### [From objects to arrays](https://courses.edx.org/courses/course-v1:W3Cx+JS.0x+3T2017/courseware/1f5a7a2285994fbd87d23829a53d29d2/9c8aeb657b10443893ad9d08f37a0870/?child=first)

#### In Javascript, an object = a table whose keys/indexes are defined!

Look at this array:

1. > **var darkVador = ['villain', 'half human half machine'];**
2. undefined
4. > darkVador[0]
5. "villain"
7. > darkVador[1]
8. "half human half machine"

And now, look at this object:

1. var darkVador = {
2. job: 'villain',
3. race: 'half human half machine'
4. };

They look a bit similar, don't they?

* + Same name of the variable that contains the object = darkVador
  + Instead of '[' and ']' that we used for defining an array, we use '{' and '}' for defining an object
  + The elements of the object (its **properties**) are separated by a comma ','
  + The pairs of keys/values are separated by ':' as in race**:** 'half human, half machine'
  + The last pair of keys/values has no ',' at the end.

**It is possible to access the object's properties with "." or with brackets**

We saw that we can use the "." operator, followed by the property name. It's also possible to use the bracket notation, and manipulate the object as an array whose indexes, instead of being 0, 1, 2 etc., are the property names!

1. > var book = {
2. title: 'Le Petit Prince',
3. author: 'Saint-Exupery'
4. };
5. undefined
7. > **var title = book.title;**
8. undefined
10. >**title;**
11. **"Le Petit Prince"**
13. > **var title = book['title'];**
14. undefined
16. >**title**
17. **"Le Petit Prince";**
19. > var author = book['author'];
20. undefined
22. > author;
23. "Saint-Exupery"

As you can see, if you look at *lines 7-10* and *13-16*, writing book.title or book['title'] is equivalent!

**In JavaScript, objects are arrays whose indexes are property names: please remember this!**

**Property declaration syntax**

**Property names: different possibilities**

We can put single or double quotes around the name of the property, or nothing at all:

1. **var louis = {age: 40}; // WE DO THIS MOST OF THE TIME!**
2. var louis = {"age": 40};
3. var louis = {'age': 40};

**In some cases we have to put quotes around the property name:**

* + When it is a reserved word from JavaScript,
  + Or it contains spaces or special characters,
  + Or it begins with a number.

Examples:

1. **book.1stPublication = '6 avril 1943'; // begins with a number**
2. **// Throws a SyntaxError**
3. book['1stPublication'] = '6 avril 1943'; // OK
4. book.date of publication = '6 avril 1943'; // spaces not allowed!
5. book['date of publication'] = '6 avril 1943'; // allowed, but avoid!

**Another classic case where the name of a property is in a variable**

In this case it is necessary  to use the syntax with '[' and ']' ...

Example:

1. > var key = 'title';
2. undefined
4. >**book[key];**
5. "Le Petit Prince"

### An object can contain another object

Example:

1. > var book = {
2. name: 'Catch-22',
3. published: 1961,
4. **author: {                 // embedded object!**
5. **givenName: 'Joseph',**
6. **familyName: 'Heller'**
7. **}**
8. };
9. undefined
11. > book.author.givenName;
12. "Joseph"
14. > book.author.familyName;
15. "Heller"

Accessing the embedded object author is done by chaining property accesses using the "." operator, like in book.author.givenName(here we access the givenName property of the object author, which is also a property of the book object).

**Elements, properties and methods**

**Some vocabulary:**

* + For **arrays**, we speak of **elements**
  + For **objects**, we talk about **properties**
  + But **a property can also be a function**, in which case it is called a **method**

**Yes, it is possible for an object's property to be a function!**

A very simple example:

1. var medor = {
2. name: 'Benji',
3. bark: function(){
4. alert('Ouarf, Ouarf!');
5. }
6. };

In this example, the bark property's value is a function, so we call bark "a method".

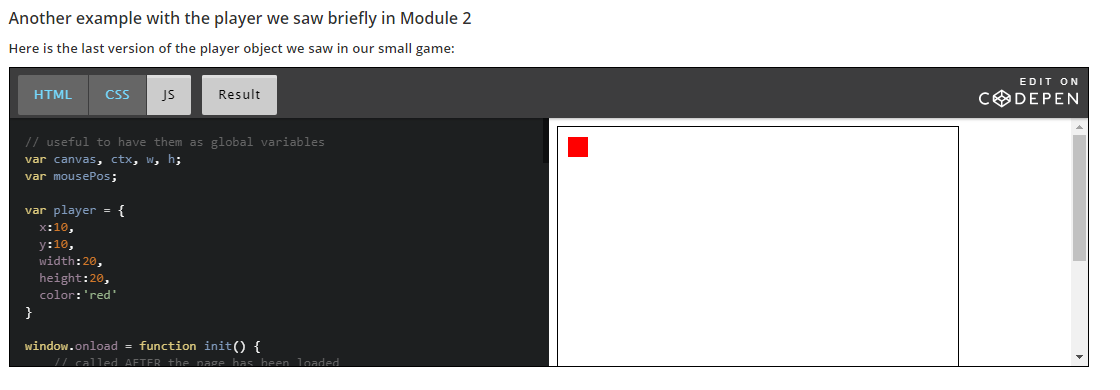
**A method is a special property that corresponds to the object's behavior**

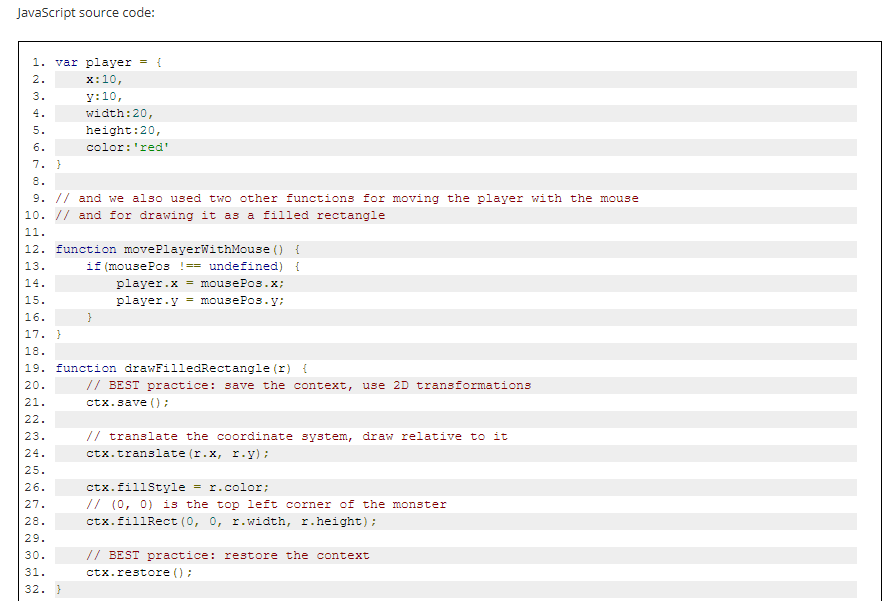
**Properties correspond to an object's DNA (its characteristics),   
and are nouns (age, name, etc.)**

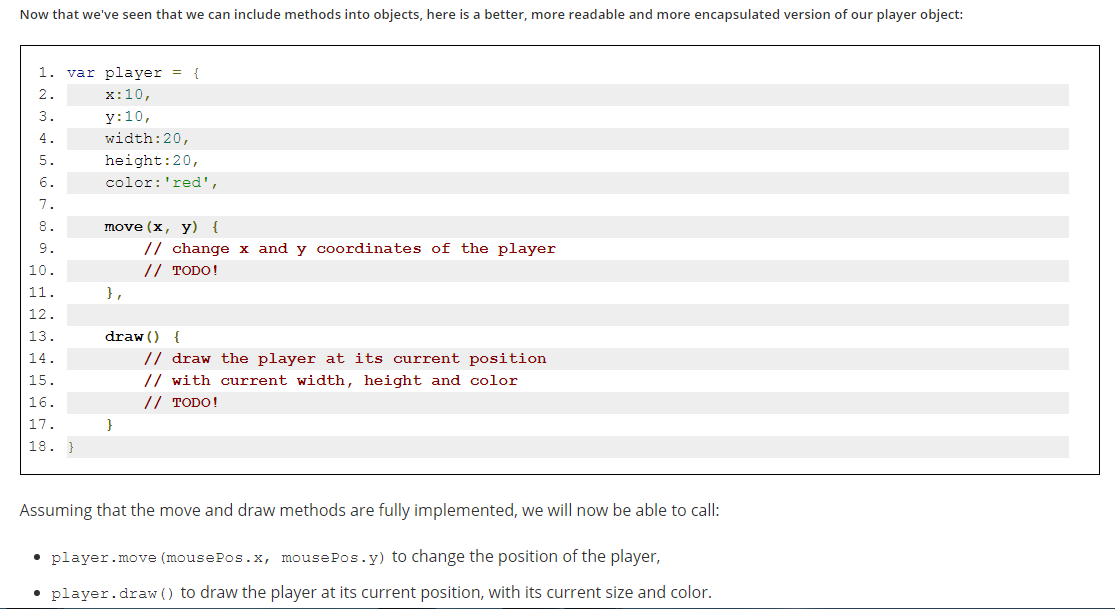
**Methods correspond to an object's behavior   
and are verbs (bark, move, changeSpeed, etc.)**

**Calling a method**

Since a method is a property we can use the '.' operator (or brackets with the method's name as a string index).







Assuming that the move and draw methods are fully implemented, we will now be able to call:

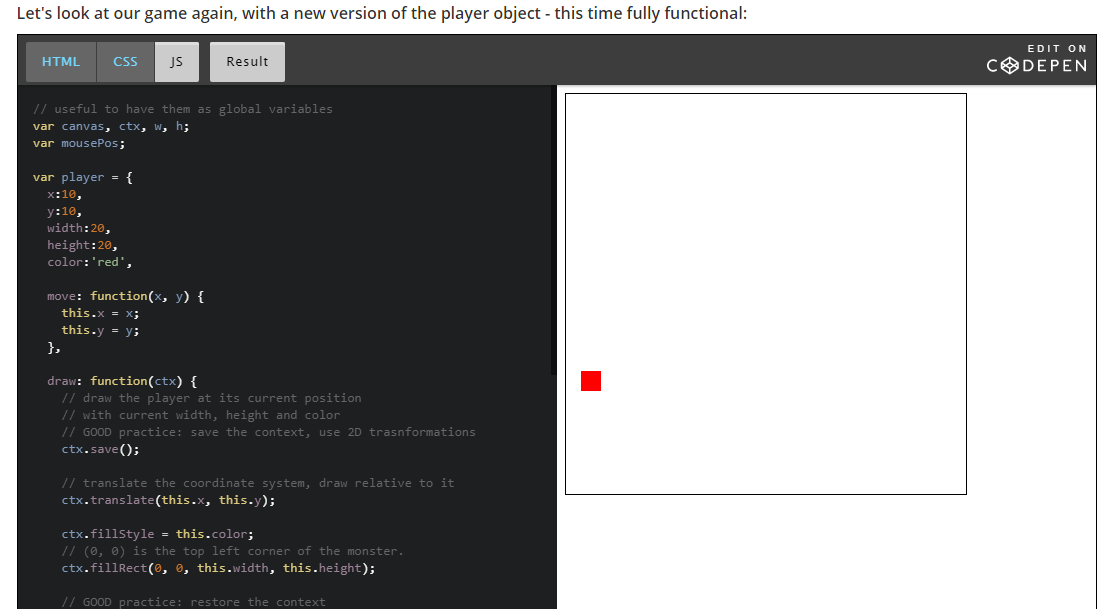
* + player.move(mousePos.x, mousePos.y) to change the position of the player,
  + player.draw() to draw the player at its current position, with its current size and color.

Readability is better, it is like asking the player to move, or asking it to draw itself. And we do not need to pass the x, y, width, height, color to the draw method: it is inside the player object, and it can access all its internal property values!

**The this keyword: Accessing properties from a method**

**The this keyword!**

When one wants to access an object property or wants to call another method from an object method, we must use the this keyword. In the code of the player object, this means "from this object".



## Adding/deleting properties and methods

### Properties and methods can be added/deleted after an object has been defined

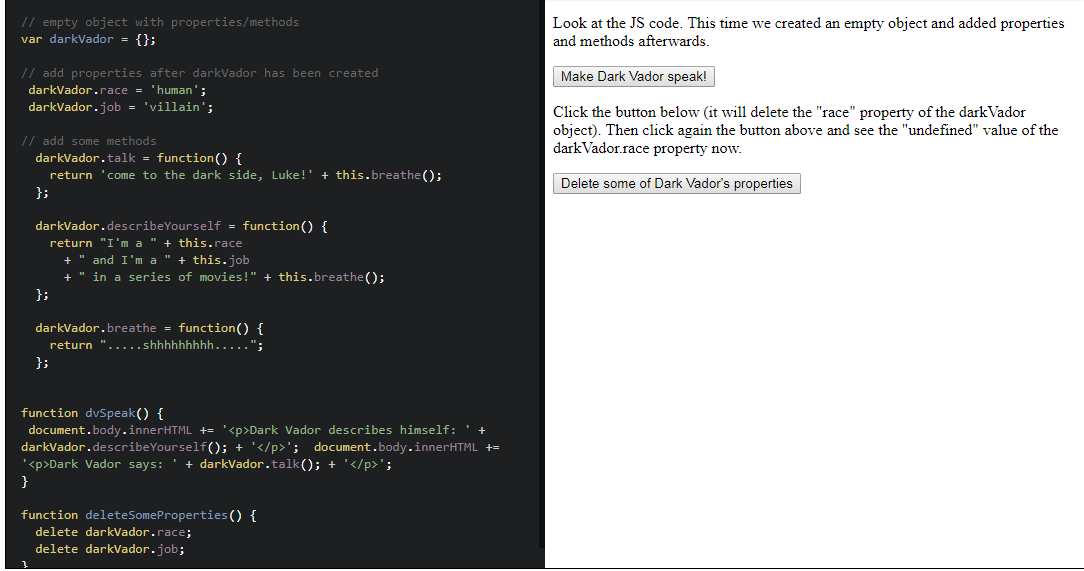
#### Unlike other object-oriented languages, it is possible in JavaScript to add or to remove properties after an object has been created

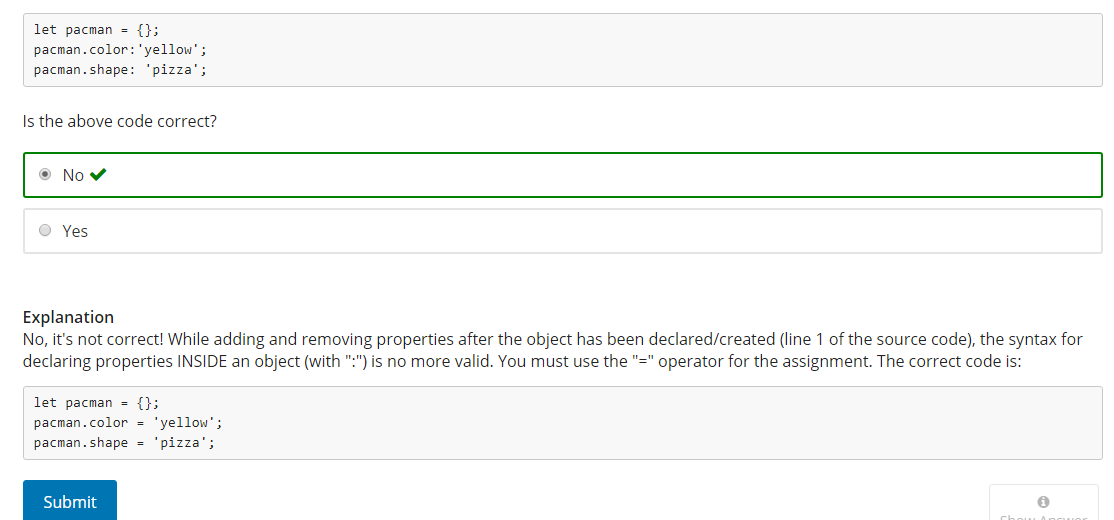


**Deleting a property or a method**

You can use the JavaScript keyword "delete" to delete an object's property (it will become undefined).

1. function deleteSomeProperties() {
2. **delete darkVador.race;**
3. **delete darkVador.job;**
4. }





## [Classes](https://courses.edx.org/courses/course-v1:W3Cx+JS.0x+3T2017/courseware/1f5a7a2285994fbd87d23829a53d29d2/c33324ad26584d6391466a8966b74abb/?child=first): [definition](https://courses.edx.org/courses/course-v1:W3Cx+JS.0x+3T2017/courseware/1f5a7a2285994fbd87d23829a53d29d2/c33324ad26584d6391466a8966b74abb/?child=first)

template to tell us how to easily create multiple objects that share the same properties and the same methods, but whose properties' values may differ.

**In many programming languages, these templates are called "**[**classes**](https://courses.edx.org/courses/course-v1:W3Cx+JS.0x+3T2017/courseware/1f5a7a2285994fbd87d23829a53d29d2/c33324ad26584d6391466a8966b74abb/?child=first)**".**

* + In JavaScript 5 (also called ES5), we did not have such a concept, instead we had "constructor functions".
  + In JavaScript 6 (ES6), we have the concept of [classes](https://courses.edx.org/courses/course-v1:W3Cx+JS.0x+3T2017/courseware/1f5a7a2285994fbd87d23829a53d29d2/c33324ad26584d6391466a8966b74abb/?child=first), and the syntax is rather similar to what we find in other object oriented programming languages.

Let's introduce these two ways of defining "pseudo [classes](https://courses.edx.org/courses/course-v1:W3Cx+JS.0x+3T2017/courseware/1f5a7a2285994fbd87d23829a53d29d2/c33324ad26584d6391466a8966b74abb/?child=first)" with ES5's function constructors, and with ES6 [classes](https://courses.edx.org/courses/course-v1:W3Cx+JS.0x+3T2017/courseware/1f5a7a2285994fbd87d23829a53d29d2/c33324ad26584d6391466a8966b74abb/?child=first)!

### ES5's constructor functions, the new keyword

With JavaScript version 5 (and previous versions), you can define a pseudo-class template called "**a constructor function**". The syntax is the same as for creating a function, except that:

* 1. **By convention, its name is Capitalized.** The first letter of the function name is in uppercase, this is a good way to know, when you read someone else's code, that this is not a regular function, but a constructor function. **Its name is a noun, the name of the class of objects you are going to build.** Example: Person, Vehicle, Enemy, Product, Circle, Ball, Player, Hero, etc.
  2. You build new objects using the **new** keyword:   
       
     Examples (Car, Hero, Ball, Product are constructor function names):  
       
     var car = **new** Car('Ferrari', 'red');  
     var luke = **new** Hero('Luke Skywalker', 'rebels");  
     var ball1 = **new** Ball(10, 10, 20, 'blue'); // x=10, y=10, radius = 20, color = 'blue'  
     var p1 = **new** Product('Epson printer P1232', '183', 'Mr Buffa'); // ref, price, customer  
     etc.
  3. **The parameters of the function are the "constructor parameters": the new object that you are building will take these as its initial properties' values.** You can build a Hero, but you must give him/her a name, a side, etc.
  4. **You define** the**property names and method names using the this keyword**. But beware: the syntax is not the same as the syntax we used for singleton/simple objects. No more ":" and "," between properties. Here we use "=" and ";" like in regular functions.  
       
     Example:   
       
     function Hero(name, side) {  
         **this.name = name;**  
         **this.side = side;**  
         **this.speak = function()** {  
             console.log("My name is " + this.name + " and I'm with the " + this.side);  
         }  
     }  
       
     In a constructor function named "Hero", you will find properties declared like this: this.name this.side; and methods declared like this: this.speak = function() {...}
  5. **Very often some properties are initialized using the constructor function parameters**, so that the newly constructed objects will get an initial value for their properties. In this case, we use the this keyword to distinguish the property from the constructor function parameter:  
       
     Example:   
       
     function Hero(name) {  
         **this.name = name;**  
         ...  
     }



### Creating objects using the new ES6 classes

ES5's constructor function syntax is not easy to read. If someone does not respect the "conventions" that we've just discussed (start the class with an uppercase, etc.), then the code may work, but it will be difficult to guess that we are not in front of a regular function.

ES6 created a class keyword and a constructor keyword, along with advanced concepts that will be the subject of an "advanced JavaScript" course.

Main changes:

* 1. **A class is simply defined using the keyword class followed by the name of the class**
  2. **The unique constructor is defined using the constructor keyword followed by the parameters**
     + **The constructor is executed when an object is created using the keyword new**Example: let h1 = new Hero('Ian Solo', 'rebels');  
         
       This will call constructor(name, side) in the example below.
  3. **A method is simply defined by its name followed by its parameters (we no more use the keyword "function")**Example: speak() {...} in the source code below.

Here is the new version of the Hero "template", this time with the ES6 class syntax:

1. **class Hero {**
2. **constructor(name, side) {**
3. this.name = name; // property
4. this.side = side; // property
5. }
6. **speak() { // method, no more "function"**
7. return "<p>My name is " + this.name +
8. ", I'm with the " + this.side + ".</p>";
9. }
10. }
12. var darkVador = new Hero("Dark Vador", "empire");

* Line 1: a class is simply defined using the keyword class followed by the name of the class. Like for constructor functions, the convention is to use a noun, capitalized.
* Line 2: the constructor is defined using the constructor keyword. **THERE CAN BE ONLY ONE CONSTRUCTOR in the class.**  A SyntaxError will be thrown if the class contains more than one occurrence of a constructor method. No more use of the function keyword. Simply use the constructor keyword followed by the parameters.

The instructions in the body of the constructor are executed when an object is created using the keyword new followed by the name of the class, with arguments between parentheses. These arguments will be passed to the constructor.

* Line 7: a method is simply defined by its name followed by its parameters. **Again, no more use of the function keyword.**

#### 

### Creating objects with functions (factories)

**Here is a new one: objects can also be created by functions that return objects (factories)**

1. function getMousePos(event, canvas) {
2. var rect = canvas.getBoundingClientRect();
3. var mxx = event.clientX - rect.left;
4. var my = event.clientY - rect.top;
6. **return { // the getMousePos function returns an object. It’s a factory**
7. **x: mx,**
8. **y: my**
9. **}**
10. }

And here is how you can use this:

1. var mousePos = getMousePos(evt, canvas);
3. console.log("Mouse position x = " + mousePos.x + " y = " + mousePos.y);

The call to getMousePos returns an object that has an x and a y property.

### Static properties and methods

#### Class properties and methods vs. instances' properties and methods

Sometimes, there are methods "attached" to a class, not to an instance of a class.

For example, imagine the Hero class we've already seen, and we would like to know how many Star Wars's heroes have been created.

Instead, object oriented programming languages have the concept of "class properties" and "class methods" that complete the "instance properties" and "instance methods" that we've seen up to this point. Hero.getNbHeroes() means "Hey, class Hero, can you tell me how many heroes have been created using your class?". Class methods define the "class behavior", and instance methods define the instances' behavior. darVador.speak(); means "Hey, Dark Vador, please, tell us something!". I speak to Dark Vador and I'm expecting something creative from him, such as "I'm your father, Luke!".

It's the same for properties. If there is a property named nbHerosCreated in the class Hero, it represents the DNA of the class, not of the instances. You can say "the Hero class has the number of heroes it created", and you can say "Dark Vador has a name and belongs to the empire side", but not "Dark Vador has a number of heroes he created". We have class properties and instance properties.

#### The static keyword is used for defining class methods

##### Class methods

How do we distinguish them? By using the static keyword. When you see a method preceded by the static keyword, it means that you see a class property or a class method.

**The static keyword defines a static method for a class.   
  
Static methods are called without instantiating their class   
and can not be called through a class instance.   
  
Consequence: do not use instance properties in their body!  
  
Static methods are often used to create utility functions for an application (source: MDN).**

##### Class properties

Class properties should be defined after the class definition, and declared using the name of the class followed by the . operator and the name of the property. Example: Point.nbPointsCreated in the example below. A best practice is to ALWAYS use them this way.

There is another way to declare Class properties (using static getters and setters -- see next section, for advanced users), but we recommend using this one for beginners.

#### Example of creation and use of class methods and properties using an ES6 class

##### Source code:

1. class Point {
2. constructor(x, y) {
3. this.x = x;
4. this.y = y;
5. **// static property**
6. **Point.nbPointsCreated++;**
7. }
9. // static method
10. **static distance(a, b) {**
11. const dx = a.x - b.x;
12. const dy = a.y - b.y;
14. return Math.sqrt(dx\*dx + dy\*dy);
15. }
16. }
17. **// static property definition is necessarily outside of the class with ES6**
18. **Point.nbPointsCreated=0;**
20. // We create 3 points
21. const p1 = new Point(5, 5);
22. const p2 = new Point(10, 10);
23. const p3 = new Point(12, 27);
25. document.body.innerHTML += "<p>Distance between points (5, 5) and (10, 10) is " +
26. **Point.distance(p1, p2)** + "</p>";
27. document.body.innerHTML += "Number of Points created is " + **Point.nbPointsCreated**;

**[Advanced] ES6 getters and setters**

**Definition**

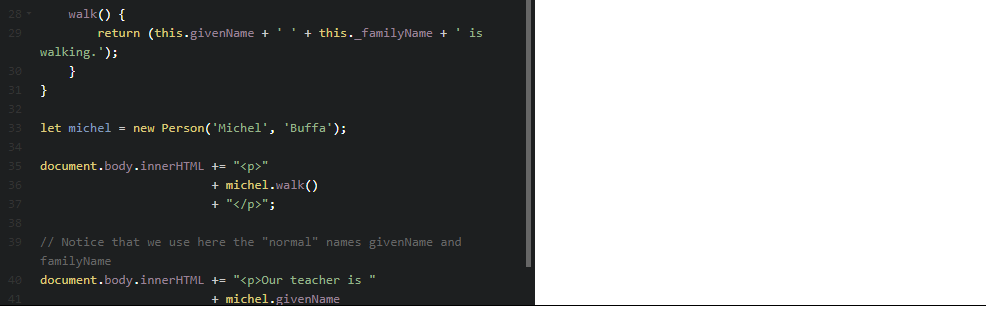
It is possible to use special methods that are called **getters**and**setters**. They allow to make some checks when one is trying to set a value to a property, or to do some processing when accessing it (for example for displaying it in uppercase, even if its value is in lowercase).

These special functions are called "getters" and "setters", and are declared using the keywords get and set followed by the name of the property they define.

Imp Note –

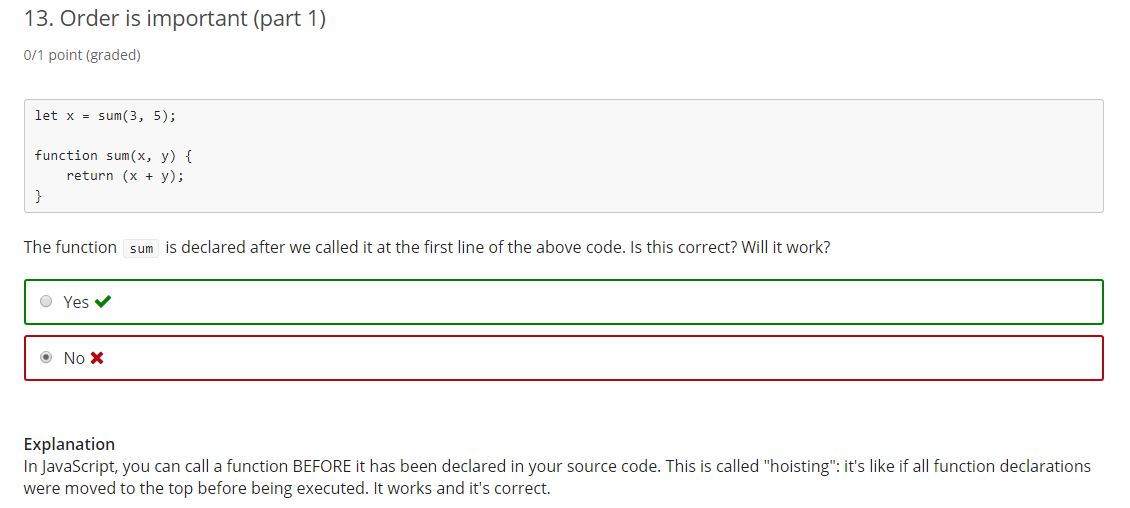
Notice that when you declare get familyName() {...} for example, you define implicitly a property whose name is "familyName" and that will be accessible using object.familyName, where object is an instance of the class. See lines 22-25 in the example above. Displaying the value of p1.familyName will call implicitly get familyName(), while p1.familyName = 'Smith'; will call set Name('Smith');

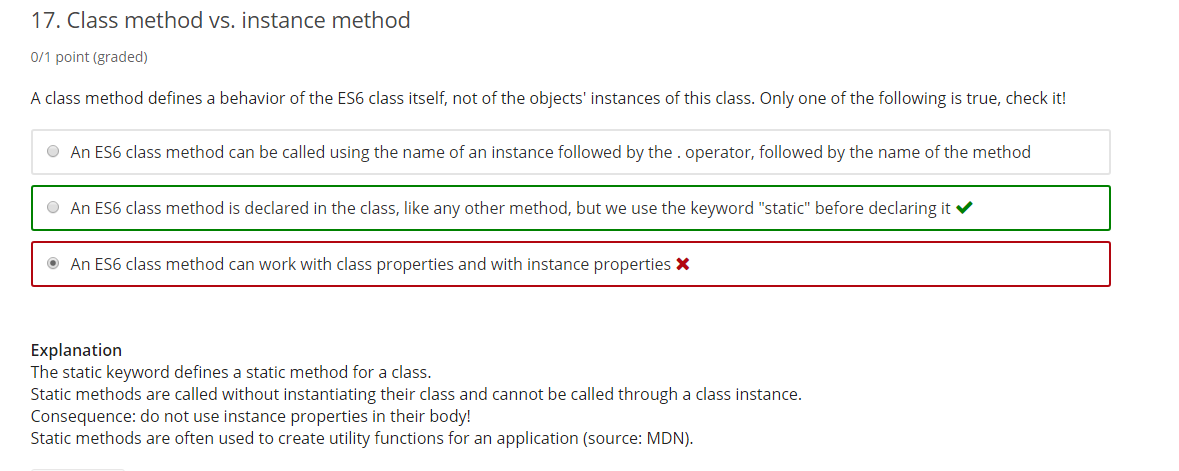
As get familyName() defines an implicit property named familyName, the convention is to use this.**\_**familyName for storing its value (the same name preceded by an underscore).

Some question –





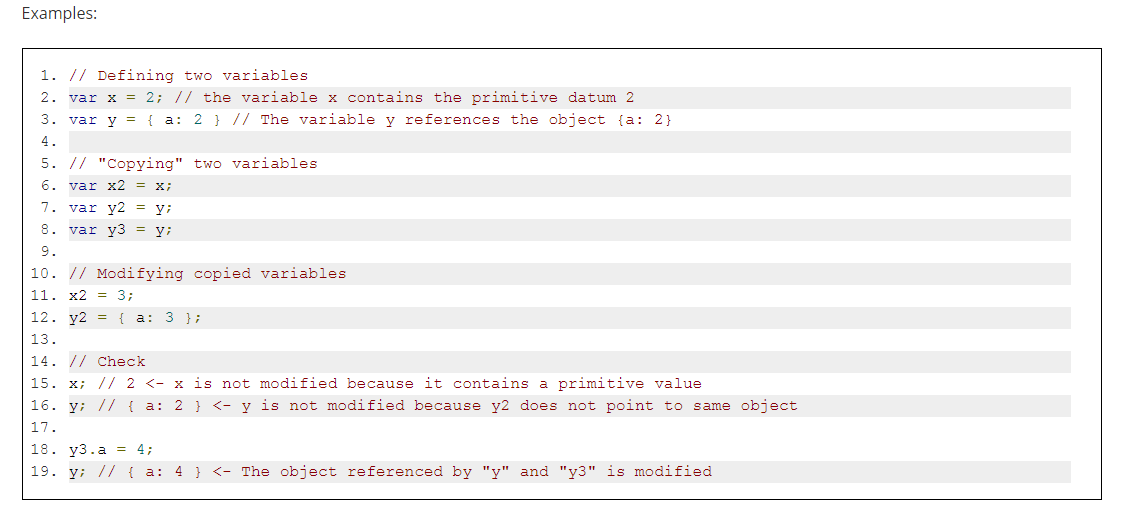


**References and objects**

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.



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

### Comparing two objectsscales

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

**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

Predefined functions are methods from the global object window:

1. > parseInt('10 little children');
2. 10
4. > window.parseInt('10 little children');
5. 10
7. > alert === window.alert
8. true
10. > prompt === window.prompt
11. true
13. > window.addEventListener === addEventListener
14. true

**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"

**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]

**The most useful methods of the built-in 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"]

### Built-in JavaScript 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" \t "_blank), 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"

**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

**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.

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!"

### Built-[in JavaScript class](https://courses.edx.org/courses/course-v1:W3Cx+JS.0x+3T2017/courseware/b108897ce3c4449c8326cda572490b8f/a32962a1e62848d890bc9a0d1599824e/?child=first): 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.

**Built-in JavaScript 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!**

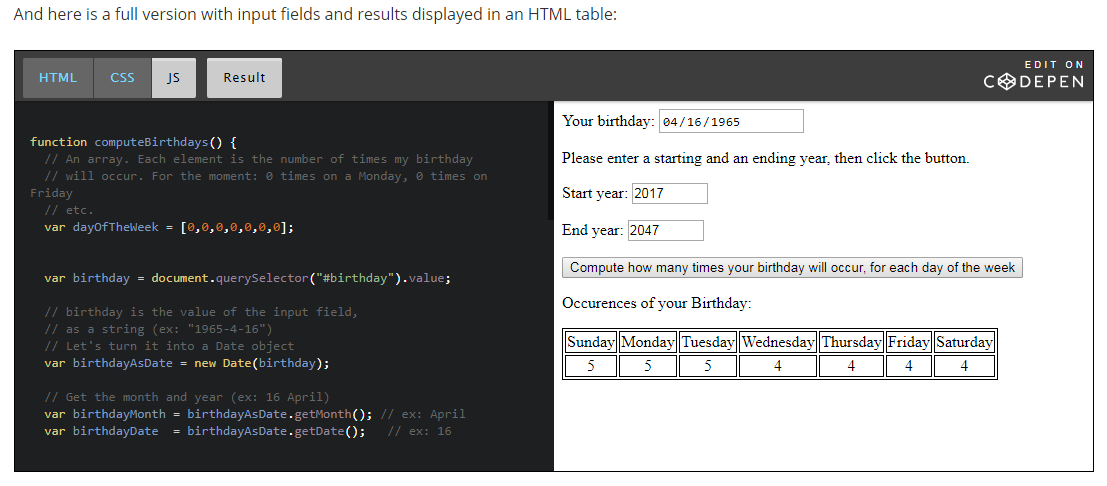
1. > var d = new Date(1965, 3, 16); // Michel Buffa's birthday
2. undefined
4. > d.getDay(); // Sunday is 0
5. 5
7. > d; // let's verify
8. **Fri** Apr 16 1965 00:00:00 GMT+0200 (CEST)
10. **> // Great, it was a Friday :-)**

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.



Important note –

* 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?

**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.
  + Link - <https://www.w3.org/WAI/tutorials/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>

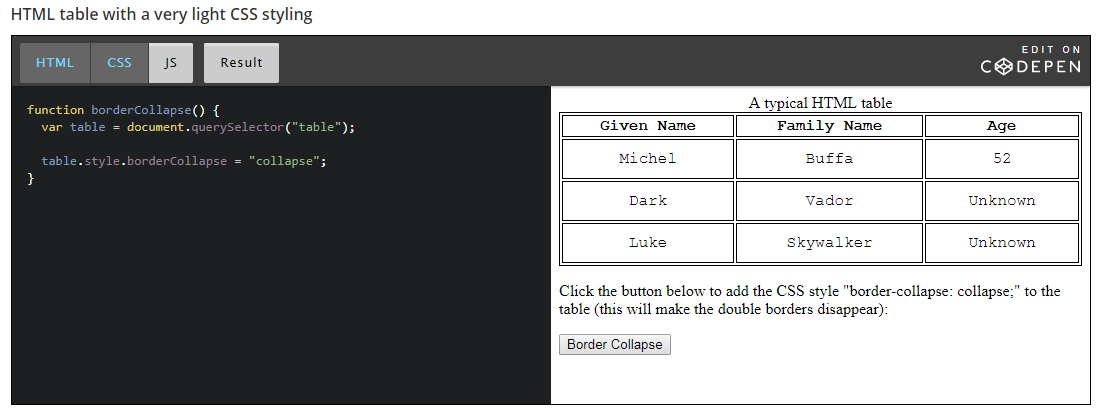
Most of the time, we add some CSS rules for rendering cell/row/table borders and for adjusting spacing between the text in the cells and the cell borders. Let's look at another example

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

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. }

Javascript code -

**Explanations**:

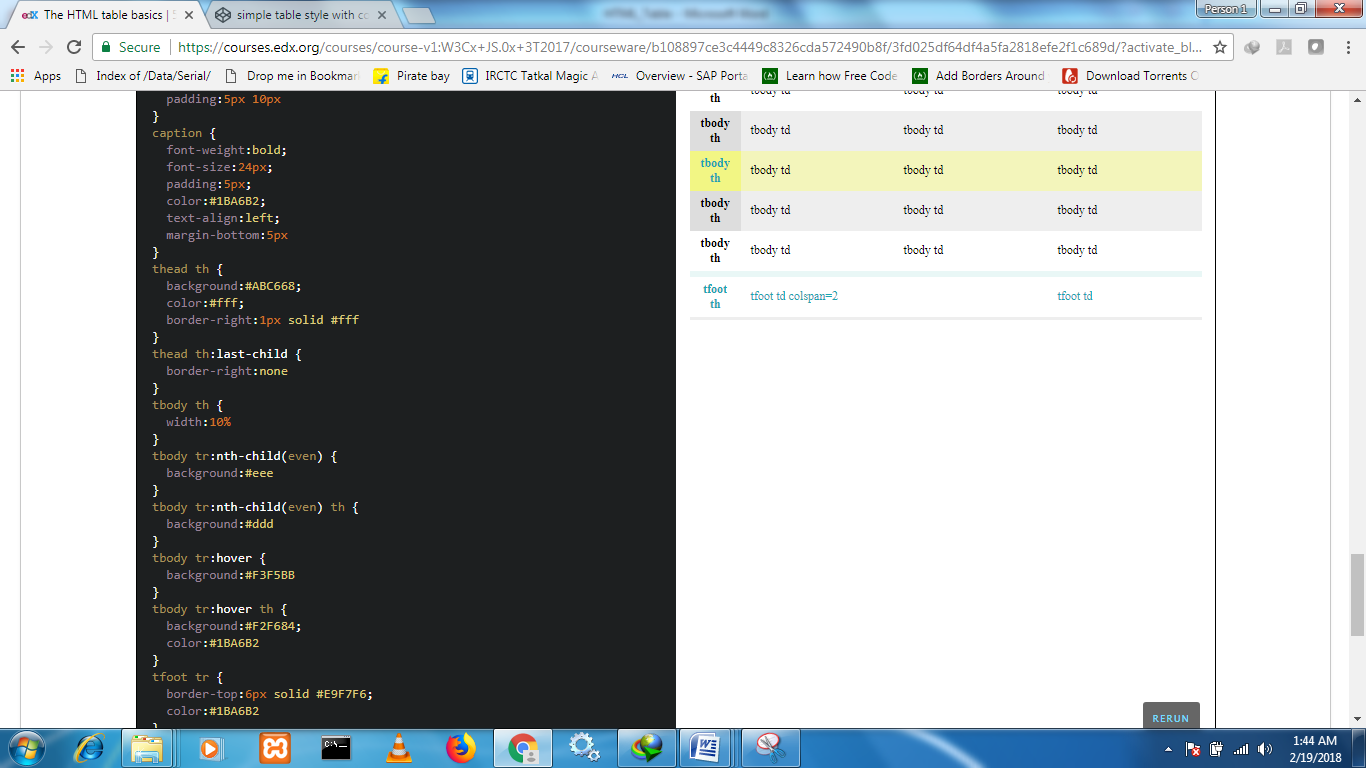
* *Line 1*: this rule says that the table will occupy the width of the window and will have a black, continuous border that is one pixel wide.
* *Line 7*: this rule says that table rows, table cells and table headers will also have a border and will use the font family Courier.
* *Line 12*: this says 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

#### 



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

**Introduction**

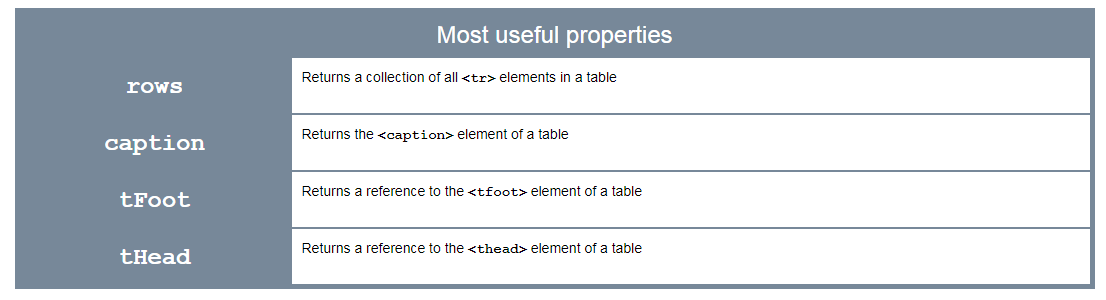
There is a JavaScript API associated with the HTML table elements that makes dynamic table management possible, enabling you to add or delete a row, add or delete a cell, modify the content of the cells, etc.

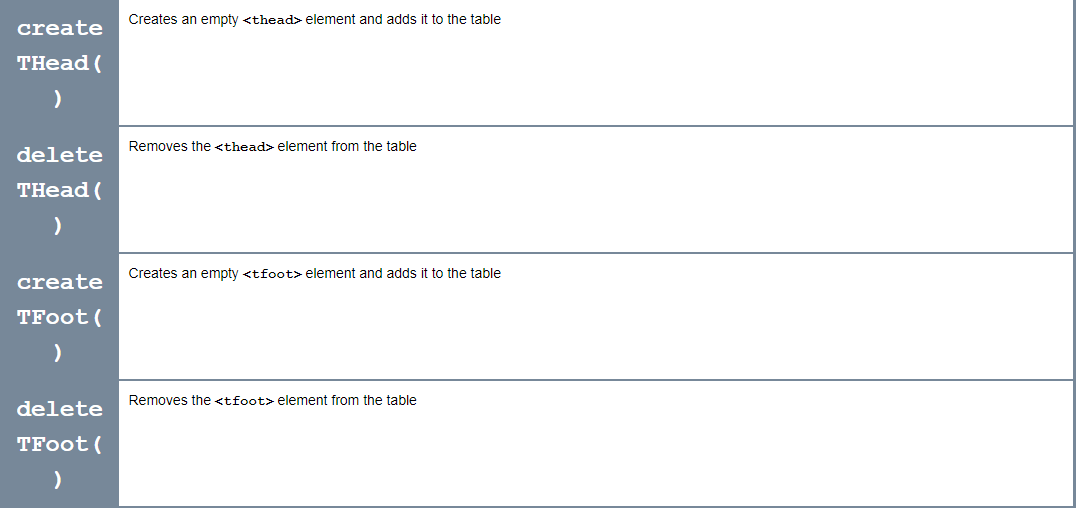
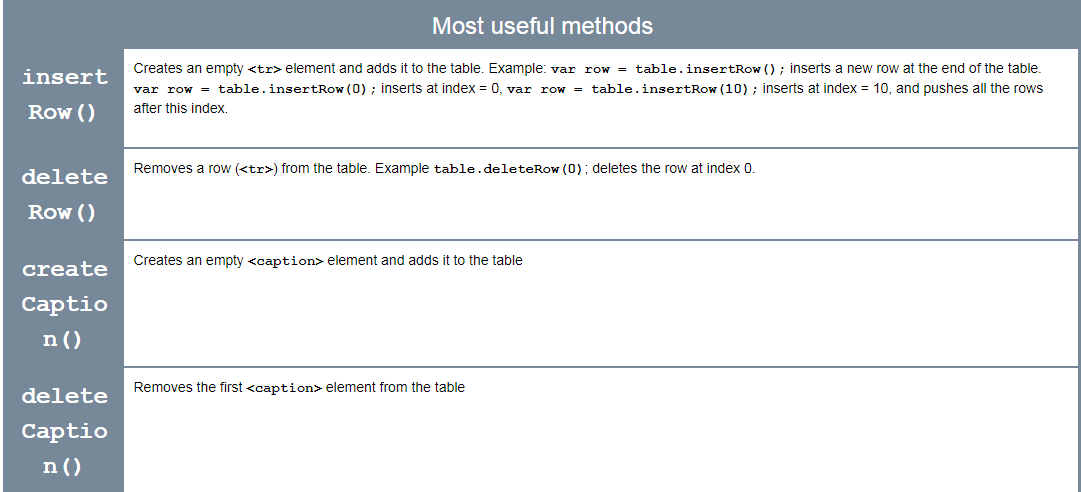
**The Table object (<table>)**

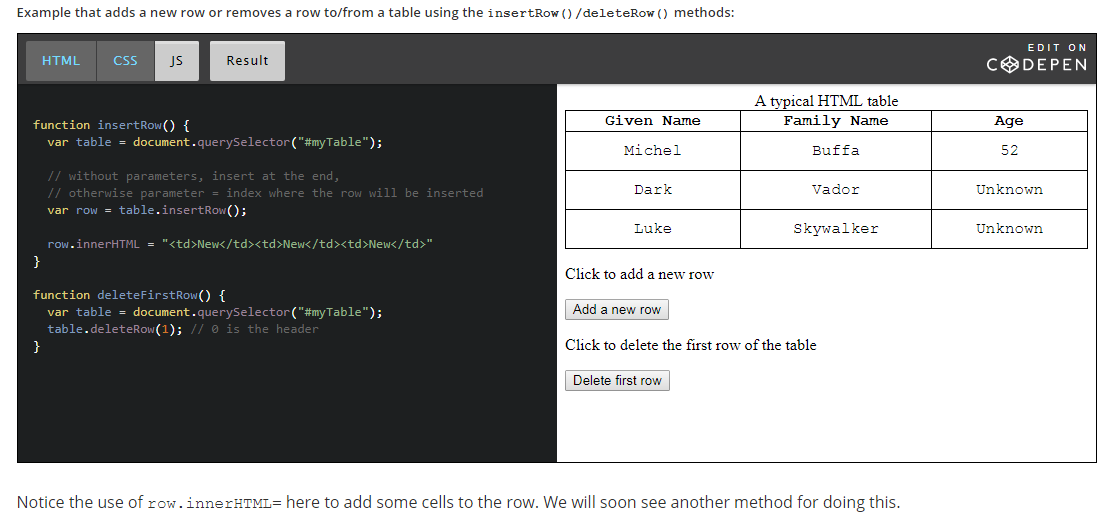
When you look for a table using the DOM API or the selector API, or when you create a table using the DOM API, you get a Table object:

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:







**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: 