«Talento Tech»

Front-End JS

Clase 11

Buenos Aires Ciudad

Clase 11: Programación Modular con Funciones

1. **Introducción a las Funciones**
    - ¿Qué son las funciones en JavaScript?
    - Ventajas de utilizar funciones
    - Ejemplo práctico: uso de funciones en un programa simple
2. **Parámetros de Entrada y Salida en Funciones**
    - Definición de parámetros
    - Tipos de funciones según los parámetros: sin parámetros, con parámetros
    - Ejemplo: Función con y sin parámetros
    - Argumentos: ¿qué son y cómo funcionan?
    - Ejemplos con múltiples argumentos
3. **Alcance de las Variables (Scope)**
4. **Programación Modular vs Funciones**
    - ¿Qué es la programación modular?
    - Diferencias entre el uso de funciones y la programación modular
    - Ventajas de la modularidad en proyectos grandes
    - Ejemplo práctico: desarrollando un sistema modular utilizando funciones
5. **Parámetros en Funciones**
    - ¿Qué son los parámetros en las funciones?
    - ¿Cómo definirlos correctamente?
    - Ejemplo práctico: función con diferentes tipos de parámetros
    - Parámetros opcionales y predeterminados en JavaScript
6. **Funciones Nativas de JavaScript**
    - ¿Qué son las funciones nativas?
    - Ejemplos de funciones nativas útiles en el día a día:
    - Por qué usar funciones nativas: eficiencia y simplicidad
    - Ejemplos prácticos de uso
7. **Comparación entre Funciones y Programación Modular**
    - Diferencias clave
    - Cuándo usar una sobre la otra
8. **Ejercicios Prácticos**

Objetivo de la clase:

En esta clase vas a comprender cómo las funciones permiten estructurar y reutilizar el código de manera más limpia y organizada. Aprenderás a definir funciones, a pasarles parámetros de entrada y a recibir valores de salida, diferenciando además conceptos como argumentos y alcance de variables (scope). También te vas a familiarizar con la idea de programación modular, viendo cómo se relaciona y se complementa con las funciones para construir programas más escalables y fáciles de mantener. Por último, conocerás las funciones nativas que ofrece JavaScript y cuándo conviene aprovecharlas para simplificar tareas cotidianas.

1. Introducción a las Funciones

Las **funciones** son uno de los pilares más importantes en la programación. Son bloques de código que te permiten organizar y reutilizar las instrucciones necesarias para realizar una tarea específica. Esto hace que el código sea más limpio, fácil de mantener y más modular.

Por ejemplo, si tuvieras que escribir varias veces un código para calcular el área de un rectángulo, sería mejor crear una función que lo haga por vos. Cada vez que necesites calcular el área, simplemente invocás la función en lugar de volver a escribir el código.

Ejemplo simple:

```
function calcularArea(largo, ancho) {

    return largo * ancho;

}

console.log(calcularArea(5, 3)); // Devuelve 15
```

🛑**¡Recordá!**
function calcularArea(a, b) {
  return a * b;
}
a y b son parámetros que la función espera recibir

la función hace la multiplicación a * b
return entrega el resultado al lugar donde fue llamada

2. Parámetros de Entrada y Salida en Funciones

Cuando hablamos de funciones, no solo nos referimos a bloques de código que hacen algo, sino que también pueden recibir información y devolver resultados. Acá es donde entran en juego los **parámetros** y los **argumentos**. Entender cómo funcionan es clave para aprovechar todo el potencial de las funciones.

Definición de Parámetros

Los parámetros son las "variables" que las funciones reciben al ser llamadas. Se definen cuando se crea la función y representan los datos que la función necesita para operar. Es como cuando le pedís a alguien que te alcance algo, y ese "algo" sería el parámetro.

```
function saludar(nombre) {

   console.log("Hola " + nombre);}
```

En este ejemplo, `nombre` es el parámetro de la función. Cuando la llamás, tenés que pasarle un **argumento**, que es el valor específico que querés usar:

```
saludar("Juan"); // Esto imprimirá "Hola Juan"
```

Tipos de Funciones Según los Parámetros

Sin Parámetros: Una función que no recibe ningún dato. Simplemente realiza una tarea y no necesita ninguna información externa.

```
function saludar() {

   console.log("Hola a todos");

}

saludar();  // Imprime: "Hola a todos"
```

Con Parámetros: Estas funciones son más flexibles porque pueden trabajar con diferentes datos dependiendo de los argumentos que les pases.

```
function sumar(a, b) {

   console.log(a + b);

}
```

```
sumar(5, 10);  // Imprime: 15
```

En este segundo caso, la función `sumar` tiene dos **parámetros** (`a` y `b`) que, cuando la llamamos, se reemplazan con los valores 5 y 10.

2. Argumentos: ¿Qué son y cómo funciona?

Cuando llamamos a una función y le pasamos datos, esos datos se conocen como **argumentos**. Los argumentos son los valores reales que pasás a los parámetros de la función. Si pensás en los parámetros como un recipiente, los argumentos son lo que llenan ese recipiente.

```
function multiplicar(a, b) {

   return a * b;

}

console.log(multiplicar(3, 4)); // Imprime: 12
```

🛑`return` → devuelve un valor, que podés reutilizar después
`console.log` → solo lo muestra en la consola, pero no lo devuelve

Acá, `3` y `4` son los **argumentos** que reemplazan a `a` y `b` en la función. Al final, el resultado es `3 * 4`, que es 12.

Ejemplos con Múltiples Argumentos

Las funciones pueden recibir varios argumentos, y podés usarlos dentro de la función de la manera que prefieras. A continuación te doy un ejemplo más avanzado:

```
function presentar(nombre, edad, ciudad) {

   console.log(nombre + " tiene " + edad + " años y vive en "
+ ciudad + ".");

}

presentar("Juan", 25, "Buenos Aires");

// Imprime: "Juan tiene 25 años y vive en Buenos Aires."
```

3. Scope Global y Local en Funciones

Uno de los conceptos más importantes a la hora de trabajar con funciones es el **alcance** o **scope**. Este define dónde y cuándo una variable es accesible dentro del código. Hay dos tipos principales: **global** y **local**.

Scope Global

Una variable con **scope global** es aquella que podés usar en cualquier parte del programa, incluso dentro de funciones. Las variables globales se declaran fuera de cualquier función.

```
let nombre = "Pedro";

function saludar() {

  console.log("Hola " + nombre);

}

saludar(); // Accede a la variable global 'nombre' y muestra: "Hola Pedro"
```

🛑**¡Importante!** En este caso, `nombre` es global, porque está definida fuera de la función `saludar`.

Scope Local

Una variable con **scope local** es aquella que solo existe dentro de una función. No se puede acceder a ella desde fuera de esa función.

```
function saludar() {

   let nombre = "Maria";

   console.log("Hola " + nombre);

}

saludar();  // Imprime: "Hola Maria"
```

```
console.log(nombre);  // Error, 'nombre' no está definida
fuera de la función
```

🛑**¡Importante!** Acá, `nombre` está definida dentro de la función `saludar`, por lo que solo es accesible dentro de esa función.

Diferencias entre Scope Global y Local

El scope global puede ser peligroso si no lo manejás bien, porque cualquier función o parte del programa puede modificar una variable global. Por eso, siempre es recomendable *limitar el uso de variables globales y trabajar más con variables locales.*

4. Programación Modular vs. Funciones

En la programación, siempre es importante que el código sea **modular**. Esto significa dividir el código en bloques más pequeños, independientes y reutilizables. Las **funciones** son la base de esta modularidad. A través de ellas, podés organizar tu código de una manera más clara y ordenada, donde cada función se ocupa de una tarea específica.

¿Qué es la Programación Modular?

La programación modular es un enfoque que consiste en dividir el código en módulos o partes pequeñas, cada una con una responsabilidad clara. Esto facilita el mantenimiento y reutilización del código. En JavaScript, las funciones son los módulos básicos.

Ventajas de la Programación Modular

1. **Mantenimiento Simplificado:** si necesitás cambiar algo, solo modificás una función en lugar de todo el programa.
2. **Reutilización:** podés usar las mismas funciones en diferentes partes de tu código sin necesidad de reescribirlas.
3. **Legibilidad:** el código se vuelve más fácil de leer, ya que cada función tiene un propósito claro.

Ejemplo de programación modular:

```
function calcularAreaRectangulo(base, altura) {
```

```
    return base * altura;}

function imprimirResultado(area) {

    console.log("El área del rectángulo es: " + area);

}

let base = 5;

let altura = 10;

let area = calcularAreaRectangulo(base, altura);

imprimirResultado(area);
```

Acá, cada función tiene una tarea específica: una calcula el área y otra imprime el resultado. Este enfoque hace que el código sea mucho más fácil de manejar.

5. Funciones Nativas de JavaScript

Las **funciones nativas** en JavaScript son herramientas predefinidas que el lenguaje te ofrece para realizar tareas comunes y necesarias sin tener que reinventar la rueda. Están listas para ser usadas y facilitan el trabajo de los desarrolladores, ya que te permiten resolver problemas rápidamente y de manera eficiente.

¿Qué son las funciones nativas?

Las funciones nativas son funciones que ya vienen integradas en el lenguaje de JavaScript. Podés usarlas directamente, sin necesidad de definirlas, ya que están incluidas en el entorno de ejecución. Estas funciones están diseñadas para realizar tareas frecuentes como mostrar alertas, manipular números, cadenas, fechas y más.

Estas funciones son parte del core del lenguaje y cubren un amplio rango de operaciones. Desde funciones simples como mostrar un mensaje en pantalla hasta otras más complejas como generar números aleatorios o trabajar con fechas.

Ejemplos de funciones nativas útiles en el día a día:

Veamos algunas funciones nativas que podés usar todos los días mientras desarrollás con JavaScript:

`alert()`: Muestra un cuadro de alerta con un mensaje.

```
alert("¡Bienvenido a mi sitio web!");
```

> **¿Para qué sirve?**
> Es una manera rápida de mostrar un mensaje emergente en la pantalla del usuario. Es ideal para notificaciones o alertas simples.

`parseInt()`: Convierte un string en un número entero.

```
let numero = parseInt("123");

console.log(numero);  // Imprime 123
```

> **¿Para qué sirve?**
> Te permite convertir un texto que contiene números en un número entero que podés usar en cálculos matemáticos.

`Math.random()`: Genera un número aleatorio entre 0 y 1.

```
let aleatorio = Math.random();

console.log(aleatorio);
```

> **¿Para qué sirve?**
> Si necesitás generar un valor aleatorio, como para un juego o para decidir algo de forma azarosa, esta función es muy útil.

`Date()`: Devuelve la fecha y hora actual.

```
let fechaActual = new Date();

console.log(fechaActual);
```

> **¿Para qué sirve?**
> Es fundamental para trabajar con fechas y horas. La podés usar para mostrar la fecha actual, calcular diferencias entre fechas o registrar cuándo ocurrió un evento.

`console.log()`: Imprime un mensaje en la consola del navegador.

```
console.log("Este es un mensaje para la consola");
```

¿Para qué sirve?
Esta es tu mejor amiga cuando estás desarrollando. Te permite ver el estado de tus variables y objetos mientras ejecutás el código, facilitando la depuración y la prueba de tu programa.

`toUpperCase()` y `toLowerCase()`: Convierte un string a mayúsculas o minúsculas.

```
let texto = "Hola Mundo";

console.log(texto.toUpperCase());  // Imprime "HOLA MUNDO"

console.log(texto.toLowerCase());  // Imprime "hola mundo"
```

¿Para qué sirve?
Son muy útiles cuando necesitás estandarizar el texto antes de compararlo o mostrarlo. Por ejemplo, para comparar strings de manera que no importe si están en mayúsculas o minúsculas.

`slice()`: Extrae una parte de un string o array.

```
let frase = "JavaScript es increíble";

let parte = frase.slice(0, 10);

console.log(parte);  // Imprime "JavaScript"
```

¿Para qué sirve?
Te permite extraer una porción de un string o array sin modificar el original. Ideal para trabajar con textos largos y obtener solo la parte que necesitás.

¿Por qué usar funciones nativas?

El uso de funciones nativas tiene varias ventajas clave:

- **Eficiencia**: las funciones nativas están optimizadas en los motores de JavaScript, como V8 (usado en Chrome). Esto significa que son más rápidas y eficientes que escribir tu propia versión de esas funciones.
- **Simplicidad**: usar funciones preexistentes simplifica el código y lo hace más legible. En lugar de tener que reinventar la lógica para cada tarea, simplemente utilizás una función que ya existe y que sabés que funciona correctamente.

- **Menos Errores**: las funciones nativas han sido probadas y validadas por millones de desarrolladores alrededor del mundo. Esto reduce el riesgo de errores o comportamientos inesperados en tu código.

Por ejemplo, ¿para qué escribir una función que convierta un string a mayúsculas cuando ya tenés `toUpperCase()` disponible? Eso es lo lindo de trabajar con un lenguaje tan popular como JavaScript: la comunidad ha resuelto muchos problemas antes que vos.

Ejemplos prácticos de uso

Ahora, veamos cómo combinar varias funciones nativas en un ejemplo práctico . Supongamos que querés pedirle al usuario que ingrese su nombre y luego mostrarle un saludo personalizado en mayúsculas.

```
let nombre = prompt("¿Cuál es tu nombre?");

if (nombre) {

  let saludo = "Hola, " + nombre.toUpperCase() + "!";

  alert(saludo);

  console.log("El usuario se llama " + nombre);}
```

Acá estamos usando `prompt()` para pedir el nombre, `toUpperCase()` para convertirlo a mayúsculas, y `alert()` para mostrar el saludo. Además, con `console.log()` registramos el nombre ingresado en la consola.

Comparación entre Funciones y Programación Modular

Hasta ahora hemos visto cómo trabajar con funciones y las ventajas de usar funciones nativas. Pero, ¿cómo se comparan las funciones con el concepto más amplio de **programación modular**?

Diferencias clave

- **Funciones**: son bloques de código que realizan una tarea específica. En general, las funciones pueden definirse de manera que resuelvan una acción concreta (como sumar dos números o mostrar un mensaje).
- **Programación Modular**: va más allá de las funciones individuales. Implica dividir todo el programa en módulos (que pueden contener varias funciones). Cada módulo es responsable de una parte del comportamiento del programa. Estos módulos son reutilizables y, generalmente, son independientes entre sí.

En resumen: una **función** es una pieza específica de código, mientras que la **programación modular** es una estrategia general para organizar todo el código en partes manejables.

¿Cuándo usar una sobre la otra?

- **Funciones**: usalas cuando necesitás realizar una tarea única y específica que podés encapsular. Es la unidad más pequeña de organización en el código.
- **Programación Modular**: es útil cuando trabajás en proyectos grandes o complejos. Dividir el código en módulos te permite gestionar mejor el crecimiento del proyecto y evitar la duplicación de código.

Ejemplo práctico: Proyecto que combina programación modular y funciones

Vamos a suponer que estás creando una pequeña aplicación que maneja un carrito de compras en una tienda online.

Paso 1: Crear funciones individuales para manejar las tareas.

```
function agregarProducto(producto) {

   console.log(producto + " fue agregado al carrito.");

}

function calcularTotal(productos) {

   let total = productos.reduce((sum, producto) => sum +
producto.precio, 0);

   console.log("El total es: $" + total);

   return total;
```

Paso 2: Modularizar el código. Podrías tener un módulo separado para el carrito de compras, otro para la interfaz de usuario, y otro para el cálculo de precios.

```
// Módulo del carrito de compras
```

```
const carrito = (function() {
    let productos = [];
    return {
        agregar: function(producto) {
            productos.push(producto);
            agregarProducto(producto.nombre);
        },
        total: function() {
            return calcularTotal(productos);
        }
    };
})();
```

Este es un ejemplo básico de cómo podés usar tanto funciones como programación modular para crear una aplicación más compleja, reutilizable y fácil de mantener.

Con esto, terminamos la ampliación del tema de **Funciones Nativas de JavaScript** y la comparación con la **Programación Modular**. ¡Espero que te haya quedado claro y que puedas aplicar estas ideas a tus proyectos!

💭**Storytelling**

¡Funciones y Modulaciones!

Tomás (Desarrollador Senior)

¡Muy buen trabajo con las estructuras de control y bucles! Ahora vamos a dar el siguiente paso para modularizar y organizar nuestro código de forma profesional. Las funciones nos van a permitir separar la lógica de nuestro programa en bloques reutilizables y fáciles de mantener.

Ejercicio práctico #1:

Calculadora Modular con Funciones

Lucía (Product Owner)

El equipo necesita una calculadora que sea clara y fácil de extender. Para eso vamos a dividir su lógica en varias funciones, cada una con una responsabilidad concreta.

Lo que harás:

- Definí cuatro funciones diferentes: una para sumar, otra para restar, otra para multiplicar y otra para dividir.
- Cada función recibirá dos parámetros numéricos y devolverá el resultado de la operación correspondiente.
- En el caso de la función dividir, acordate de verificar que el divisor no sea cero. Si lo es, mostrá un mensaje de error en la consola.
- Probá el correcto funcionamiento creando dos variables numéricas de ejemplo y llamando a las cuatro funciones con ellas.

🛑**¡Importante!** Mostrá los resultados de cada operación en la consola de forma clara y legible usando `console.log()`.

Tips de Tomás:
Recordá la sintaxis de `function` para declarar funciones. Podés cambiar los valores de prueba para verificar distintos resultados. Usá comentarios si te ayuda a organizar el código.

Ejercicio práctico #2:

Generación y visualización de productos con formato JSON

Lucía (Product Owner)

En nuestro próximo sprint queremos preparar un catálogo de productos dinámico. Vamos a practicar la creación de datos estructurados usando objetos, simulando un escenario real de desarrollo.

Lo que vas a hacer:

1. Creá una función llamada `generarProductos()` que no reciba parámetros.
2. Dentro de la función, armá un array llamado `productos` con al menos 5 objetos. Cada producto debe tener:

- `id`: un identificador numérico único
- `name`: nombre del producto
- `description`: descripción breve
- `amount`: precio en formato numérico
- Retorná el array completo desde la función.

🛑**¡Importante!** Fuera de la función, guardá el resultado en una variable y recorré el array con un bucle (`for` o `forEach`) para mostrar en consola cada producto con todos sus datos.

Tips de Tomás:
Repasá la estructura de un objeto JSON. Usá nombres de propiedades claros y coherentes. Probá agregar un producto más si querés practicar

Buenos Aires aprende

Agencia de Habilidades para el Futuro

Buenos Aires Ciudad