CURSO DE PROGRAMACIÓN JAVA FULLSTACK

JavaScript

4. Programación Orientada a Objetos. AJAX. Promesas.

Arturo Bernal Mayordomo

Edición: Noviembre 2019

Contenido

1. - Objetos y clases en JavaScript

JavaScript es un lenguaje orientado a objetos, y hasta la versión ES2015, trabajar con clases se hacía de una forma diferente a otros lenguajes orientados a objetos conocidos. Sin embargo, esto ha cambiado, y ahora tenemos una sintaxis más familiar para programadores que conozcan Java, C#, PHP, etc.

1.1 *JSON*

JSON o **J**ava**S**cript **O**bject **N**otation es una notación especial usada para crear objetos genéricos en JavaScript. Está siendo un formato cada vez más popular para el intercambio de datos entre aplicaciones, o para almacenar información en bases de datos noSQL (MongoDB por ejemplo). Para más información consulta json.org.

Antes de nada, vamos a ver cómo crear un objeto genérico en JavaScript sin usar JSON, y le añadiremos algunas propiedades y métodos (sí, en JavaScript, podemos añadir propiedades y métodos a un objeto en cualquier momento). Para crear un objeto genérico (y por tanto, vacío), usamos la clase Object.

Como podemos ver, añadimos (o accedemos a) propiedades al objeto usando el punto (**object.property**) o la notación de array asociativo (**object[property]**).

```
let obj = new Object();
obj.nombre = "Peter"; // Añadimos la propiedad 'nombre' usando la notación del punto
obj["edad"] = 41; // Añadimos la propiedad 'edad' usando la notación de un array asociativo
obj.getInfo = function() { // Creamos un nuevo método → getInfo()
   return "Mi nombre es " + this.nombre + " y tengo " + this.edad
}

console.log(obj.getInfo()); // Imprime "Mi nombre es Peter y tengo 41"
console.log(obj.nombre); // Imprime "Peter". Accedemos al nombre usando la notación del punto
console.log(obj["nombre"]); // Imprime "Peter". Ahora accedemos con la notación del array asociativo
var prop = "nombre";
console.log(obj[prop]); // Imprime "Peter". Podemos acceder a la propiedad “nombre” a partir de una
variable que almacena el nombre de dicha propiedad (sólo para la notación de array)
```

Ahora, haremos lo mismo pero usando la notación JSON. Es equivalente a lo que hemos hecho antes, pero lo que haremos será asignarle las propiedades con los dos puntos ‘:’ en lugar del igual ‘=’. Las propiedades iniciales serán declaradas entre llaves, pero podemos añadir más luego como hicimos en el ejemplo anterior. Lo equivalente a usar new Object(), sería dejar las llaves vacías **{}** (objeto vacío).

```
var obj = {
   nombre: "Peter",
   edad: 41,
   getInfo: function () { // Método
      return "Mi nombre es " + this.nombre + " y tengo " + this.edad
   }
}
```

Array de objetos

En JSON, como en JavaScript, los corchetes se usan para crear arrays. Dentro de un array podemos almacenar otros objetos (en formato JSON siempre), valores primitivos, u otros arrays.

```
var persona = {
   nombre: "Peter",
   edad: 41,
   trabajos: [ // trabajos es un array de objetos JSON
      {
         descripción: "Payaso triste",
         duración: "2003-2005"
      },
      {
         descripción: "Sexador de pollos",
         duración: "2005-2015"
      }
   ]
```

```
}

console.log(persona.trabajos[1].descripción); // Imprime "Sexador de pollos"
```

1.2 *Clases en ES2015*

En ES2015, podemos declarar nuevas clases de forma similar a cómo lo hacemos en otros lenguajes como Java, C#, PHP, etc. Aunque todavía, está más limitado (todo es de ámbito público, por ejemplo) que las posibilidades que nos dan estos otros lenguajes. Para quien conozca ES5, internamente está utilizando funciones constructoras y prototype (ver anexo).

```
class Product { }

console.log(typeof Product); // Imprime “function” (Mira la explicación de funciones constructoras del anexo)
```

Para crear un constructor tenemos que implementar un método llamado constructor() que será llamado automáticamente cuando se instancie un nuevo objeto.

```
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}
```

Así es como declaramos métodos de una clase. Internamente se añaden al prototype de la función constructora (ES5 → ver anexo).

```
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
```

1.3 *Herencia*

En ES2015, una clase puede heredar de otra utilizando la palabra reservada **extends**. Heredará todas las propiedades y métodos de la clase padre. Por supuesto, podremos sobrescribirlos en la clase hija, aunque seguiremos pudiendo llamar a los métodos de la clase padre utilizando la palabra reservada **super**. De hecho, si creamos un constructor en la clase hija, debemos llamar al constructor padre.

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

  sayHello() {
    console.log(`Hola, soy ${this.name}`);
  }

  sayType() {
    console.log("Soy un usuario");
  }
}
```

```
class Admin extends User {
  constructor(name) {
    super(name); // Llamamos al constructor de User
  }

  sayType() { // Método sobrescrito
    super.sayType(); // Llamamos a User.sayType (método del padre)
    console.log("Pero también un admin");
  }
}

let admin = new Admin("Anthony");
admin.sayHello(); // Prints "Hola, soy Anthony"
admin.sayType(); // Imprime "Soy un usuario" y "Pero también un admin"
```

1.4 *Métodos estáticos*

En ES2015 podemos declarar métodos estáticos, pero no propiedades estáticas. Estos métodos se llaman directamente utilizando el nombre de la clase y no tienen acceso al objeto this (no hay objeto instanciado).

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

console.log(User.getRoles()); // ["user", "guest", "admin"]
let user = new User("john");
// No podemos llamar a un método estático desde un objeto!!!
console.log(user.getRoles()); // Uncaught TypeError: user.getRoles is not a function
```

1.5 *Valor primitivo y valor string*

Cuando un objeto es convertido a string (concatenándolo por ejemplo), el método toString (heredado de Object) es automáticamente llamado. Por defecto, imprimirá el tipo de objeto "[object Object]", pero podemos sobreescribir el comportamiento de este método en el objeto (o en su prototype).

```
class Warrior {
  constructor(name, vitality) {
    this.vitality = vitality;
    this.name = name;
  }
}

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

console.log(w1.toString()); // Imprime "[object Object]"
console.log("Warrior (w1): " + w1); // Imprime "Warrior(w1): [object Object]" (llama al toString() de Object)

class Warrior2 {
  constructor(name, vitality) {
    this.vitality = vitality;
    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)"
```

Cuando comparamos objetos usando los operadores relacionales (>, <, =>, =<), obtenemos el el valor primitivo por defecto (por defecto toString()). Si sobrescribimos el método **valueOf()** (heredado de Objeto), devuelve otro valor primitivo, que será usado para este tipo de comparaciones.

```
class Warrior {
  constructor(name, vitality) {
    this.vitality = vitality;
    this.name = name;
  }

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

  valueOf() {
    return this.vitality; // El valor primitivo será la vitalidad
  }
}

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

console.log(w1 > w2); // Imprime true: 150(w1) > 100(w2)
```

1.6 *Desestructurar objetos*

Desde ES2015, también es posible desestructurar propiedades de objetos. El funcionamiento es similar a desestructurar un array, pero usamos llaves '**{}**' en lugar de corchetes.

```
let usuario = {
  id: 3,
  nombre: "Pedro",
  email: "peter@gmail.com"

}

let {id, nombre, email} = usuario;
console.log(nombre); // Imprime "Pedro"

// Esta función recibirá un objeto como primer parámetro y lo desestructurará
function imprimirUsuario({id, nombre, email}, otraInfo = "Nada") {
  ...
}

otraInfo(usuario, "No es muy listo");
```

Existe la posibilidad de asignar diferentes nombres desde las propiedades en las variables desestructuradas → (**nombrePropiedad: nombreVariable**):

```
let {id: idUsuario, nombre: nombreUsuario, email: emailUsuario} = usuario;
console.log(nombreUsuario); // Imprime "Pedro"
```

2. - **Promesas**

Es una característica que estaba sólo disponible usando jQuery o en la libería Q (que AngularJS usa), y ahora está soportado de forma nativa.

Una **promesa** es un objeto que se crea para resolver una acción asíncrona. El constructor recibe una función con dos parámetros, **resolve** y **reject**. *resolve* se llama cuando la acción acaba correctamente devolviendo (opcionalmente) algún dato, mientras que *reject* se usa cuando se produce un error. Una promesa acaba cuando se llama a una de estas dos funciones.

Cuando obtenemos una promesa, tenemos que suscribirnos a ella llamando al método **then** y luego pasándole una función que manejará el resultado. Esta función se ejecutará cuando se resuelva la promesa correctamente.

```
function getPromise() {
  return new Promise((resolve, reject) => {
    console.log("Promesa llamada...");
    setTimeout(function() {
      console.log("Resolviendo la promesa...");
      resolve(); // Promesa resuelta!.
    }, 3000); // Esperamos 3 segundos y acabamos la promesa
  });
}

getPromise().then(() => console.log("La promesa ha acabado!"));

console.log("El programa continúa. No espera que termine la promesa (operación asíncrona)");
```

También podemos encadenar varias llamadas al método then, ya que este a su vez devuelve otra promesa con el valor que devuelve (return).

```
function getPromise() {
  return new Promise((resolve, reject) => {
    setTimeout(function() {
      resolve(1);
    }, 3000); // Después de 3 segundos, termina la promesa
  });
}

getPromise().then(num => {
  console.log(num); // Imprime 1
  return num + 1;
}).then(num => {
  console.log(num); // Imprime 2
}).catch(error => {
  // Si se produce un error en cualquier parte de la cadena de promesas... (o promesa original
rechazada)
});
```

Iremos directamente al bloque **catch** si **lanzamos un error** dentro de una sección **then**. O si la promesa original se rechaza con reject.

```
getProducts().then(products => {
  if(products.length === 0) {
    throw 'No hay productos!'
  }
  return products.filter(p => p.stock > 0);
}).then(prodsStock => {
  // Imprime los productos con stock en la página
}).catch(error => {
  console.error(error);
});
```

Con el objetivo de hacer nuestro código más fácil de leer, podemos separar las funciones del bloque **then** (o catch) en lugar de usar funciones anónimas o arrow.

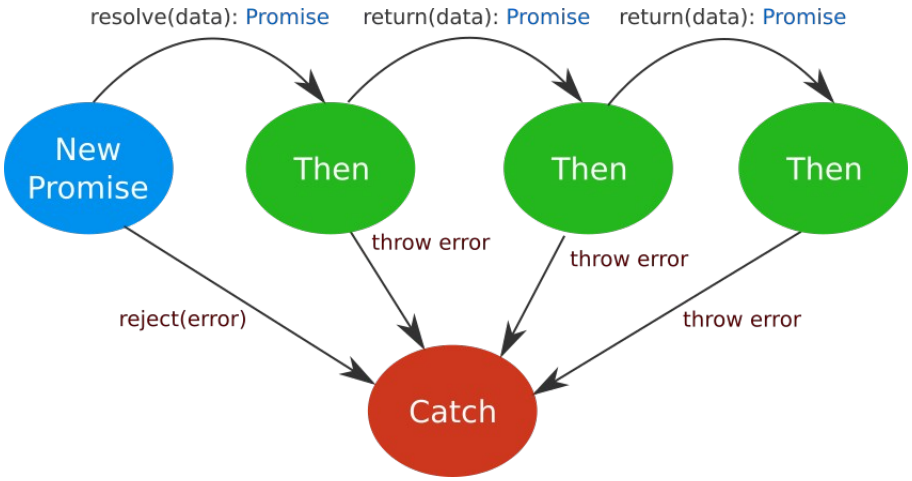
```
function filterStock(products) {
```

```
    if(products.length === 0) {
        throw 'No hay productos!'
    }
    return products.filter(p => p.stock > 0);
}

function showProducts(prodsStock) {
    // Imprime el producto en la página
}

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

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



Tanto el método **then** como el método **catch** pueden devolver un valor directamente o dentro de una Promesa.

También tenemos el método Promise.all, al que enviaremos un array de promesas y se resolverá cuando todas las promesas creadas se hayan resuelto (por ejemplo, varias llamadas HTTP al servidor), devolviendo un array con todos los resultados generados. También existe Promise.race, que recibe un array de promesas y devuelve el resultado de la primera que acaba.

2.1

3. - AJAX

Aunque librerías JQuery o frameworks como Angular tienen métodos y funciones para realizar **AJAX** al servidor, ya que hasta ahora simplificaban bastante el proceso, Actualmente es igual de simple hacer lo mismo sin depender de ninguna librería externa gracias a la API Fetch.

Una llamada AJAX es una petición al servidor web tras haberse cargado la página, que no implica volver a recargar la página actual. Es por tanto una buena opción para ahorrar ancho de banda, ya que el servidor envía sólo los datos que necesita (por ejemplo para actualizar alguna información sobre un producto en la web) y no toda la página.

AJAX es el acrónimo de **Asynchronous JavaScript And XML** (recuerda que HTML deriva de XML), pero hoy en día JSON es el formato usado para enviar y recibir datos del servidor (JSON se integra de forma nativa con JavaScript). **Asíncrono** significa que si realizamos una petición, la respuesta no la recibiremos de forma inmediata. Utilizaremos promesas para gestionar estas peticiones.

3.1 *Métodos básicos de HTTP*

La petición y respuesta del servidor será mediante el protocolo HTTP. Cuando hacemos una petición HTTP, el navegador envía al servidor: cabeceras (como *useragent* el cual identifica al navegador, las preferencias de idioma, etc.), el tipo de petición HTTP y parámetros o datos (si son necesarios).

Hay muchos tipos de petición que podemos enviarle al servidor. Los más usados cuando realizamos una llamada AJAX (o accedemos a un servicio web) son:

- **GET** → Recibe datos y normalmente no modifica nada. Es equivalente a un SELECT en SQL. Cuando usamos este método, la información normalmente se envía concatenada en la URL (formato URIEncoded).
- **POST** → Operación para insertar un nuevo dato (no necesariamente) en el servidor. Equivale a INSERT en SQL. Los datos a insertar se envían dentro del cuerpo de la petición HTTP.

- **PUT** → Esta operación actualiza datos existentes en el servidor. Equivale a UPDATE en SQL. La información que identifica el objeto a actualizar es enviada en la URL (como en GET), y los datos a modificar se envían aparte en el cuerpo de la llamada (como en POST).
- **DELETE** → Esta operación elimina un dato del servidor, como la operación DELETE de SQL. La información que identifica al objeto a eliminar será enviada en la URL (como en GET).

3.2 *La API Fetch*

Esta nueva API de JavaScript nos facilita mucho el trabajo comparado con las llamadas clásicas al servidor usando XMLHttpRequest. Está soportada por todos los navegadores modernos. Más información.

Para utilizar esta API, tenemos la función global **fetch**. Esta función devuelve una promesa con un objeto del tipo Response. La promesa puede ser rechazada (reject) cuando hay un error con la comunicación que impide recibir una respuesta, pero no cuando el servidor devuelva un código de error.

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

3.3 *Enviar formularios con archivos usando FormData*

Si necesitamos enviar un formulario que contiene archivos por AJAX, podemos utilizar la clase FormData. El servidor recibirá los archivos por separado y el resto de la información en formato **urlencoded**.

```
<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>
```

Podemos instanciar el objeto FormData y añadirle los valores manualmente:

```
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
})
```

O podemos añadirle el formulario entero con todo lo que contiene (después podríamos seguir añadiendo más datos si queremos):

```
fetch(`${SERVER}/products`, {
   method: 'POST',
   body: new FormData(document.getElementById("addProduct")),
   headers: {
      'Content-Type': 'application/json'
   }
})
```

3.4 *Enviar archivos en JSON codificados con Base64*

Si nuestro servidor requiere que le pasemos todos los datos en formato JSON, los archivos deben ser convertidos a string. Para ello los codificaremos utilizando el formato Base64.

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

   // Cuando se seleccione un archivo, lo procesamos
   document.getElementById("photo").addEventListener('change',  () => {
      var file   = document.getElementById("photo").files[0];
      var reader = new FileReader();

      reader.addEventListener("load", () => { //Evento de conversión a Base64 completa (asíncrono)
         imagePreview.src = reader.result; // Mostramos la imagen cargada en un elemento <img>
      }, false);

      if (file) { // Si se ha seleccionado un archivo válido (convertir a Base64)
         reader.readAsDataURL(file);
      }
   });
});
```

De esta manera, podemos incluir el archivo dentro de un objeto JSON como un dato más. En el servidor se decodificará y se guardará otra vez como archivo:

```
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'
   }
});
```

3.5 *Ejemplos*

1. Ejemplo de una petición **GET**. Recibiremos una respuesta JSON con un array de productos, que recorreremos y crearemos la estructura HTML de cada producto para añadir al 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");
   // Imagen
   let imgTD = document.createElement("td");
   let img = document.createElement("img");
   img.src = `${SERVER}/img/${product.photo}`;
   imgTD.appendChild(img);

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

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

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

2. Ejemplo de llamada **POST**. Los datos se envían al servidor en formato JSON. Recibiremos a su vez otra respuesta JSON con el producto insertado. Si todo va bien, el producto recibido se insertará en el 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));
}
```

3.6 *Encapsular llamadas AJAX con clases*

Para facilitar un poco más las llamadas AJAX al servidor, podemos crearnos una clase con unos cuantos métodos estáticos.

```
"use strict";

class Http {
    static async ajax(method, url, headers = {}, body = null) {
        const resp = await fetch(url, { method, headers, body});
        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);
  }
}
```

También es buena idea crear clases intermediarias para gestionar las peticiones al servidor. Por ejemplo, vamos a crear una clase que se encargue de gestionar lo relacionado con productos (AJAX, crear elementos DOM, 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;
    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() { // Devuelve una fila de tabla <tr> con la información
    let tr = document.createElement("tr");
    // Imagen
    let imgTD = document.createElement("td");
    let img = document.createElement("img");
    img.src = `${SERVER}/${this.photo}`;
    imgTD.appendChild(img);

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

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

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

En el archivo JavaScript principal el código ahora queda más limpio:

```
function getProducts() { // Obtener productos y añadirlos al DOM
   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() { // Añadir un producto al servidor e insertarlo en el DOM (tabla)
   let prod = new Product({
      name: document.getElementById("name").value,
      description: document.getElementById("description").value,
      photo: imagePreview.src
   });

   prod.add().then(prod => { // Recibimos el producto añadido a la base de datos
      let tbody = document.querySelector("tbody");
      tbody.appendChild(prod.toHTML());
   }).catch(error => alert(error.toString()));
}
```

4. - async / await

Las instrucciones **async** y **await** fueron introducidas en el estándar ES2017. Se utilizan para que la sintaxis al trabajar con promesas sea más amigable.

await se utiliza para esperar que una promesa termine y nos devuelva su valor. El problema es que eso bloquea la ejecución del siguiente código hasta que la promesa acabe. Esto implicaría bloquear el programa durante un tiempo indeterminado. Por ello, sólo podemos usarlas dentro de funciones precedidas con la palabra **async**. Una función async siempre nos devuelve una promesa, que contendrá el valor devuelto con la palabra return.

```
static async getProducts() { // Devuelve una promesa (async)
   const resp = await Http.get(`${SERVER}/products`); // Esperamos la respuesta del servidor
   return response.products.map(prod => new Product(prod));
}
```

El método de arriba equivale a esto:

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