UNIT 1

JAVASCRIPT

Part 3 - Annex
Object Oriented Programming. AJAX. Promises.

Client-side Web Development
2nd course – DAW
IES San Vicente 2020/2021
Author: Arturo Bernal Mayordomo

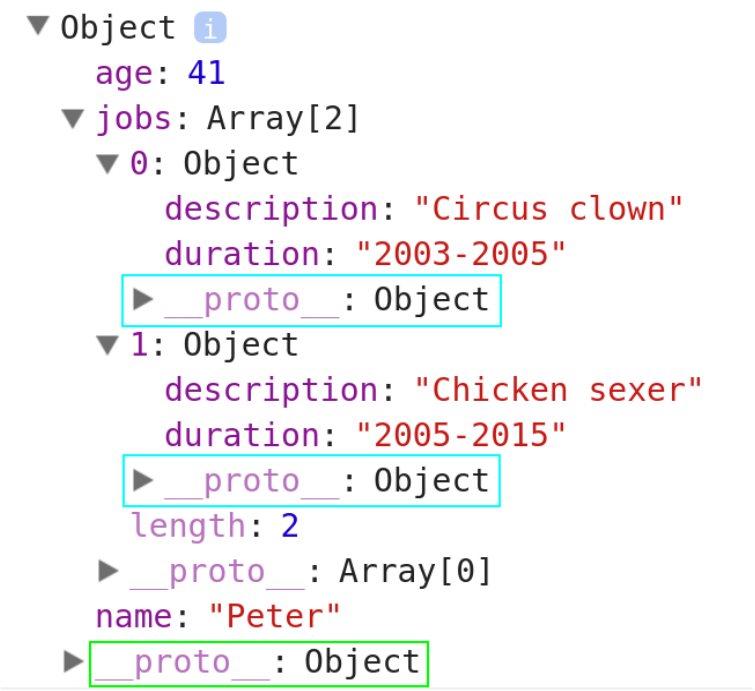
Index

Objects and classes in JavaScript (ES5)

Prototype

The **prototype** is a property (and also an object) that is present in all JavaScript objects. It's not usually accessible directly except from the **constructor function** (we'll learn about them soon). However, most browsers let us access the prototype through a property called __proto__ (This is not recommended, at least not in a production environment).

```
console.log(person);
```



Prototype can be understood as some special form of inheritance. When looking for a property, or a method, JavaScript will try to find it in the object. If it's not there, it will look for it inside the object prototype. If still not there, it will search in the object prototype's prototype, and so on… (until undefined is found).

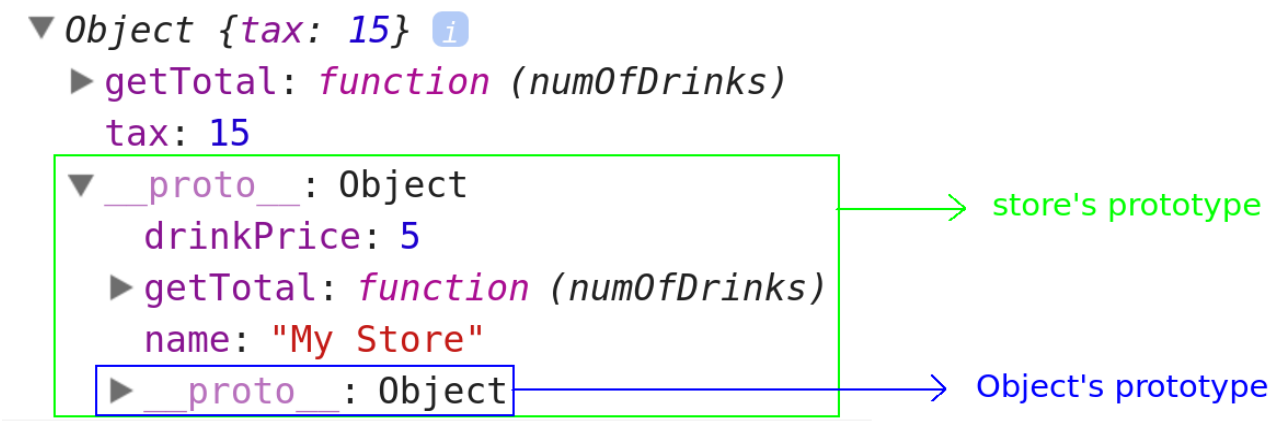
```
console.log(typeof person.prototype); // Prints undefined (we don't have access though this property)
console.log(typeof person.__proto__); // Prints object (not recommended to modify this property)
console.log(typeof person.__proto__.toString); // Prints function. toString() is defined by default in the prototype
```

Inheritance through prototype

If we create an object using JSON, the **Object** class prototype will be assigned to it. It's like saying that this generic object will inherit all Object's properties and methods through its prototype. It's similar to what happens in Java, that all classes (and their objects) inherit from the class Object.

We can use **Object.create(existingObject)** to create an **empty** object with the prototype set to the object we pass as an argument (inherits properties and methods).

```
let store = {
   name: "My Store",
   drinkPrice: 5,
   getTotal: function(numOfDrinks) {
      return numOfDrinks * this.drinkPrice;
   }
}

let liquorStore = Object.create(store); // store will be the prototype of liquorStore
// When we access a property or method that doesn't exist in the object, looks in its prototype
console.log(liquorStore.getTotal(5)); // Prints 25. Calls liquorStore.prototype.getTotal
liquorStore.tax = 15; // Drinks should be 15% more expensive
liquorStore.getTotal = function(numOfDrinks) { // We "override" getTotal method to apply tax
   return numOfDrinks * (this.drinkPrice * (1 + this.tax / 100));
}

console.log(liquorStore.getTotal(5)); // Now prints 28.75 (25 + 15%)
console.log(liquorStore.__proto__.getTotal(5)); // Prints 25. We can still access store's original method

console.log(liquorStore.__proto__.__proto__ === Object.prototype); // Prints true. We can still access the
prototype inherited from object (from store's prototype)
```



Advanced property definition

A lot of people don't know that object's properties in JavaScript are configurable (read only, getters, setters,…). By default, a property is writable, can be listed in a foreach loop, and can be reconfigured.

```
'use strict';
let warrior = {
   vitality: 100
}

let vitProps = Object.getOwnPropertyDescriptor(warrior, 'vitality'); // Arguments → object, 'property'
console.log(vitProps); // Prints Object {value: 100, writable: true, enumerable: true, configurable: true}
```

If we want to create a new property (or method) we can just assign to it a new value by writing **warrior.strength = 10** for example. Here's another way to do the same (but with some differences):

```
Object.defineProperty(warrior, 'strength', {
   value: 10
});
```

```
let strengthProps = Object.getOwnPropertyDescriptor(warrior, 'strength');
console.log(strengthProps); // Prints Object {value: 10, writable: false, enumerable: false, configurable: false}
```

As you can see, if we define a new property using **Object.defineProperty**, by default is not writable, can't be listed in a foreach loop and is not reconfigurable.

```
warrior.strength = 20; → Uncaught TypeError: Cannot assign to read only property 'strength' of object
'#<Object>' (writable: false)

for(let prop in warrior) { // Will print only -> vitality: 100 (strength is not iterable or enumerable)
   console.log(prop + ": " + warrior[prop]);
}

Object.defineProperty(warrior, 'strength', {
   value: 20,
   writable: true
}); → Uncaught TypeError: Cannot redefine property: strength (configurable: false)
```

Getters and setters

We can also define getter and setter functions for any property. When we read or write a property, these functions (if they exist) will be called automatically.

If we create a setter or getter (or both) on a property name, we must use a different name inside the object (putting '_' before the name for example). If we use the same exact name, when we get the original value inside the getter function, that getter function will be called recursively (infinite recursive calls). The same happens with setters.

```
'use strict';
let warrior = {
   _name: 'Peter', // We use underscore '_' to avoid calling getter/setter recursively
   _vitality: 100,
   bonus: 20
}

Object.defineProperty(warrior, 'vitality', {
   get: function() {
      // If we used 'vitality' instead of '_vitality', we'd have infinite recursive calls to this getter
      return this._vitality + this.bonus;
   }
});

console.log(warrior.vitality); // Prints 120 (_vitality + bonus). Calls vitality getter function

Object.defineProperty(warrior, 'name', {
   get: function() {
      return this._name;
   },
   set: function(newName) {
      // If a null or empty name is provided, we assign a default
      this._name = newName || "Super Warrior"
   }
});

warrior.name = ''; // Empty name -> assigns default in setter
console.log(warrior.name); // Prints "Super Warrior" (default value in setter)
```

Constructor functions

The most similar concept to a class in JavaScript (ES5) are **constructor functions**. Any function can be a constructor function, but when we really need a function to become a constructor (like a class) name should start with an **uppercase** letter, and the properties of objects instantiated from that function must begin with the prefix **this**.

This is a comparison between creating an object directly with JSON and using a constructor function to do the same thing:

```
'use strict';
let warrior = {
   vitality: 100
}

function Warrior(vitality) {
   this.vitality = vitality; // Property of every object
}

let warrior2 = new Warrior(120);

console.log(warrior); // Prints Object {vitality: 100}
console.log(warrior2); // Prints Warrior {vitality: 120}
console.log(typeof warrior); // prints object
console.log(typeof warrior2); // prints object
console.log(typeof Warrior); // Prints function
```

Like with JSON, we also can assign a function to a property (method). As said before, in constructor functions we must use '**this**' prefix.

```
let warrior = {
   vitality: 100,
   receiveDamage: function (damage) {
      this.vitality -= damage;
   }
}

function Warrior(vitality) {
   this.vitality = vitality; // Property of every object
   this.receiveDamage = function (damage) {
      this.vitality -= damage;
   }
}

let warrior2 = new Warrior(120);

warrior.receiveDamage(20);
warrior2.receiveDamage(20);
console.log(warrior.vitality); // Prints 80 (100-20)
console.log(warrior2.vitality); // Prints 100 (120-20)
```

Accessing prototype from constructor functions

As always, objects created with constructor functions have a prototype assigned. As we saw earlier, in JSON the prototype is inherited from Object. But if we instantiate an object using a constructor function, a new prototype is created an assigned to every object created from it (they all share the same prototype object). This new prototype has Object's prototype inside it, so we can still access methods inherited

from Object.

```
let warrior2 = new Warrior(120);

console.log(warrior.__proto__ == Object.prototype); // Prints true, uses Object's prototype
console.log(warrior2.__proto__ == Object.prototype); // Prints false. Has its own prototype
console.log(warrior2.__proto__.__proto__ == Object.prototype); // Prints true, Warrior's prototype inherits Object's prototype

let warrior3 = new Warrior(90);

console.log(warrior2.__proto__ == warrior3.__proto__); // Prints true. They share the same prototype object
```

Using a constructor function name, we have access to the prototype directly (no need to use __proto__), so we can establish common properties and methods shared by all instantiated objects.

Let's see why it's usually preferable to create methods using the **prototype** instead of using a property inside the constructor function:

```
'use strict';
function Warrior(vitality) {
   this.vitality = vitality; // Property of every object
   this.receiveDamage = function (damage) {
      this.vitality -= damage;
   };
}

let w1 = new Warrior(100);
let w2 = new Warrior(100);
console.log(w1 == w2); // Prints false. Same values but not the same object (reference)
console.log(w1.receiveDamage == w2.receiveDamage); // Prints false. Same code, but different functions!!
```

Every time we create a new object, memory is reserved to store all its properties and a reference to that memory zone where the object resides is returned. If we define a method using a property, that method will be created in memory with the object, so every object will have its own methods (1000 different objects → 1000 different methods in memory with the same code).

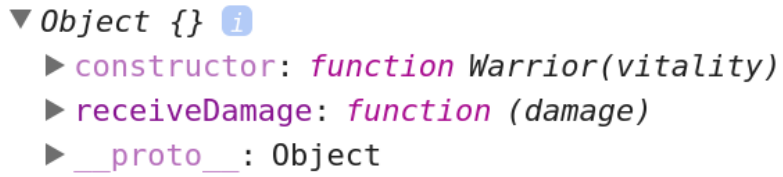
As we mentioned earlier, when we access a property or method, JavaScript searches in the object, and if JavaScript doesn't find it there, it will start searching in the prototype (and then in the prototype's prototype, …). So, if all objects created from a constructor class **share** the same prototype object, we should define shared methods there. It's similar to create objects from a JSON definition using Object.create as we saw earlier.

```
'use strict';
function Warrior(vitality) {
   this.vitality = vitality; // Property of every object
}

Warrior.prototype.receiveDamage = function (damage) {
   this.vitality -= damage;
};

let w1 = new Warrior(100);
let w2 = new Warrior(100);
console.log(w1.receiveDamage == w2.receiveDamage); // Prints true. Same reference in memory!
```

```
w1.receiveDamage(20);
w2.receiveDamage(30);
console.log(w1.vitality); // Prints 80 (100-20)
console.log(w2.vitality); // Prints 70 (100-30)
```

We don't have to worry about accessing each object properties if a method is shared between 2 or more objects, because when we use '**this**' it's referring to the object that called the method.

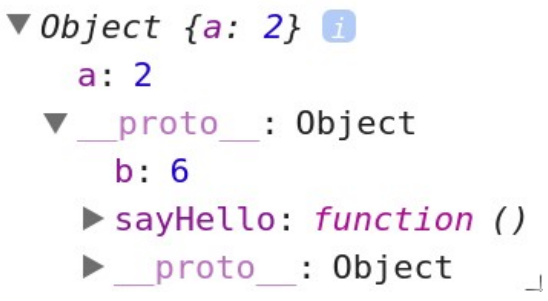
```
console.log(Warrior.prototype);
```



Object new methods

Object.setPrototypeOf(objA, objB) → objB now becomes the prototype of objA (objA inherits all the properties and methods from objB).

```
let objA = {
   a: 2
}

let objB = {
   b: 6,
   sayHello() {
      console.log("Hello from B");
   }
}

Object.setPrototypeOf(objA, objB);
console.log(objA); // objA's prototype is now objB
objA.sayHello(); // Prints "Hello from B"
```



Object.assign(target, obj1, obj2) → Copies the properties from obj1 and obj2 into target. If a property name is in both obj1 and obj2, obj2's property overwrites it.

```
let obj1 = {
   a: 2,
   b: 1
}

let obj2 = {
   b: 6,
   c: 4
}

let obj3 = {};
Object.assign(obj3, obj1, obj2);
console.log(obj3); // Object {a: 2, b: 6, c: 4}
```

This method is useful when we receive an object in a function parameter and if some property we need inside that object is not defined, create a "default" object with default properties and take the properties we don't have from there.

```
function getConfiguration(timeout, options = {}) {
      let defaults = {
         container: '#products',
         classItem: 'product',
         title: 'Example',
         description: 'Example description'
      };
      // Properties defined in options will override defaults
      let settings = Object.assign({}, defaults, options);
      ...
   }
```

Object.keys(object) → Returns a string array with all the properties names.

Object.values(object) → Returns an array with all the properties values.

Object.entries(object) → Returns an array that contains other arrays of key-value pairs (property → value). We can create a Map object from this array.

```
let foods = {
   baguette: '    ',
   egg: '    ',
   kiwi: '    ',
   sandwich: '    ',
   salad: '    '
}

console.log(Object.keys(foods)); // ["baguette", "egg", "kiwi", "sandwich", "salad"]
console.log(Object.values(foods)); // ["     ", "     ", "     ", "     ", "     "]
console.log(Object.entries(foods)); // [["baguette", "     "], ["egg", "     "], ["kiwi", "     "], ["sandwich", "     "],
["salad", "     "]]
```

Functions: advanced features

Using call and apply

We now how to call functions in the usual way (nameOfFunction(parameters…)). We also know that the keyword **this** is used in a constructor function or in a method to reference the actual object. But, what happens if we use the this keyword inside a normal function?.

```
'use strict';
function foo () {
   console.log(this); // strict mode -> undefined. normal -> window object
}

foo();
```

Well, as you can see the behavior is different between strict and normal mode. That's why using **this** inside a normal function is not very common. However, we can pass that function an object (whichever we want) that will be referenced by **this**. To do that, we must call the function in a special way (using **call** method).

```
'use strict';
function foo () {
   console.log(this.num);
}

foo.call({num: 13}); // Prints 13
foo.call({num: 42}); // Prints 42
```

What happens if the function receives arguments?. We just have to include them after the object we send to be referenced by **this**.

```
'use strict';
function foo(name, age) {
   console.log(this.name + " is " + (this.age >= age?'older':'younger') +
   " than " + name);
}

let person = {
   name: "Peter",
   age: 42
}
foo.call(person, "Louise", 31); // Prints "Peter is older than Louise"
```

We can do the same with the method **apply**. The difference is that we must include the function arguments in an array, but the result is the same.

```
foo.apply(person, ["Louise", 31]); // Prints "Peter is older than Louise"
```

IIFEs

An IIFE or Immediately Invoked Function Expression, is a self executing function

which context exist separately from the global context. A simple IIFE would be this:

```
'use strict';
(function() {
   console.log("I'm automatically called");
})(); // Prints "I'm automatically called"
```

The most important matter is that inside the function, everything is isolated from the global scope. However we can pass global objects to this IIFE inside the last parenthesis separated by comma. This is used to create global objects that represent a library like Jquery, so if they need to use external functions or variables, don't mess with the global scope of the application that uses the library.

```
'use strict';
(function() {
   let num = 14;
})();

console.log(num); → Uncaught ReferenceError: num is not defined
```

Another example passing a global object to the IIFE:

```
'use strict';
let Library = {}; // Object that contains the library methods and properties

(function(library) {
   let localName = "Peter"

   library.sayHello = function() { // Creating a new method inside the IIFE
      console.log("Hello " + localName); // From here we can access localName
   }
})(Library); // We pass the global object to the IIFE function

Library.sayHello(); // Prints "Hello Peter"
```

Iterators

Iterators are a new way to loop over arrays and other collections. For example, from an array we can access a special method using square bracket notation called **[Symbol.iterator]**. We haven't studied symbols in JavaScript as they are a very advanced feature, but lets think about them as properties that never return the same value, so what we really get is a new different iterator every time we call the corresponding method.

```
let array = [12, 15 , 36];
let iterator = array[Symbol.iterator](); // We get an iterator function
console.log(iterator.next()); // Prints {value: 12, done: false}
```

As you can see, iterators don't return just the value when calling **next()**. Instead, it returns a JSON object with the properties value (current value) and done (if the iterator has finished iterating). It won't return {**done:true**} until we call next() after getting the last value.

```
console.log(iterator.next()); // Prints {value: 12, done: false}
console.log(iterator.next()); // Prints {value: 15, done: false}
console.log(iterator.next()); // Prints {value: 36, done: false}
console.log(iterator.next()); // Prints {value: undefined, done: true}
```

This is how we can use an iterator with a loop to iterate through all values from an array (or any collection that implements it):

```
let current = iterator.next();
while(!current.done) {
    console.log(current.value);
    current = iterator.next();
}

// Another example
let current;
while(!(current = iterator.next()).done) {
   console.log(current.value);
}
```

We can also create our own iterator, for example to iterate through the properties of an object.

```
let iterableObject = {
  name: "Peter",
  bloodType: "B-",
  age: 50,
  employed: true,
  [Symbol.iterator]: function() {
     let properties = Object.keys(this); // Aray with the names of the properties
     let count = 0; // Current property index
     let isDone = false; // When we have finished, we'll return done: true

     let next = () => { // Next function. This is what we return
        if(count === properties.length) isDone = true;
        return { done: isDone, value: this[properties[count++]] };
     }
```

```
        return { next }; // An object with only the next method in it
    }
}

let iterator = iterableObject[Symbol.iterator]();
let current = iterator.next();
while(!current.done) {
    console.log(current.value);
    current = iterator.next();
}
```

The **for..of** loop internally uses an iterator. It didn't work with objects before, but if we implement an iterator on an object, it will work.

```
// propValue gets directly the current.value (it's not an object)
for(let propValue of iterableObject) {
    console.log(propValue); // Prints "Peter" - "B-", 50 and true
}
```

Generators

A generator is a special kind of function. It's a concept similar to an iterator function because you can call it multiple times and it will return a different value each time until it finishes. In fact, you can think of it that is a more simple way to declare an iterator (it can be used for more purposes than just iterate through an object or a collection).

You declare a generator the same way you declare a normal function but putting an asterisk '*' just before the name of the function. Let's see an example:

```
function *names() {
   yield "Mary";
   yield "John";
}
```

```
let nameGen = names(); // It works like an iterator
console.log(nameGen.next()); // Prints {value: "Mary", done: false}
console.log(nameGen.next()); // Prints {value: "John", done: false}
console.log(nameGen.next()); // Prints {value: undefined, done: true}
```

The **yield** instruction is like **return**, but with one big difference. Each time we call the next() method, the function is going to return the next yield instruction after the one it returned the last time it was called.

As you can see it's the same concept as an iterator, and we can do the same with it, like iterate over its values with a **for..of** loop, use **spread** and **destructure** (like with arrays).

```
for(name of names()) {
   console.log(name);
}

let nameArray = [...names()];
console.log(nameArray.join(",")); // Prints "Mary,John"
```

```
let [n1,n2] = names();
console.log(n1); // Prints "Mary"
console.log(n2); // Prints "John"
```

Now, lets see how we do the same as with iterators to iterate through an object's properties, but this time using a generator function instead:

```
let iterableObject = {
   name: "Peter",  bloodType: "B-",   age: 50,  employed: true,
   [Symbol.iterator]: function *() { // Note the asterisk here!
      let properties = Object.keys(this); // Array with the names of the properties
      for(let property of properties) {
         yield this[property]; // On each call to next() we return the next property
      }
   }
}

for(let propValue of iterableObject) {
   console.log(propValue); // Prints "Peter" - "B-", 50 and true
}
```

AJAX with XMLHttpRequest

Although usually a library like JQuery or a framework like Angular is used to make **AJAX** calls to the server because they offer much more functionality and make the job simpler, it's not a bad idea to learn some AJAX basics now. It will make easier to understand in the near future what those libraries/frameworks offer to us.

An AJAX call is simply a request to the web server made after the page has been loaded and doesn't imply reloading the page or loading a new one. It's also a good option for saving bandwidth and resources because the server sends only the data that's needed (for example, to update some information about a product in the page) and not a full web page.

AJAX stands for **Asynchronous JavaScript And XML** (remember that HTML derives from XML), but nowadays JSON is becoming the format of choice to send/receive data from the server (JSON integrates natively with JavaScript). Being **asynchronous** means that if we make a request, we're not going to receive an immediate response. Instead, we must pass a function that will be called when the information is received (or an error occurs). Until then, the web page code will not be blocked and will continue to respond normally to events, execute pending code, etc.

HTTP basic methods

Server requests and responses will be sent/received using the HTTP protocol. When making an HTTP request, the browser will send to the server: **headers** (like the useragent which identifies the browser, preferred language, etc.), the **HTTP request method**, and the **data** (if needed).

There are many request methods we can use to make an HTTP call to the server, but the most used ones when making an AJAX call (or accessing web services) are:

- **GET** → Retrieves data and usually won't modify anything. It's equivalent to SELECT in SQL. When using this method, the data will usually be sent in the URL (URIEncoded format).
- **POST** → Operation intended to insert new data and store it in the server. It's usually equivalent to INSERT in SQL. Data is sent inside the HTTP request's content field. Login requests are made with POST instead of GET so the data is not visible in the URL.
- **PUT** → Operation that updates existing data in the server. Similar to UPDATE in SQL. Data which identifies the object (id for example) to update is sent in the URL, and the data to modify is sent in the content field like with POST.
- **DELETE** → Operation that deletes existing data in the server (like DELETE in SQL). Data which identifies what to delete is sent in the URL.

XMLHttpRequest

To create a request for the server, we must instantiate a **XMLHttpRequest** object. After that, we call its **open** method. This method receives 3 parameters: **HTTP method** ("GET", "POST", "PUT", "DELETE"), the **url** of the resource in the server, and a third boolean parameter that indicates if the call is **asynchronous** (recommended), or it will block the browser until a response it's received.

To add headers to the HTTP request, we can use **setRequestHeader** method after calling **open**. After that, we'll just call **send** method and the request data will be sent to the server (empty if it's a GET or DELETE request, or a string containing the data if it's a POST or PUT request).

```
let http = new XMLHttpRequest();
http.open('POST', '/mypage.php', true);
// urlencoded means var1=value1&var2=value2... format (using EncodeURIComponent with values)
http.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
http.send("data1=" + encodeURIComponent(data1) + "&data2=" + encodeURIComponent(data2));
```

XMLHttpResponse

Sending a request to the server goes through many different states until a response is received. When the request process goes into a new state, a **readystatechange** event is fired by the XMLHttpRequest object. We must attach a handler function to that event if we're going to do something when a response is received.

Inside this function we should check 2 properties of the XMLHttpRequest object. The first one is **readyState**, which indicates the connection status with an integer (0: request not initialized (1: server connection established, 2: request received, 3: processing request, 4: request finished and response is ready), and also the **status** property, which (when the response has been received) indicates if everything went ok or there was an error using HTTP response codes (200 → OK, 404 → Resource not found, 500 → Server internal error). When readyState equals 4 and status is equal to 200, we have received a successful response.

```
http.addEventListener('readystatechange', function() {
   // readyState = 4 -> response received. status = 200 -> OK (no error)
   if(http.readyState === 4 && http.status === 200) {
      document.getElementById("id1").innerHTML = http.responseText;
   }
});
```

Progress Events

There are events which are simpler to handle than **readystatechange**. Those events are called **progress events**, and some of them are:

- **progress**: Can be fired multiple times during request. It indicates that the client is uploading data to the server. The event object has some useful properties that allow us to show the percentage transferred to the user.
    - **lengthComputable** (boolean) → If the total transfer size is known

    - **loaded (number)** → Bytes already transferred to the server
    - **total (number)** → Total bytes to be transferred (if unknown is zero)

- **load**: Fires once when a response from the server has been received (same as readyState == 4). Note that this response can be a valid response (200) or an HTTP error (check status property).
- **error**: Fires if an error sending data ocurred (network failure for example).
- **abort**: Fires if the request has been aborted. This can be done using the **abort()** method with the XMLHttpRequest object (after is sent).
- **timeout**: Fires if the request isn't completed in an amount of time (sending data and receiving response). You can establish this time setting the **timeout** property with the XMLHttpRequest object (before sending).
- **loadend**: Fires when the HTTP request has ended, doesn't matter if correctly, with an error or aborted. This event is the last one to occur.

Example

```
let http = new XMLHttpRequest();
http.open('POST', 'myserver.com/products', true);
http.send(JSON.stringify(product)); // Sending data in JSON format

http.addEventListener('load', (event) => {
   if(http.status === 200) {
      let response = JSON.parse(http.responseText); // Data received in JSON format
      // Do something with the response
   } else {
      showError();
   }
});

http.addEventListener("error", showError);
http.addEventListener("abort", showError);
http.addEventListener("progress", showProgress);

function showError(event) {
   alert("An error ocurred or the transfer was aborted");
}

function showProgress(event) {
  if (evt.lengthComputable) {
    let percentComplete = evt.loaded / evt.total;
    // Show percentage somewhere
  } else {
    // Unable to compute progress information since the total size is unknown
  }
}
```

Submit files and forms using FormData

If we need to send a form by AJAX which has one or more file inputs, we can use the FormData class to get all the data and send it. In this case, we won't send a JSON

object, but **urlencoded** information instead.

```
<form id="addProduct">
   <p><input type="text" name="name" id="name" placeholder="Product's name" required></p>
   <p><input type="text" name="description" id="description" placeholder="Description" required></p>
   <p>Photo: <input type="file" name="photo" id="photo" required></p>
   <button type="submit">Add</button>
</form>
```

We can create a FormData object and add the value pairs manually:

```
let formData = new FormData();
formData.append("name", document.getElementById("name").value);
formData.append("description", document.getElementById("description").value);
formData.append("photo", document.getElementById("photo").files[0]);

let http = new XMLHttpRequest();
http.open('POST', SERVER + '/product');
http.send(formData);
```

Or we can create directly a FormaData object containing everything from a form (It's the same as before, and we could add more data if we want to):

```
let http = new XMLHttpRequest();
http.open('POST', SERVER + '/product');
http.send(new FormData(document.getElementById("addProduct")));
```

Submit files in JSON using Base64 encoding

If our server needs the information in JSON format, files have to be converted into strings. A format like Base64 is the most used for this purpose.

```
window.addEventListener("load", (e) => {
   ...

   // When a file (image) is selected we process it
   document.getElementById("photo").addEventListener('change',  () => {
      let file   = document.getElementById("photo").files[0];
      let reader = new FileReader();

      reader.addEventListener("load", () => { //Converted into Base64 event (async)
         imagePreview.src = reader.result;
      }, false);

      if (file) { // File has been selected (convert to Base64)
         reader.readAsDataURL(file);
      }
   });
});
```

Then we can send it inside a JSON object (and decode the Base64 fields in the server and save them to files):

```
let prod = {
    name: document.getElementById("name").value,
    description: document.getElementById("description").value,
    photo: imagePreview.src
};
```

```
let http = new XMLHttpRequest();
http.open('POST', SERVER + '/product/json');
http.send(JSON.stringify(prod));
```

Examples

1. In this example of a **GET** method request, we'll receive HTML code containing information about products in JSON format, so we have to create the HTML elements, write that data and append them in the DOM.

```
function getProducts() {
    let http = new XMLHttpRequest();
    http.open('GET', `${SERVER}/product`);
    http.send();

    http.addEventListener('load', (event) => {
        if(http.status === 200) {
            let response = JSON.parse(http.responseText); // Data received in JSON format
            response.products.forEach((p) => appendProduct(p));
        } else {
            showError();
        }
    });
}

function appendProduct(product) {
    let tbody = document.querySelector("tbody");
    let tr = document.createElement("tr");
    // Image
    let imgTD = document.createElement("td");
    let img = document.createElement("img");
    img.src = `${SERVER}/img/${product.photo}`;
    imgTD.appendChild(img);

    // Name
    let nameTD = document.createElement("td");
    nameTD.textContent = product.name;

    // Description
    let descTD = document.createElement("td");
    descTD.textContent = product.description;

    tr.appendChild(imgTD);
    tr.appendChild(nameTD);
    tr.appendChild(descTD);
    tbody.appendChild(tr);
}
```

2. This is a **POST** example. Data will be sent in **JSON** format. We'll receive a response with an ok property (true or false), and the inserted product or the error message. If everything goes well, the returned product will be inserted in the DOM.

```
function addProduct() {
    let prod = {
        name: document.getElementById("name").value,
        description: document.getElementById("description").value,
        photo: imagePreview.src
```

```
    };

    let http = new XMLHttpRequest();
    http.open('POST', SERVER + '/product/json');
    http.send(JSON.stringify(prod));

    http.addEventListener("error", (event) => alert("An error ocurred or the transfer was aborted"));

    http.addEventListener('load', (event) => {
        if(http.status === 200) {
            let response = JSON.parse(http.responseText); // Data received in JSON format
            appendProduct(response.product);
        } else {
            alert(`Error uploading product. Status: ${http.status}`);
        }
    });
}
```

Encapsulating AJAX calls in Promises

Let's go back and see how we could handle an AJAX request using event handler function, and how we could manage errors, etc. As you can see, the code could get too big because it has so much responsibility (handle multiple scenarios, errors).

```
function getProduct(id) {
    let http = new XMLHttpRequest();
    http.open('GET', '/products/' + id, true);
    http.send();

    http.addEventListener('load', function() { // Event load is fired when http.readyState === 4
        if(http.status === 200) { // OK
            let prod = JSON.parse(http.responseText);
            if(prod.error) { // The server sends us an error
                // Process error ...
            } else {
                // Process data.....
            }
        } else {
            // There's been an error, process it
        }
    });

    http.addEventListener('error', function() {
        // There's been an error, process it
    });
}

getProduct(12); // The (big) code inside the function will handle everything
```

Now let's create a Promise and use its resolve and reject methods with the same AJAX call inside it. Note that this time, results and error processing are made outside that code making it cleaner.

```
function getProduct(id) {
    return new Promise(function(resolve, reject) {
        let http = new XMLHttpRequest();
        http.open('GET', '/products/' + id, true);
        http.send();
```

```
        http.addEventListener('load', function() { // Event load is fired when http.readyState === 4
            if(http.status === 200) { // OK
                let prod = JSON.parse(http.responseText);
                if(prod.error) {
                    reject(prod.error); // We return the error message rejecting the promise
                } else {
                    resolve(prod); // We "return" the product
                }
            } else {
                reject('Error HTTP: ' + http.status);
            }
        });

        http.addEventListener('error', function() {
            reject('Error in the http request for a product');
        });
    });
}

getProduct(12) // We subscribe to the promise
    .then(function(product) { // Promise is resolved and returns a product
        // Process product data
    })
    .catch(function(error){ // Promise is rejected (error)
        // Process/show error message
    });
```