UNIVERSIDAD CATÓLICA DE SANTA MARÍA

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

SESIÓN N° 04:

Fundamentos de Javascript

I

OBJETIVOS

- Analizar la estructura y sintaxis del código JavaScript, diferenciando entre conceptos clave como variables, funciones y objetos.
- Evaluar la eficiencia y adecuación de diferentes enfoques de JavaScript para resolver desafíos comunes de programación.
- Implementar funcionalidades centrales de JavaScript, incluyendo la manipulación del DOM, el manejo de eventos y las operaciones asíncronas, para crear aplicaciones web interactivas.
- Depurar código JavaScript de manera efectiva, utilizando las herramientas de desarrollo del navegador y técnicas de manejo de errores para identificar y resolver problemas.
- Valorar la importancia de escribir código JavaScript limpio, bien estructurado y mantenible.
- Apreciar el papel de JavaScript en la mejora de la experiencia del usuario y en la creación de contenido web dinámico.

II

TEMAS A TRATAR

- Introducción a javascript
- ¿Qué es JavaScript?
- Formas de declarar javascript
- Recursos del lenguaje
- Funciones
- Objetos en javascript
- Arrays
- Objetos globales
- Resumen

III

MARCO TEORICO

1. INTRODUCCIÓN A JAVASCRIPT

Si ya estás familiarizado con HTML y CSS, sabes que HTML crea la estructura o esqueleto de una página web, mientras que CSS se encarga de su presentación visual. JavaScript, el tercer pilar de la web, añade **dinamismo, funcionalidad y lógica** a tus creaciones. Permite que las páginas web respondan a las acciones del usuario, ejecuten cálculos, se comuniquen con servidores y mucho más.

¿Por qué JavaScript es esencial?

- Aplicaciones interactivas: Desde actualizaciones de contenido en tiempo real hasta mapas interactivos y animaciones, JavaScript impulsa la mayoría de las características que hacen que la web sea atractiva y funcional. (Freeman & Robson, 2014).
- Resolución de problemas: JavaScript te permite abordar problemas del mundo real mediante la programación. Puedes tomar datos del usuario, procesarlos y generar respuestas personalizadas, como en aplicaciones de mensajería o calculadoras en línea. (Haverbeke, 2018)

- Manipulación del DOM: JavaScript interactúa con el Document Object Model (DOM), una representación estructurada de la página web. Esto te permite seleccionar elementos específicos, modificar su contenido, estilo y comportamiento en tiempo real. (Duckett, 2014)

2. ¿QUÉ ES JAVASCRIPT?

JavaScript es un lenguaje de programación multipropósito que se ejecuta principalmente en el navegador web. Es:

- Interpretado: No requiere compilación previa, lo que facilita su uso y agiliza el desarrollo. (Flanagan, 2020)
- Versátil: Puede utilizarse tanto en el frontend (interacción con el usuario) como en el backend (lógica del servidor) gracias a tecnologías como Node.js. (Cantelon, Harter, Holowaychuk, & Rajlich, 2018)
- Esencial para la web moderna: La mayoría de los sitios web utilizan JavaScript para ofrecer una experiencia rica e interactiva a los usuarios. (Zakas, 2011)

3. FORMAS DE DECLARAR JAVASCRIPT

Existen tres formas principales de incluir JavaScript en una página web:

1. En línea: Directamente dentro de un elemento HTML, utilizando un atributo de evento.

```
HTML
<button onclick="alert('¡Hola Mundo!')">¡Haz clic aquí!</button>
```

   - Explicación: El código JavaScript (alert('¡Hola Mundo!')) se ejecuta cuando ocurre el evento onclick (clic del ratón) en el botón.

2. En las etiquetas <script>: Dentro de la sección <head> o <body> del HTML.

```
HTML
<script>
    // Tu código JavaScript aquí
    console.log("¡Hola desde el script!");
</script>
```

   - Explicación: El código JavaScript se coloca entre las etiquetas <script>. Se ejecuta cuando el navegador carga la página o en el momento en que se encuentra la etiqueta <script>.

3. En un archivo .js vinculado: La forma más recomendada, ya que promueve la organización y la reutilización del código.
   - Archivo script.js:

```
JavaScript
console.log("¡Hola desde un archivo externo!");
```

   - Dentro del HTML:

```
HTML
<script src="script.js"></script>
```

   - Explicación: El código JavaScript se almacena en un archivo externo (script.js) y se vincula al HTML mediante el atributo src de la etiqueta <script>.

4. RECURSOS DEL LENGUAJE

A. VARIABLES

- Definición: Espacios de almacenamiento en memoria para guardar datos que pueden cambiar durante la ejecución del programa. Se declaran con let o var.

```
JavaScript
```

```
let nombre = "María";
var edad = 30;
```

- let: Introduce un ámbito de bloque, lo que significa que la variable solo es accesible dentro del bloque de código donde se declara (entre llaves {}). (Flanagan, 2020)
- var: Tiene un ámbito de función, lo que significa que la variable es accesible dentro de toda la función donde se declara. (Zakas, 2011)
- Constantes: Similares a las variables, pero su valor no puede ser modificado después de su declaración. Se declaran con const.

```
JavaScript
const PI = 3.14159;
```

B. CONSOLELOG

Muestra un mensaje en la consola web (o del intérprete JavaScript).

Parámetros:

- **obj1 ... objN**: Una lista de objetos JavaScript para mostrar. Las representaciones en texto de cada uno de los objetos se agregan y muestran juntas (al final una tras otra), en el orden listado.
- **Msg**: Un texto (mensaje) conteniendo cero o más sustituciones de cadenas (sustituciones de strings).
- **subst1 ... substN**: Objetos JavaScript con la sustitución a reemplazar dentro del texto (msg). Esto brinda control adicional en el formato de salida del texto.

C. SCOPE

El **ámbito** es el contexto actual de ejecución en el que los valores y expresiones son "visibles" o pueden ser referenciados. Si una variable o expresión no está en el ámbito actual, no estará disponible para su uso. Los ámbitos también pueden estar estratificados en una jerarquía, de modo que los ámbitos hijos tienen acceso a los ámbitos padres, pero no a la inversa.

JavaScript tiene los siguientes tipos de **ámbitos**:

- Ámbito global: El ámbito por defecto para todo el código que se ejecuta en modo script.
- Ámbito del módulo: El ámbito del código que se ejecuta en modo módulo.
- Ámbito de la función: El ámbito creado con una función.

Además, las variables declaradas con **let** o **const** pueden pertenecer a un ámbito adicional:

Ámbito de bloque: El ámbito creado con un par de llaves (un bloque).

Una función crea un **ámbito**, de manera que (por ejemplo) una variable definida exclusivamente dentro de la función no puede ser accedida desde fuera de la función o dentro de otras funciones. Por ejemplo, lo siguiente no es válido:

```
function exampleFunction() {
    var x = "declarada dentro de la función";
    // x solo se puede utilizar en exampleFunction
    console.log("funcion interna");
    console.log(x);
}
console.log(x);  // error
```

Sin embargo, el siguiente código es válido debido a que la variable se declara fuera de la función, lo que la hace global:

```
var x = "función externa declarada";
exampleFunction();

function exampleFunction() {
```

```
    console.log("funcion interna");
    console.log(x);
}
console.log("funcion externa");
console.log(x);
```

El ámbito de bloque es sólo para las declaraciones **let** y **const**, pero no las declaraciones **var**.

```
{
  var x = 1;
}
console.log(x); // 1
{
  const x = 1;
}
console.log(x); // ReferenceError: x is not defined
```

D. TIPOS DE DATOS

Siete tipos de datos que son primitivos:

- Booleano. true y false.
- null. Una palabra clave especial que denota un valor nulo. (Dado que JavaScript distingue entre mayúsculas y minúsculas, null no es lo mismo que Null, NULL o cualquier otra variante).
- undefined. Una propiedad de alto nivel cuyo valor no está definido.
- Number. Un número entero o un número con coma flotante. Por ejemplo: 42 o 3.14159.
- BigInt. Un número entero con precisión arbitraria. Por ejemplo: 9007199254740992n.
- String. Una secuencia de caracteres que representan un valor de texto. Por ejemplo: "Hola"
- Symbol (nuevo en ECMAScript 2015). Un tipo de dato cuyas instancias son únicas e inmutables
- Object

Aunque estos tipos de datos son una cantidad relativamente pequeña, permiten realizar funciones útiles con tus aplicaciones. Los otros elementos fundamentales en el lenguaje son los **Objetos** y las **funciones**. Puedes pensar en objetos como contenedores con nombre para los valores, y las funciones como procedimientos que puedes programar en tu aplicación.

Hasta ahora hemos analizado los dos primeros, pero hay otros.

a) NÚMEROS

Puedes almacenar números en variables, ya sea números enteros como 30 (también llamados enteros — "integer") o números decimales como 2.456 (también llamados números flotantes o de coma flotante — "number"). No es necesario declarar el tipo de las variables en JavaScript, a diferencia de otros lenguajes de programación. Cuando le das a una variable un valor numérico, no incluye comillas:

```
let myAge = 17;
```

b) CADENAS DE CARACTERES (STRINGS)

Las **strings** (cadenas) son piezas de texto. Cuando le das a una variable un valor de cadena, debes encerrarlo entre comillas simples o dobles; de lo contrario, JavaScript intenta interpretarlo como otro nombre de variable.

```
let dolphinGoodbye = 'Hasta luego y gracias por todos los peces';
```

c) BOOLEANOS

Los booleanos son valores verdadero/falso — pueden tener dos valores, true o false. Estos, generalmente se utilizan para probar una condición, después de lo cual se ejecuta el código según corresponda. Así, por ejemplo, un caso simple sería:

```
let iAmAlive = true;
```

Mientras que en realidad se usaría más así:

```
let test = 6 < 3;
```

Aquí se está usando el operador "menor que" (<) para probar si 6 es menor que 3. Como era de esperar, devuelve false, ¡porque 6 no es menor que 3! Aprenderás mucho más sobre estos operadores más adelante en el curso.

d) ARREGLOS

Un arreglo es un objeto único que contiene múltiples valores encerrados entre corchetes y separados por comas. Intenta ingresar las siguientes líneas en tu consola:

```
let myNameArray = ['Chris', 'Bob', 'Jim'];
let myNumberArray = [10, 15, 40];
```

Una vez que se definen estos arreglos, puedes acceder a cada valor por su ubicación dentro del arreglo. Prueba estas líneas:

```
myNameArray[0]; // debería devolver 'Chris'
myNumberArray[2]; // debe devolver 40
```

Los corchetes especifican un valor de índice correspondiente a la posición del valor que deseas devolver. Posiblemente hayas notado que los arreglos en JavaScript tienen índice cero: el primer elemento está en el índice 0.

e) OBJETOS

En programación, un objeto es una estructura de código que modela un objeto de la vida real. Puedes tener un objeto simple que represente una caja y contenga información sobre su ancho, largo y alto, o podrías tener un objeto que represente a una persona y contenga datos sobre su nombre, estatura, peso, qué idioma habla, cómo saludarlo, y más.

Intenta ingresar la siguiente línea en tu consola:

```
let dog = { name : 'Spot', breed : 'Dalmatian' };
```

Para recuperar la información almacenada en el objeto, puedes utilizar la siguiente sintaxis:

```
dog.name
```

f) TIPADO DINÁMICO

JavaScript es un "lenguaje tipado dinámicamente", lo cual significa que, a diferencia de otros lenguajes, no es necesario especificar qué tipo de datos contendrá una variable (números, cadenas, arreglos, etc.).

Por ejemplo, si declaras una variable y le das un valor entre comillas, el navegador trata a la variable como una cadena (string):

```
let myString = 'Hello';
```

Incluso si el valor contiene números, sigue siendo una cadena, así que ten cuidado:

```
let myNumber = '500'; // Vaya, esto sigue siendo una cadena
typeof myNumber;
myNumber = 500; // mucho mejor — ahora este es un número
typeof myNumber;
```

Intenta ingresar las cuatro líneas anteriores en tu consola una por una y ve cuáles son los resultados. Notarás que estamos usando un operador especial llamado typeof — esto devuelve el tipo de datos de la variable que escribes después. La primera vez que se llama, debe devolver string, ya que en ese punto la variable myNumber contiene una cadena, '500'. Échale un vistazo y ve qué devuelve la segunda vez que lo llamas.

g) CONSTANTES EN JAVASCRIPT

Muchos lenguajes de programación tienen el concepto de una **constante** — un valor que, una vez declarado no se puede cambiar. Hay muchas razones por las que querrías hacer esto, desde la seguridad (si un script de un tercero cambia dichos valores, podría causar problemas) hasta la depuración y la comprensión del código (es más difícil cambiar accidentalmente valores que no se deben cambiar y estropear cosas claras).

En los primeros días de JavaScript, las constantes no existían. En JavaScript moderno, tenemos la palabra clave **const**, que nos permite almacenar valores que nunca se pueden cambiar:

```
const daysInWeek = 7;
const hoursInDay = 24;
```

const funciona exactamente de la misma manera que **let**, excepto que a **const** no le puedes dar un nuevo valor. En el siguiente ejemplo, la segunda línea arrojará un error:

```
const daysInWeek = 7;
daysInWeek = 8;
```

E. CAMBIAR TIPO DE DATOS

JavaScript es un lenguaje tipado dinámicamente. Esto significa que no tienes que especificar el tipo de dato de una variable cuando la declaras. También significa que los tipos de datos se convierten automáticamente según sea necesario durante la ejecución del script.

Así, por ejemplo, puedes definir una variable de la siguiente manera:

```
var answer = 42;
```

Y luego, puedes asignarle una cadena a esa misma variable, por ejemplo:

```
answer = 'Gracias por todo el pescado...';
```

Debido a que JavaScript se tipifica dinámicamente, esta asignación no genera un mensaje de error.

5. FUNCIONES

Las funciones son uno de los bloques de construcción fundamentales en JavaScript. Una función en JavaScript es similar a un procedimiento — un conjunto de instrucciones que realiza una tarea o calcula un valor, pero para que un procedimiento califique como función, debe tomar alguna entrada y devolver una salida donde hay alguna relación obvia entre la entrada y la salida. Para usar una función, debes definirla en algún lugar del ámbito desde el que deseas llamarla.

A. DEFINIR FUNCIONES

a) DECLARACIÓN DE FUNCIÓN

Una definición de función (también denominada declaración de función o expresión de función) consta de la palabra clave **function**, seguida de:

- El nombre de la función.
- Una lista de parámetros de la función, entre paréntesis y separados por comas.
- Las declaraciones de JavaScript que definen la función, encerradas entre llaves, { ... }.

Por ejemplo, el siguiente código define una función simple llamada square ("cuadrado"):

```
function square(number) {
  return number * number;
}
```

La función square toma un **parámetro**, llamado number. La función consta de una declaración que dice devuelva el parámetro de la función (es decir, number) multiplicado por sí mismo. La instrucción return especifica el valor devuelto por la función:

```
return number * number;
```

Los parámetros primitivos (como un number) se pasan a las funciones por valor; el valor se pasa a la función, pero si la función cambia el valor del parámetro, este cambio no se refleja globalmente ni en la función que llama.

Si pasas un objeto (es decir, un valor no primitivo, como Array o un objeto definido por el usuario) como parámetro y la función cambia las propiedades del objeto, ese cambio es visible fuera de la función, como se muestra en el siguiente ejemplo:

```
function myFunc(theObject) {
  theObject.make = 'Toyota';
}
var mycar = { make: 'Honda', model: 'Accord', year: 1998 };
var x, y;
x = mycar.make; // x obtiene el valor "Honda"
myFunc(mycar);
y = mycar.make; // y obtiene el valor "Toyota"
                // (la propiedad make fue cambiada por la función)
```

b) EXPRESIONES FUNCTION

Si bien la declaración de función anterior sintácticamente es una declaración, las funciones también se pueden crear mediante una expresión **function**.

Esta función puede ser **anónima**; no tiene por qué tener un nombre. Por ejemplo, la función square se podría haber definido como:

```
const square = function(number) { return number * number }
```

```
var x = square(4) // x obtiene el valor 16
```

Sin embargo, puedes proporcionar un nombre con una expresión **function**. Proporcionar un nombre permite que la función se refiera a sí misma y también facilita la identificación de la función en el seguimiento de la pila de un depurador:

```
const factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1) }
console.log(factorial(3))
```

Las expresiones **function** son convenientes cuando se pasa una función como argumento a otra función. El siguiente ejemplo muestra una función **map** que debería recibir una función como primer argumento y un arreglo como segundo argumento.

```
function map(f, a) {
  let result = []; // Crea un nuevo arreglo
  let i; // Declara una variable
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
  return result;
}
```

En el siguiente código, la función recibe una función definida por una expresión de función y la ejecuta por cada elemento del arreglo recibido como segundo argumento.

```
function map(f, a) {
  let result = []; // Crea un nuevo arreglo
  let i; // Declara una variable
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
  return result;
}
const f = function(x) {
   return x * x * x;
}
let numbers = [0, 1, 2, 5, 10];
let cube = map(f,numbers);
console.log(cube);
```

La función devuelve: [0, 1, 8, 125, 1000].

En JavaScript, una función se puede definir en función de una condición. Por ejemplo, la siguiente definición de función define **myFunc** solo si **num** es igual a 0:

```
var myFunc;
if (num === 0) {
  myFunc = function(theObject) {
    theObject.make = 'Toyota';
  }
}
```

Además de definir funciones como se describe aquí, también puedes usar el constructor **Function** para crear funciones a partir de una cadena en tiempo de ejecución, muy al estilo de eval().

Un **método** es una **función** que es propiedad de un **objeto**.

c) LA EXPRESIÓN DE FUNCIÓN FLECHA (=>)

Una expresión de función flecha tiene una sintaxis más corta y su léxico se une a este valor (ver

arrow functions para más detalles):

```
([param] [, param]) => { instrucciones }
param => expresión
param
```

param: El nombre de un argumento. Si no hay argumentos se tiene que indicar con (). Para un único argumento no son necesarios los paréntesis. (como foo **=>** 1)

instrucciones o expresión: múltiples instrucciones deben ser encerradas entre llaves. Una única expresión no necesita llaves. La expresión es, así mismo, el valor de retorno implícito de esa función.

B. LLAMAR FUNCIONES

Definir una función no la ejecuta. Definirla simplemente nombra la función y especifica qué hacer cuando se llama a la función.

Llamar a la función en realidad lleva a cabo las acciones especificadas con los parámetros indicados. Por ejemplo, si defines la función square, podrías llamarla de la siguiente manera:

```
square(5);
```

La declaración anterior llama a la función con un argumento de 5. La función ejecuta sus declaraciones y devuelve el valor 25.

Las funciones deben estar dentro del ámbito cuando se llaman, pero la declaración de la función se puede elevar (cuando aparece debajo de la llamada en el código), como en este ejemplo:

```
console.log(square(5));
/* ... */
function square(n) { return n * n }
```

El ámbito de una función es la función en la que se declara (o el programa completo, si se declara en el nivel superior).

Nota: Esto solo trabaja cuando se define la función usando la sintaxis anterior (es decir, function funcName() {}). El siguiente código no trabajará.

Esto significa que la **elevación de función** solo trabaja con declaraciones de función, no con expresiones de función.

```
console.log(square)    // square se eleva con un valor inicial undefined.
console.log(square(5)) // Error de tipo no detectado: square no es una función
const square = function(n) {
  return n * n;
}
```

Los argumentos de una función no se limitan a cadenas y números. Puedes pasar objetos completos a una función. La **función show_props()** (definida en Trabajar con objetos es un ejemplo de una función que toma un objeto como argumento.

Una función se puede llamar a sí misma. Por ejemplo, aquí hay una función que calcula factoriales de forma recursiva:

```
function factorial(n) {
  if ((n === 0) || (n === 1))
    return 1;
  else
```

```
    return (n * factorial(n - 1));
}
```

Luego, podrías calcular los factoriales de 1 a 5 de la siguiente manera:

```
var a, b, c, d, e;
a = factorial(1); // a obtiene el valor 1
b = factorial(2); // b obtiene el valor 2
c = factorial(3); // c obtiene el valor 6
d = factorial(4); // d obtiene el valor 24
e = factorial(5); // e obtiene el valor 120
```

Hay otras formas de llamar funciones. A menudo hay casos en los que una función se tiene que llamar dinámicamente, o el número de argumentos de una función varía, o en los que el contexto de la llamada a la función se tiene que establecer en un determinado objeto específico en tiempo de ejecución.

Resulta que las funciones en sí mismas son **objetos** y, a su vez, estos **objetos** tienen métodos. (Consulta el objeto **Function**. Uno de estos, el método apply(), se puede utilizar para lograr este objetivo.

C. ÁMBITO DE FUNCTION

No se puede acceder a las variables definidas dentro de una función desde cualquier lugar fuera de la función, porque la variable se define solo en el ámbito de la función. Sin embargo, una función puede acceder a todas las variables y funciones definidas dentro del ámbito en el que está definida.

En otras palabras, una función definida en el ámbito global puede acceder a todas las variables definidas en el ámbito global. Una función definida dentro de otra función también puede acceder a todas las variables definidas en su función principal y a cualquier otra variable a la que tenga acceso la función principal.

```
// Las siguientes variables se definen en el ámbito global
var num1 = 20,
    num2 = 3,
    name = 'Chamahk';
// Esta función está definida en el ámbito global
function multiply() {
  return num1 * num2;
}
multiply(); // Devuelve 60
// Un ejemplo de función anidada
function getScore() {
  var num1 = 2,
      num2 = 3;
  function add() {
    return name + ' anotó ' + (num1 + num2);
  }
  return add();
}
getScore(); // Devuelve "Chamahk anotó 5"
```

D. FUNCIONES ANIDADAS Y CIERRES

Puedes anidar una función dentro de otra función. La función anidada (interna) es privada de su función contenedora (externa).

También forma un **cierre**. Un **cierre** es una expresión (comúnmente, una función) que puede tener variables libres junto con un entorno que une esas variables (que "cierra" la expresión).

Dado que una función anidada es un **cierre**, significa que una función anidada puede "heredar" los argumentos y variables de su función contenedora. En otras palabras, la función interna contiene el ámbito de la función externa.

Para resumir:

Solo se puede acceder a la función interna desde declaraciones en la función externa.

La función interna forma un **cierre**: la función interna puede usar los argumentos y variables de la función externa, mientras que la función externa no puede usar los argumentos y variables de la función interna.

El siguiente ejemplo muestra funciones anidadas:

```
function addSquares(a, b) {
  function square(x) {
    return x * x;
  }
  return square(a) + square(b);
}
a = addSquares(2, 3); // devuelve 13
b = addSquares(3, 4); // devuelve 25
c = addSquares(4, 5); // devuelve 41
```

Dado que la función interna forma un cierre, puedes llamar a la función externa y especificar argumentos tanto para la función externa como para la interna:

```
function outside(x) {
  function inside(y) {
    return x + y;
  }
  return inside;
}
fn_inside = outside(3);  // Piensa en ello como: dame una función que agregue 3 a
                         // lo que sea que le des eso
result = fn_inside(5);   // devuelve 8
result1 = outside(3)(5); // devuelve 8
```

Preservación de variables

Observa cómo se conserva x cuando se devuelve inside. Un cierre debe conservar los argumentos y variables en todos los ámbitos a los que hace referencia. Dado que cada llamada proporciona argumentos potencialmente diferentes, se crea un nuevo cierre para cada llamada a outside. La memoria se puede liberar solo cuando el inside devuelto ya no es accesible.

Esto no es diferente de almacenar referencias en otros objetos, pero a menudo es menos obvio porque uno no establece las referencias directamente y no las puede inspeccionar.

Funciones multianidadas

Las funciones se pueden anidar de forma múltiple. Por ejemplo:

- Una función (A) contiene una función (B), que a su vez contiene una función (C).
- Ambas funciones B y C forman cierres aquí. Por tanto, B puede acceder a A y C puede acceder a B.
- Además, dado que C puede acceder a B que puede acceder a A, C también puede acceder a A.

Por tanto, los cierres pueden contener múltiples ámbitos; contienen de forma recursiva el ámbito de las funciones que la contienen. Esto se llama encadenamiento de alcance. (La razón por la que se llama "encadenamiento" se explica más adelante).

Considera el siguiente ejemplo:

```
function A(x) {
```

```
  function B(y) {
    function C(z) {
      console.log(x + y + z);
    }
    C(3);
  }
  B(2);
}
A(1); // registra 6 (1 + 2 + 3)
```

En este ejemplo, **C** accede a **y** de **B** y a **x** de **A**.

Esto se puede hacer porque:

- B forma un cierre que incluye a A (es decir, B puede acceder a los argumentos y variables de A).
- C forma un cierre que incluye a B.
- Debido a que el cierre de B incluye a A, el cierre de C incluye a A, C puede acceder a los argumentos y variables de B y de A. En otras palabras, C encadena los ámbitos de B y A, en ese orden.

Sin embargo, lo contrario no es cierto. **A** no puede acceder a **C**, porque **A** no puede acceder a ningún argumento o variable de **B**, del que **C** es una variable. Por lo tanto, **C** permanece privado solo para **B**.

a) CONFLICTOS DE NOMBRES

Cuando dos argumentos o variables en el ámbito de un cierre tienen el mismo nombre, hay un conflicto de nombres. Tiene más prioridad el ámbito anidado. Entonces, el ámbito más interno tiene la mayor prioridad, mientras que el ámbito más externo tiene la más baja. Esta es la cadena de ámbito. El primero de la cadena es el ámbito más interno y el último es el ámbito más externo. Considera lo siguiente:

```
function outside() {
  var x = 5;
  function inside(x) {
    return x * 3.5;
  }
  return inside;
}
outside()(10); // devuelve 20 en lugar de 10
```

El conflicto de nombre ocurre en la declaración `return x` y está entre el parámetro `x` de inside y la variable `x` de outside. La cadena de ámbito aquí es {inside, outside, objeto global}. Por lo tanto, `x` de inside tiene precedencia sobre `x` de outside y 20 (x) de inside se devuelve en lugar de 10 (x de outside).

b) CIERRES

Los **cierres** son una de las características más poderosas de JavaScript. JavaScript permite el anidamiento de funciones y otorga a la función interna acceso completo a todas las variables y funciones definidas dentro de la función externa (y todas las demás variables y funciones a las que la función externa tiene acceso).

Sin embargo, la función externa no tiene acceso a las variables y funciones definidas dentro de la función interna. Esto proporciona una especie de encapsulación para las variables de la función interna.

Además, dado que la función interna tiene acceso a el ámbito de la función externa, las variables y funciones definidas en la función externa vivirán más que la duración de la ejecución de la función externa, si la función interna logra sobrevivir más allá de la vida de la función externa. Se crea un cierre cuando la función interna de alguna manera se pone a disposición de cualquier ámbito fuera de la función externa.

```
var pet = function(name) {  // La función externa define una variable llamada "name"
  var getName = function() {
    return name;             // La función interna tiene acceso a la variable
                             // "name" de la función externa
  }
  return getName; // Devuelve la función interna, exponiéndola así a ámbitos
externos
}
myPet = pet('Vivie');
myPet();                     // Devuelve "Vivie"
```

Puede ser mucho más complejo que el código anterior. Se puede devolver un objeto que contiene métodos para manipular las variables internas de la función externa.

```
var createPet = function(name) {
  var sex;

  return {
    setName: function(newName) {
      name = newName;
    },

    getName: function() {
      return name;
    },

    getSex: function() {
      return sex;
    },

    setSex: function(newSex) {
      if(typeof newSex === 'string' && (newSex.toLowerCase() === 'male' ||
        newSex.toLowerCase() === 'female')) {
        sex = newSex;
      }
    }
  }
}

var pet = createPet('Vivie');
pet.getName();                  // Vivie
pet.setName('Oliver');
pet.setSex('male');
pet.getSex();                   // male
pet.getName();                  // Oliver
```

En el código anterior, la variable **name** de la función externa es accesible para las funciones internas, y no hay otra forma de acceder a las variables internas excepto a través de las funciones internas. Las variables internas de las funciones internas actúan como almacenes seguros para los argumentos y variables externos. Contienen datos "persistentes" y "encapsulados" para que trabajen las funciones internas. Las funciones ni siquiera tienen que estar asignadas a una variable o tener un nombre.

```
var getCode = (function() {
  var apiCode = '0]Eal(eh&2';  // Un código que no queremos que los externos puedan
                               // modificar...
  return function() {
    return apiCode;
  };
})();
```

```
getCode();    // Devuelve el apiCode
```

Precaución: Hay una serie de trampas para tener en cuenta al usar cierres:

Si una función encerrada define una variable con el mismo nombre que una variable en el ámbito externo, entonces no hay forma de hacer referencia a la variable en el ámbito externo nuevamente. (La variable de ámbito interno "anula" la externa, hasta que el programa sale del ámbito interno).

```
var createPet = function(name) {  // La función externa define una variable llamada
                                  // "name".
  return {
    setName: function(name) {     // La función envolvente también define una
                                  // variable llamada "name".
      name = name;                // ¿Cómo accedemos al "name" definido por la
                                  // función externa?
    }
  }
}
```

c) USAR EL OBJETO ARGUMENTS

El arguments de una función se mantiene en un objeto similar a un arreglo. Dentro de una función, puedes abordar los argumentos que se le pasan de la siguiente manera:

```
arguments[i]
```

Donde **i** es el número ordinal del argumento, comenzando en **0**. Entonces, el primer argumento que se pasa a una función sería **arguments[0]**. El número total de argumentos se indica mediante **arguments.length**.

Usando el objeto **arguments**, puedes llamar a una función con más argumentos de los que formalmente declara aceptar. Esto suele ser útil si no sabes de antemano cuántos argumentos se pasarán a la función. Puedes usar **arguments.length** para determinar el número de argumentos que realmente se pasan a la función, y luego acceder a cada argumento usando el objeto **arguments**.

Por ejemplo, considera una función que concatena varias cadenas. El único argumento formal para la función es una cadena que especifica los caracteres que separan los elementos a concatenar. La función se define de la siguiente manera:

```
function myConcat(separator) {
   var result = ''; // inicia list
   var i;
   // itera a través de arguments
   for (i = 1; i < arguments.length; i++) {
      result += arguments[i] + separator;
   }
   return result;
}
```

Puedes pasar cualquier número de argumentos a esta función, y concatena cada argumento en una "lista" de cadenas:

```
// devuelve "red, orange, blue, "
myConcat(', ', 'red', 'orange', 'blue');
// devuelve "elephant; giraffe; lion; cheetah"
myConcat('; ', 'elephant', 'giraffe', 'lion', 'cheetah');
// devuelve "sage. basil. oregano. pepper. perejil. "
myConcat('. ', 'salvia', 'albahaca', 'orégano', 'pimienta', 'perejil');
```

Nota: La variable **arguments** es "similar a un arreglo", pero no es un arreglo. Es similar a un **arreglo** en el sentido de que tiene un índice numerado y una propiedad **length**. Sin embargo, no posee todos los métodos de manipulación de arreglos.

d) PARÁMETROS DE FUNCIÓN

A partir de ECMAScript 2015, hay dos nuevos tipos de parámetros: parámetros predeterminados y parámetros resto.

d.1) PARÁMETROS PREDETERMINADOS

En JavaScript, los parámetros de las funciones están predeterminados en **undefined**. Sin embargo, en algunas situaciones puede resultar útil establecer un valor predeterminado diferente. Esto es exactamente lo que hacen los parámetros predeterminados.

Sin parámetros predeterminados (preECMAScript 2015)

En el pasado, la estrategia general para establecer valores predeterminados era probar los valores de los parámetros en el cuerpo de la función y asignar un valor si eran **undefined**.

En el siguiente ejemplo, si no se proporciona ningún valor para **b**, su valor sería **undefined** al evaluar **a * b**, y una llamada a **multiply** normalmente habría devuelto **NaN**. Sin embargo, esto se evita con la segunda línea de este ejemplo:

```
function multiply(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;

  return a * b;
}
multiply(5); // 5
```

Con parámetros predeterminados (posECMAScript 2015)

Con parámetros predeterminados, ya no es necesaria una verificación manual en el cuerpo de la función. Simplemente puedes poner 1 como valor predeterminado para b en el encabezado de la función:

```
function multiply(a, b = 1) {
  return a * b;
}

multiply(5); // 5
```

Para obtener más detalles, consulta parámetros predeterminados en la referencia.

d.2) PARÁMETROS REST

La sintaxis del parámetro **rest** nos permite representar un número indefinido de argumentos como un arreglo.

En el siguiente ejemplo, la función **multiply** usa parámetros **rest** para recopilar argumentos desde el segundo hasta el final. Luego, la función los multiplica por el primer argumento.

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map(x => multiplier * x);
}
var arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

e) FUNCIONES FLECHA

Una expresión de **función flecha** (anteriormente, y ahora conocida incorrectamente como función de flecha gruesa) tiene una sintaxis más corta en comparación con las expresiones de función y no tiene su propio **this**, **arguments**, **super** o **new.target**. Las funciones flecha siempre son anónimas.

Dos factores influyeron en la introducción de las funciones flecha: funciones más cortas y no vinculantes de **this**.

e.1) FUNCIONES MÁS CORTAS

En algunos patrones funcionales, las funciones más cortas son bienvenidas. Compara:

```
var a = [
  'Hidrógeno',
  'Helio',
  'Litio',
  'Berilio'
];

var a2 = a.map(function(s) { return s.length; });

console.log(a2); // logs [8, 6, 7, 9]
var a3 = a.map(s => s.length);
console.log(a3); // logs [8, 6, 7, 9]
```

e.2) SIN THIS SEPARADO

Hasta las **funciones flecha**, cada nueva función definía su propio valor **this** (un nuevo objeto en el caso de un constructor, indefinido en llamadas a funciones en modo estricto, el objeto base si la función se llama como un "método de objeto", etc.). Esto resultó ser poco menos que ideal con un estilo de programación orientado a objetos.

```
function Person() {
  // El constructor Person() define `this` como él mismo.
  this.age = 0;

  setInterval(function growUp() {
    // En modo no estricto, la función growUp() define `this`
    // como el objeto global, que es diferente del `this`
    // definido por el constructor Person().
    this.age++;
  }, 1000);
}

var p = new Person();
```

En ECMAScript 3/5, este problema se solucionó asignando el valor en **this** a una variable que se podría cerrar.

```
function Person() {
  var self = this; // Algunos eligen `that` en lugar de` self`.
                   // Elige uno y se congruente.
  self.age = 0;

  setInterval(function growUp() {
    // La retrollamada se refiere a la variable `self` de la cual
    // el valor es el objeto esperado.
    self.age++;
  }, 1000);
```

```
}
```

Alternativamente, podrías crear una función vinculada para que el valor **this** adecuado se pasara a la función **growUp**().

Una función flecha no tiene su propio **this** se utiliza el valor de **this** del contexto de ejecución adjunto. Por lo tanto, en el siguiente código, **this** dentro de la función que se pasa a **setInterval** tiene el mismo valor que **this** en la función adjunta:

```
function Person() {
  this.age = 0;

  setInterval(() => {
    this.age++; // |this| propiamente se refiere al objeto person
  }, 1000);
}
var p = new Person();
```

E. CIUDADANO DE PRIMERA CLASE

Un lenguaje de programación se dice que tiene Funciones de primera clase cuando las funciones en ese lenguaje son tratadas como cualquier otra variable. Por ejemplo, en ese lenguaje, una función puede ser pasada como argumento a otras funciones, puede ser retornada por otra función y puede ser asignada a una variable.

Ejemplo: Asignar función a una variable

```
const foo = function() {
   console.log("foobar");
}
// Invocación usando una variable
foo();
```

Asignamos una **Función Anónima** a una **Variable**, la cual utilizamos para invocar la función añadiendo paréntesis () al final.

Nota: Aunque la función no sea anónima (función nombrada), se puede utilizar la variable para invocarla. Nombrar las funciones puede ser útil cuando estamos depurando el código. Pero no afectará como invocamos a la función.

Ejemplo: Pasar la función como argumento

```
function diHola() {
   return "Hola ";
}
function saludar(saludo, nombre) {
  console.log(saludo() + nombre);
}
// Pasamos `diHola` como argumento de la función `saludar`
saludar(diHola, "JavaScript!");
```

Pasamos nuestra función **diHola()** como argumento de la función **saludar()**, esto explica como tratamos la función como un valor.

Nota: Una función que pasamos como argumento a otra función, se llama **Callback function**. **diHola** es una función **Callback**.

Ejemplo: Devolver una función

```
function diHola() {
```

```
   return function() {
      console.log("¡Hola!");
   }
}
```

En este ejemplo; Necesitamos devolver una función desde otra función - Podemos devolver una función porque JavaScript trata la función como un **value**.

Nota: Una función que devuelve una función se llama **Higher-Order Function**.

Volviendo al ejemplo; Ahora, necesitamos invocar la función **diHola** y su Función Anónima devuelta. Para ello, tenemos dos maneras:

1- Usando una variable

```
const diHola = function() {
   return function() {
      console.log("¡Hola!");
   }
}
const miFuncion = diHola();
miFuncion();
```

De esta manera, devolverá el mensaje ¡Hola!.

Nota: Debes usar otra variable para que devuelva el mensaje. Si invocas **diHola** directamente, devolverá la función en si misma sin invocar a la función devuelta.

2- Usando paréntesis doble

```
function diHola() {
   return function() {
      console.log("¡Hola!");
   }
}
diHola()();
```

Usamos paréntesis doble ()() para invocar también a la función retornada.

F. CLOSURE

Una **clausura** o **closure** es una función que guarda referencias del estado adyacente (ámbito léxico). En otras palabras, una clausura permite acceder al ámbito de una función exterior desde una función interior. En JavaScript, las clausuras se crean cada vez que una función es creada.

a) ÁMBITO LÉXICO

Consideremos el siguiente ejemplo:

```
function iniciar() {
  var nombre = "Mozilla";  // La variable nombre es una variable local creada por
                           // iniciar.
  function mostrarNombre() {  // La función mostrarNombre es una función interna,
                              // una clausura.
    alert(nombre);  // Usa una variable declarada en la función externa.
  }
  mostrarNombre();
}
iniciar();
```

La función **iniciar**() crea una variable local llamada **nombre** y una función interna llamada **mostrarNombre**(). Por ser una función interna, esta última solo está disponible dentro del cuerpo de **iniciar**(). Notemos a su vez que **mostrarNombre**() no tiene ninguna variable propia; pero, dado que las funciones internas tienen acceso a las variables de las funciones externas, **mostrarNombre**() puede acceder a la variable nombre declarada en la función **iniciar**().

Ejecuta el código y observa que la sentencia **alert**(), dentro de **mostrarNombre**(), muestra con éxito el valor de la variable nombre, la cual fue declarada en la función externa. Este es un ejemplo de ámbito léxico, el cual describe cómo un analizador sintáctico resuelve los nombres de las variables cuando hay funciones anidadas. La palabra léxico hace referencia al hecho de que el ámbito léxico se basa en el lugar donde una variable fue declarada para determinar dónde esta variable estará disponible. Las funciones anidadas tienen acceso a las variables declaradas en su ámbito exterior.

G. AMBITO DE UNA FUNCIÓN

Las funciones internas no pueden ser llamadas desde un ámbito global, eso quiere decir que para llamar a una función interna tenemos que estar dentro del ámbito donde fue creada. En cambio, desde una función interna se puede llegar al ámbito externo recibiendo las llamadas de las funciones tal como mostraremos en el siguiente código:

```
//Ámbito de una función
function dos() {
    console.log("Funcion dos")
    function tres() {
        console.log("Funcion tres")
        uno()
    }
    tres()
}
function uno() {
    console.log("Funcion uno")
    tres() // causa error
}
dos()
uno()
```

H. OPERADORES

Un operador es básicamente un símbolo matemático que puede actuar sobre dos valores (o variables) y producir un resultado. En la tabla de abajo aparecen los operadores más simples, con algunos ejemplos para probarlos en la consola del navegador.

Operador	Explicación	Símbolo (s)	Ejemplo
Suma/concatena	Se usa para sumar dos números, o juntar dos cadenas en una.	+	6 + 9; "Hola " + "mundo!";
Resta, multiplicación, división	Estos hacen lo que esperarías que hicieran en las matemáticas básicas.	-, *, /	9 - 3; 8 * 2; // La multiplicación en JS es un asterisco 9 / 3;
Operador de asignación	Los has visto anteriormente: asigna un valor a una variable.	=	let miVariable = 'Bob';
identidad/igualdad	Comprueba si dos valores son iguales entre sí, y devuelve un valor de true/false (booleano).	===	let miVariable = 3; miVariable === 4;
Negación, distinto (no igual)	En ocasiones utilizado con el operador de identidad, la negación es en JS el equivalente al operador	!, !==	La expresión básica es true, pero la comparación devuelve false porque lo

	lógico NOT — cambia true por false y viceversa.		hemos negado:
			let miVariable = 3;
			!miVariable === 3;
			Aquí estamos comprobando "miVariable NO es igual a 3". Esto devuelve false, porque miVariable ES igual a 3.
			
			let miVariable = 3;
			miVariable !== 3;

I. MANEJO DE EXCEPCIONES

Puedes lanzar excepciones usando la sentencia **throw** y manejarlas usando las sentencias **try...catch**.

- sentencia throw
- sentencia try...catch

a) TIPOS DE EXCEPCIÓN

En JavaScript se puede lanzar casi cualquier objeto. Sin embargo, no todos los objetos lanzados son iguales. Aunque es común lanzar números o cadenas como errores, a menudo es más efectivo utilizar uno de los tipos de excepción creados específicamente para este propósito:

- Las excepciones de ECMAScript
- DOMException y DOMError

b) SENTENCIA THROW

Utilice la sentencia **throw** para lanzar una excepción. Una sentencia **throw** especifica el valor a lanzar:

```
Throw expresión;
```

Puede lanzar cualquier expresión, no sólo expresiones de un tipo específico. El siguiente código lanza varias excepciones de diferentes tipos:

```
throw 'Error2';   // String type
throw 42;         // Number type
throw true;       // Boolean type
throw {toString() { return "I'm an object!"; } };
```

c) TRY...CATCH

La sentencia **try** consiste en un bloque **try** que contiene una o más sentencias. Las llaves {} se deben utilizar siempre, incluso para un bloque de una sola sentencia. Al menos un bloque **catch** o un bloque **finally** debe estar presente. Esto nos da tres formas posibles para la sentencia try:

- try...catch
- try...finally
- try...catch...finally

Un bloque **catch** contiene sentencias que especifican que hacer si una excepción es lanzada en el bloque **try**. Si cualquier sentencia dentro del bloque **try** (o en una función llamada desde dentro del bloque **try**) lanza una excepción, el control cambia inmediatamente al bloque **catch**. Si no se lanza ninguna excepción en el bloque **try**, el bloque **catch** se omite.

El bloque **finally** se ejecuta después del bloque **try** y el/los bloque(s) **catch** ya hayan finalizado su ejecución. Éste bloque siempre se ejecuta, independientemente de si una excepción fue lanzada o capturada.

Puede anidar una o más sentencias **try**. Si una sentencia **try** interna no tiene un bloque **catch**, se ejecuta el bloque **catch** de la sentencia **try** que la encierra.

Usted también puede usar la declaración **try** para manejar excepciones de JavaScript.

d) BLOQUE CATCH INCONDICIONAL

Cuando solo se utiliza un bloque **catch**, el bloque **catch** es ejecutado cuando cualquier excepción es lanzada. Por ejemplo, cuando la excepción ocurre en el siguiente código, el control se transfiere a la cláusula **catch**.

```
try {
   throw "myException"; // genera una excepción
}
catch (e) {
   // sentencias para manejar cualquier excepción
   logMyErrors(e); // pasa el objeto de la excepción al manejador de errores
}
```

El bloque **catch** especifica un identificador (**e** en el ejemplo anterior) que contiene el valor de la excepción. Este valor está solo disponible en el **scope** del bloque **catch**.

e) BLOQUES CATCH CONDICIONALES

Tambien se pueden crear "bloques **catch** condicionales", combinando bloques **try...catch** con estructuras **if...else if...else** como estas:

```
try {
    myroutine();  // puede lanzar tres tipos de excepciones
} catch (e) {
    if (e instanceof TypeError) {
        // sentencias para manejar excepciones TypeError
    } else if (e instanceof RangeError) {
        // sentencias para manejar excepciones RangeError
    } else if (e instanceof EvalError) {
        // sentencias para manejar excepciones EvalError
    } else {
       // sentencias para manejar cualquier excepción no especificada
       logMyErrors(e); // pasa el objeto de la excepción al manejador de errores
}
```

6. OBJETOS EN JAVASCRIPT

JavaScript está completamente construido en base a clases y objetos muchos elementos son en realidad objetos como las funciones y los arrays, veremos ahora algunos de los más importantes.

A. OBJETOS

Los **objetos** en JavaScript, como en tantos otros lenguajes de programación, se pueden comparar con objetos de la vida real. El concepto de **Objetos** en JavaScript se puede entender con objetos tangibles de la vida real.

En JavaScript, un objeto es una entidad independiente con propiedades y tipos. Compáralo con una taza, por ejemplo. Una taza es un objeto con propiedades. Una taza tiene un color, un diseño, un peso, un material del que está hecha, etc. Del mismo modo, los objetos de JavaScript pueden

tener propiedades que definan sus características.

a) OBJETOS Y PROPIEDADES

Un **objeto** de JavaScript tiene propiedades asociadas a él. Una **propiedad** de un **objeto** se puede explicar como una variable adjunta al **objeto**. Las propiedades de un **objeto** básicamente son lo mismo que las variables comunes de JavaScript, excepto por el nexo con el objeto. Las propiedades de un **objeto** definen las características del **objeto**. Accedes a las propiedades de un objeto con una simple notación de puntos:

```
objectName.propertyName
```

Como todas las **variables** de JavaScript, tanto el nombre del **objeto** (que puede ser una variable normal) como el nombre de la propiedad son sensibles a mayúsculas y minúsculas. Puedes definir propiedades asignándoles un valor. Por ejemplo, vamos a crear un objeto llamado **myCar** y le vamos a asignar propiedades denominadas **make**, **model**, y **year** de la siguiente manera:

```
var myCar = new Object();
myCar.make = 'Ford';
myCar.model = 'Mustang';
myCar.year = 1969;
```

El ejemplo anterior también se podría escribir usando un iniciador de objeto, que es una lista delimitada por comas de cero o más pares de nombres de propiedad y valores asociados de un objeto, encerrados entre llaves ({ }):

```
var myCar = {
    make: 'Ford',
    model: 'Mustang',
    year: 1969
};
```

Las propiedades no asignadas de un **objeto** son **undefined** (yno **null**).

```
myCar.color; // undefined
```

También puedes acceder o establecer las propiedades de los objetos en JavaScript mediante la notación de corchetes [ ]. Los objetos, a veces son llamados arreglos asociativos, debido a que cada propiedad está asociada con un valor de cadena que se puede utilizar para acceder a ella. Por lo tanto, por ejemplo, puedes acceder a las propiedades del objeto **myCar** de la siguiente manera:

```
myCar['make']  = 'Ford';
myCar['model'] = 'Mustang';
myCar['year']  = 1969;
```

El nombre de la propiedad de un objeto puede ser cualquier cadena válida de JavaScript, o cualquier cosa que se pueda convertir en una cadena, incluyendo una cadena vacía. Sin embargo, cualquier nombre de propiedad que no sea un identificador válido de JavaScript (por ejemplo, el nombre de alguna propiedad que tenga un espacio o un guión, o comience con un número) solo se puede acceder utilizando la notación de corchetes. Esta notación es muy útil también cuando los nombres de propiedades son determinados dinámicamente (cuando el nombre de la propiedad no se determina hasta el tiempo de ejecución). Ejemplos de esto se muestran a continuación:

```
// Se crean y asignan cuatro variables de una sola vez, separadas por comas
```

```
var myObj = new Object(),
    str = 'myString',
    rand = Math.random(),
    obj = new Object();

myObj.type                  = 'Sintaxis de puntos';
myObj['fecha de creación'] = 'Cadena con espacios';
myObj[str]                  = 'Valor de cadena';
myObj[rand]                 = 'Número aleatorio';
myObj[obj]                  = 'Object';
myObj['']                   = 'Incluso una cadena vacía';

console.log(myObj);
```

Por favor, ten en cuenta que todas las claves con notación en corchetes se convierten a cadenas a menos que estas sean símbolos, ya que los nombres de las propiedades (claves) en JavaScript pueden solo pueden ser cadenas o símbolos (en algún momento, los nombres privados también serán agregados a medida que progrese la propuesta de los campos de clase, pero no las usarás con el formato [ ]). Por ejemplo, en el código anterior, cuando la clave **obj** se añadió a **myObj**, JavaScript llamará al método **obj.toString**(), y usará la cadena resultante de esta llamada como la nueva clave.

También puedes acceder a las propiedades mediante el uso de un valor de cadena que se almacena en una variable:

```
var propertyName = 'make';
myCar[propertyName] = 'Ford';

propertyName = 'model';
myCar[propertyName] = 'Mustang';
```

Puedes usar la notación de corchetes con **for...in** para iterar sobre todas las propiedades enumerables de un objeto. Para ilustrar cómo funciona esto, la siguiente función muestra las propiedades del objeto cuando pasas el objeto y el nombre del objeto como argumentos a la función:

```
function showProps(obj, objName) {
  var result = ``;
  for (var i in obj) {
    // obj.hasOwnProperty() se usa para filtrar propiedades de la cadena de
                           // prototipos del objeto
    if (obj.hasOwnProperty(i)) {
      result += `${objName}.${i} = ${obj[i]}\n`;
    }
  }
  return result;
}
```

Por lo tanto, la llamada a la función **showProps(myCar, "myCar")** devolverá lo siguiente:

```
myCar.make = Ford
myCar.model = Mustang
myCar.year = 1969
```

7. ARRAYS

Son grupos o conjuntos de datos identificados mediante un solo nombre o identificador, haciendo uso de un índice para diferenciar la posición en la que se almacena cada dato.

El objeto **Array** de JavaScript es un objeto global que es usado en la construcción de **arrays**, que son objetos tipo lista de alto nivel.

Los **arrays** son objetos similares a una lista cuyo prototipo proporciona métodos para efectuar operaciones de recorrido y de mutación. Tanto la longitud como el tipo de los elementos de un array son variables. Dado que la longitud de un array puede cambiar en cualquier momento, y los datos se pueden almacenar en ubicaciones no contiguas, no hay garantía de que los arrays de JavaScript sean densos; esto depende de cómo el programador elija usarlos.

A. CREAR UN ARRAY

```
let frutas = ["Manzana", "Banana"]

console.log(frutas.length)
// 2
```

a) ACCEDER A UN ELEMENTO DE ARRAY MEDIANTE SU ÍNDICE

```
let primero = frutas[0]
// Manzana

let ultimo = frutas[frutas.length - 1]
// Banana
```

b) RECORRER UN ARRAY

```
frutas.forEach(function(elemento, indice, array) {
    console.log(elemento, indice);
})
// Manzana 0
// Banana 1
```

c) AÑADIR UN ELEMENTO AL FINAL DE UN ARRAY

```
let nuevaLongitud = frutas.push('Naranja') // Añade "Naranja" al final
// ["Manzana", "Banana", "Naranja"]
```

d) ELIMINAR EL ÚLTIMO ELEMENTO DE UN ARRAY

```
let ultimo = frutas.pop() // Elimina "Naranja" del final
// ["Manzana", "Banana"]
```

e) AÑADIR UN ELEMENTO AL PRINCIPIO DE UN ARRAY

```
let nuevaLongitud = frutas.unshift('Fresa') // Añade "Fresa" al inicio
// ["Fresa" ,"Manzana", "Banana"]
```

f) ELIMINAR EL PRIMER ELEMENTO DE UN ARRAY

```
let primero = frutas.shift() // Elimina "Fresa" del inicio
// ["Manzana", "Banana"]
```

g) ENCONTRAR EL ÍNDICE DE UN ELEMENTO DEL ARRAY

```
frutas.push('Fresa')
// ["Manzana", "Banana", "Fresa"]

let pos = frutas.indexOf('Banana') // (pos) es la posición para abreviar
// 1
```

B. ITERADORES

Son los mecanismos bajo los cuales vamos a recorrer un array de una manera estructurada y programada:

```
//for
let numeros = new Array(10,14,25,6,7,8,9)
for(let index=0; index <numeros.length;index++)
{
    console.log("Index ",index, "valor: ",numeros[index])
}

//foreach
let numeros = new Array(10,14,25,6,7,8,9)
let index=0
numeros.forEach(element => {
    console.log("Index ",index++, "valor: ",element)
});

//while
let numeros = new Array(10, 14, 25, 6, 7, 8, 9)
let index = 0
while (true) {
    if (index == numeros.length)
        break
    console.log("Index ", index, "valor: ", numeros[index]);
    index++;
}
//do..while
let numeros = new Array(10, 14, 25, 6, 7, 8, 9)
let index = 0
do{
    if (index == numeros.length)
        break
    console.log("Index ", index, "valor: ", numeros[index]);
    index++;
}while(true)
```

8. OBJETOS GLOBALES

El término "objetos globales" (u objetos incorporados estándar) aquí no debe confundirse con el

objeto global. Aquí, los objetos globales se refieren a objetos en el ámbito global. Se puede acceder al objeto global en sí usando el operador this en el ámbito global (pero solo si no se usa el modo estricto ECMAScript 5, en ese caso devuelve undefined). De hecho, el alcance global consiste en las propiedades del objeto global, incluidas las propiedades heredadas, si las hay.

```
const fruits = ['Apple', 'Banana'];

console.log(fruits.length); // 2
console.log(fruits[0]);     // "Apple"
```

9. RESUMEN

La guía de práctica se enfoca en los **Fundamentos de JavaScript**, un lenguaje de programación clave en el desarrollo de aplicaciones web dinámicas e interactivas. El documento proporciona un marco teórico sólido que abarca desde la introducción a JavaScript hasta conceptos más avanzados como la manipulación del DOM, el manejo de eventos y la programación asíncrona.

La guía comienza estableciendo los **objetivos** de aprendizaje, que se alinean con los niveles superiores de la Taxonomía de Bloom, abarcando los dominios cognitivo, psicomotor y afectivo. Los estudiantes no solo adquirirán conocimientos teóricos sobre JavaScript, sino que también desarrollarán habilidades prácticas para implementar y depurar código, valorando al mismo tiempo la importancia de escribir código limpio y mantenible.

El **marco teórico** detalla los conceptos fundamentales de JavaScript. Se explican los diferentes **tipos de datos** (números, cadenas, booleanos, arrays y objetos), así como los **operadores** que permiten realizar operaciones sobre ellos. Se exploran las **sentencias condicionales** (if/else, switch) y los **bucles** (for, while, do...while) que controlan el flujo de ejecución del programa.

Las **funciones** se presentan como bloques de código reutilizables que encapsulan tareas específicas. Se describen las diferentes formas de declarar funciones, el uso de parámetros y valores de retorno, así como el concepto de **ámbito** (scope) y **clausuras** (closures). El **manejo de excepciones** se aborda mediante el uso de bloques try...catch...finally, que permiten controlar errores en tiempo de ejecución de manera elegante.

La guía también profundiza en **objetos** y **arrays**, estructuras de datos esenciales en JavaScript. Se explica cómo crear, manipular e iterar sobre objetos y arrays, resaltando su importancia en la organización y gestión de datos.

La **manipulación del DOM** se introduce como la capacidad de JavaScript para interactuar con la estructura de una página web, permitiendo modificar elementos y contenido de forma dinámica. El **manejo de eventos** se explora como el mecanismo para responder a las acciones del usuario, como clics o pulsaciones de teclas.

Finalmente, se ofrece una breve introducción a la **programación asíncrona** en JavaScript, incluyendo conceptos como callbacks, promesas y async/await, que son cruciales para manejar operaciones que toman tiempo, como la comunicación con servidores.

IV

(La práctica tiene una duración de 4 horas) ACTIVIDADES

1. EXPERIENCIA DE PRÁCTICA N° 01: FORMAS DE DECLARAR CÓDIGO JAVASCRIPT

a) DECLARACIÓN DE CÓDIGO EN LINEA

1. Agregar un documento html con una estructura definida con el nombre de index.html, la página web debe tener el número de estructura una semántica suficiente para la aplicación de código JavaScript.
2. Identificar los elementos del documento html que recibirán definiciones de código JavaScript.

3. Agregar las instrucciones JavaScript necesarias para entregar dinamismo y funcionalidad a los elementos html elegidos.
4. Grabar el documento html y visualizarlo en el navegador web.
5. Probar la funcionalidad agregada a los elementos html a través de código JavaScript.
6. corregir cualquier error que se presente y volver a ejecutar la visualización de la página web y la funcionalidad en los controles elegidos mediante JavaScript.

b) DECLARACIÓN DE CÓDIGO EN ETIQUETAS SCRIPT

1. Agregar un documento html con una estructura definida con el nombre de index.html, la página web debe tener el número de estructura una semántica suficiente para la aplicación de código JavaScript.
2. Identificar los elementos del documento html que recibirán definiciones de código JavaScript.
3. Agregar dentro de las etiquetas script el código html que afectará los elementos html elegidos.
4. Agregar las instrucciones JavaScript necesarias para entregar dinamismo y funcionalidad a los elementos html elegidos.
5. Grabar el documento html y visualizarlo en el navegador web.
6. Probar la funcionalidad agregada a los elementos html a través de código JavaScript.
7. corregir cualquier error que se presente y volver a ejecutar la visualización de la página web y la funcionalidad en los controles elegidos mediante JavaScript.

c) DECLARACIÓN DE CÓDIGO EN ARCHIVO VINCULADO

1. Agregar un documento html con una estructura definida con el nombre de index.html, la página web debe tener el número de estructura una semántica suficiente para la aplicación de código JavaScript.
2. Identificar los elementos del documento html que recibirán definiciones de código JavaScript.
3. Agregar un archivo con el nombre de main.js
4. agregar las instrucciones JavaScript necesarias para entregar dinamismo y funcionalidad a los elementos html elegidos dentro del archivo main.js.
5. Agregar las instrucciones JavaScript necesarias para entregar dinamismo y funcionalidad a los elementos html elegidos.
6. Grabar el documento html y visualizarlo en el navegador web.
7. Probar la funcionalidad agregada a los elementos html a través de código JavaScript.
8. corregir cualquier error que se presente y volver a ejecutar la visualización de la página web y la funcionalidad en los controles elegidos mediante JavaScript.

2. EXPERIENCIA DE PRÁCTICA N° 02: USO DE RECURSOS DEL LENGUAJE

a) USO DE VARIABLES Y CONSTANTES

1. Crear un bloque de código cualquiera.
2. Declarar una variable con ámbito de bloque en el bloque de código anterior.
3. Declarar una constante en ámbito del bloque anterior.
4. Demostrar que las variables y constante tienen el determinado ámbito de bloque
5. Grabar y visualizar los cambios.
6. Corregir y actualizar la grabación.

b) ELEVACIÓN DE VARIABLES

1. Escriba el siguiente código:

```
console.log(x === undefined); // true
var x = 3;

// devolverá un valor de undefined
var myvar = 'my value';
```

```
(function() {
  console.log(myVar); // undefined
  var myvar = 'valor local';
})();
```

2. Describa como se produce el hoisting de la variable.

c) USO DE CONSOLELOG Y SCOPE

1. Utilice console.log() para imprimir los valores de las variables sin asignar.
2. Crear funciones anidadas y verifique el alcance(scope) de las variables y funciones, utilice console.log() en la verificación de ámbito.
3. Grabar y visualizar los cambios.
4. Corregir y actualizar la grabación.

d) USO DE TIPOS DE DATOS Y CAMBIO DE TIPOS DE DATOS

1. Escriba el siguiente código:

```
var answer = 42;
answer = 'Gracias por todo el pescado...';
```

2. Describa el proceso para cambiar el tipo de dato a una variable.
3. Escribir el siguiente código:

```
parseInt("F", 16);
parseInt("17", 8);
parseInt("15", 10);
parseInt(15.99, 10);
parseInt("FXX123", 16);
parseInt("1111", 2);
parseInt("15*3", 10);
parseInt("12", 13);

parseInt("Hello", 8); // No es un número en absoluto
parseInt("0x7", 10);  // No es de base 10
parseInt("546", 2);   // Los dígitos no son válidos para representaciones binarias.
```

4. Describa el proceso anterior.
5. Grabar y visualizar los cambios.
6. Corregir y actualizar la grabación.
7. Escribir el siguiente código:

```
var howMany = 10;

alert("howMany.toString() is " + howMany.toString()); // displays "10"

alert("45 .toString() is " + 45 .toString()); //displays "45"

var x = 7;
alert(x.toString(2))      // Displays "111"
```

8. Describa el proceso anterior.
9. Grabar y visualizar los cambios.
10. Corregir y actualizar la grabación.

3. EXPERIENCIA DE PRÁCTICA N° 03: USO DE FUNCIONES

a) CIUDADANO DE PRIMERA CLASE

1. Escribir el siguiente código:

```
const foo = () => {
  console.log("foobar");
}
foo(); // Invoke it using the variable
// foobar
```

2. Describa el proceso anterior.
3. Grabar y visualizar los cambios.
4. Corregir y actualizar la grabación.
5. Escribir el siguiente código:

```
function sayHello() {
  return "Hello, ";
}
function greeting(helloMessage, name) {
  console.log(helloMessage() + name);
}
// Pass `sayHello` as an argument to `greeting` function
greeting(sayHello, "JavaScript!");
// Hello, JavaScript!
```

6. Describa el proceso anterior.
7. Grabar y visualizar los cambios.
8. Corregir y actualizar la grabación.
9. Escribir el siguiente código:

```
function sayHello() {
  return () => {
    console.log("Hello!");
  }
}
```

10. Describa el proceso anterior.
11. Grabar y visualizar los cambios.
12. Corregir y actualizar la grabación.
13. Establecer las condiciones que debe tener una función de primera clase.

b) CLOSURE

1. Escriba el siguiente código ejemplo:

```
function makeFunc() {
  const name = 'Mozilla';
  function displayName() {
    console.log(name);
  }
  return displayName;
}

const myFunc = makeFunc();
myFunc();
```

2. Describa el proceso anterior.
3. Grabar y visualizar los cambios.
4. Corregir y actualizar la grabación.

5. Establecer las condiciones que debe tener una función que utiliza el principio de clusures.

c) ÁMBITO DE FUNCIÓN

1. Escriba el siguiente código ejemplo:

```
function exampleFunction() {
  const x = "declared inside function";  // x can only be used in exampleFunction
  console.log("Inside function");
  console.log(x);
}

console.log(x);  // Causes error
```

2. Describa el proceso anterior.
3. Grabar y visualizar los cambios.
4. Escriba el siguiente código ejemplo:

```
const x = "declared outside function";

exampleFunction();

function exampleFunction() {
  console.log("Inside function");
  console.log(x);
}

console.log("Outside function");
console.log(x);
```

5. Describa el proceso anterior.
6. Grabar y visualizar los cambios.
7. Escriba el siguiente código ejemplo:

```
function f() {
  try {
    console.log(0);
    throw 'bogus';
  } catch (e) {
    console.log(1);
    return true;    // this return statement is suspended
                    // until finally block has completed
    console.log(2); // not reachable
  } finally {
    console.log(3);
    return false;   // overwrites the previous "return"
    console.log(4); // not reachable
  }
  // "return false" is executed now
  console.log(5);   // not reachable
}
console.log(f()); // 0, 1, 3, false
```

8. Describa el proceso anterior.
9. Grabar y visualizar los cambios.

d) MANEJO DE EXCEPCIONES

1. Escriba el siguiente código ejemplo:

```
function UserException(message) {
  this.message = message;
```

```
  this.name = 'UserException';
}
function getMonthName(mo) {
  mo--; // Adjust month number for array index (1 = Jan, 12 = Dec)
  const months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
    'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
  if (months[mo] !== undefined) {
    return months[mo];
  } else {
    throw new UserException('InvalidMonthNo');
  }
}

let monthName;

try {
  // statements to try
  const myMonth = 15; // 15 is out of bound to raise the exception
  monthName = getMonthName(myMonth);
} catch (e) {
  monthName = 'unknown';
  console.error(e.message, e.name); // pass exception object to err handler
}
```

2. Describa el proceso anterior.
3. Grabar y visualizar los cambios.

4. EXPERIENCIA DE PRÁCTICA N° 04: USO DE OBJETOS

a) CREACIÓN DE OBJETOS

1. Agregar objetos a su proyecto utilizando iniciadores de objetos.
2. Agregar objetos a su proyecto utilizando métodos constructores.
3. Agregar objetos a su proyecto utilizando el método object.create().
4. Grabar y probar los cambios hechos en el código.

b) AGREGANDO PROPIEDADES A LOS OBJETOS

1. Agregar propiedades a los objetos de a su proyecto utilizando iniciadores de objetos.
2. Agregar objetos a su proyecto utilizando métodos constructores.
3. Agregar objetos a su proyecto utilizando el método object.create().
4. Grabar y probar los cambios hechos en el código.

5. EXPERIENCIA DE PRÁCTICA N° 05: USO DE ARRAYS

a) CREACIÓN DE ARRAYS

1. Crear 3 arrays diferentes(numérico, strings y objetos) bajo los 3 métodos de creación que existen para los arreglos.
2. Hacer rutinas para el ingreso de datos a los 3 arreglos.
3. Grabar y probar los cambios hechos en el código.
4. Visualizar los resultados en la página web.
5. Verificar los resultados y corregir errores.

b) MANIPULACIÓN DE ARRAYS

1. Aplicar sobre los arreglos creados los diferentes tipos de acciones incluidos dentro del documento de práctica.
2. Grabar y probar los cambios hechos en el código.
3. Visualizar los resultados en la página web.

4. Verificar los resultados y corregir errores.

c) USO DE ITERADORES DE ARRAYS

1. Aplicar sobre los arreglos creados los diferentes tipos de Iteradores incluidos dentro del documento de práctica.
2. Grabar y probar los cambios hechos en el código.
3. Visualizar los resultados en la página web.
4. Verificar los resultados y corregir errores.

6. EXPERIENCIA DE PRÁCTICA N° 06: USO DE OBJETOS GLOBALES

a) USO DE OBJETO WINDOW

1. Agregar una función que nos permita visualizar las propiedades del objeto Window.
2. En la función del punto anterior agregar el uso de algunos métodos objeto Window.
3. Grabar las instrucciones agregadas.
4. Visualizar los cambios realizados y corregir los errores producidos.

b) USO DEL OBJETO ARRAY

1. Agregar una función que nos permita visualizar las propiedades del objeto Array.
2. En la función del punto anterior agregar el uso de algunos métodos objeto Array.
3. Grabar las instrucciones agregadas.
4. Visualizar los cambios realizados y corregir los errores producidos.

c) USO DE OBJETO NUMBER

1. Agregar una función que nos permita visualizar las propiedades del objeto Number.
2. En la función del punto anterior agregar el uso de algunos métodos objeto Number.
3. Grabar las instrucciones agregadas.
4. Visualizar los cambios realizados y corregir los errores producidos.

V
EJERCICIOS PROPUESTOS

1. **Calculadora de Propinas:** Crea una función que calcule el monto de la propina a dejar en un restaurante. La función debe tomar como entrada el monto total de la cuenta y el porcentaje de propina deseado, y devolver el monto de la propina y el total a pagar (cuenta + propina).
2. **Conversor de Temperatura:** Escribe un programa que permita al usuario ingresar una temperatura en grados Celsius o Fahrenheit y la convierta a la otra escala. Utiliza funciones para realizar las conversiones y muestra los resultados en la consola.
3. **Generador de Contraseña Segura:** Diseña una función que genere contraseñas seguras de una longitud determinada. La contraseña debe incluir una combinación de letras mayúsculas y minúsculas, números y símbolos. Asegúrate de que la función sea flexible para permitir al usuario especificar la longitud deseada y los tipos de caracteres a incluir.

VI
CUESTIONARIO

1. ¿Cuál es el rol principal de JavaScript en el desarrollo web, y cómo se diferencia de HTML y CSS?
2. Explica las tres formas de incorporar código JavaScript en una página web, destacando sus ventajas y desventajas.
3. ¿Por qué es importante la separación de código al trabajar con JavaScript en proyectos web más grandes?

4. Compara y contrasta las palabras clave let, var y const en JavaScript. ¿Cuándo deberías usar cada una y por qué?
5. ¿Qué es el hoisting en JavaScript y cómo afecta la declaración y el uso de variables?
6. Describe los diferentes tipos de ámbito (scope) en JavaScript y cómo determinan la accesibilidad de las variables.
7. ¿Cuál es la diferencia entre el ámbito local y el ámbito global en JavaScript? Proporciona ejemplos.
8. Explica la utilidad de console.log() en el desarrollo de aplicaciones JavaScript. ¿Cómo puedes utilizarlo para depurar tu código?
9. Enumera los tipos de datos primitivos en JavaScript y proporciona un ejemplo de cada uno.
10. ¿Qué es un objeto en JavaScript? ¿Cómo se diferencia de los tipos de datos primitivos?
11. Describe cómo crear un objeto en JavaScript utilizando diferentes métodos (literal de objeto, constructor, Object.create()).
12. ¿Cómo se agregan, modifican y eliminan propiedades y métodos de un objeto en JavaScript?
13. ¿Qué es un array en JavaScript? ¿Cómo se accede a sus elementos y se modifican?
14. Explica la importancia de los iteradores en JavaScript. Proporciona ejemplos de cómo utilizar forEach, map y filter para trabajar con arrays.
15. ¿Qué son las funciones en JavaScript y por qué son fundamentales en la programación?
16. Compara y contrasta las diferentes formas de declarar funciones en JavaScript: declaración de función, expresión de función y función flecha.
17. ¿Qué es un closure en JavaScript y cómo se relaciona con las funciones anidadas? Proporciona un ejemplo práctico de su uso.
18. ¿Por qué es importante el manejo de excepciones en JavaScript? ¿Cómo puedes utilizar los bloques try...catch...finally para gestionar errores de manera efectiva?
19. ¿Qué es el DOM (Document Object Model) y cómo interactúa JavaScript con él?
20. Explica cómo seleccionar elementos del DOM utilizando métodos como getElementById, querySelector y querySelectorAll.
21. Describe cómo puedes modificar el contenido, el estilo y los atributos de los elementos del DOM utilizando JavaScript.
22. ¿Qué son los eventos en JavaScript y cómo puedes utilizarlos para crear interactividad en tus aplicaciones web?
23. Explica el concepto de programación asíncrona en JavaScript. ¿Por qué es importante y qué desafíos presenta?
24. Compara y contrasta los diferentes enfoques para manejar la asincronía en JavaScript: callbacks, promesas y async/await.
25. ¿Cómo puedes utilizar JavaScript para mejorar la experiencia del usuario en una aplicación web?
26. Investiga y describe una biblioteca o framework de JavaScript que te interese. ¿Cuáles son sus principales características y ventajas?

VII

REFERENCIAS

D. F. Silva, R. M. (2021). "A Study on the Performance of Java Virtual Machine Garbage Collectors. *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 1*(35), 23-31.

Duckett, J. (2014). *JavaScript & jQuery: Interactive Front-End Web Development.* Wiley.

Flanagan, D. (2020). *JavaScript: The Definitive Guide (7th ed.).* O'Reilly Media.

Haverbeke, M. (2018). *Eloquent JavaScript: A Modern Introduction to Programming (3rd ed.).* No Starch Press.

Hortsmann, C. (2009). *Big Java.* San José: Pearson.

Oracle. (1994). *The Java™ Tutorials*. (Oracle) Recuperado el 28 de 07 de 2024, de https://docs.oracle.com/javase/tutorial/index.html

Zakas, N. C. (2011). *Professional JavaScript for Web Developers (3rd ed.).* Wiley.