Bloque 3

Programación con JavaScript

Programación Orientada a Objetos. AJAX. Promesas.

Programación con JavaScript
Cefire 2017/2018
Autor: Arturo Bernal Mayordomo

Index

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.

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”
```

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
```

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"
```

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
```

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)
```

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"
```

AJAX

Aunque normalmente se usa una librería como JQuery o un framework como Angular para hacer llamadas **AJAX** al servidor, porque ofrecen muchas más funcionalidades y hacen que el trabajo sea más simple, no es mala idea aprender los conceptos básicos de AJAX directamente en JavaScript. Esto, nos va a ayudar a comprender la capa de abstracción que nos ofrecen ciertas librerías o frameworks.

Una llamada AJAX es simplemente una petición al servidor web para que nos ofrezca algo tras haberse cargado la página y que no implique que se tenga que recargar una página entera otra vez. 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 está siendo la forma 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. Le indicaremos una función que se llamará cuando recibamos la información (o se produzca un error). Hasta entonces, la página web no estará bloqueada y continuará respondiendo de forma normal a los eventos, etc.

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).

XMLHttpRequest

Para poder realizar una petición AJAX al servidor, debemos instanciar un objeto **XMLHttpRequest**. Después, llamamos al método **open** para establecer la comunicación. Se reciben 3 parámetros: **método HTTP** ("GET", "POST", "PUT", "DELETE"), la **url** del servidor, y un booleano que indique si la llamada es **asíncrona** (recomendado), o si bloqueará la ejecución de la aplicación hasta recibir la respuesta.

Para añadir cabeceras a la petición HTTP, podemos usar el método **setRequestHeader** después de llamar a **open**. Tras esto, podremos llamar al método **send** y enviar el cuerpo de la petición (vacío si es una petición GET o DELETE, o con un string con los datos si es una petición POST o PUT).

```
var http = new XMLHttpRequest();
http.open('POST', '/mypage.php', true);
// urlencoded significa var1=value1&var2=value2... (los valores deben codificarse con EncodeURIComponent)
http.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
http.send("data1=" + encodeURIComponent(data1) + "&data2=" + encodeURIComponent(data2));
```

XMLHttpResponse

El envío de una petición al servidor pasa a través de diferentes estados hasta que se recibe una respuesta. Cuando la petición cambia de estado, se genera un evento **readystatechange** sobre el objeto XMLHttpRequest. Debemos añadir un manejador para cuando recibamos la respuesta.

Dentro de la función comprobaremos dos propiedades del objeto XMLHttpRequest. La primera es **readyState**, que nos indica el estado de la conexión (0: la petición no se ha iniciado, 1: se ha establecido conexión con el servidor, 2: la petición ha sido recibida, 3: se está procesando la petición, 4: la petición ha finalizado y la respuesta está lista), y también la propiedad del **status**, la cual (cuando la respuesta ha sido recibida) indica si todo ha ido bien, o se ha producido un error, usando los códigos de respuesta HTTP (200 → Todo correcto, 404 → Recurso no encontrado, 500 → Error interno del servidor, etc.). Cuando **readyState** vale 4 y el **status** es igual a 200, habremos recibido una respuesta con éxito.

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

Tipos de eventos

Además del evento **readystatechange**, existen otros eventos más simples que se lanzan sólo en situaciones completas y no en cada cambio de estado:

- progress: Puede activarse varias veces durante el envío de datos al servidor. Indica el porcentaje de datos transferidos al servidor (útil para detectar el progreso cuando estamos subiendo archivos por ejemplo). Propiedades:
  - **lengthComputable** (boolean) → Si se conoce el total de bytes a transferir

    - **loaded (number)** → Bytes transferidos al servidor
    - **total (number)** → Total de bytes a transferir (0 si no se conoce)
- **load**: Se lanza cuando se recibe la respuesta del servidor (equivale a readyState == 4). Esta respuesta puede ser correcta (código 200 o similar) o un error HTTP (comprueba la propiedad ***status***).
- **error**: Se lanza cuando se produce un error al enviar la información (fallo de la red por ejemplo).
- **abort**: Ocurre cuando se aborta (cancela) la petición. Esto se puede hacer de manualmente llamando al método **abort()** del objeto XMLHttpRequest.
- **timeout**: Cuando no se recibe respuesta pasado un determinado tiempo. Puedes establecer este tiempo en ms con la propiedad **timeout** del objeto XMLHttpRequest (antes de enviar datos).
- **loadend**: Este evento se dispara siempre que termine una petición HTTP independientemente de si se ha recibido respuesta, ha ocurrido un error, se ha abortado, o ha saltado un timeout. Útil para dejar de mostrar una animación de carga por ejemplo.

Ejemplo

```
let http = new XMLHttpRequest();
http.open('POST', 'myserver.com/products', true);
http.send(JSON.stringify(product)); // Enviando datos en formato JSON

http.addEventListener('load', (event) => {
   if(http.status === 200) {
      let response = JSON.parse(http.responseText); // Datos recibidos en formato JSON
      // Do something with the response
   } else {
      showError();
   }
});

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

function showError(event) {
   alert("Ocurrió un error o se abortó la conexión");
}

function showProgress(event) {
  if (evt.lengthComputable) {
    let percentComplete = evt.loaded / evt.total;
    // Actualizamos una barra de progreso o el porcentaje
  } else {
    // No se puede saber el progreso porque el tamaño total del envío se desconoce
  }
}
```

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]);

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

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

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

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
};

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

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() {
   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); // Datos recibidos en formato JSON
         response.products.forEach((p) => appendProduct(p));
      } else {
         showError();
      }
   });
}

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
```

```
    };

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

    http.addEventListener("error", (event) => alert("Ocurrió un error o la transferencia fue abortada"));

    http.addEventListener('load', (event) => {
        if(http.status === 200) {
            let response = JSON.parse(http.responseText); // Datos recibidos en formato JSON
            appendProduct(response.product); // Añadimos el producto al DOM
        } else {
            alert(`Error añadiendo producto. Status: ${http.status}`);
        }
    });
}
```

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. Están mejor preparadas para manejar las respuestas asíncronas que el concepto de enviar una función manejadora por parámetro a otra función.

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

```
function getPromise() {
   return new Promise(function(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 se espera a la promesa (operación asíncrona)");
```

Volvamos atrás en el tiempo y veamos cómo manejar una respuesta AJAX usando una función manejadora con el evento 'load', y cómo gestionar los errores, etc. Como podemos ver, el código que obtendríamos es bastante grande porque tiene que administrar bastantes escenarios.

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

   http.addEventListener('load', function() { // El evento load se lanza cuando http.readyState === 4
      if(http.status === 200) { // OK
         var prod = JSON.parse(http.responseText);
         if(prod.error) { // El servidor nos devuelve un error
            // Procesa el error ...
         } else {
            // Procesa los datos.....
         }
      } else {
         // Ha habido un error, y queremos procesarlo
      }
   });

   http.addEventListener('error', function() {
      // Ha habido un error y queremos procesarlo
```

```
    });
}

getProduct(12); // El código de la función lo manejará todo
```

Ahora vamos a usar una promesa para gestionar la llamada AJAX. Esta vez el resultado se gestiona fuera del código de la llamada, por lo que queda más limpio.

```
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() { // Evento load se lanza cuando http.readyState === 4
            if(http.status === 200) { // OK
                let prod = JSON.parse(http.responseText);
                if(prod.error) {
                    reject(prod.error); // Devolvemos un error rechazando la promesa
                } else {
                    resolve(prod); // Resolvemos la promeso devolviendo el producto
                }
            } else {
                reject('Error HTTP: ' + http.status);
            }
        });

        http.addEventListener('error', function() {
            reject('Error en la petición http de un producto');
        });
    });
}

getProduct(12) // Nos subscribimos a la promesa
    .then(function(product) { // La promesa está resuelta y devuelve un producto
        // Procesamos los datos
    })
    .catch(function(error){ // La promesa es rechazada (error)
        // Procesamos/mostramos el mensaje de error
    });
```

El método **then** puede devolver a su vez valores con return. Este valor se devuelve a su vez como promesa para que podamos concatenar varias acciones. Estamos formando una cadena de promesas donde cada una se dedica a una fase del procesamiento de los datos.

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

getPromise().then(function(num) {
    console.log(num); // Imprime 1
    return num + 1;
}).then(function(num) {
    console.log(num); // Imprime 2
}).catch(function (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(function (products) {
   if(products.length === 0) {
      throw 'No hay productos!'
   }
   return products.filter(p => p.stock > 0);
}).then(function (prodsStock) {
   // Imprime los productos con stock en la página
}).catch(function (error) {
   console.error(error);
})
```

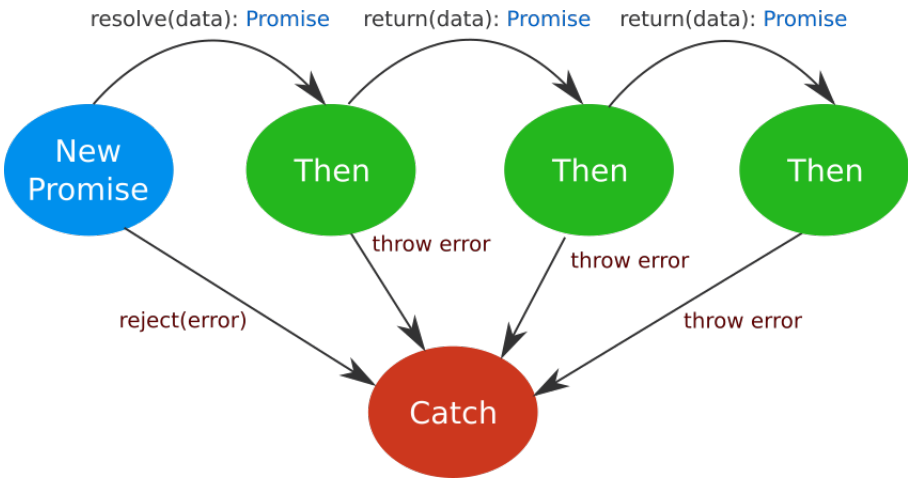
Con el objetivo de hacer nuestro código más fácil de leer, podemos separar las funciones del bloque **then** (o catch) en lugar de usar funciones anónimas o lambdas.

```
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 una nueva Promesa (en lugar de un valor directamente como hemos visto en el método then). Esta promesa se resolverá y su valor devuelto se recibe en el siguiente **then** → concatenando promesas.

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.

Encapsular llamadas AJAX con clases y promesas

Una buena forma de organizar nuestras peticiones AJAX sería encapsularlas en promesas (separación de código y mejor mantenibilidad). Tambien es una buena práctica crear clases y métodos que trabajen con dichas promesas.

Empezaremos creando una clase Http para gestionar llamadas AJAX:

```
class Http {
  static ajax(method, url, data) {
    return new Promise((resolve, reject) => {
      let http = new XMLHttpRequest();
      http.open(method, url);
      http.send(data);

      http.addEventListener("error", (errorEvent) => reject(errorEvent.message));

      http.addEventListener('load', (event) => {
        if (http.status === 200) {
          let response = JSON.parse(http.responseText);
          resolve(response); // Respuesta correcta del servidor
        } else {
          reject(Error(http.statusText)); // Error
        }
      });
    });
  }

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

  static post(url, data) {
    return Http.ajax('POST', url, data);
  }

  static put(url, data) {
    return Http.ajax('PUT', url, data);
  }

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

También es una 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 {
  constructor(prodJSON) {
    this.id = prodJSON.id || -1; // -1 valor por defecto (por si no hay id en el objeto)
    this.name = prodJSON.name;
    this.description = prodJSON.description;
    this.photo = prodJSON.photo;
  }

  static getProducts() { // Obtener productos (GET)
    return Http.get(`${SERVER}/product`).then((response) => {
      // Podemos lanzar un error para ir directamente al próximo catch
      if (!response.products) throw "El servidor no ha devuelto productos!";
```

```
            return response.products.map(prod => new Product(prod)); // Respuesta correcta
        });
    }

    add() { // Añadir producto (POST)
        return Http.post(`${SERVER}/product`, JSON.stringify(this))
            .then((response) => {
                if (!response.ok) throw response.error;
                return new Product(response.product);
            });
    }

    update() { // Actualizar producto (PUT)
        return Http.put(`${SERVER}/product/${this.id}`, JSON.stringify(this))
            .then((response) => {
                if (!response.ok) throw Error(response.error);
                return new Product(response.product);
            });
    }

    delete() { // Borrar producto (DELETE)
        return Http.delete(`${SERVER}/product/${this.id}`).then((response) => {
            if (!response.ok) throw Error(response.error);
            return response.ok;
        });
    }

    toHTML() {
        ...
    }
}
```

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
    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()));
}
```