw in the ES6 Point class in module 4      // We always call such methods using the name of the class followed      // by the dot operator      this.listOfContacts.sort(ContactManager.compareByName);    }    // class method for comparing two contacts by name    static compareByName(c1, c2) {      // JavaScript has builtin capabilities for comparing strings      // in alphabetical order      if (c1.name < c2.name)        return -1;      if (c1.name > c2.name)        return 1;      return 0;    }    printContactsToConsole() {      this.listOfContacts.forEach(function (c) {        console.log(c.name);      });    }    load() {      if (localStorage.contacts !== undefined) {        // the array of contacts is savec in JSON, let's convert        // it back to a reak JavaScript object.        this.listOfContacts = JSON.parse(localStorage.contacts);      }    }    save() {      // We can only save strings in local Storage. So, let's convert      // ou array of contacts to JSON      localStorage.contacts = JSON.stringify(this.listOfContacts);    }    displayContactsAsATable(idOfContainer) {      // empty the container that contains the results      let container = document.querySelector("#" + idOfContainer);      container.innerHTML = "";      if (this.listOfContacts.length === 0) {        container.innerHTML = "<p>No contacts to display!</p>";        // stop the execution of this method        return;      }      // creates and populate the table with users      var table = document.createElement("table");      // iterate on the array of users      this.listOfContacts.forEach(function (currentContact) {        // creates a row        var row = table.insertRow();        row.innerHTML = "<td>" + currentContact.name + "</td>"          + "<td>" + currentContact.email + "</td>"      });      // adds the table to the div      container.appendChild(table);    }  } |

Note that we've also added some buttons for playing with the load/save features we implemented in the previous page:

* Add some new contacts to the list using the form,
* Save them by clicking on the save button,
* Empty the list, click the empty button,
* Reload the list... you can see that contacts have been correctly saved and restored!

* 1. **Discussion and optional projects**

Here is the discussion forum for this part of the course. Please either post your comments/observations/questions or share your creations.

See below for proposed optional projects to help you practice!

#### Optional projects

* Improve the CSS of the contact manager table.
* Add more complicated features to the contact manager:
  + Add an extra column with a trash bin icon in it (you can use this [one](http://i.imgur.com/yHyDPio.png)). When you click on this icon, delete the contact.  
      
    Hint: find a way to get the index of the current row in the click event listener, so that you can easily delete the contact from the array. You can add a "HTML data attribute" using trashbin.dataset.contactId = 3; for example, when you create the img element of the trash bin, do something like this:  
      
    let trashbin = document.createElement("img");  
    trashbin.src =  "http://i.imgur.com/yHyDPio.png";  
    trashbin.dataset.contactId = 3; // 3 is the current row index and corresponds  
                                    // to contact at index = 3 in the array of contacts  
      
    It's like adding a data-contactId attribute to the HTML of the img element. Then in the event listener, use evt.target.dataset.contactId to get it back.
  + Add a search form for searching a contact by name: rebuild the table to display only contacts that match. More difficult: reduce the table as you type!
  + Improve the CSS of this ugly-looking table! ;-)
  + Add a header on the table and try to make the table sortable when you click on the header of one column (e.g., clicking on "email" will sort the table by email).
* [ADVANCED] Using [the classList JavaScript interface](https://developer.mozilla.org/en-US/docs/Web/API/Element/classList): elem.classList.add("name of a CSS class"), remove, and toggle methods, allow the user to manipulate CSS classes from JavaScript. Try to make the table of contacts editable. Click on a cell and it will become editable (tip: use both a label and an input field). When you click, you hide the label and show the input field, and when you click outside of the input field, you do the reverse. Use the "blur" event to detect when clicks occur outside).