La Guía del Programador

Resumen hiperconcentrado de los conocimientos y conceptos más importantes

Variables

Las usamos **para almacenar tipos de datos y valores**. Tipos más comunes: string, number, boolean, object, array.

```
// Usamos let cuando sabemos que la variable puede redefinirse o cambiar su tipo
let suma = 20;

// Usamos const cuando sabemos que la variable no va a ser redefinida. A tener en
cuenta que si es un objeto o un array, sus propiedades pueden ser modificadas,
pero no se puede cambiar el tipo, o sea no puede pasar de objeto a string por
ejemplo

// Object
const objeto = {
  propiedad: "contenido",
};
// Array
const productos = ["Arroz", "Fideos"];
// Boolean
let mayorEdad = true;
// Number
let edad = 18;
// String
let nombre = "Ricardo";
```

Condicionales

Los usamos para hacer nuestros algoritmos más inteligentes. Cuando una condición se cumple, se ejecuta un bloque de código, de lo contrario, puede ejecutarse otro.

```
if (suma > 19) {
  console.log("Suma es mayor que 19.");
} else if (suma == 19) {
  console.log("Suma es igual que 19.");
} else {
  console.log("Suma es menor que 19.");
}
```

Alternativa con switch:

```
let operacion = prompt("¿Qué operación querés hacer? (+ - / *)");
switch (operacion) {
  case "+":
    // Algoritmo para sumar variables
    break;
  case "-":
    // Algoritmo para restar variables
    break;
  case "/":
    // Algoritmo para dividir variables
    break;
  case "*":
    // Algoritmo para multiplicar variables
    break;
  // Si ingresó otra cosa en el prompt, ingresa al default
  default:
    alert("El operador ingresado es inválido.");
}
```

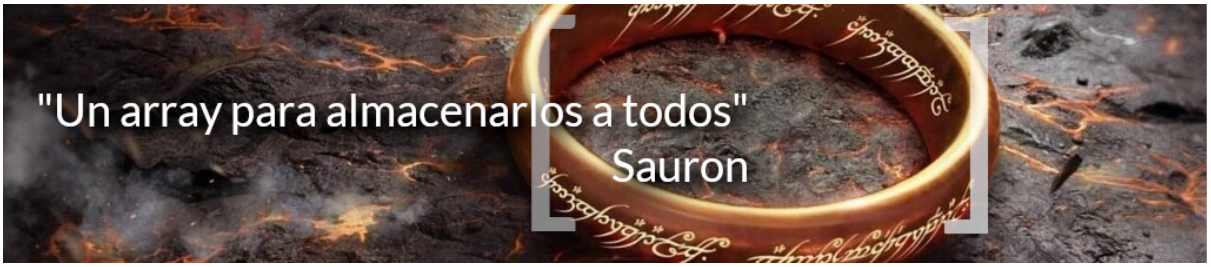
Funciones

Las funciones son extremadamente útiles y necesarias en programación, **nos permiten aislar ciertos algoritmos y porciones de código para poder reutilizarlos en cualquier parte de nuestra aplicación**.

```
function sumar(valor1, valor2) {
  // Me suma los dos valores recibidos como parámetros
  // y los devuelve con un return
  return valor1 + valor2;
}
let resultado1 = sumar(20, 20); // 40
let resultado2 = sumar(10, 10); // 20
```

Arrays

Los arrays son listas de variables, y estas variables pueden ser de cualquier tipo. **Básicamente una variable para almacenar variables**.



Los arrays tienen métodos MUY útiles que nos permiten trabajar con ellos de manera eficiente según el problema a resolver al que nos enfrentemos:

```
const listaVariada = ["Ricardo", "Ana", "Vicente", "María", 20, true, objeto];
// Con find buscamos elementos y lo almacenamos en una variable
let ana = listaVariada.find((elemento) => elemento == "Ana");
// indexOf devuelve el índice del elemento
let indiceRicardo = listaVariada.indexOf("Ricardo");
// Elimina uno o varios elemento elemento del array a partir de índice indicado
listaVariada.splice(indiceRicardo, 1);
// Pobre Ricky, lo volvemos a poner en el array con push
listaVariada.push("Ricardo");
```

Estos son algunos de los más usados, pero hay muchos más.

Ciclos

Los usamos **para recorrer listas (arrays) o repetir bloques de código X cantidad de veces**.

```
// For incremental
for (let i = 1; i < 11; i++) {
  let resultado = tabla * i;
  alert(tabla + " x " + i + " es igual a " + resultado);
}
// For facha
for (const item of mochila) {
  let indiceItem = mochila.indexOf(item);
}
```

Objetos

Los objetos son **una herramienta poderosa que nos permite agrupar muchos datos (propiedades) en una sola variable**, llamada objeto. Por ejemplo un producto:

```
const producto = {
    nombre: "Celular Samsung",
    stock: 10,
    precio: 100
}
```

Métodos

Son funciones internas del objeto.

objeto.metodo()

Constructor
El primer método que se llama automáticamente cuando se crea (instancia) un objeto desde una clase "molde". Es la encargada de recibir los parámetros y definir las propiedades del objeto (con this).

```
class Pokemon {
  constructor(nivel, energia) {
    this.nivel = nivel;
    this.energia = energia;
  }
}
const pikachu = new Pokemon(1, 10);
```

Alternativa para crear un objeto simplificado:

```
const pikachu = {
    nivel: 1,
    energia: 10
}
```

Esta alternativa no se puede usar como molde al no tener una clase, por lo que tampoco tiene método constructor.

Las dos opciones son válidas, solo que en algunos casos una opción puede resultar mejor que otra.

Si tenemos varios tipos de objetos, y estos comparten sus métodos y propiedades (ejemplo clase Pokémon y sus 150 Pokemones) lo ideal es que usemos una clase.

Pero en proyectos más pequeños es probable que con la versión simple nos alcance.

PRO TIP: Está bueno aclarar que internamente para JavaScript prácticamente todo es un objeto, es por eso que incluso hasta las variables string o number tienen métodos.

```
let numero = 20;
let numeroATexto = numero.toString() // de 20 a “20”
```

Y por supuesto los arrays, que tienen métodos muy útiles.

El poder del DOM

El DOM nos brinda **el poder de MANIPULAR los elementos de nuestro HTML de forma dinámica y en tiempo real, sin necesidad de recargar la página**.

```
// Creo un div de la nada (mágeco)
let div = document.createElement("div");
// Le agrego HTML dentro del div
div.innerHTML = "<h2>¡Hola Coder!</h2>";
// Añado el elemento como hijo de body con append, y aparece en el HTML
document.body.append(div);
```

El DOM no solo nos permite crear sino también **nos permite alterar propiedades y los atributos de cualquier elemento existente**.

```
// Agregarle o quitarle clases de CSS a cualquier elemento
const divMensaje = document.querySelector(".mensaje");
divMensaje.classList.add("fondoRojo"); // Agrega clase fondoRojo
divMensaje.classList.remove("fondoAzul"); // Remueve clase fondoAzul
```

```
// Obtener o alterar atributos de cualquier elemento. Suponiendo que tengamos este
div: <div id="23" class=".producto">
const divProducto = document.querySelector(".producto");
const idProducto = divProducto.getAttribute("id"); // 23
// Sugar syntax
const idProducto = divProducto.id; // 23
```

```
// Crear nuestros propios atributos con dataset para pasar información de
JavaScript a HTML y viceversa <div id="23" dataset-stock="14"
dataset-categoria="Alimentos" class=".producto">
const divProducto = document.querySelector(".producto");
const stockProducto = divProducto.stock; // 14
const categoriaProducto = divProducto.categoria; // Alimentos
```

Eventos

Es **la herramienta que tenemos para saber cuándo el usuario está interactuando con nuestro HTML**.

Con el método addEventListener le indicamos a JavaScript que esté atento a que cuando tal evento ocurra dentro de tal elemento, ejecute una función con las instrucciones que nos interese a nosotros.

```
const elemento = document.querySelector("#btnEnviar");
elemento.addEventListener("click", () => {
  // Contenido de la función, este bloque de código se va a ejecutar cuando se
  // dispare el evento "click" dentro del evento
  console.log("Hiciste click en el elemento #btnEnviar");
});
```

Sintaxis alternativa al addEventListener:

```
elemento.onclick = () => {
  console.log("Hiciste click en el elemento #btnEnviar");
};
```

La lista de eventos más usados (en caso de usar addEventListener quitar el “on”):

Evento	Descripción	Elementos para los que está definido
onchange	Deseleccionar un elemento que se ha modificado	<input>, <select>, <textarea>
onclick	Pinchar y soltar el ratón	Todos los elementos
onfocus	Seleccionar un elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pulsar una tecla (sin soltar)	Elementos de formulario y <body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	La página se ha cargado completamente	<body>
onmouseout	El ratón "sale" del elemento (pasa por encima de otro elemento)	Todos los elementos
onmouseover	El ratón "entra" en el elemento (pasa por encima del elemento)	Todos los elementos
onsubmit	Enviar el formulario	<form>

Un método que nos es muy útil a la hora de hacer botones o queremos evitar el comportamiento por defecto del HTML, por ejemplo si tenemos una etiqueta `<a link="http://google.com" id="btnGoogle">` y queremos evitar que cuando hacemos click vaya a la dirección indicada, lo podemos hacer gracias a preventDefault.

```
elemento.addEventListener("click", (event) => {
  event.preventDefault();
  alert("No va a ir a google.com");
});
```

Storage

Gracias al storage tenemos la posibilidad de **almacenar variables y datos de forma PERSISTENTE en el navegador**.

```
// Ej de variable o dato que queremos almacenar
let variable = "Ricky";
// Con el método setItem guardamos datos en el storage. El primer parámetro es la clave (similar a un ID de una base de datos), y el segundo parámetro es el valor, el dato que queremos guardar
localStorage.setItem('miVariable1', variable);
// Con el método getItem obtenemos datos del storage. Con solo pasarle la clave que le asociamos al dato almacenado nos devuelve lo que guardamos
let datoAlmacenado = localStorage.getItem('miVariable1');
// Imprimo en consola el dato almacenado, que se va a imprimir de forma persisten incluso si cierro y abro la ventana
console.log(datoAlmacenado); // Ricky
```

Ahora bien, **si queremos guardar datos más complejos como objetos o arrays, debemos usar el poder de JSON**. Esto es debido a que el storage sólo permite almacenar datos en forma de string.

```
// Array vacío
let arrayProductos = [];
// Le agrego producto en forma de objeto
arrayProductos.push({
    nombre: "Arroz",
    precio: 100,
    cantidad: 1
});
// Interactuando con el storage
// Con el método stringify de JSON lo que hacemos en convertir nuestro array en un string en formato JSON, para poder guardarlo de forma COMPATIBLE con el storage
localStorage.setItem('carrito', JSON.stringify(arrayProductos));
// Y cuando necesitemos volver a utilizar el array, nos va a devolver un string. Un array u objeto en forma de string no nos sirve, por lo que usamos el método parse para convertir el string en su forma original
let arrayStorage = JSON.parse(localStorage.getItem('carrito'));
// Nos devuelve el array original
console.log(arrayStorage);
```

Operadores avanzados: Sugar Syntax

Son **simplemente alternativas a la sintaxis original, pero con menos código** (o más facheras, como nos gusta decir). Si bien no nos hace mejores programadores utilizarlas, sí es importante conocerlas por si el día de mañana nos toca interpretar código ajeno (algo muy probable en nuestras carreras).

Operador ternario

```
let nombre1 = "Ricky"; // Viene del prompt
let edad = 40; // Viene del prompt
// Sintaxis original
if (mayorEdad >= 18) {
  mayorEdad = true;
} else {
  mayorEdad = false;
}
// Alternativa con OPERADOR TERNARIO
let mayorEdad = edad >= 18 ? true : false;
```

Operador lógico OR

```
// Sintaxis original
if (carritoStorage) {
  this.productos = carritoStorage;
} else {
  this.productos = [];
}
// Operador lógico OR
this.productos = carritoStorage || [];
```

Negador lógico

```
let modoOscuro = true;
// Sintaxis original para invertir un booleano
if (modoOscuro) {
  modoOscuro = false;
} else {
  modoOscuro = true;
}
// Negador lógico, nos sirve para INVERTIR booleanos
modoOscuro = !modoOscuro;
```

Variables “falsy”

Es importante entender cuándo JavaScript interpreta cuando una variable es “falsy” o no a la hora de usar condiciones y operadores lógicos. Acá un ejemplo simple que lo demuestra:

```
// Lista de variables Falsy
let variable;
variable = false;
variable = 0;
variable = "";
variable = null;
variable = undefined;
variable = NaN;

if (variable) {
  console.log("La variable es válida (true):", variable);
} else {
  console.log("No entró en el condicional (falsy):", variable);
}
```

Asincronismo y Fetch

El asincronismo en JavaScript tiene un objetivo muy claro: conectarnos con un servidor (ya sea externo como una API o local con un archivo .json por ejemplo) para **obtener o enviar información sin necesidad de recargar la página**. Gracias a las promesas y Fetch lo podemos hacer de manera muy sencilla.

Promesas

Las promesas nos ayudan a manejar las respuestas a las peticiones asíncronas; saber cuando se realizaron satisfactoriamente, obtener el resultado, o cuando salieron mal. Y lo podemos hacer con los siguientes métodos:

```
promesa
  // Si la promesa salió bien, con el método .then recibimos el resultado que nos
interesa
  .then((resultado) => {
    console.log('Conexión exitosa:', resultado);
  })
  // En caso de que ocurriese un error, con en el método .catch capturamos el
error y le avisamos al usuario
  .catch((error) => {
    console.log('Hubo un error:', error);
  });
```

Fetch

Este método nos permite hacer conexiones asíncronas y como resultado nos devuelve una promesa, entonces para manejar las resoluciones de la conexión se hace de manera casi idéntica.

```
fetch("https://pokeapi.co/api/v2/pokemon/" + numeroPokemon)
  .then((response) => response.json()) // Si la respuesta es en formato JSON, debemos agregar un then previo para convertirla en objeto
  // Ahora si, tenemos el resultado en un objeto que recibimos parámetro
  .then((pokemon) => {
    // Mostramos el pokemon en consola
    console.log(pokemon);
  })
  // En caso de introducir un número inexistente o algun caracter erróneo manejamos el error
  .catch((error) => {
    console.log("Ese pokémon no existe.");
  });
```

Fetch con Sugar Syntax

La sintaxis alternativa del Fetch no solo es más fachera sino que nos permite trabajar de forma bloqueante; esto significa que no se va a ejecutar la siguiente línea de código hasta que finalice la conexión, teniendo como resultado un código más simple y más fácil de leer. Pero si tengo que esperar a que finalice la conexión... ¿no estaría perdiendo la habilidad asincrónica? Buena pregunta, y la respuesta es no, porque para usar esta sintaxis fachera debemos declararla dentro de una función asincrónica.

Veamos un ejemplo:

```
// Le agregamos async antes de declarar la función. Esto la vuelve una función asincrónica, lo que significa que se va a ejecutar en segundo plano mientras el resto del código sigue corriendo
async function cargarPokemon(numeroPokemon) {
  // Aunque internamente esta función es bloqueante (solo a nivel local) porque usamos await, que hace que no se ejecute la siguiente línea hasta obtener respuesta de la conexión
  const response = await fetch("https://pokeapi.co/api/v2/pokemon/" + numeroPokemon);
  const pokemon = await response.json();
  console.log(pokemon);
}
```

El Camino del Programador

Cómo enfocarse en aprender la lógica y en adquirir la mentalidad del programador

1. No hace falta entender TODO en profundidad y a la perfección.

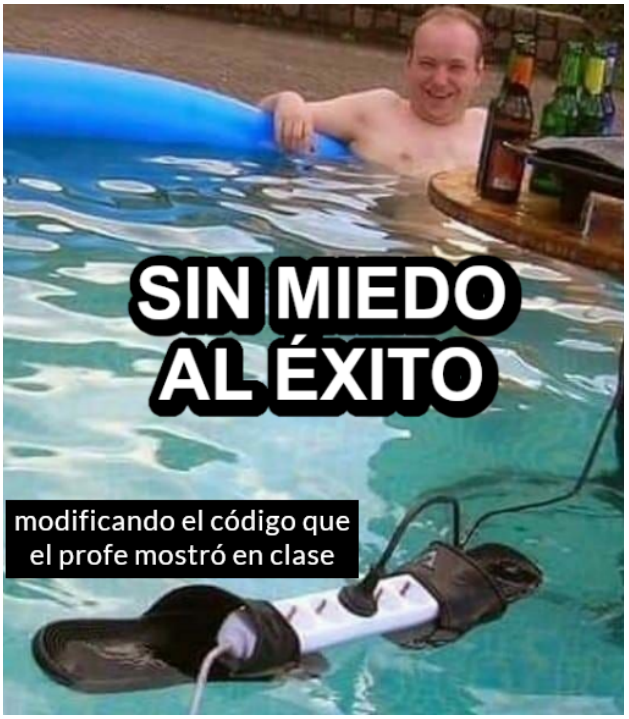
```
// Ejemplo: Tal vez no lo entienda a la perfección, pero vi en clase que con
// este método puedo buscar un elemento del array y cuando lo encuentra me lo
// devuelve en una variable, por ejemplo cuando necesito saber si un producto
// está en el carrito
enCarrito(nuevoProducto) {
 return productos.find((producto) => (producto.nombre == nuevoProducto.nombre));
}
```

Saber su utilidad es más que suficiente para empezar a utilizarlo cuando lo necesite. Con la práctica y el tiempo ya lo voy a entender en profundidad.

2. Se aprende jugando y picando código, no mirando.

Copiar código y modificarlo, cambiarle los nombres a las variables, probar cosas diferentes, trabajar y sortear los errores, debuggear con console.log(), ahí es cuando más se aprende.

No hay que tenerle miedo a romper el código o a los errores, porque es ahí donde se aprende de verdad.



3. Este curso es solo el comienzo de sus exitosas carreras, por lo que es REQUISITO y parte del camino como programadores aprender a usar las poderosas herramientas que tienen a su disposición.

- **Google**: Las problemas y las dudas que tienen, alguien ya las se las hizo antes que ustedes y las respuestas están a un tipeo y a un click de distancia.
- **YouTube**: Mucho material y muy bien explicado, desde las cosas más simples como variables y funciones a cosas más complejas como el manejo del DOM y los eventos.
- **ChatGPT**: TREMENDA ventaja que tenemos a favor de las generaciones de programadores anteriores. Un asistente personal que puede explicarnos en detalle lo que se nos ocurra; desdé el cómo funciona hasta el cómo hacerlo.

Estas son herramientas poderosísimas de las que se nutren no solo los programadores en formación como ustedes, sino también los programadores profesionales de alto rango (seniors).

Si logran adquirir esta mentalidad, no van a tener límites.

Tarea:

Aprender a aprender de Freddy Vega