«Talento Tech»

Back-End

Node JS

Clase 03

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 3 - Funciones, Arrays y Methods

Temario:

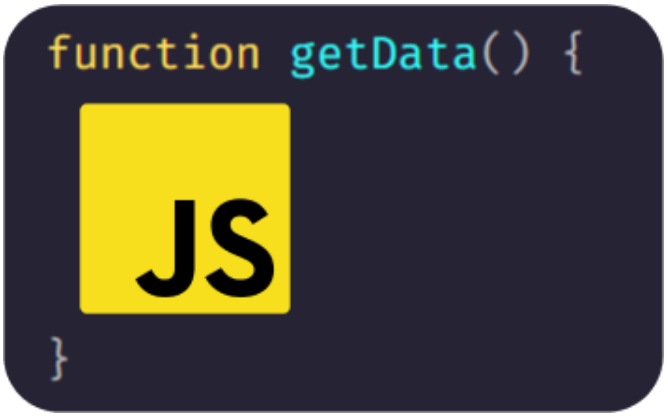
1. **Funciones:**
   - Declaración de funciones
   - HOF: Higher order functions (funciones como argumentos)
2. **Arrays:**
   - Declaración
   - Métodos de arrays más utilizados
3. **Template Literals**:
   - Uso de backticks para strings dinámicos y multilínea

Objetivos de la Clase

En esta clase, se repasará el concepto de funciones, sus diferentes tipos y cómo declararlas. Además, se explorarán las Higher Order Functions y su propósito en la programación. También se estudiará la estructura de los Arrays, junto con sus métodos más útiles. Finalmente, se integrará una nueva forma de crear cadenas de texto dinámicas para enriquecer las habilidades de programación de los estudiantes.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Funciones



¿Qué son las funciones?

Es un concepto fundamental de la programación que nos permite crear algoritmos que encapsulan determinada acción, funcionalidad o lógica que necesitemos reutilizar.

No son algo exclusivo de la sintaxis de Javascript, sino más bien una herramienta fundamental en los lenguajes de programación aunque cada uno tenga formas distintas de declararlas.

Cuando se desarrolla una aplicación, solemos repetir consecuentemente diversas instrucciones. Esto presenta una serie de problemas a futuro ya que:

- El código de la aplicación es mucho más largo y repetitivo.
- Si se quiere modificar alguna de las instrucciones repetidas, se deben hacer tantas modificaciones como cantidad de veces se haya escrito esa instrucción, lo que se convierte en un trabajo muy pesado y muy propenso a cometer errores.

En palabras simples, una función es un conjunto de instrucciones que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente.

Estas nos permiten abstraer la solución a un problema dentro de una instrucción de código y adaptarla para que tenga diversos resultados dependiendo de su uso o aplicación.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Declaración de una *Función.*

En javascript tenemos varias formas para crear una función, conozcamos las funciones **declaradas**, las **expresadas** y las **arrow functions**.

Funciones Declaradas

```
function search() {
  // proceso encapsulado
}
```

Se utiliza la palabra reservada ***function*** seguida del nombre de la función pegada a un juego de paréntesis y un bloque de llaves donde se ingresará el código que realizará la función.

Funciones Expresadas

```
const search = function () {
 // proceso encapsulado
}
```

En este caso la función es **anónima** (sin nombre) y se guarda dentro de una variable o constante tradicional.

La diferencia fundamental entre las funciones declaradas y las funciones expresadas es que estas últimas sólo están disponibles a partir de la inicialización de la variable. Si ejecutamos la variable antes de declararla, nos dará un error.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Invocar funciones.

Una vez hayamos declarado nuestra nueva función, para poder utilizarla debemos **invocarla**. Para ello escribiremos el nombre de la función seguida de un par de paréntesis.

```
function subtract() {
  return 20 - 10;
}

subtract(); // 10
```

Como podemos observar en la imagen anterior, utilizar la sentencia `subtract();` nos permite ejecutar el código interno dentro de la función del mismo nombre. Esto nos **retornará** el resultado el cual podremos guardar dentro de una nueva variable.

Palabra reservada RETURN.

Existen casos donde nuestras funciones deben ejecutar cierto proceso sin necesidad de devolver un resultado, como por ejemplo en la WEB al querer borrar un elemento de nuestro HTML que ya no necesitamos o cuando queremos imprimir un valor en la consola de la terminal, sin embarg o en la mayoría de los casos nuestras funciones abstraen o guardan comportamientos que finalmente retornan un dato que tendrá que ser utilizado en otra parte de nuestro código, como por ejemplo el resultado de la resta de 2 valores.

Para capturar nuevamente ese valor que resulta de la ejecución de la función debemos utilizar la palabra reservada **`return`** antes del valor a devolver.

Ese valor se puede guardar en una variable para ser utilizado más adelante o implementarlo directamente desde la invocación de la función en otra parte del código que así lo permita.

```
function subtract() {
  return 20 - 10;
}

const result = subtract();

// imprimimos el resultado en la terminal
console.log(result);
```

En los casos donde una función no posee la sentencia `return`, se considera que es del tipo `void` o vacío, es decir, que solamente ejecuta el código interno pero no devuelve ningún resultado.

Arrow Functions.

Introducidas al lenguaje a partir de la especificación **ES6** del estándar **EcmaScript**, son otra forma de declarar nuestras funciones y hoy en día juegan un papel fundamental en la forma en la que se escriben programas de Javascript moderno.

Una de las principales ventajas es que nos permiten declarar funciones que ocupen tan solo una línea de código.

Para declarar un arrow function usamos la siguiente sintaxis:

```
const subtract = () => { /* rutina o proceso */ };
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Cómo podemos observar, es similar a la declaración de una función **expresada** solo que prescindimos de la palabra reservada `function` y luego de los paréntesis colocamos una "flecha" compuesta por un signo `=` y un signo `>`. A continuación de ella el uso de llaves es **OPCIONAL** y en caso de **NO** usarlas, la función tendrá un `return` implícito para lo que esté inmediatamente después **en la misma línea**.

```
const subtract = () => 20 - 10; // return implícito
```

Si bien en el ejemplo anterior no fue necesario utilizar la palabra reservada `return`, es importante mencionar que esto solo es posible en los casos donde la función sea de una sola línea, en el caso de las funciones multilínea, agregamos el par de llaves `{}`, quedando de la siguiente manera:

```
const subtract = () => {
  const resultado = 20 - 10;

  return resultado;  // return explícito
};
```

Parámetros y argumentos.

Hasta ahora solo vimos que cada declaración de una función, ya sea declarada, expresada o de flecha llevaba consigo un par de paréntesis vacíos. Estos no están ahí de casualidad ya que su función principal es la de recibir parámetros.

Un parámetro es un nombre que reservamos como una suerte de comodín para indicarle a la función que al momento de ser invocada deberán pasar un valor en esa posición para luego ser utilizado dentro de la misma función.

Cuando invocamos a la función, el valor literal que le pasamos para que ocupe el lugar del **parámetro** es conocido como **argumento**.

Para poner los términos claros:

- Un **parámetro** es una variable listada dentro de los paréntesis en la declaración de función (es un término reservado para el momento de la declaración).

```
// agrego 2 parámetros a la función
const subtract = (numA, numB) => {
  const resultado = numA - numB;
  return resultado;
};
```

- Un **argumento** es el valor que es pasado a la función cuando esta es llamada (es el término para el momento en que se llama).

```
 /* Los valores 20 y 10
 * son los argumentos que toman
 * el lugar de los parámetros numA y numB
 */
subtract(20, 10);
```

Al momento de crear nuestra función, debemos tener claro cuántos parámetros vamos a esperar o necesitar y colocarlos dentro de los paréntesis separados por cada uno por una coma, siendo que no existe una cantidad límite de parámetros permitidos.

Estos nos van a permitir crear funciones “multiuso” es decir, que no operen bajo datos estáticos, sino que el resultado que devuelva la función dependa de los datos que recibe como input, encapsulando un comportamiento idéntico frente a distintos escenarios.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

HOF: Funciones de orden superior

Una función de orden superior, también conocida como **callback**, es una función que se pasa a otra función como un argumento, que luego se invoca dentro de la función externa para completar algún tipo de rutina o acción. Es decir, pasamos una función B por parámetro a una función A, de modo que la función A puede ejecutar esa función B de forma genérica desde su código, y nosotros podemos definirlas desde fuera de dicha función.

```
function addition(a, b) {
  return a + b;
}

function subtract(a, b) {
  return a - b;
}

function calculator(a, b, action) {
  return action(a, b);
}

calculator(20, 10, addition); // 30
calculator(20, 10, subtract); // 10
```

En el ejemplo anterior, creamos una función llamada `calculator` que espera tres parámetros, 2 valores y una acción. Aprovechamos esta lógica para pedirle que ejecute la función `adition` en caso que deseemos sumar ambos valores o `subtract` si lo que buscamos es restarlos. De esta manera, es la función `calculator` la encargada de ejecutar de forma interna la acción recibida por parámetro.

De este comportamiento nace su famoso nombre **callback**, ya que el anidado de funciones con este comportamiento nos permite encadenar la ejecución de una función detrás de otra.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Arrays

Declaración

Los arrays, arreglos o vectores son estructuras de datos utilizadas en la mayoría de los lenguajes de programación como una herramienta ideal para el agrupamiento de valores en forma de lista indexada.

En javascript, un array es una colección de variables o datos que pueden ser todas del mismo tipo o cada una de un tipo de dato diferente diferente. Su utilidad se comprende mejor con un ejemplo sencillo: si una aplicación necesita manejar los días de la semana, se podrían crear siete variables de tipo texto lo cual representaría mayor dificultad a la hora de tratar esa información como un conjunto de datos relacionados.

```
const var1 = 'Lunes';
const var2 = 'Martes';
const var3 = 'Miércoles';
const var4 = 'Jueves';
const var5 = 'Viernes';
const var6 = 'Sábado';
const var7 = 'Domingo';
const week = ['Lunes', 'Martes', 'Miércoles',
'Jueves', 'Viernes', 'Sábado', 'Domingo'];
```

Imaginemos que en lugar de los días de la semana queremos guardar todos los países del mundo, necesitaríamos declarar más de 200 variables diferentes.

Como podemos observar, para declarar un array debemos utilizar la misma sintaxis que para la declaración de variables, salvo que luego del operador de asignación “=” colocaremos un par de corchetes donde alojaremos los datos de nuestro array separados por comas.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Acceso a los arrays

```
const fruits = ['Pera', 'Manzana', 'Frutilla', 'Durazno'];
```

En este ejemplo declaramos un array de 4 elementos, donde si bien nosotros conocemos su extensión en este caso, muchas veces es necesario consultar este dato ya sea porque no lo sabemos o porque fue modificado a través de la ejecución del programa.

Para saber cuántos elementos posee un array debemos acceder a su propiedad **`length`** (al igual que sucede con los strings), de este modo obtendremos la cantidad de elementos que existen en nuestra colección.

```
const fruits = ['Pera', 'Manzana', 'Frutilla', 'Durazno'];

console.log(fruits.length); // 4
```

Otro punto importante es que cada elemento dentro de un array posee un **índice** único. Esté índice va en orden ascendente y **comienza en 0** desde el primer al último elemento del array.

```
 // index          0          1           2           3
const fruits = ['Pera', 'Manzana', 'Frutilla', 'Durazno'];
```

Tal como muestra el ejemplo anterior, si bien este array posee una extensión de 4 elementos, su índice llega hasta el 3 debido a que comienza desde la posición 0. Por esta razón es de suma importancia entender la diferencia entre **`length`** (cantidad) e **`index`** (índice o posición).

Sabiendo esto, ahora podemos consultar los elementos de un array en relación a la posición que ocupan dentro de él colocando entre corchetes el número de la posición que representa ese elemento dentro del array.

```
// index          0          1          2          3
const fruits = ['Pera', 'Manzana', 'Frutilla', 'Durazno'];

console.log(fruits[2]); // frutilla
```

Métodos de Array

Son funciones que poseen los array de forma nativa, con ellas seremos capaces de agregar, quitar, buscar, filtrar, modificar, cortar, y muchas cosas más sobre los elementos internos de nuestros arreglos.

Por ejemplo, mediante el método `.at()` podemos acceder a un elemento del array de igual modo que mediante el uso de corchetes, cambiando la sintaxis de `fruits[2]` por `fruits.at(2)` lo que tendría exáctamente el mismo resultado.

En este apartado conoceremos algunos de los métodos de array más utilizados.

Añadir o eliminar elementos

En este caso tendremos a disposición los métodos ***push***, ***unshift***, ***pop*** y ***shift*** que cabe destacar, son métodos mutables lo que significa que modificarán el array original al estar aplicando el resultado directamente sobre el mismo.

Para **añadir** elementos al inicio o al final del array, utilizaremos **push** y **unshift**:

```
const fruits = ['Pera', 'Manzana', 'Frutilla', 'Durazno'];

fruits.push('Kiwi');
// ['Pera', 'Manzana', 'Frutilla', 'Durazno', 'Kiwi']
fruits.unshift('Kiwi');
// ['Kiwi', 'Pera', 'Manzana', 'Frutilla', 'Durazno']
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Para **eliminar** elementos al inicio o al final del array, utilizaremos **pop** y **shift**:

```
const fruits = ['Pera', 'Manzana', 'Frutilla', 'Durazno'];

fruits.pop(); // ['Pera', 'Manzana', 'Frutilla']
fruits.shift(); // ['Manzana', 'Frutilla', 'Durazno']
```

Unir elementos de un array en una cadena (string)

En este caso utilizamos el método `.join()` que recibe como parámetro el separador de nuestros elementos.

```
const fruits = ['Pera', 'Manzana', 'Frutilla', 'Durazno'];

fruits.join(' - '); //'Pera - Manzana - Frutilla - Durazno'
```

Recorrer los elementos de un array

Nos permite **recorrer** los elementos de un array y ejecutar una acción frente a cada iteración. El método `.forEach()` no devuelve nada y espera que se le pase por parámetro una función de **callback** que se ejecutará por cada elemento del array.

En el siguiente ejemplo, **fruit** toma el valor actual de la iteración y ejecuta la función `(fruit) => console.log(fruit)` para imprimir cada valor en la consola.

```
const fruits = ['Pera', 'Manzana', 'Frutilla', 'Durazno'];

fruits.forEach((fruit) => console.log(fruit));

// Pera
// Manzana
// Frutilla
// Durazno
```

Filtrar elementos en un array

Para ello usaremos el método `.filter()` a quien le debemos pasar una función de **callback** con la condición que deben cumplir los elementos para ser filtrados.

```
const prices = [125, 237, 58, 1920, 418];

prices.filter((price) => price >= 200);
// [237, 1920, 418]
```

Modificar o crear arrays a partir de otros

Hay situaciones en las que tenemos un array y queremos crear nuevos subarrays a partir del original, o simplemente deseamos modificarlo para hacer ciertos cambios, pero de una forma más general y no tener que hacerlo elemento a elemento.

- `.slice(start, end)`: Devuelve los elementos desde la posición **start** hasta **end** (excluido). **Inmutable**
- `.splice(start, size)`: Altera el array, eliminando **size** (cantidad de elementos) desde posición **start**. **Mutable**

- `.copyWithin(pos, start, end)`: Altera el array, modificando desde **pos** y copiando los ítems desde **start** a **end**. **Mutable**
- `.fill(element, start, end)`: rellena el array con **element** desde **start** hasta **end**. **Mutable**

Crear nuevos arrays a partir de una condición

El método `.map()` es un método muy potente y útil para trabajar con arrays, puesto que su objetivo es devolver un nuevo array donde cada uno de sus elementos será lo que devuelva la función callback por cada uno de los elementos del array original.

```
const prices = [125, 237, 58, 1920, 418];

prices.map((price) => price *= 1.21);

// devuelve un array con todos los precios + el 21%
// [151.25, 287.77, 70.18, 2329.2, 505.78]
```

Acumular los valores de una array

El método `.reduce()` se encarga de recorrer todos los elementos del array, e ir acumulando sus valores (o alguna operación diferente) y sumarlo todo, para devolver su resultado final.

```
const prices = [125, 237, 58, 1920, 418];
prices.reduce((total, price) => total + price, 0);
// 2758

/* total toma el 0 como "valor inicial" y en cada iteración
* se le va sumando el price actual hasta recorrer todo el
* array
*/
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Iteradores de array

Si bien es posible recorrer arrays con los ciclos tradicionales como un for, existen una estructura para este fin que simplifica mucho el trabajo a realizar.

Esta es la estructura **for of**, que propone la siguiente sintaxis:

```
const fruits = ['Pera', 'Manzana', 'Frutilla', 'Durazno'];

for (let fruit of fruits) {
  console.log(fruit);
}

// Pera
// Manzana
// Frutilla
// Durazno
```

Template Literals

Cadenas de texto más simples

Los literal strings o template literals son una característica de JavaScript introducida en ES6 que simplifica la creación y manipulación de cadenas de texto. Se escriben utilizando **backticks** (``) en lugar de comillas simples o dobles.

Características principales:

1. **Interpolación de variables**: Permiten insertar valores dinámicamente dentro de una cadena utilizando la sintaxis `${expresión}`.

```
const userName = 'Juan';
const age = 30;
console.log(`Hola, mi nombre es ${userName}
y tengo ${age} años.`);
// Hola, mi nombre es Juan y tengo 30 años.
```

2. Soporte para varias líneas: Se pueden escribir cadenas que abarcan múltiples líneas sin necesidad de usar caracteres especiales como `\n`.

```
const message = `Esta es una cadena
que ocupa varias
líneas.`;

console.log(message);
// Esta es una cadena
// que ocupa varias
// líneas.
```

3. Inclusión de expresiones complejas: Además de variables, se pueden incluir operaciones y funciones directamente.

```
const price = 150;
const tax = 1.21;

console.log(`El precio final es de $${price * tax}.`);
// El precio final es de $181.5.
```

En conclusión, los **template literals** son una herramienta poderosa y flexible que optimiza el manejo de cadenas en JavaScript, especialmente en casos donde se mezclan variables, expresiones y texto.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejercicio Práctico

Calculando con Precisión

Ejercicio 1

Después de demostrar tus primeros pasos con Node.js, Matías y Sabrina te plantean un reto más interesante.

“Imaginá que los precios de los productos del cliente están listos para cargarse en el sistema”, te dicen. Pero hay un detalle: antes de enviarlos, tenés que calcular el IVA del 21% y presentarlos de manera clara.

Tu misión:

1. Creá un array con 10 números que representen los precios de los productos.
2. Utiliza un método de array para calcular el precio con IVA incluido para cada valor.

Ejercicio 2

Ahora que tenés los valores con IVA calculados, es momento de mostrarlos como un desarrollador profesional. Matías te desafía a usar template literals para estructurar tu salida de esta forma:

```
El precio es: ${valor}.- IVA incluido.
```

Imprimí cada precio ajustado en la consola siguiendo este formato. Cada mensaje debe ser claro y profesional, como si estuvieras preparando un reporte para el cliente.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

¿Puedes demostrar tu capacidad para manejar datos y presentar resultados de manera impecable?

¡Este es tu momento!

Materiales y Recursos Adicionales:

MDN Web Docs: Funciones - Guía completa para aprender sobre funciones en JavaScript.

MDN Web Docs: Arrays - Documentación sobre métodos de arrays.

MDN Web Docs: Template Literals - Explicación detallada de los template literals.

Eloquent JavaScript - Un libro interactivo sobre JavaScript que incluye capítulos sobre funciones, arrays y manipulación de cadenas.

Videos introductorios de freeCodeCamp para repasar funciones y arrays.

Preguntas para Reflexionar:

- ¿Cómo puedes determinar qué tipo de función (declarada, expresada o de flecha) es más adecuado para un caso específico?
- ¿Qué ventajas tiene usar métodos como `.map()` y `.filter()` frente a un bucle tradicional para trabajar con arrays?
- ¿Cómo mejorarías tu código para que sea más legible y eficiente al manipular datos con arrays?
- ¿Qué otros casos prácticos se te ocurren para usar template literals en una aplicación real?

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Próximos Pasos:

Clases y Objetos: Introducción a la Programación Orientada a Objetos (POO). Aprenderemos qué son las clases, cómo crear objetos y cómo trabajar con métodos y propiedades.

Módulos y librerías: Aprenderemos todo lo necesario para trabajar con código modular y de terceros.

Manejo de Promesas: Exploraremos cómo manejar procesos asíncronos utilizando promesas y `async/await` en JavaScript.

Buenos Aires Ciudad