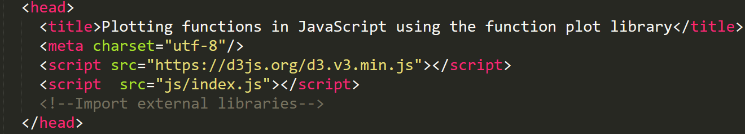
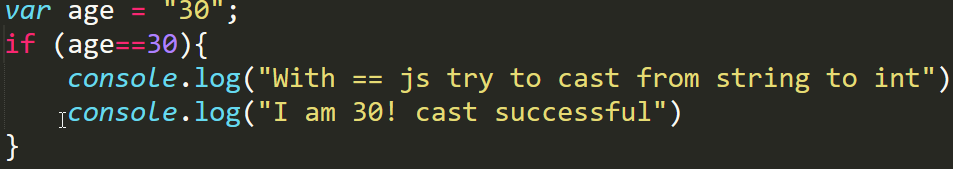
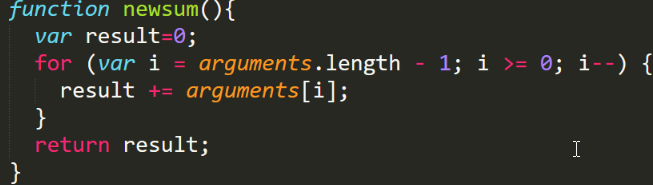
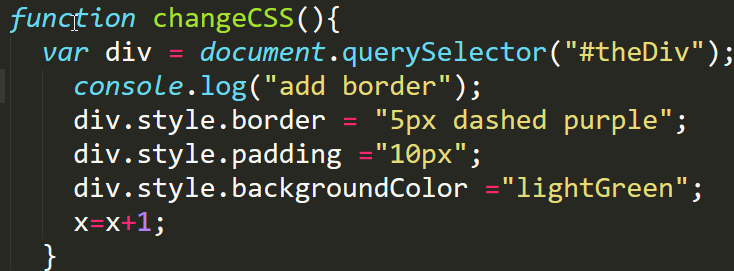
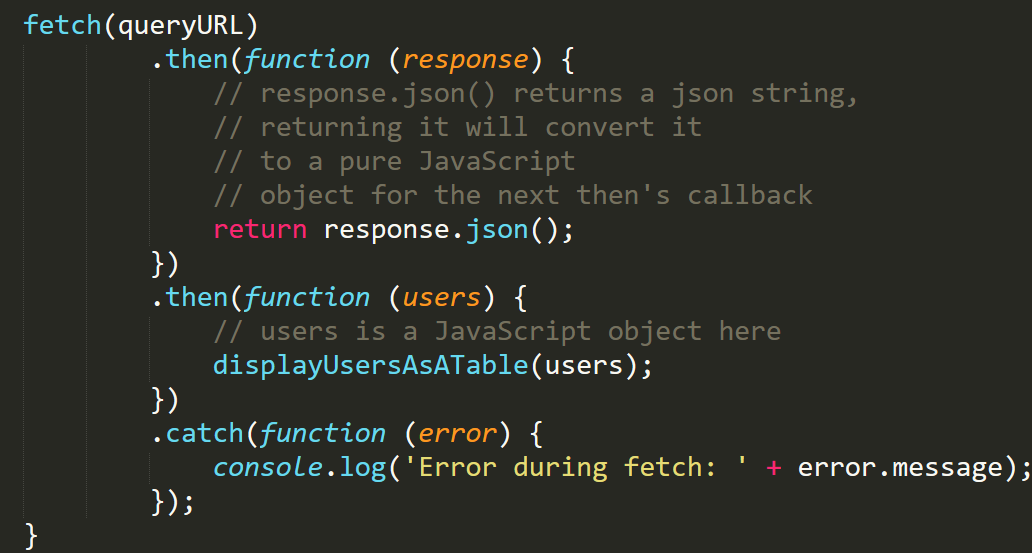
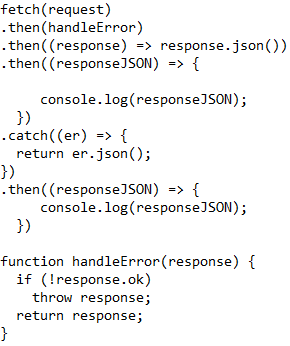
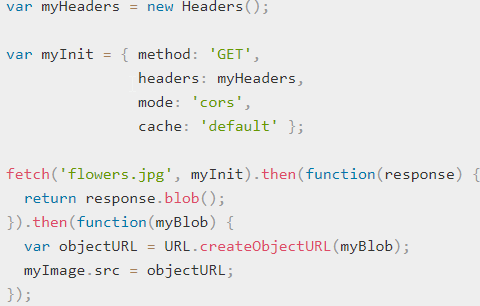
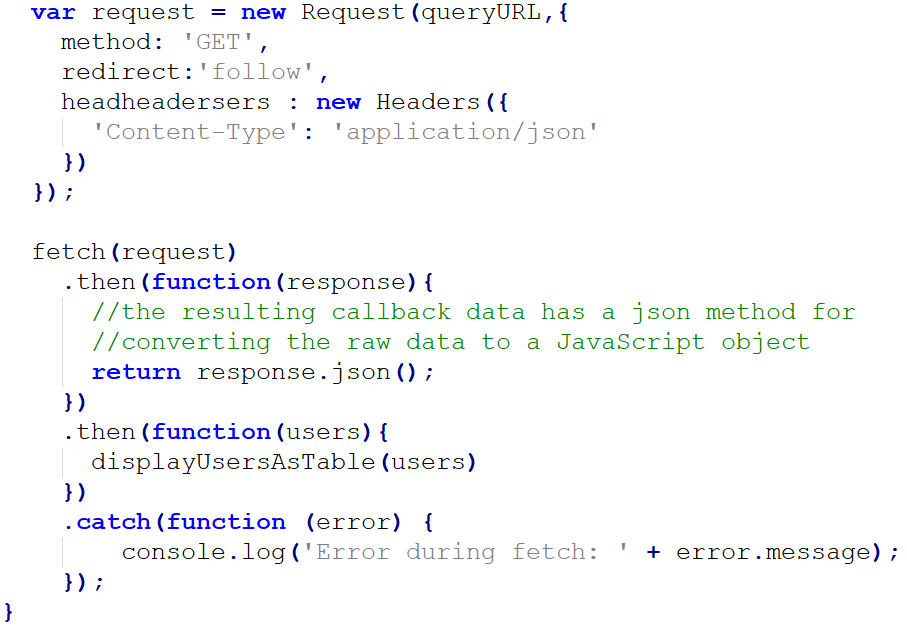
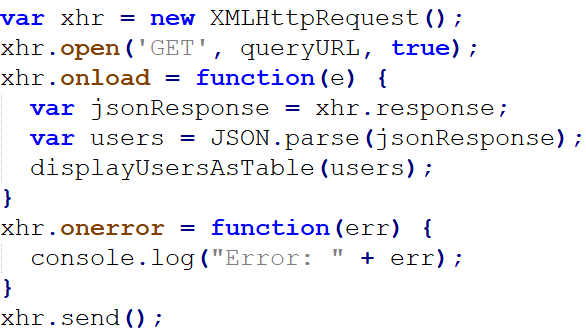
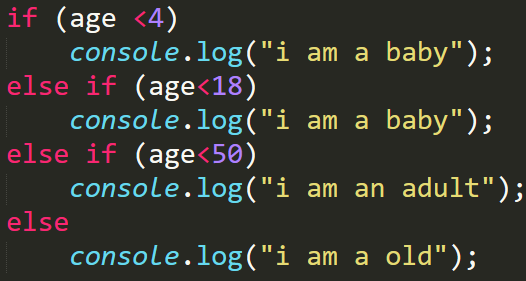
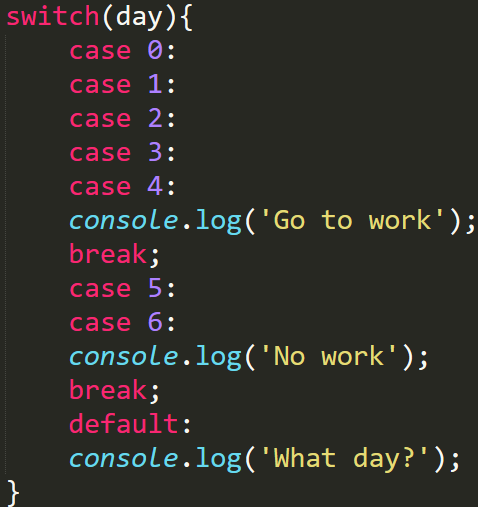
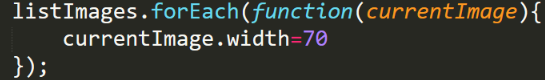
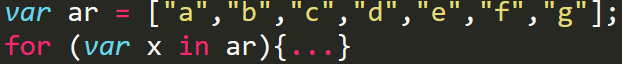
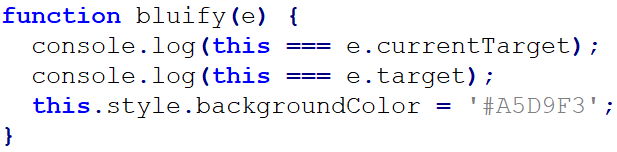
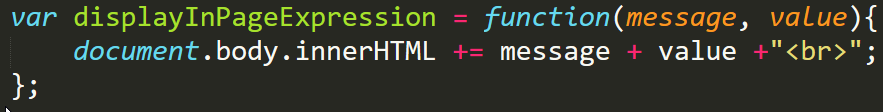
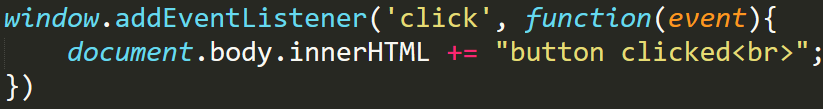
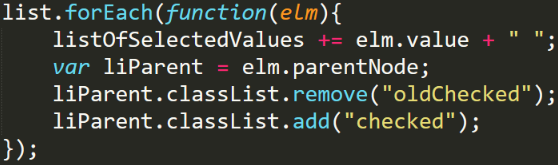
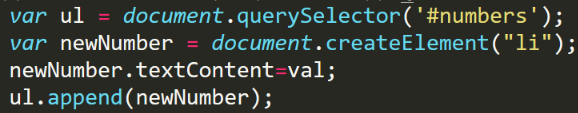
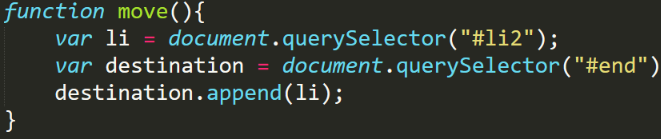
JavaScript

**W3Cx: JS.0x JavaScript Introduction**

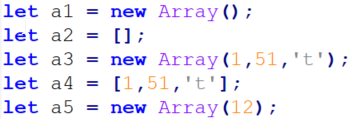
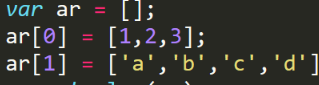
**01 Introduction to JavaScript**

* There are different ways to import a piece of JavaScript inside an HTML page, the most common is to use the <script> tag to import js files.
  + This can be used to import external libraries (1) or our index.js file (2)
* Inside <script> we can also directly write some JS code, but this is a bad practice because will break the separation between representation and functionality.
* The page can be configured to execute JS code in precise moment of its life. A typical example is after the page loaded to generate dynamically calculated html. In this case we use the property onload of tag body to call the function plot.
* debugger; The debugger statement stops the execution of JavaScript, and calls (if available) the debugging function.
  + It’s equivalent to a break point
  + you activate debugging in your browser opening the devtools (F12 key)
* There are only few data types: number(float) - string - Boolean - undefined - object - null(object)
  + undefined means a variable has been declared but has not yet been assigned a value
  + null instead is an assignment value. It can be assigned to a variable as a representation of no value
  + Array are objects: var a = [1,2,3,4];
    - Dimension is not fixed you can add a new element a[4]=8;
  + Numbers variable have 3 specials values : +Infinity | -Infinity | NaN Infinity (or +Infinity) represents all number values greater than 1.79769313486231570e+308 and -Infinity represents values smaller than -1.79769313486231570e+308. Nan represents not-a-number value.
    - NaN works similar to sql null value: Nan is equal to nothing, not even to itself. But you do have a function to check the NaN value: isNaN(0/0) //true
    - Any operation with Infinity gives Infinity as a result
* There are two commands to create a variable: let and var.
  + var always defines a function variable local to the whole function
  + let instead has as scope only the block of instruction where variable is created
  + Variable created outside a function are always global variables.
* Use === Instead of == JavaScript utilizes two different kinds of equality operators: === /!== and == /!= It is considered best practice to always use the first set when comparing. The triple-equals operator never does type coercion. It returns true if both operands reference the same object, or in case of value types, have the same value. However, when using with == and != you can run into issues when working with different types: js will try to coerce the values to enable the comparison.
  + For example it will try to cast a string to number.
* Function are declared using the keyword function
  + function plot(){...}
  + function always returns a value. if a return statement is not present then it will return undefined.
  + functions can have a variable numbers of unspecified arguments, an array named arguments is created automatically in each function, it contains all the call parameters of the function.
* If necessary from JS we can read an HTML element using its id: with the command document.querySelector("#identifier"). Just add a # before the id when using a CSS selector
  + Pay attention on the timing of this call: if the target element has not been already rendered the function will return null.
*  From JS it’s possible to change the css of html’s elements using the style properties:
* There are different ways to call a JS function from an HTML page:
  + We have already seen the property onload of body tag.
  + Js can handle human interaction as the click on a button:
    - <button onclick="buildTable()">
  + Same example with classic input fields:
    - <input type="text" id="inputField" oninput="showWhatWeTyped()"/>
  + All event type of all the different tags can be managed by a js function
* The 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() method that provides an easy, logical way to fetch resources asynchronously across the network. The simplest use of fetch() takes one argument, the path to the resource you want to fetch, and returns a promise containing the response (a Response object). In this case the response is a Json so we can convert into an object using response.json() method.
  + Fetch only rejects a promise when a network error is encountered, although this usually means permissions issues or similar. Basically it will only reject a promise if the user is offline, or some unlikely networking error occurs, such a DNS lookup failure.
  + fetch provides a simple ok flag that indicates whether an HTTP response’s status code is in the successful range or not. if(response.ok) ... you probably want to create a generic error handling function you can use for all of your fetch calls
  + fetch() accepts a second optional parameter, an header that allows you to control a number of different settings:
* A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future. A Promise is in one of these states: pending, fulfilled, rejected. A pending promise can either be fulfilled with a value, or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's “then” method are called. (If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.)
* Fetch can be used also with a Request object Only the first constructor parameter (queryURL) is mandatory the second is equivalent to the header seen before.
* ****The same request can be made using XhR2 XMLHttpRequest:
* It’s necessary to configure two call backs that will be used depending on the result of the call: in case of success onload otherwise onerror. Pay attention that onload include all the different answer types: for example a 400(bad request) will use onload instead a no answer will trigger onerror.
  + xhr.status to check the result type: if(xhr.status===404) …
  + xhr.response would be the same of this.response
  + This type of request doesn’t return a promise.
* The call is asynchronous started in back ground when the *send()* command is hit. On competitions the result will be elaborated with a call back to onload/onerror method.

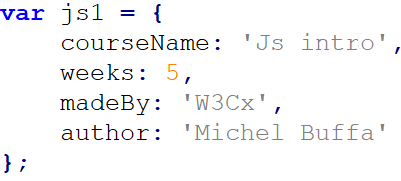
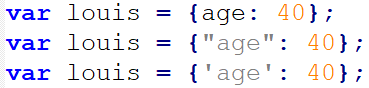
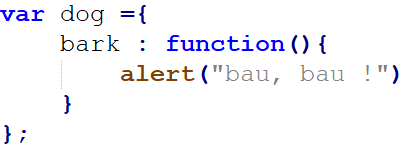
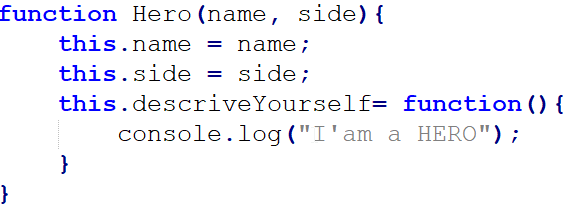
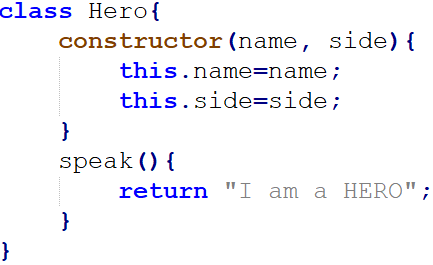
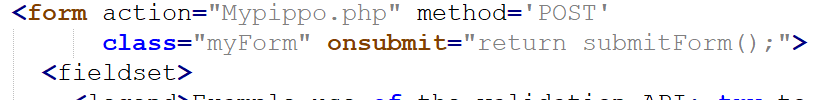
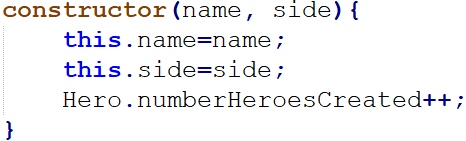
**02 Adding interactivity to HTML documents**

* If operator is present inside the language, with its friends else if and else.
* The following values are evaluated as false : false - undefined - null - 0 - NaN - empty string
  + Everything else is evaluated as true!
  + var b = (0/0) || 4 ; b will be 4!
* Switch statement work in the classic way:
* Ternary operator is present inside js: var max = (min<10) ? 10 : min;
* The classic do – while – do while loops are presents
* For can be used in three different ways
  + Classic for loop
  + ForEach works in a little different way: you need to pass a callback function that take as parameter the object we are iterating on
    - The callback function can take up to 3 parameters: 1° current element, 2° element's index, 3°array itself
    - a.forEach(function(value, index, array){…}
  + With the in function: at each execution of the block statement, x is assigned with the name of one of the properties (the keys) of the object.
  + pay attention x is 0 - 1 -2 -3 ... and not “a” – “b” – “c”…
* *Anonymous function*: functions without a name that can be assigned to a variable. The variable can be used to execute the function
* *Inline functions*: sometime a method need a function as parameter. We can pass an existing function, an anonymous function or we can create a function on the fly using an inline function:
  + In the example we are registering an event listener that react on mouse click. The callback function used is an inline function.
* An event often used to execute js code is window.onload = …
  + This event occurs when a window has loaded (including all its resources: images, etc.). This is very useful when you want to run JS code and be sure that the DOM is ready (to be sure that a document.getElementById(...) or document.querySelector(...) will not raise an error because the document has not been loaded and elements you are looking for are not ready).
  + Usually, the onload event listener corresponds to a JavaScript function that can be seen as "the main" function of your Web application.
* There are different ways to add an eventListener to an HTML object.
  + There’s the method addEventListener that can add any type of event, it take two parameters: event to manage and callback function.
  + If present we can use the direct property: window.onkeyup = processKeyUp;
  + Of course we can always use an anonymous or inline function.
* When a function is used as a listener we have access to the event properties currentTarget and target. Remember that events bubble by default so the difference between the two is: target is the element that triggered the event (e.g., the user clicked on) currentTarget is the element that the event listener is attached to.
  + Second condition is true if target and currentTarget are the same object
* To select some elements from JS we can not only use the querySelector command but also the querySelectorAll(tagName). This command returns all the elements that match the search criteria:
  + let listImages = document.querySelectorAll("img");
  + Return all the tag images
* querySelectorAll also works using the class of the HTML elements. document.querySelectorAll('.className'); just add a '.' before the class name when using a CSS selector.
* Another way to select an element is document.getElementById(“id”); this will return the single element with the specified id
* It’s possible to change the HTML inside an element using the property innerHTML.( innerHTML corresponds to all the sub DOM contained inside the tag) This field is a string and can be used to write directly htmal tag:
  + p.innerHTML = ‘<p>The house …</p>'
  + p.innerHTML += <p>…</p>' It’s also possible to add new part to the existing content
* JS can change the classes [css selectors] of an element (using classList property), in this way its representation will change dynamically without the need to work directly on element style properties
  + div.classList.remove(‘…’);
  + div.classList.add(‘…’);
  + div.classList.contains('foo'); will return true if the element contains the foo class
  + Interesting is the method toggle: div.classList.toggle('foo'); Check if class contains the class "foo", if it does, "foo" is removed, otherwise it's added
* Instead of injecting directly html code from the JS we can work to a higher level, creating and adding element.
  + In the example we use the document.createElement(‘li’); command to generate a new li element. And with the append method we add it to the intended ul.
* We can move an element from one point to another of the document: appending an existing document will remove from its parent and add to the new one

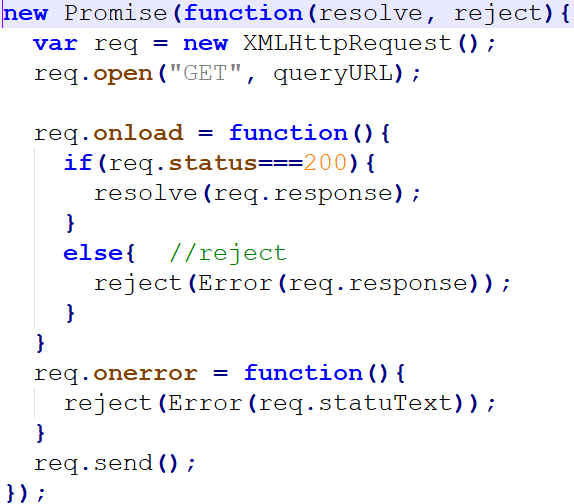
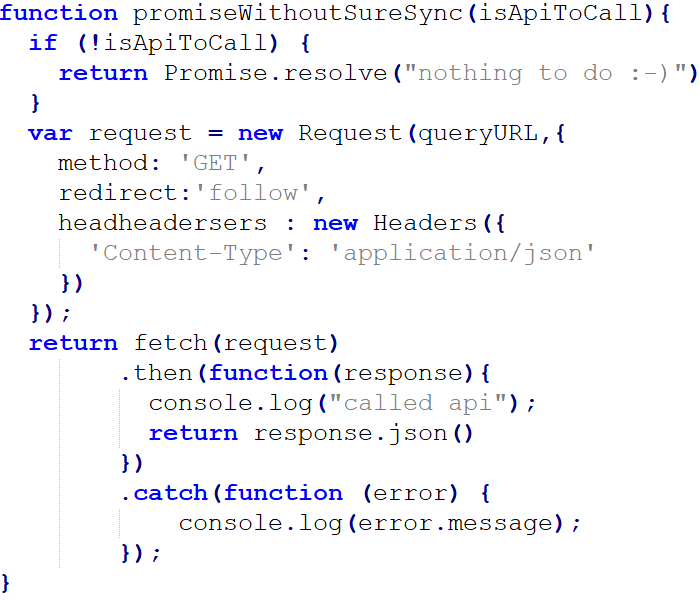
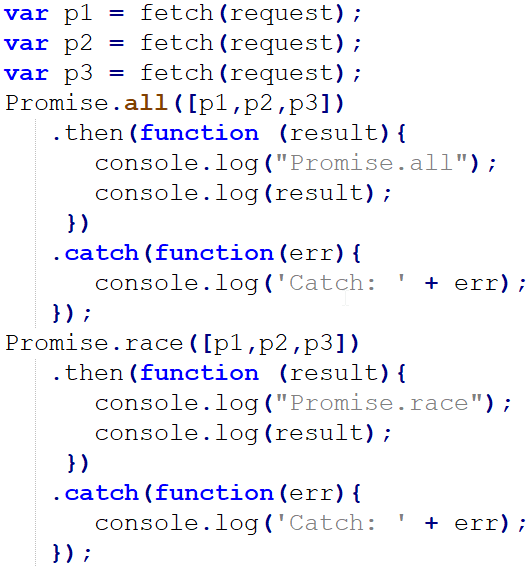
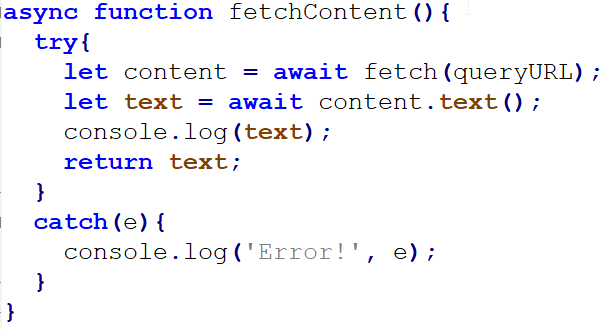
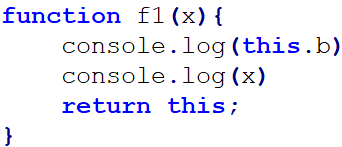
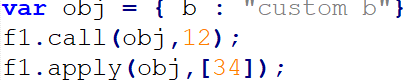
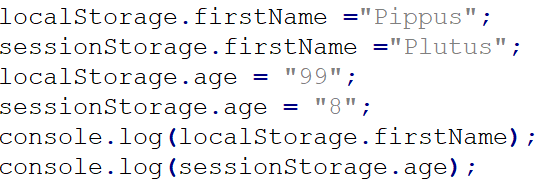
**03 Playing with some HTML5 APIs**

* Arrays can be declared in different ways: a1 and a2 are empty arrays instead a3 and a4 contains some elements. If we call new Array passing a single parameter (a5) this will be the size of the array and each elements will be undefined (**bad practice**).
* Arrays don’t have a fixed dimension, you can change it dynamically at run time, To add an element to an existing array we can use letters.push('j'); or letters[letters.length]='h';
  + If you add element specifying its position pay attention to not live holes (undefined elements) in the array.
  + it's possible to change array length at run time Letters.length=10; The new elements will be filled with undefined
  + Letters.pop() remove last element
* Arrays have a sort function but you need to tell it how to sort the array passing a comparing function. This function takes 2 parameters (a,b) and return -1 if a comes before b, 1 if it is positioned after and 0 if are equal.
  + persons.sort(compareByAge);
* We can build arrays of arrays and each one can have different dimension and type on contained objects.
* Strings are similar to array but you cannot add elements to an existing string; you can’t use the push/pop methods for adding - removing characters at the end of the string. Strings are not modifiable at all:
  + s[0] = 'U'; will give back no error but it's not affecting s
  + var s = s + "hello"; builds a new string and assigns it to s.
* s.slice(4,9) - s.substring(4,9) return a part of the original string. Take two parameters: the start and end index of the slice (element at end index will NOT be included in the slice)
  + There is a difference between slice and substring when the second parameter is negative. Stay away from negative parameters **bad practice.**

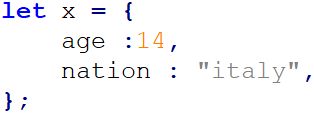
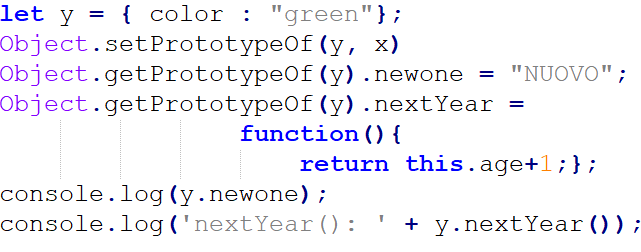
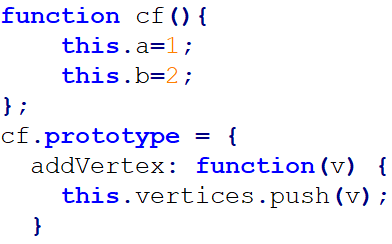
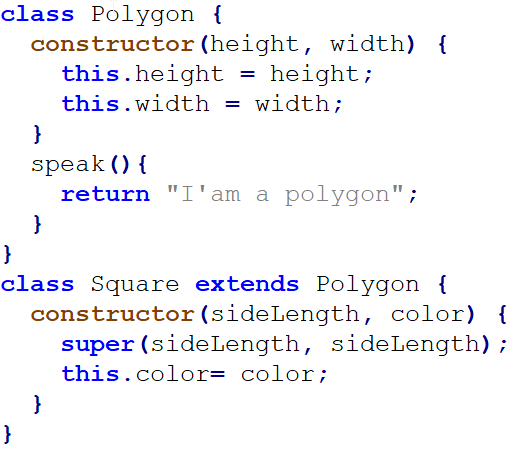
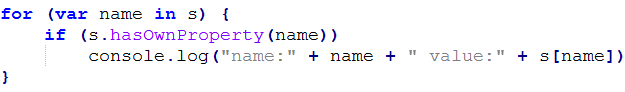
**04 Structuring data**

* Objects literals or singleton objects are the most basic form of objects, they can be created simply by opening brace and listing its properties.
* It is possible to access the object's properties using dot or brackets: js1.author - js1['author']
  + Objects are arrays whose indexes are property names.
  + Second mode is rare used but necessary to access properties with strange named: js1['date of Publication'] (name with spaces)
* Property names can be declared in three different but equivalent ways.
* As in Json a property can be a complex object.
* Literal objects can have methods.
* To access an object property or call another method from an object method, we must use the this keyword. This.bark();
* It’s possible to create an empty object and add things after var lin ={}; lin.color ='red';
* You can delete an object property:
  + delete lin.color;
  + If accessed after the deletion it will return undefined
* Constructor function is the JS old way to create multiple object. It’s a function that happens to be called with the new operator.
* Syntax is not the same as singleton, no more ":" and "," between properties; we use "=" and ";" like in regular functions.
  + To build new objects using the new keyword:
  + let ian = new Hero('Ian Solo', 'Rebels')
  + By convention constructor functions start with capital letter
* JS6 introduces the class keyword. It’s now possible to create classes in a more standard way.
  + Class name must start with capitalized letter.
  + There can be only one constructor in a class.
  + When defining functions the keyword function is no more needed.
* Unlike functions, classes must be declared BEFORE using them, this because function declarations are "hoisted" and class declarations are not. Hoisting is JavaScript's default behaviour of moving declarations to the top.
* Class instances are created using new command:
  + let luke = new Hero('Luke', 'Rebels')
* A class can have static properties but they must be defined outside the class declaration. Class’s methods can use static properties without any problem.
  + We can add one to the Hero class:
  + Hero.numberHeroesCreated = 0;
  + We can change the constructor to use the static properties

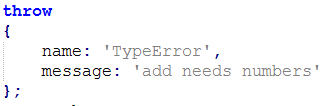
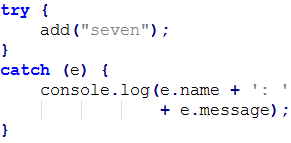
**05 Working with forms**

* Primitive types are passed by copy, objects by references.
  + Comparing two objects will only return true if they point to the same object
* JS environment defines a “global object”. When this environment is a Web server this global object is named window. The global variables (created outside functions and classes) defined with the keyword var are properties of this window object. Let instead does not create a property on the global window object.
  + console.log(window)
* JavaScript and Form. Using JS is the only way to make complex server side data check on a form before sending it to the back end. With the onsubmit property we can call a js function to make the validation. The form will be really submitted only if the function returns true.
  + It’s possible to make the call to back end directly inside the function and return false to avoid the standard operation.
  + Pay attention to use the return command when implementing onsubmit.
* It’s possible to create a promise to wrap every type of code using the method new Promise(…) In this way it’s possible to use .then handlers even with XMLHttpRequest.
  + The constructor takes a function with two parameters (resolve, reject), these are functions used to fulfil or reject the promise, calling automatically the appropriate handling.
  + It's up to the developer to manually call resolve or reject based on the result of their given task.
* Sometimes you do not always need to complete an async tasks within the promise. If it's possible that an async action is taken it’s best to return a promise so that you can always count on a promise coming out of a given function
  + If the async operation isn’t necessary we can keep the coherence of the returned value by calling Promise.resolve(…) to return a promise in every case.
  + This promise has already completed so it’s successful handler is immediately called.
* It’s possible to manage relation between multiple promises using Promise.all([p1,p2,p3]) and Promise.race([p1,p2,p3]) .
  + Promise.all(…) .then will fired when all tree fetch requests complete with success; result will be an array with all the responses. If any promise is rejected the catch fires immediately.
  + Promise.race(…) takes the result of the first fetch to complete.
* Promise are powerful but in presence of long chains of asynchronous operations .then and .catch create a not so readable code. async and await keywords are used for simplifying promise handling. To use await inside a method you need to declare it async. When the awaited promise is hit the asynchronous routine (fetch in this case) runs and execution of further code halts (though not blocking) until the async action finishes. Essentially you get to keep your code "inline" without the need for callback.
  + Variable content is of type response and not promise, await make the necessary conversion.
  + For error handling use the classic try/catch construct.
  + You can use async-await with every type of method: anonymous, lambda, method’s arguments, class methods
* Lambda are an improvement over anonymous functions. Before arrow functions, every new function defined its own this value (a new object in the case of a constructor, the base object if the function is called as an "object method", etc.), this proved to be less than ideal with an object-oriented style of programming. Lambda doesn’t have it's own this value, it’s lexically bound to the enclosing scope.
  + There is no arguments variable with lambda function.
  + lambda with no parameters should be written with a pair of empty parentheses: var x = () => {a+b}
* The real meaning of this. In most cases, the value of this is determined by how a function is called.
  + In the global execution context (outside of any function), this refers to the global object: this.b=77
  + Inside a function, the value of this depends on how the function is called. Whene the value of this is not set by the call, this will default to the global object: console.log(f1(10));
  + We can use call or apply to bound an object to the this value of a function call. If the value passed as this is not an object, an attempt will be made to convert it to an object using the internal ToObject operation.
  + In arrow functions, this retains the value of the enclosing lexical context's this.
  + When a function is called as a method of an object, its this is set to the object the method is called on.
  + Inside a constructor this is bound to the new object being created.
  + As a DOM event handler: when a function is used as an event handler, its this is set to the element the event fired from.
* Js specification define two API to persist key-value data in Web clients, the main difference is data longevity. Using localStorage the data will remain until deleted, while using sessionStorage the data are erased when the tab/browser is closed.
  + Both key and value are string

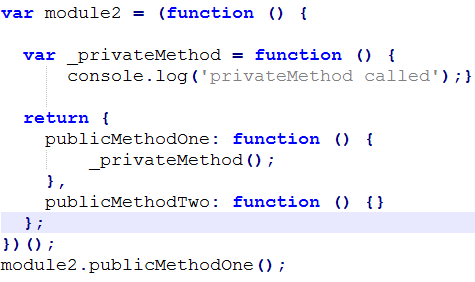
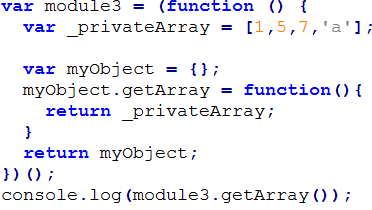
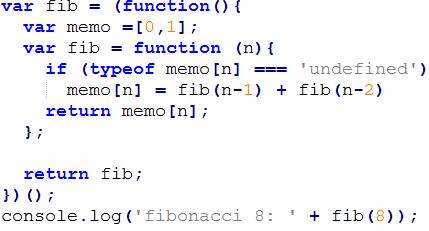
**06 Prototype Chain**

* Each object has a private property which holds a link to another object called its prototype. That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. Null has no prototype.
* JavaScript objects are dynamic "bags" of properties, when trying to access a property of an object, the property will not only be sought on the object but on the prototype of the object, the prototype of the prototype, and so on until either a property with a matching name is found or the end of the prototype chain is reached.
* If we create a singleton object it has Object.prototype as its Prototype, Object.prototype has null as its prototype. Prototype chain will be : x 🡪 Object.prototype 🡪 null
* We can change the prototype of an object y using Object.setPrototypeOf(y, x)
  + set object prototype of y to x
* To retrieve the properties of an object we can use Object.getPrototypeOf(y).
  + y.\_\_proto\_\_ is equivalent.
  + As seen in picture we can add properties to our prototype but this will create a code difficult to read and understand.
* For array and function prototype chains are a little different: array 🡪 Array.prototype 🡪 Object.prototype 🡪 null || function 🡪 Function.prototype 🡪 Object.prototype 🡪 null.
* When using constructor function we can define a prototype to use. Each constructor has an associated prototype object (cf.prototype), when a new object is created, it’s parent is set to the prototype object (always the same object) associated with the constructor that created it.
  + It’s possible to add new properties to our prototype: cf.prototype.b = 3;
* When using class keyword the syntax to define a prototype is a little different. We use extends and super. Extends sets up the prototype inheritance relationship. Super calls the constructor of the base class; it musts always be the first line of code in the constructor.
  + this's not defined until 'super' is called
  + As with constructor function the prototype object is always the same independently by the number of instances created. You can delete a property or a method of a prototype and it will destroy it for every instance you created.
* To know if an a property is of our current object or of one of its prototypes we can use method s.hasOwnProperty(name). This will return true only if we are not using the prototype chain.
  + It’s interesting to note that this works a little different with class keyword. Using an instance of Square with the previous code will show that colour, height and width are all properties of Square 😐.

**07 Exception**

* JavaScript provides an exception mechanism to manage errors. When a mishap is detected the program should throw an exception:
  + throw should receive an object containing a name property and a message property containing the error description.
  + The object can have other properties but this two must always be present.
* To manage an exception it is necessary to use a try / catch block, if an exception is throw within a try block control will go to the catch clause.
  + A try have only a single catch statement, if it’s necessary to manage different type of exception you need to inspect the type (name) of the exception.

**08 Private Methods**

* JavaScript doesn’t have privacy, but is possible to implement it using Immediately-Invoked-Function-Expressions, this are inline functions created and immediately called to return an object. The object’s scope creates “privacy” by returning only the parts we need, leaving the others out of the global scope. In the example the method ‘\_privateMethod’ is not directly accessible but can be used calling publicMethodOne.
  + \_privateMethod is ‘private’ because is inside the scope of the returning object but not in the general scope.
* Using the same trick is possible to have private object (in this case an array)
*  In simple cases we only need to hide a single implementation detail and return a complete object may not be necessary; it’s possible to use the same principle to return a function. The array memo (that stores the previously calculated value for optimization purpose) is not visible from outside to avoid unrequested manipulation.