****

**Javascript Introduction**

**- Introduction to Javascript –**

**edX – W3CX(Microsoft)**

1. **Module 4 Outline**

.

### **What you will learn in Module 4**

We continue exploring JavaScript Object Oriented Programming:

* You will learn new things about JavaScript object properties and methods
* You will see how to build many instances of objects using ES5 constructor functions and the new ES6 classes
* You will learn how to use the "new" keyword for building objects, and about the mysterious "this" keyword you already met in some examples
* You will learn about "class properties and methods" (vs. "instance properties and methods")

1. **Objects (part 2): properties and methods**
   1. **Introduction**

You're already familiar with the concept of objects, but so far we've only seen one simple form, called "objects literals" or "singleton objects". I think we've referred to them as "simple objects" in the course. Here is an example:

1. var js1 = {
2. courseName**:** 'JavaScript intro',
3. weeks**:** 5,
4. madeBy: 'W3Cx',
5. author: 'Michel Buffa' // no "," after the last property!, even if ES5/6 accept it
6. }

And we access properties values using the "." operator, like this:

1. > js1.author
2. "Michel Buffa"
4. > js1.weeks
5. 5

**However, we haven't explained 90% of what is going on, and what we can do with "objects".** Our objective in this module, is to explain the most important features of objects, while keeping it simple (more advanced topics will be taught in a future "JavaScript Advanced" course, such as prototypes, context binding, etc.).

**Features you will learn:**

* The relationship between JavaScript objects and arrays,
* What a "reference" is in a programming language,
* How to embed methods in your objects (functions inside an object),
* The "this" object that you very often encounter in Object Oriented JavaScript code,
* How to add methods and properties to your objects,
* How to make multiple objects of the same class using ES6 classes,
* The built-in JavaScript objects and classes: Array, String, RegExp, Date, Math, Error, etc. And, we will remind you about objects such as navigator, document, window, screen, etc.

**Features you will learn in an upcoming course:**

* JavaScript prototypes,
* Inheritance,
* Advanced manipulations of properties and methods,
* Methods such as bind, call, etc., that can be useful for changing the value of "this",
* And more!

**Knowledge check 4.2.1 (not graded)**

var myBoss = {

positon: 'The Boss!',

givenName: 'John, John the Boss!',

office: 31

};

**How do we call an object defined like the one shown above?**

* A JavaScript object literal
* No special name, objects are objects, that's all
  1. **From objects to arrays**

**[ --Live coding video: object's properties -- ]**

### **From objects to arrays**

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

**Important note:** Darth Vader is called "Dark Vador" in the French versions of SW, and, as a French tutor, I think it's cool to give to one of the heroes an international name. :-)

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!**

**Knowledge check 4.2.2 (not graded)**

let michel = {

job:'Your Teacher'

}

**Check the correct proposal:**

* michel.job and michel['job'] are equivalent
* michel.job and michel['job'] are NOT equivalent
  1. **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"
   1. **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).

* 1. **Elements, properties and methods**

**[ -- Live coding video: object methods -- ]**

### **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

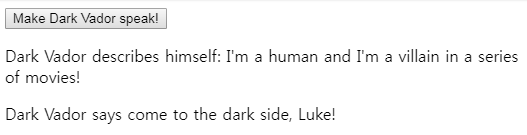
**Properties correspond to an object's DNA (its characteristics),  
and are nouns (age, name, etc.)**

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

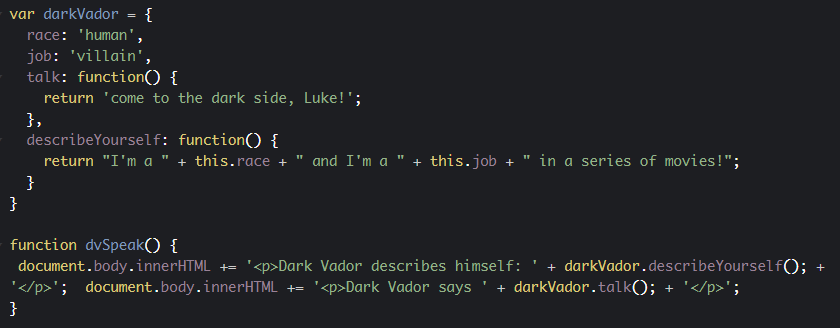
#### Calling a method

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

Let's see some examples:

****

****

****

JavaScript source code:

1. var**darkVador**= {
2. race: 'human',
3. job: 'villain',
4. **talk: function() {**
5. **return 'come to the dark side, Luke!';**
6. **}**
7. }
9. function dvSpeak() {
10. document.body.innerHTML += '<p>Dark Vador says ' +**darkVador.talk();** + '</p>';
11. }

In *line 1*, we created a simple object named darkVador, that has two properties (race and job) and a method (talk).

In the dvSpeak function, at *line 10*, we call darkVador's talk method. The syntax is a mix between the one for accessing a property (with the '.' operator), and the one for calling a function (with parentheses and ';' at the end).

When we write darkVador.talk(), we are executing the talk method of the object darkVador, but in plain English, we're just asking Dark Vador to talk. We invoke its behavior!

**Knowledge check 4.2.3 (not graded)**

let anObject = {

propertyName1: propertyValue1,

propertyName2: propertyValue2,

methodName1: function(...) {

// some code....

},

methodName2: function(...) {

// some code....

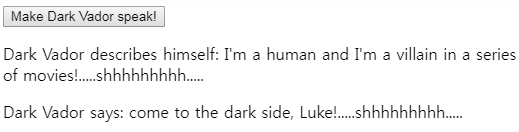
}

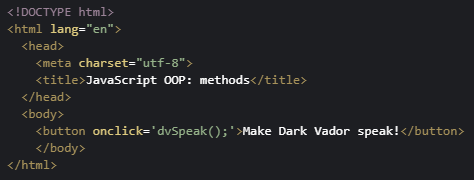
}

**In a JavaScript object, properties are nouns and methods are verbs. True or false?**

* True
* False
  1. **“this”: accessing properties**

#### Let's see the Dark Vador example with the use of this in a method

****

****

JavaScript source code:

1. var darkVador = {
2. race: 'human',
3. job: 'villain',
4. talk: function() {
5. return 'come to the dark side, Luke!' + **this.breathe();**
6. },
7. describeYourself: function() {
8. return "I'm a " + **this.race**+ " and I'm a " + **this.job**+
9. " in a series of movies!" + **this.breathe();**
10. },
11. breathe() {
12. return ".....shhhhhhhhh.....";
13. }
14. }
16. function dvSpeak() {
17. document.body.innerHTML += '<p>Dark Vador describes himself: ' +
18. darkVador.describeYourself(); +
19. '</p>';
20. document.body.innerHTML += '<p>Dark Vador says: ' +
21. darkVador.talk(); +
22. '</p>';
23. }

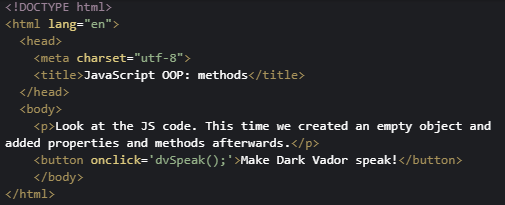
In this example, notice that the describeYourself method from the darkVador object uses the two properties name and job using the this keyword. We also call the breathe method from the two methods describeYourself and talk, using this.breathe();

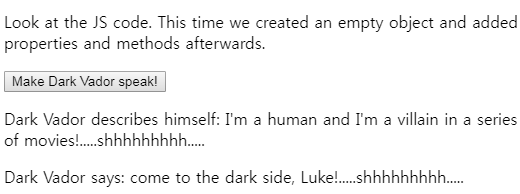
* 1. **Adding/deleting properties and methods**

### 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

Examples:

****

****

JavaScript code extract:

1. // empty object with properties/methods
2. var darkVador = {};
4. // add properties after darkVador has been created
5. darkVador.race = 'human';
6. darkVador.job = 'villain';
8. // add some methods
9. darkVador.talk = function() {
10. return 'come to the dark side, Luke!' + this.breathe();
11. };

Lines 5, 6 and 9: we can add properties and methods after the object has been created empty at line 2.

#### Deleting a property or a method

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

Example:

JavaScript code extract:

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

**Knowledge check 4.2.4 (not graded)**

let pacman = {};

pacman.color:'yellow';

pacman.shape: 'pizza';

**Is the above code correct?**

1. No
2. Yes

* 1. **Discussion topic**

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

#### Suggested topic of discussion:

* We simplified the explanations for "this" in this introductory course. Normally, "this" is the current object when you use it inside an "object literal"
* But... we also met "this" in event listeners (see in [this example from the course](https://codepen.io/w3devcampus/pen/gmygzV?editors=1000). Look at the onchange = "changePageBackgroundColor(**this.value**);") ...  
    
  In fact, the "this" keyword can be confusing in JavaScript. The key thing to remember is that it is bound to the calling object when the function is called, not when the function is created.  
    
  And in the case of event listeners, the callbacks are called by the browser... You can conclude that it's a good habit not to have event listeners in your objects: just use methods in which there is no confusion about "this".  
    
  Let's discuss that (or "this"?) in the forum :-)

1. **Objects (part 3): creating multiple objects**
   1. **Classes: definition**

### **Introduction: the concept of "class" in object oriented programming languages**

So far in this course, we've only used singleton objects: objects that only occur once: player, darkVador, etc.

Ok, this is not quite true, I'd forgotten that we created many balls in the module 2 game. We'll come back to this example further down the page!

But even with the balls from module 2, we did not use a 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.

For example, imagine Luke Skywalker, Ian Solo and Dark Vador. What do they have in common? They all are Star Wars heroes, they all have a name, they all belong to one side (the good/bad people, or rebels vs empire), etc. Imagine that we have a way of programming that describes not the objects themselves, but a "model", a "template" for these objects. We could call it StarWarsHero and use it for creating our heroes' objects.

Imagine the balls from module 2: they all had the same shape (circle), the same x, y, radius and color properties, but they were all different. They all belonged to the same class of object (ball), but they were all different in terms of their properties' values.

**In many programming languages, these templates are called "classes".**

* 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, 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" with ES5's function constructors, and with ES6 classes!

* 1. **ES5’s constructor functions, the “new” keyword**

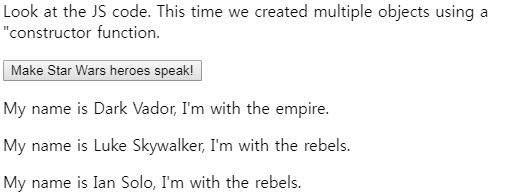
**[--Live coding video: ES5 constructor functions, the "new" keyword --]**

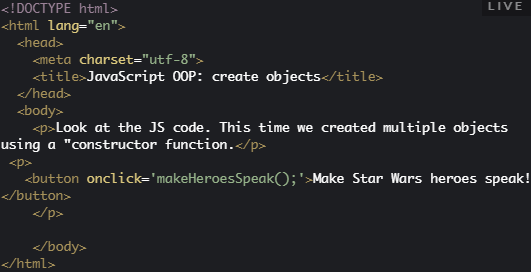
### **ES5's constructor functions, the new keyword**

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;**  
       ...  
   }

#### Full interactive example that uses a constructor function

****

****

JavaScript source code:

1. **function Hero(name, side) {**
2. this.name = name; // code outside of methods is usually for initializing
3. this.side = side; // the properties. Very often, they match the parameters
4. this.speak = function() {
5. return "<p>My name is " + this.name +
6. ", I'm with the " + this.side + ".</p>";
7. }
8. **}**
10. **var darkVador = new Hero("Dark Vador", "empire");**
11. **var luke = new Hero("Luke Skywalker", "rebels");**
12. **var ianSolo = new Hero("Ian Solo", "rebels");**
14. function makeHeroesSpeak() {
15. document.body.innerHTML += darkVador.speak();
16. document.body.innerHTML += luke.speak();
17. document.body.innerHTML += ianSolo.speak();
18. }

*Lines 1-9*: see how the constructor function is declared: the function name starts with an uppercase letter '**H**ero'. The parameters have the same name as the properties they correspond to (name, side). And in the first source code lines after the function declaration, we initialize some properties using these parameters (*lines 2 and 3*). We use the this keyword to distinguish the property and the parameter. You will often see things like: this.name = name; this.age = age; etc.

*Lines 11-13*: creation of three heroes. We use the same constructor function (Hero) along with the new keyword. Luke, darkVador and ianSolo ARE each a Hero, and share the same properties (name, side, *lines 2 and 3*) and the same behavior (they can speak, they all have a speak method, declared at *line 5*).

* 1. **Creating objects using the new ES6 classes**

**[ -- Live coding video: ES6 classes -- ]**

### 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:

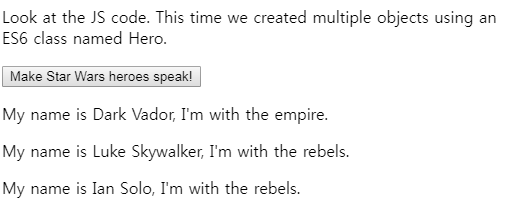
1. **A class is simply defined using the keyword class followed by the name of the class**
2. **The unique constructor is defined using the constructor keyword followed by the parameters**
   * **The constructor is executed when an object is created using the keyword new**Example: let h1 = new Hero('Ian 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.

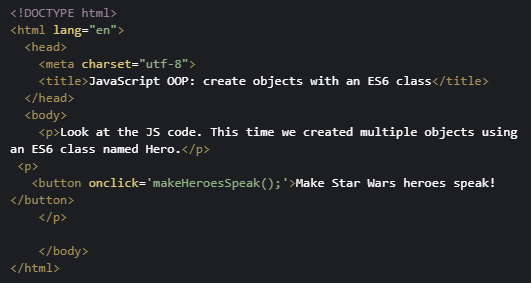
Here is the new version of the Hero "template", this time with the ES6 class syntax:

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.**

#### Interactive example that uses an ES6 class to create Star Wars heroes



****

****

* 1. **You must declare a class before using it!**

### **You must declare a class before using it!**

Unlike functions, classes must be declared BEFORE using them.

An important difference between function declarations and class declarations is that function declarations are "hoisted" and class declarations are not. This means that you can call a function BEFORE it has been declared in your source code. This is not the case with ES6 classes!

**You first need to declare your class and then access it, otherwise code like the following will throw a ReferenceError**:

Incorrect version => you try to create an instance of a class before it has been declared:

1. var p = new Rectangle(); // ReferenceError
3. class Rectangle {...}

Correct version =>

1. class Rectangle {...}
3. var p = new Rectangle(); // WORKS !
   1. **Creating objects with functions (factories)**

#### We have already seen three different ways to create objects (literals, constructor functions and ES6 classes)

Objects can be created as “literals” :

1. var darkVador = { firstName:’Dark’, lastName:’Vador’};

Objects can be created with the keyword new and a constructor function or an ES6 class:

1. var darkVador = new Hero(’Dark Vador’, ’empire’);

#### 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.

* 1. **Static properties and methods**

**[ -- Live coding video: 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. If zero hero has been created, it's obvious that we could not use this property with an instance of the class such as Dark Vador: darkVador.getNbHeroes(); this would make no sense.

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

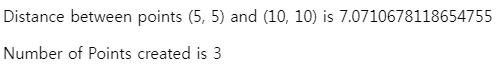
Class properties should be defined after the class definition, and declared using the name of the class followed by the . operator and the name of the property. Example: Point.nbPointsCreated in the example below. A best practice is to ALWAYS use them this way.

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**;

****

* 1. **[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).

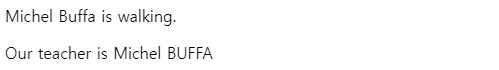
These special functions are called "getters" and "setters", and are declared using the keywords get and set followed by the name of the property they define.

Typical use (lines 7 and 11):

1. class Person {
2. constructor(givenName, familyName) {
3. this.givenName = givenName; // "normal name"
4. **this.\_familyName = familyName; // starts with "\_"**
5. }
6. **get familyName()** {
7. return **this.\_familyName**.toUpperCase();
8. }
9. **set familyName(newName)** {
10. // validation could be checked here such as
11. // only allowing non numerical values
12. **this.\_familyName**= newName;
13. }
14. walk() {
15. return (this.givenName + ' ' + **this.\_familyName**+ ' is walking.');
16. }
17. }
18. let p1 = new Person('Michel', 'Buffa');
19. console.log(p1.familyName); // will display BUFFA in the devtool console
20. // this will call implicitly get familyName();
21. p1.familyName = 'Smith';    // this will call implicitly set familyName('Smith');

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).

****

* 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 a few optional projects.

#### Suggested topics

* Did you now that ES6 classes are just "a syntactic sugar"? In fact they are equivalent to constructor functions from ES5...
* There are two sorts of object-oriented languages: class-based languages and prototype-based languages.  
  JavaScript is a prototype-based language. In this introductory course, we managed to avoid this term! Without getting into too much details, you might be curious about prototypes and maybe read some Web pages related to those.  
  And yes, ES6 classes are not "real classes"... They are meant to make developers' lives easier, i.e., for the developers who already know a class-based language such as Java, C#, etc.
* [Advanced] There is a powerful way to define "pseudo classes" using constructor functions: it's called ["The Black Box Model"](https://hacks.mozilla.org/2014/08/black-box-driven-development-in-javascript/). I recommend it to those of you who are rather comfortable with Object Oriented Programming concepts. Give it a try!

#### Optional projects

* Try to write one of the example from the previous modules without using any single time the keyword "function", use only ES6 classes and instances. In case of problems -> go the the forum and share your experience, this will be very useful for all students to see what sort of problems can occur when moving from a functional approach to an object-oriented approach
* Build an ES6 class-based contact manager!

1. Try to build a small database (in a JavaScript array) that will hold your contacts. You will use ES6 classes for defining:  
   1) a Contact class, with givenName, familyName, phoneNumber, etc. and  
   2) an HTML set of input fields (not inside a form) for creating new contacts + an "Add contact" button. When you click on the button, it calls an addContact() callback of your own that will create a new contact and add it to your database (using the push method on arrays).
2. [ADVANCED] input fields and buttons inside a form!  
   **Beware**: either do not put your input fields and buttons inside a <form> or the buttons will submit the form (this is their default behavior, unless you add an attribute type="button" to the buttons). Or you might also declare <form onsubmit = "return processMyForm();">, this will call the method processMyForm  
   (You can change this name if you like) when the form is submitted. In the processMyForm method, get the content of the input fields, build a contact, add it to the array etc. And then, do not forget to return false to avoid the submission of the HTML form).
3. It would be cool to also have a listContact() function that will generate a list of contacts (create <ul>...</ul> with <li>...</li> inside, one for each contact).
4. Now, try to write an ES6 class ContactManager (or you could also use an object literal for that..., but let's try  practicing ES6 classes!), that will have the array of contact as a property.
5. Create an instance db of this class:  
   let db = new ContactManager();
6. Add in the ContactManager class an add(c) and a list() method (for adding a contact c to the array of contacts, and for listing the contacts).
7. Now, when you press the buttons, the addContact() method from step 1 will call db.addContact(c) where db is the instance of your ContactManager class and c is an instance of the Contact class.
8. Feel free to customize this project with nice CSS, etc.