UNIT 1

JAVASCRIPT

Part 3
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....3
JSON....3
ES2015 class basics....4
Class inheritance....5
Static class members....5
Primitive and string value....6
Destructuring objects....7
Promises....8
Promise.resolve and Promise.reject....10
Promise.all and Promise.race....10
AJAX....11
HTTP basic methods....11
Fetch API....12
Submit files and forms using FormData....12
Submit files in JSON using Base64 encoding....12
Examples....13
Encapsulating AJAX requests in classes....14
async / await....17

Objects and classes in JavaScript

JavaScript is an Object Oriented language, and, until ES2015 creating classes was very different from other OO languages. However this has changed and the new syntax will be familiar to programmers who know Java, C#, etc.

JSON

JSON or **J**ava**S**cript **O**bject **N**otation is an special notation used to create generic objects in JavaScript. It has become a very popular format for data exchange with web services or storage like noSQL databases (MongoDB for example). You can learn everything about JSON notation in json.org.

First, we'll see an example of how to create a generic object in JavaScript, and add some properties and methods (yes, in JavaScript, you can add properties and methods to an object not present in the original class). We create a generic object using the Object class.

As you'll see, we can access properties (or add them) using dot notation (**object.property**) or associative array notation (**object[property]**).

```
let obj = new Object();
obj.name = "Peter"; // Add 'name' property using dot notation
obj["age"] = 41; // Add 'age' property using array notation
obj.getInfo = function() { // Creating a new method
   return "My name is " + this.name + " and I'm " + this.age
}

console.log(obj.getInfo()); // Prints "My name is Peter and I'm 41"
console.log(obj.name); // Prints "Peter". Dot notation
console.log(obj["name"]); // Prints "Peter". Associative array notation
let prop = "name";
console.log(obj[prop]); // Prints "Peter". We can store the name of the property in a variable (only array notation)
```

Now, we'll do the same but using JSON this time. In JSON format, we assign a value to a property using colon ':' instead of equal '='. Properties and methods are declared inside curly brackets (we can add more later). In this case, the equivalent to new Object(), we be the **{}** (empty object) value.

```
let obj = {
   name: "Peter",
   age: 41,
   getInfo () { // This is new syntax in ES2015, in previous versions we should use getInfo: function()
      return "My name is " + this.name + " and I'm " + this.age
   }
}
```

Arrays

In JSON notation, square brackets are used to create arrays. Inside those arrays we can store objects, primitive values, and other arrays.

```
let person = {
```

```
    name: "Peter",
    age: 41,
    jobs: [ // jobs is an array of objects JSON)
        {
            description: "Circus clown",
            duration: "2003-2005"
        },
        {
            description: "Chicken sexer",
            duration: "2005-2015"
        }
    ]
}

console.log(person.jobs[1].description); // Prints "Chicken sexer"
```

ES2015 class basics

Since ES2015, we can declare new classes in a similar way we do it in other languages like Java, C#, PHP, etc (See Annex for information about how to do it in ES5). It's still more limited, for example, there's no public or private scope) everything is public.

```
class Product { }

console.log(typeof Product); // Prints function (See annex for constructor functions)
```

To create a constructor, we must create a method called **constructor()**. It will be automatically called when a new object is instantiated.

```
class Product {
    constructor(title, price) {
        this.title = title;
        this.price = price;
    }
}

let p = new Product("Product", 50);
console.log(p); // Product {title: "Product", price: 50}
```

Creating methods inside a class has the same effect as adding them to the prototype of a constructor class in ES5 (see annex).

```
class Product {
    ...
    getDiscount(discount) {
        let totalDisc = this.price * discount / 100;
        return this.price - totalDisc;
    }
}

let p = new Product("Product", 50);
console.log(p.getDiscount(20)); // Prints 40
console.log(p.getDiscount === Product.prototype.getDiscount); // Prints true
```

Class inheritance

In ES2015, a class can inherit from another class by using the keyword **extends**. This will inherit all properties and methods from the parent class. Of course, we can override them in the child class. We still can call the parent methods (and constructor) using the keyword **super**. In fact, if we create a constructor in the child class, we **must** call the parent's constructor from there.

```
class User {
  constructor(name) {
    this.name = name;
  }

  sayHello() {
    console.log(`Hello from ${this.name}`);
  }

  sayType() {
    console.log("I'm a user");
  }
}

class Admin extends User {
  constructor(name) {
    super(name); // User constructor
  }

  sayType() { // Overriding method
    super.sayType(); // Calling User.sayType
    console.log("But also an admin");
  }
}

let admin = new Admin("Anthony");
admin.sayHello(); // Prints "Hello from Anthony"
admin.sayType(); // Prints "I'm a user" and "But also an admin"
```

Static class members

Since ES2015 we can declare **static** methods in class (but no static properties). These methods are called directly with the class name and don't have access to the **this** property (no instance).

```
class User {
  ...
  static getRoles() {
    return ["user", "guest", "admin"];
  }
}

console.log(User.getRoles()); // ["user", "guest", "admin"]
let user = new User("john"); // We can't call a static method from an object
console.log(user.getRoles()); // Uncaught TypeError: user.getRoles is not a function
```

Primitive and string value

When an object is converted to a string (concatenation for example), the method toString (inherited from Object) is automatically called. By default it will print "[object Object]", but we can override its behavior declaring this method in the class.

```
'use strict';
class Warrior {
   constructor(name, vitality) {
      this.vitality = vitality; // Property of every object
      this.name = name;
   }
}

let w1 = new Warrior("James Warrior", 150);

console.log(w1.toString()); // Prints "[object Object]"
console.log("Warrior (w1): " + w1); // Prints "Warrior(w1): [object Object]" (calls toString())

class Warrior2 {
   constructor(name, vitality) {
      this.vitality = vitality; // Property of every object
      this.name = name;
   }

   toString() {
      return this.name + " (" + this.vitality + ")";
   }
}

let w2 = new Warrior2("Peter Strong", 100);

console.log("Warrior2 (w2): " + w2); // Prints "Warrior(w1): Peter Strong (100)"
```

When comparing objects using relational operators (>, <, =>, =<), the primitive value of an object is obtained and compared. If we override **valueOf()** method (inherited from Object), returning another primitive value, it will be used for this type of comparisons. If ValueOf is not present, it will compare by string value (**toString()**).

```
'use strict';
class Warrior {
   constructor(name, vitality) {
      this.vitality = vitality; // Property of every object
      this.name = name;
   }

   toString() {
      return this.name + " (" + this.vitality + ")";
   }

   valueOf() {
     return this.vitality; // The primitive value will be object's vitality
   }
}

let w1 = new Warrior("James Warrior", 150);
let w2 = new Warrior("Peter Strong", 100);

console.log(w1 > w2); // Prints true!. Now: 150(w1) > 100(w2)
```

Destructuring objects

Sinde ES2015, we can destructure object properties. It's the same process as destructuring an array, but we use curly braces '**{}**' instead of square brackets.

```
let user = {
   id: 3,
   name: "Peter",
   email: "peter@gmail.com"

}

let {id, name, email} = user;
console.log(name); // Prints "Peter"

// Instead of an array, this function will receive an object as first parameter
function printUserData({id, name, email}, otherInfo = "None") {
   ...
}

printUserData(user, "He's not too smart");
```

Also, we can assign different names from the object's properties on the destructured variables → (propertyName: varName):

```
let {id: userId, name: userName, email: userEmail} = user;
console.log(userName); // Prints "Peter"
```

Combining objects

We can use object destructuration to create new objects combining properties from 2 or more objects. When a property is present in 2 or more objects, the last object's value will remain.

```
function configGame(options) {
   let defaults = {
      name: "Player 1",
      level: 1,
      difficulty: "normal",
      genre: "female"
   };

   let config = {...defaults, ...options};
   console.log(config);
}

let options = {
   name: "Killer Master",
   genre: "male"
};
configGame(options); // {name: "Killer Master", level: 1, difficulty: "normal", genre: "male"}
```

Promises

Now, in ES2015, a feature that was only available by using jQuery or the Q Library (AngularJs for example uses it), is natively supported. We are talking about promises, a better way to handle asynchronous request that passing a handler function as a parameter (remember AJAX from the first unit?).

A **Promise** is an object that is created with an anonymous function. This function receives two parameters, **resolve** and **reject**. *resolve* is called when the Promise finishes correctly returning (optional) some data, while *reject* is called when we want to throw an error. A Promise ends whenever one of this two functions are called.

When we get a Promise, we have to subscribe to it by calling the method **then** and passing to it an anonymous function. This function will be executed when that Promise resolves (We'll see what happens if it calls reject soon...).

```
function getPromise() {
   return new Promise(function(resolve, reject) {
      console.log("Promise called...");
      setTimeout(function() {
         console.log("Resolving promise...");
         resolve(); // Promise resolved!. Now it returns!
      }, 3000); // After 3 seconds, we end the promise
   });
}

getPromise().then(function() {
   console.log("The promised has ended!");
});

console.log("The program continues. Doesn't wait for the promise (asynchronous)");
```

Other thing we can do when using then to get Promises results, is to chain more calls to **then**. The first call will be processed when the promise resolves, and the next **then** will receive what the previous returned.

```
function getPromise() {
   return new Promise((resolve, reject) => {
      setTimeout(() => resolve(1), 3000); // After 3 seconds, we end the promise
   });
}

getPromise().then((num) => {
   console.log(num); // Prints 1
   return num + 1;
}).then((num) => {
   console.log(num); // Prints 2
}).catch((error) => {
   // If there's an error... (promise rejected)
});
```

We also can go directly to the **catch** section if we **throw an error** inside a **then** section (any of them). We can also do the same by calling **Promise.reject("error")** that simulates a promise rejection.

```
getProducts().then((products) => {
   if(products.length === 0) {
      throw 'There are no products!'
      // Promise.reject('There are no products!'); does the same
   }
   return products.filter(p => p.stock > 0);
}).then((prodsStock) => {
   // Print them on the page
}).catch((error) => {
   console.error(error);
})
```

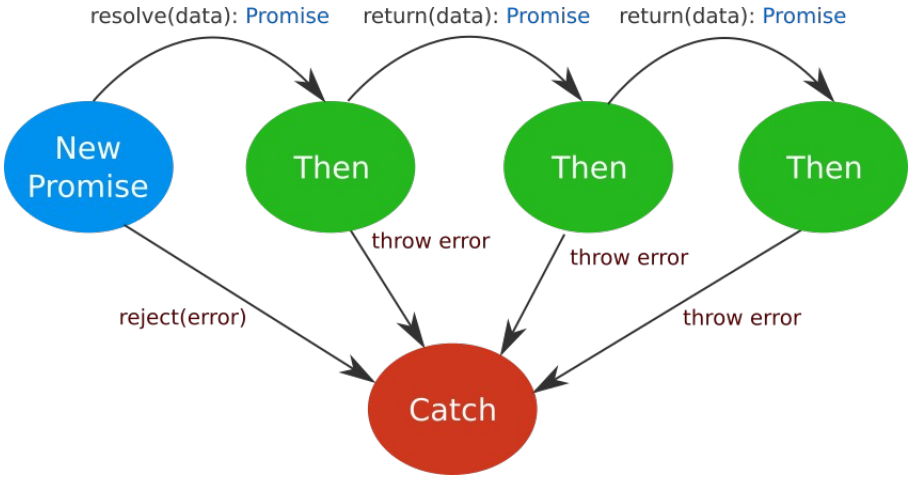
To make our code more readable, we can separate the functions and reference them from the **then** (or catch) section (the parameter will be passed normally).

```
function filterStock(products) {
   if(products.length === 0) {
      throw 'There are no products!'
      // Promise.reject('There are no products!'); does the same
   }
   return products.filter(p => p.stock > 0);
}

function showProducts(prodsStock) {
   // Print them on the page
}

function showError(error) {
   console.error(error);
}

getProducts()
   .then(filterStock)
   .then(showProducts)
   .catch(showError);
```



There's another Promise method, **.finally**, that can be chained at the end and executes no matter if the Promise was resolved or rejected. It can be used, for example, to hide a loading animation.

```
promise.then(...).catch(...).finally(() => /*Hide loading animation*/);
```

Promise.resolve and Promise.reject

When a function returns a Promise (calls a web server), but sometimes we can return a value immediately (or an error), we still must return a Promise (a function or method should return the same data type always). If we don't need to make an asynchronous call, or wait for anything, we can return a Promise with a value using **Promise.resolve**, or return a Promise with an error using **Promise.reject**.

```
function getInstantPromise() {
   // Equivalent to: return new Promise((resolve, reject) => resolve(25));
   return Promise.resolve(25);
}

getInstantPromise.then(val => console.log(val)); // Prints 25

function getRejectedPromise() {
   // Equivalent to return new Promise((resolve, reject) => reject('Error'));
   return Promise.reject('Error');
}

getRejectedPromise.catch(error => console.log(error)); // Prints "Error"
```

Promise.all, Promise.race, Promise.allSettled

Sometimes we need to create several promises (for example, several AJAX calls), and wait for every one of them to finish in order to do something. We can group all created Promises in a collection (Array for example) and use the **Promise.all** method to wait for them.

```
function make3AjaxCalls() {
   let p1 = Http.get(...); // returns a Promise
   let p2 = Http.get(...); // returns a Promise
   let p3 = Http.get(...); // returns a Promise

   return Promise.all([p1, p2, p3]);
}

make3AjaxCalls.then(results => /* results -> array with the 3 returned values */);
```

If any Promise is rejected (error), all other promises will be ignored and the error will be thrown, going directly to the catch method.

We can use **Promise.allSettled** instead of Promise.all, if we want to wait for all promises, even if one or more of them throw an error.

```
function make3AjaxCalls() {
   let p1 = Http.get(...); // returns a Promise
   let p2 = Http.get(...); // returns a Promise
   let p3 = Http.get(...); // returns a Promise

   return Promise.allSettled([p1, p2, p3]);
}

make3AjaxCalls.then(results => results.forEach(result => {
   if(result.status === 'fulfilled') {
      console.log(result.value);
   } else { // result.status === 'rejected'
```

```
        console.log(result.reason);
    }
}));
```

Another interesting method is **Promise.race**. It receives a collection of promises (similarly to Promise.all), but only returns the value of the first promise that resolves (finishes in first place), ignoring the rest.

```
function make3AjaxCalls() {
    let p1 = Http.get(...); // returns a Promise
    let p2 = Http.get(...); // returns a Promise
    let p3 = Http.get(...); // returns a Promise

    return Promise.race([p1, p2, p3]);
}

make3AjaxCalls.then(result => /* value of the fastest promise */);
```

AJAX

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.

Fetch API

The new JavaScript Fetch API makes much easier to make calls to the server than the classic AJAX with XMLHttpRequest. This API is supported by the newest versions of all the most used browsers. More info here.

To make use of this API, we just have to use the **fetch** global function. This function returns a **Promise** with a Response object. The promise only will be rejected when there's a network failure or something that makes impossible to complete the request. The simplest request is sending the url as a parameter (GET):

```
fetch(`${SERVER}/products`).then(resp => {
   if(!resp.ok) throw new Error(resp.statusText);
   return resp.json(); // promise
}).then(respJSON => {
   respJSON.products.forEach(p => appendProduct(p));
}).catch(error => console.error(error));
```

Notice that when transforming the response to JSON, this operation also returns a Promise. It's easier to return that promise and process it in the next **then** method.

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]);

fetch(`${SERVER}/products`, {
   method: 'POST',
   body: 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):

```
fetch(`${SERVER}/products`, {
   method: 'POST',
   body: 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
};

fetch(`${SERVER}/products`, {
   method: 'POST',
   body: JSON.stringify(prod),
   headers: {
      'Content-Type': 'application/json'
   }
})
```

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() {
   fetch(`${SERVER}/products`).then(resp => {
      if(!resp.ok) throw new Error(resp.statusText);
      return resp.json(); // promise
   }).then(respJSON => {
      respJSON.products.forEach(p => appendProduct(p));
   }).catch(error => console.error(error));
}


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 the inserted product or an error status (resp.ok === false). 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
    };

    fetch(`${SERVER}/products`, {
        method: 'POST',
        body: JSON.stringify(prod),
        headers: {
            'Content-Type': 'application/json'
        }
    }).then(resp => {
        if(!resp.ok) throw new Error(resp.statusText);
        return resp.json(); // promise
    }).then(respJSON => {
        appendProduct(respJSON.product);
    }).catch(error => console.error(error));
}
```

Encapsulating AJAX requests in classes

It can be a good idea to create a generic class to make HTTP requests that eliminates the need to repeat some code on each server call, and also returns the server's response formatted in JSON.

```
class Http {
    static ajax(method, url, headers = {}, body = null) {
        return fetch(url, { method, headers, body})
            .then(resp => {
                if(!resp.ok) throw new Error(resp.statusText);
                return resp.json(); // promise
            });
    }

    static get(url) {
        return Http.ajax('GET', url);
```

```
    }

    static post(url, data) {
        return Http.ajax('POST', url, {'Content-Type': 'application/json'}, JSON.stringify(data));
    }

    static put(url, data) {
        return Http.ajax('PUT', url, {'Content-Type': 'application/json'}, JSON.stringify(data));
    }

    static delete(url) {
        return Http.ajax('DELETE', url);
    }
}
```

It's also a good idea to create intermediate classes for handling these requests to the server. For example, let's create a class called Product that handles everything related to products (AJAX calls, creating the DOM element, etc.):

```
class Product {
    static getProducts() {
        return Http.get(`${SERVER}/products`).then((response) => {
            return response.products.map(prod => new Product(prod));
        });
    }

    constructor(prodJSON) {
        this.id = prodJSON.id || -1; // -1 is default value (not present)
        this.name = prodJSON.name;
        this.description = prodJSON.description;
        this.photo = prodJSON.photo;
    }

    add() {
        return Http.post(`${SERVER}/products`, this)
            .then((response) => {
                return new Product(response.product);
            });
    }

    update() {
        return Http.put(`${SERVER}/products/${this.id}`, this)
            .then((response) => {
                return new Product(response.product);
            });
    }

    delete() {
        return Http.delete(`${SERVER}/products/${this.id}`);
    }

    toHTML() { // In this example, it returns a <tr> with the product info
        let tr = document.createElement("tr");
        // Image
        let imgTD = document.createElement("td");
        let img = document.createElement("img");
        img.src = `${SERVER}/${this.photo}`;
        imgTD.appendChild(img);

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

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

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

It's now easier to get products and add a new one from the main JavaScript file:

```
function getProducts() {
    Product.getProducts().then(prods => {
        products = prods;
        let tbody = document.querySelector("tbody");
        products.forEach(p => {
            tbody.appendChild(p.toHTML());
        });
    }).catch(error => alert(error.toString()));
}

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

    prod.add().then(prod => {
        let tbody = document.querySelector("tbody");
        tbody.appendChild(prod.toHTML());
    }).catch(error => alert(error.toString()));
}
```

async / await

Keywords **async** and **await** have been introduced in the ES2017 standard. They have been created to make the syntax more readable when processing promises.

await is used to wait for a Promise to resolve and return its value. The problem is that waiting for a Promise, blocks the execution of the current function's code until that Promise ends. This is a big problem if we used **await** in the main thread, because we would block all JavaScript code execution (events, function calls, etc.).

That's why JavaScript only allows the use of **await** inside an **async** function. An async function can wait for other promises to end (**await**) and also returns a Promise.

```
static async getProducts() { // Returns a Promise
   const resp = await Http.get(`${SERVER}/products`); // Wait for the promise and get the value
   return response.products.map(prod => new Product(prod));
}
```

The above method is equivalent to this:

```
static getProducts() {
    return Http.get(`${SERVER}/products`).then((response) => {
        return response.products.map(prod => new Product(prod));
    });
}
```