Algunos conceptos básicos de JavaScript para este curso

En este documento vamos a dar un breve repaso a algunos conceptos de JavaScript que utilizaremos a lo largo del curso, y con los que conviene que nos empecemos a familiarizar desde ya, si no los hemos utilizado aún. En concreto, trataremos:

- Formas de definir variables y constantes
- El uso de "funciones flecha" (*arrow functions*), también conocidas como "expresiones lambda", como alternativa a la definición clásica de funciones, o al uso de funciones anónimas.
- Cómo gestiona JavaScript el código asíncrono, mediante *callbacks*, promesas o la especificación *async/await*

1. Declaración de variables y constantes

Para empezar, trataremos sobre la declaración de **variables**. La forma más típica que podemos encontrar en Internet para declarar variables en JavaScript es mediante la palabra reservada `var`, que permite declarar variables de cualquier tipo. Por ejemplo:

```
var nombre = "Nacho";
var edad = 41;
```

Sin embargo, esta forma de declarar variables tiene algunos inconvenientes, como por ejemplo, y sobre todo, el hecho de declarar una variable de forma local a un ámbito, y que pueda ser utilizada desde fuera de ese ámbito, porque la variable es válida dentro de la función donde se ha definido. Así, por ejemplo, este código funcionaría, y mostraría "Nacho" como nombre en ambos casos, a pesar de que, intuitivamente, la variable `nombre` no debería existir fuera del *if*:

```
if (2 > 1)
{
    var nombre = "Nacho";
    console.log("Nombre dentro:", nombre);
}
console.log("Nombre fuera:", nombre);   // "Nacho"
```

Para evitar estas vulnerabilidades, emplearemos la palabra reservada `let`, en lugar de `var`, para declarar variables:

```
if (2 > 1)
{
    let nombre = "Nacho";
    console.log("Nombre dentro:", nombre);
}
console.log("Nombre fuera:", nombre);   // Variable no definida
```

De esta forma, el ámbito de cada variable queda restringido al bloque donde se declara, y el código anterior provocaría un error.

Recordemos también que podemos emplear la palabra `const` para definir **constantes** en el código. Esto será particularmente útil tanto para definir constantes convencionales (como un texto o número fijo, por ejemplo) como para cargar librerías, como veremos en sesiones posteriores.

```
const pi = 3.1416;
```

2. Funciones y *arrow functions*

Veamos ahora las diferentes formas de definir funciones o métodos que existen en JavaScript, y con ello, introduciremos un concepto que se ha vuelto muy habitual, y que utilizaremos a menudo en estos apuntes. Se trata de una notación alternativa para definir métodos o funciones, las llamadas *arrow functions* (funciones flecha o funciones lambda).

2.1. Las funciones tradicionales

Comencemos por un ejemplo sencillo. Supongamos esta función tradicional que devuelve la suma de los dos parámetros que se le pasan:

```
function sumar(num1, num2)  {
    return num1 + num2;
}
```

A la hora de utilizar esta función, basta con llamarla en el lugar deseado, pasándole los parámetros adecuados. Por ejemplo:

```
console.log(sumar(3, 2)); // Mostrará 5
```

2.2. Las funciones anónimas

Esta misma función también podríamos expresarla como una función anónima. Estas funciones se declaran "sobre la marcha", y se suelen asignar a una variable para poderlas nombrar o llamar después:

```
let sumarAnonimo = function(num1, num2) {
    return num1 + num2;
};
console.log(sumarAnonimo(3, 2));
```

2.3. Las *arrow functions*

Las "funciones flecha" o *arrow functions* suponen una forma de definir funciones que emplea una expresión lambda para especificar los parámetros por un lado (entre paréntesis) y el código de la función por otro entre llaves, separados por una flecha. Se prescinde de la palabra reservada `function` para definirlas.

La misma función anterior, expresada como *arrow function*, quedaría así:

```
let sumar = (num1, num2) => {
    return num1 + num2;
};
```

Al igual que ocurre con las funciones anónimas, se puede asignar su valor a una variable para usarlo más adelante, o bien definirlas sobre la marcha en un fragmento de código determinado.

De hecho, el código anterior puede simplificarse aún más: en el caso de que la función simplemente devuelva un valor, se puede prescindir de las llaves y de la palabra `return`, quedando así:

```
let sumar = (num1, num2) => num1 + num2;
```

Además, si la función tiene un único parámetro, se pueden prescindir de los paréntesis. Por ejemplo, esta función devuelve el doble del número que recibe como parámetro:

```
let doble = num => 2 * num;
console.log(doble(3));          // Mostrará 6
```

2.3.1. Uso directo de *arrow functions*

Como comentábamos antes, las *arrow functions*, así como las funciones anónimas, tienen la ventaja de poder utilizarse directamente en el lugar donde se precisan. Por ejemplo, dado el siguiente listado de datos personales:

```
let datos = [
    {nombre: "Nacho", telefono: "966112233", edad: 41},
    {nombre: "Ana", telefono: "911223344", edad: 36},
    {nombre: "Mario", telefono: "611998877", edad: 15},
    {nombre: "Laura", telefono: "633663366", edad: 17}
];
```

Si queremos filtrar las personas mayores de edad, podemos hacerlo con una función anónima combinada con la función `filter`:

```
let mayoresDeEdad = datos.filter(function(persona) {
    return persona.edad >= 18;
})
console.log(mayoresDeEdad);
```

Y también podemos emplear una *arrow function* en su lugar:

```
let mayoresDeEdad = datos.filter(persona => persona.edad >= 18);
console.log(mayoresDeEdad);
```

Notar que, en estos casos, no asignamos la función a una variable para usarla más tarde, sino que se emplean en el mismo punto donde se definen. Notar también que el código queda más compacto empleando una *arrow function*.

2.4. *Arrow functions* y funciones tradicionales

La diferencia entre las *arrow functions* y la nomenclatura tradicional o las funciones anónimas es que con las *arrow functions* no podemos acceder al elemento `this`, o al elemento `arguments`, que sí están

disponibles con las funciones anónimas o tradicionales. Así que, en caso de necesitar hacerlo, deberemos optar por una función normal o anónima, en este caso.

3. Los *callbacks*

Uno de los dos pilares en los que se sustenta la programación asíncrona en JavaScript lo conforman los *callbacks*. Un *callback* es una función A que se pasa como parámetro a otra B, y que será llamada en algún momento durante la ejecución de B (normalmente cuando B finaliza su tarea). Este concepto es fundamental para dotar a Node.js (y a JavaScript en general) de un comportamiento asíncrono: se llama a una función, y se le deja indicado lo que tiene que hacer cuando termine, y mientras tanto el programa puede dedicarse a otras cosas.

Un ejemplo lo tenemos con la función `setTimeout` de JavaScript. A esta función le podemos indicar una función a la que llamar, y un tiempo (en milisegundos) que esperar antes de llamarla. Ejecutada la línea de la llamada a `setTimeout`, el programa sigue su curso y cuando el tiempo expira, se llama a la función *callback* indicada.

Probemos a escribir este ejemplo en un archivo llamado `callback.js` en nuestra subcarpeta "*ProyectosNode/Pruebas/PruebasSimples*":

```
setTimeout(function() {console.log("Finalizado callback");}, 2000);
console.log("Hola");
```

Si ejecutamos el ejemplo, veremos que el primer mensaje que aparece es el de "Hola", y pasados dos segundos, aparece el mensaje de "Finalizado callback". Es decir, hemos llamado a `setTimeout` y el programa ha seguido su curso después, ha escrito "Hola" por pantalla y, una vez ha pasado el tiempo estipulado, se ha llamado al *callback* para hacer su trabajo.

Utilizaremos *callbacks* ampliamente durante este curso. De forma especial para procesar el resultado de algunas promesas que emplearemos (ahora veremos qué son las promesas), o el tratamiento de algunas peticiones de servicios.

4. Las promesas

Las promesas son otro mecanismo importante para dotar de asincronía a JavaScript. Se emplean para definir la finalización (exitosa o no) de una operación asíncrona. En nuestro código, podemos definir promesas para realizar operaciones asíncronas, o bien (más habitual) utilizar las promesas definidas por otros en el uso de sus librerías.

A lo largo de este curso utilizaremos promesas para, por ejemplo, enviar operaciones a una base de datos y recoger el resultado de las mismas cuando finalicen, sin bloquear el programa principal. Pero para entender mejor qué es lo que haremos, llegado el momento, conviene tener clara la estructura de una promesa y las posibles respuestas que ofrece.

4.1. Crear una promesa. Elementos a tener en cuenta

En el caso de que queramos o necesitemos crear una promesa, se creará un objeto de tipo `Promise`. A dicho objeto se le pasa como parámetro una función con dos parámetros:

- La función *callback* a la que llamar si todo ha ido correctamente
- La función *callback* a la que llamar si ha habido algún error

Estos dos parámetros se suelen llamar, respectivamente, `resolve` y `reject`. Por lo tanto, un esqueleto básico de promesa, empleando *arrow functions* para definir la función a ejecutar, sería así:

```
let nombreVariable = new Promise((resolve, reject) => {
    // Código a ejecutar
    // Si todo va bien, llamamos a "resolve"
    // Si algo falla, llamamos a "reject"
});
```

Internamente, la función hará su trabajo y llamará a sus dos parámetros en uno u otro caso. En el caso de `resolve`, se le suele pasar como parámetro el resultado de la operación, y en el caso de `reject` se le suele pasar el error producido.

Veámoslo con un ejemplo. La siguiente promesa busca los mayores de edad de la lista de personas vista en un ejemplo anterior. Si se encuentran resultados, se devuelven con la función `resolve`. De lo contrario, se genera un error que se envía con `reject`. Copia el ejemplo en un archivo llamado `prueba_promesa.js` en la carpeta "*ProyectosNode/Pruebas/PruebasSimples*" de tu espacio de trabajo:

```
let datos = [
    {nombre: "Nacho", telefono: "966112233", edad: 41},
    {nombre: "Ana", telefono: "911223344", edad: 36},
    {nombre: "Mario", telefono: "611998877", edad: 15},
    {nombre: "Laura", telefono: "633663366", edad: 17}
];

let promesaMayoresDeEdad = new Promise((resolve, reject) => {
    let resultado = datos.filter(persona => persona.edad >= 18);
    if (resultado.length > 0)
        resolve(resultado);
    else
        reject("No hay resultados");
});
```

La función que define la promesa también se podría definir de esta otra forma:

```
let promesaMayoresDeEdad = listado => {
    return new Promise((resolve, reject) => {
        let resultado = listado.filter(persona => persona.edad >= 18);
        if (resultado.length > 0)
            resolve(resultado);
        else
            reject("No hay resultados");
    });
};
```

Así no hacemos uso de variables globales, y el array queda pasado como parámetro a la propia función, que devuelve el objeto `Promise` una vez concluya. Deja definida la promesa de esta segunda forma en el archivo fuente de prueba.

4.2. Consumo de promesas

En el caso de querer utilizar una promesa previamente definida (o creada por otros en alguna librería), simplemente llamaremos a la función u objeto que desencadena la promesa, y recogemos el resultado. En este caso:

- Para recoger un resultado satisfactorio (`resolve`) empleamos la cláusula `then` .
- Para recoger un resultado erróneo (`reject`) empleamos la cláusula `catch` .

Así, la promesa anterior se puede emplear de esta forma (nuevamente, empleamos *arrow functions* para procesar la cláusula `then` con su resultado, o el `catch` con su error):

```
promesaMayoresDeEdad(datos).then(resultado => {
    // Si entramos aquí, la promesa se ha procesado bien
    // En "resultado" podemos acceder al resultado obtenido
    console.log("Coincidencias encontradas:");
    console.log(resultado);
}).catch(error => {
    // Si entramos aquí, ha habido un error al procesar la promesa
    // En "error" lo podemos consultar
    console.log("Error:", error);
});
```

Copia este código bajo el código anterior en el archivo `prueba_promesa.js` creado anteriormente, para comprobar el funcionamiento y lo que muestra la promesa.

Notar que, al definir la promesa, se define también la estructura que tendrá el resultado o el error. En este caso, el resultado es un vector de personas coincidentes con los criterios de búsqueda, y el error es una cadena de texto. Pero pueden ser el tipo de dato que queramos.

4.3. La especificación async/await

Desde *ECMAScript7* se tiene disponible una nueva forma de trabajar con promesas, a través de la especificación **async/await**. Es una forma más cómoda de llamar a funciones asíncronas y recoger su resultado antes de llamar a otra, sin necesidad de ir anidando cláusulas `then` para enlazar el resultado de una promesa con la siguiente.

Hay que tener en cuenta que se trata de una especificación relativamente reciente (mediados de 2017), y por tanto puede que algunos navegadores aún no la soporten. No entraremos en detalles sobre cómo utilizarla de momento. Lo haremos más adelante, cuando estemos familiarizados con las promesas.