



# ULAGOS

## UNIDAD II

### Fundamentos de JS

INGENIERÍA EN INFORMÁTICA

Docente: Victor Saldivia Vera

Correo: [victor.saldivia@ulagos.cl](mailto:victor.saldivia@ulagos.cl)

# TABLA DE CONTENIDO

## VARIABLES Y SINTAXIS BÁSICA

Var, Let, Const y Estructura de Código y Scopes

01



## TIPOS DE DATOS

Datos más utilizados en JavaScript

02



## OPERADORES

Operadores matemáticos

03



04

## CONDICIONALES/CICLOS

Sentencias de comparación y bucles



05

## FUNCIONES

Métodos con variables locales y variables externas



06

## EJERCICIOS CON HTML

Ejercicios con HTML



# PARTE I

INTRODUCCIÓN A JS



Brendan Eich creador de JS



## UN POCO DE HISTORIA

JavaScript fue creado por **Brendan Eich** en **1995** mientras trabajaba en **Netscape Communications**.

JavaScript es un lenguaje de programación de alto nivel y orientado a objetos que se ha convertido en uno de los pilares fundamentales de la web. A lo largo de su historia, ha experimentado importantes hitos que han contribuido a su popularidad y evolución.

En **1997** JavaScript fue estandarizado por primera vez bajo el nombre de ECMAScript por Ecma International. ECMAScript define la especificación del lenguaje JavaScript y garantiza su compatibilidad entre diferentes implementaciones.



Luego en el año **2005**, jQuery una biblioteca de JavaScript simplificó en gran medida la manipulación del DOM (Document Object Model) y el manejo de eventos en los navegadores. jQuery se convirtió en una de las bibliotecas más populares y facilitó el desarrollo web en múltiples navegadores.

# EXPANSIÓN HACIA EL LADO DEL SERVIDOR

Con el surgimiento de **Node.js** el año 2009 como entorno de tiempo de ejecución basado en el motor de JavaScript V8, significó la expansión de JavaScript, lo que permitió ejecutar el lenguaje en el lado del servidor. Con el nacimiento de Node.js brindó a los desarrolladores la capacidad de crear aplicaciones web de alto rendimiento y escalables en JavaScript.



# 2009



# 2023



Con el precedente de Node.js, surgieron más frameworks de JavaScript se han convertido en referentes en el desarrollo de aplicaciones web modernas. Angular, desarrollado por Google, React, creado por Facebook, y Vue.js, un framework progresivo, han simplificado la creación de interfaces de usuario interactivas y reactivas. Sin nombrar a muchos más entornos que existen en la actualidad.

# ES6 and JS

ECMAScript 6      Javascript

Una hito importante en la historia de JavaScript es la nueva versión de **ES6**. Este estándar indica como debe ser interpretado el lenguaje, en cada tecnología como: navegadores, servidores de node.js, en el desarrollo de aplicaciones, etc

El salto de **ES5** a **ES6** significó muchas mejoras del lenguaje por ejemplo: simplificación en el proceso de código orientado a objetos, el uso de las arrows functions, el uso de promesas, y las nuevas declaraciones de variables: let y const. Actualmente ya nos encontramos en la versión ES14.

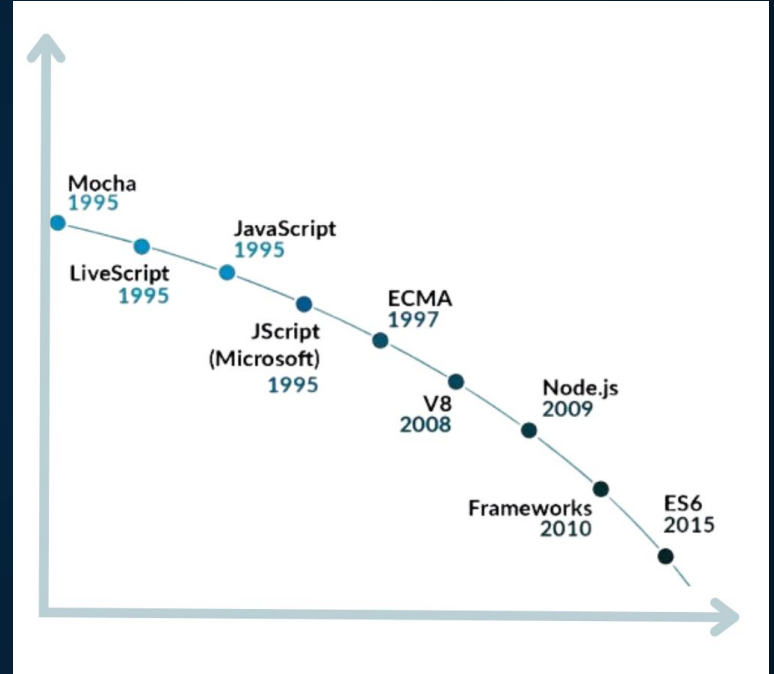


Figura 1. Línea de Tiempo de hitos más relevantes de JS



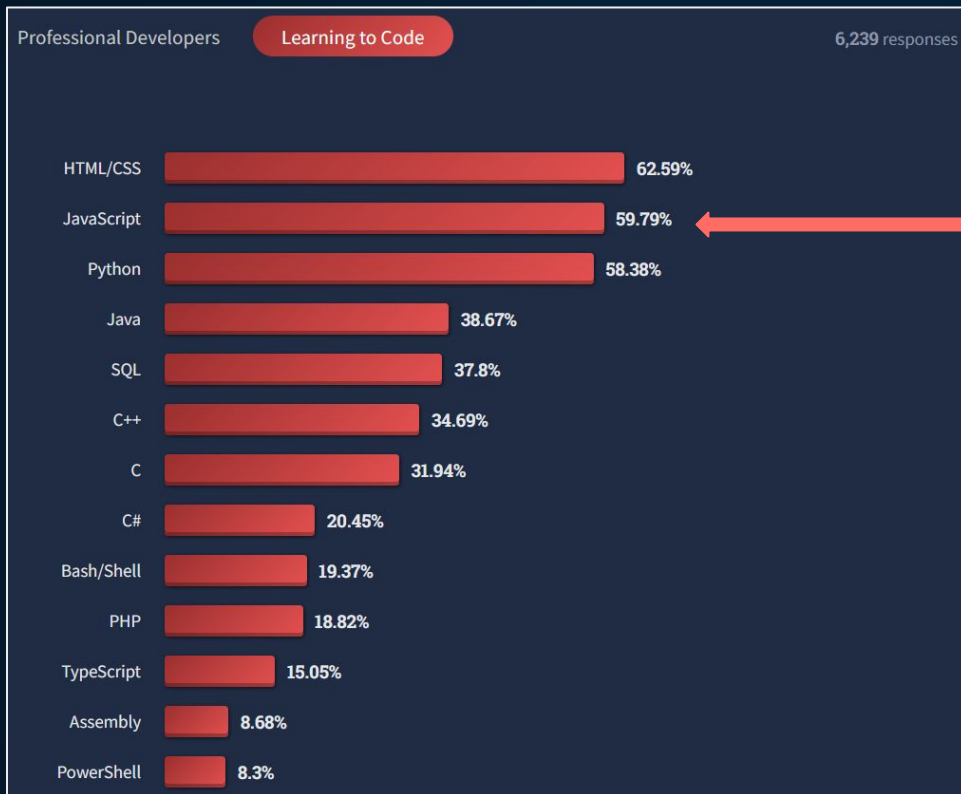


Fuente: StackOverflow - Lenguajes Más Populares por Developers 2022

El mercado actual nos indica que es indispensable para cualquier profesional del área de desarrollo conozca **JavaScript**.

El 2022 se **consagró como el décimo año consecutivo** en el que **JavaScript** ocupó el lugar del lenguaje de programación más utilizado. Es el más popular del mundo, con más de 14 millones de profesionales trabajando con este lenguaje en todo el planeta. Además, es el lenguaje de los navegadores web.

Según la encuesta realizada por **StackOverflow** el **67,9%** de profesionales tech trabajan con JavaScript.



Importante destacar que **JavaScript** es uno de los lenguajes de programación con una curva de aprendizaje baja, por eso es uno más recomendado para empezar, porque es muy versátil y amigable con las personas que no tienen ninguna experiencia a la hora de programar.

Según la encuesta de **StackOverflow** con un **59,79%** de los encuestados principiantes, prefieren a JavaScript como lenguaje de programación a la hora de dar sus primeros pasos en el área de la programación.

Fuente. StackOverflow - Lenguajes Más Populares por Principiantes 2022

# TÉRMINOS A ENTENDER

## ANTES DE EMPEZAR CON VARIABLES DE JS



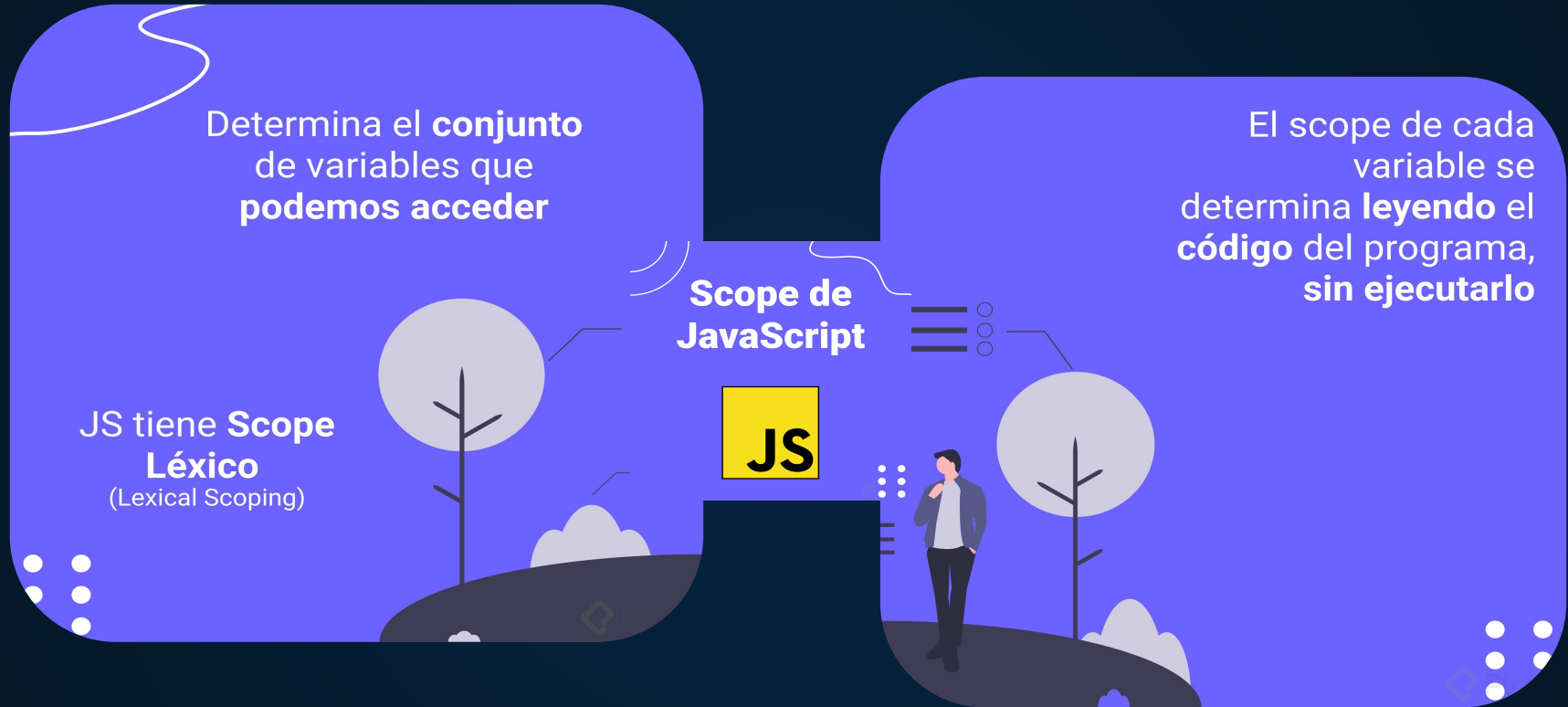
### SCOPE (ALCANCE)

El scope en JavaScript se refiere al alcance o contexto en el cual una variable o función es accesible. Determina la visibilidad y disponibilidad de las variables y funciones en diferentes partes de un programa. En **JS**, existen tres tipos principales de scope: **scope global**, **scope de bloque** y **scope local**, las cuales se explicarán en capítulos, más adelante.



### HOISTING

Es un comportamiento de JavaScript, que se refiere a la forma en que el lenguaje mueve las declaraciones de variables y funciones al principio del ámbito de ejecución, antes de que se ejecute el código. Esto quiere decir, que se puede usar una variable o una función antes de haber sido declarada, aunque no es una buena práctica.



Cuando hablamos de **Scope** nos referimos al **entorno**

En cada **función** héroe significa una algo **diferente**

```
function teamCap () {
  var héroe = "Capitan America";
  console.log(héroe);
  // Capitan America
}

function teamIronMan () {
  var héroe = "Iron Man";
  console.log(héroe);
  // Iron Man
}
```

Scope de  
JavaScript

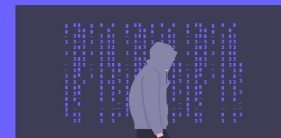


En JS tenemos Diferentes  
Scopes

**Global Scope**

**Local Scope**

- Block Scope
- Function Scope



## Global Scope

Cuando **declaramos** fuera de toda **función** o **bloque de código**

```
var comic = 'X-Men'

function comprar() {
  console.log('compraste el comic ${comic}')
  // compraste el comic X-Men
}

function leer() {
  console.log('estas leyendo ${comic}')
  // estas leyendo X-Men
}

comprar()
leer()
```

La variable **comic** se **podrá** usar a lo largo de **todo** nuestro **programa**

## Scope de JavaScript



## Block Scope

**Bloque de código** como ser un **if**, **while**, **for**, etc. es decir que están **entre llaves**

```
function mutantes() {
  const mutante = 'Wolverine'
  if (true) {
    // esta variable se quedara solo
    // entre estas llaves
    const mutante = 'Ciclope'
    console.log(mutante)
    // "Ciclope"
  }
  console.log(mutante)
  // "Wolverine"
}
```

variables **declaradas** con **let** o **const**

## Function Scope

Accesibles dentro de toda la función, pero **no fuera** de la misma

```
function justiceLeague() {
  var heroes = ['Batman', 'Superman', 'Aquaman']
  console.log(heroes)
  // ["Batman", "Superman", "Aquaman"]
}

function teenTitans() {
  var heroes = ['Robin', 'Raven', 'Terra']
  console.log(heroes)
  // ["Robin", "Raven", "Terra"]
}

console.log(heroes)
// ReferenceError:
// heroes is not defined
```

variables **declaradas**  
con **var**

Scope de  
JavaScript

JS

Para evitar que las funciones se vuelvan **impredecibles** debemos **declarar dentro del scope más reducido**

JS

# 01. VARIABLES



# VARIABLES

En JavaScript, existen 3 formas de declarar variables: **var**, **let** y **const**. Cada una tiene sus propias características y diferencias que se explica a continuación:

## VAR

Permite la reasignación de valores y tiene un scope global o de función. Es sometida a hoisting.

## LET

Permite la reasignación de valores y tiene un scope de bloque. No está sometida a hoisting.

## CONST

No permite la reasignación de valores y puede tener un scope global, de función o de bloque. No es sometida a hoisting.

Antes de la llegada de **ES6 (ECMAScript 6)**, las declaraciones **var** eran las que se utilizaban. Sin embargo, hay problemas asociados relacionados con variables declaradas con **var**. Por eso fue necesario que surgieran nuevas formas de declarar variables, y es así donde aparece **let** y **const**.

## VAR v/s LET

Uno de los problemas es, cuando se declara una **variable** con **var** dentro de una función o fuera de cualquier bloque, su alcance es la función o el ámbito global. Esto puede causar problemas de colisión de variables y dificultar la mantención del código. Las variables declaradas con **var** se mueven al principio de su ámbito (**hoisting**), esto puede generar comportamientos inesperados y dificultar la comprensión del código.

Por último, las variables **var** pueden ser reasignadas y redeclaradas dentro del mismo ámbito, lo que puede llevar a errores sutiles. Aquí es donde aparece la variable **let**, solucionando estos problemas y ofreciendo un comportamiento más predecible y seguro en cuanto a scope y hoisting.



```
1  var estatura = 1.71
2  let peso = 60
3  const nombre = "Victor"
4
```

Código 1. Variables en JS

En scripts más antiguos, siempre van a encontrar **var** en lugar de **let**:



Código 2. Variable de tipo var

## En Resumen...

Es recomendable utilizar **let** en lugar de **var** en la mayoría de los casos, porque entrega un mejor control sobre el ámbito de las variables y evita problemas futuros en el código. Sin embargo, en contextos más antiguos o específicos, donde se requiere el comportamiento de **var**, aún se puede utilizar sin problemas.

# ¿CÓMO IMPRIMIR UN MENSAJE EN JS?

---

**alert ( )** 

**console.log ( )** 

**document.write ( )** 

# LA ETIQUETA SCRIPT

Los programas de JavaScript se pueden insertar en casi cualquier parte de un **documento HTML** con el uso de la etiqueta **<script>**, como se muestra en el **Código 3**.

La etiqueta **<script>** contiene código JavaScript que se ejecuta automáticamente cuando el navegador procesa la etiqueta.

Más adelante se explicará en detalle la estructura de una página web en HTML, solo basta con saber la estructura básica y fundamental para ejecutar un código JS.



```
1  <!DOCTYPE HTML>
2  <html>
3      <body>
4          <p>Antes del script ... </p>
5              <script>
6                  alert('¡Hola, mundo!');
7              </script>
8          <p>... Desp és del script.</p>
9      </body>
10 </html>
```

Código 3. Hola Mundo en JS

# SCRIPT EXTERNO

Como regla general, solo los **scripts más simples** se colocan en el **HTML**. Los más complejos están en archivos separados. La ventaja de un archivo separado es que el navegador lo descargará y lo almacenará en caché.

Si tenemos mucho código de JavaScript, es de buena práctica colocarlo en un archivo aparte.

Los archivos de **script** se adjuntan a **HTML** con el atributo **src**, como se muestra en el **Código 4**.

```
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5      <meta charset="UTF-8">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Document</title>
8  </head>
9
10 <body>
11     <!-- Hola Mundo Externo desde un Archivo JS -->
12     <script src="../JS/01.holamundo.js"></script>
13 </body>
14 </html>
```

Código 4. Hola Mundo en JS con un Script Externo

## **02. TIPOS DE DATOS**

# TIPOS DE DATOS

Como en todo lenguaje de programación existen diferentes tipos de datos.

Vamos a conocer algunos de los tipos de datos más utilizados en JavaScript. Cubriremos en términos generales cada uno de ellos.

**NUMBERS**



**STRINGS**



**BOOLEANS**



**NULL**



**UNDEFINED**



**OBJECT Y SYMBOL**





# NUMBERS





```

1  let edad = 29;
2  let sueldo = 1200000;
3  const PI = 3.14;
  
```

# NUMBERS

El tipo number representa tanto números enteros como de punto flotante. En JavaScript, **no existe un tipo de dato específico para representar números enteros o números con decimales**, a diferencia de Python.

Al igual que en otros lenguajes se pueden realizar diferentes operaciones aritméticas como: suma, resta, multiplicación y división.

Además de los números comunes, existen los llamados **“valores numéricos especiales”** que también pertenecen a este tipo de datos: **Infinity**, **-Infinity** y **NaN**.

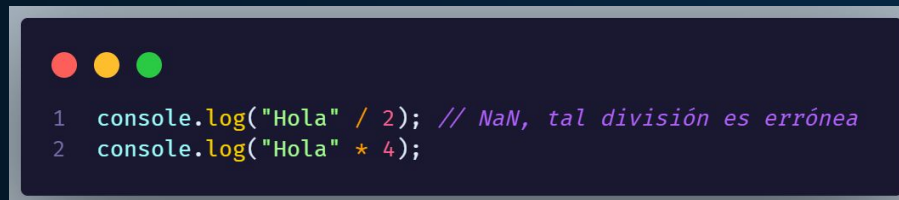
# INFINITY

**Infinity** representa el Infinito matemático  $\infty$ . Es un valor especial que es mayor que cualquier número.

Una manera de obtenerlo como resultado sería dividiendo por cero:



```
1 console.log(edad / 0);
2 console.log(Infinity)
```



```
1 console.log("Hola" / 2); // NaN, tal división es errónea
2 console.log("Hola" * 4);
```

# NaN

**NaN** representa un error de cálculo. Es el resultado de una operación matemática incorrecta o indefinida, por ejemplo:

# NUMBERS

## (Notación Científica y BigInt)

Los números en notación científica se representan utilizando la notación exponencial. Se utiliza el formato **xEn**, donde **x es el coeficiente** y **n es el exponente**. El x puede ser un número decimal o entero, y el n puede ser positivo o negativo.

Por ejemplo, el número 1 millón en notación científica sería representado como **1e6** (como se muestra en la imagen de la izquierda).

En cuanto a los **BigInt**, se utilizan para representar enteros de tamaño muy grande, superando el límite de Number. Los BigInt se crean utilizando el **sufijo n** al final de un número entero.



```
1  const n_grande = 1e6; // 1 millón
2  const n_pequeno = 1e-6; // 0.000001
```



```
1  const bigInt = 1234567890123456789012345678901234567890n;
```

# STRINGS



# STRINGS

Las cadenas de texto (strings) son un tipo de dato que representa una secuencia de caracteres. En JavaScript, podemos crear strings utilizando comillas simples o comillas dobles. También se puede ocupar backticks (comillas invertidas).



```
1 let string1 = "Hola ¿Cómo estás?";  
2 let string2 = '¡Buenas Tardes!';  
3 let frase = `Este es un saludo: ${string1}`;
```

Tanto el uso de comillas dobles o simples no tiene ninguna diferencia. Los backticks son comillas de "funcionalidad extendida". Nos permiten incrustar variables y expresiones en una cadena de caracteres encerrándolas en `${...}`, como se muestra en el ejemplo de arriba.

# BACKTICKS

Es importante el uso de los **backticks** para imprimir estas cadenas extendidas. Nos permiten incrustar variables y expresiones en una cadena de caracteres encerrándolas en `${...}`, como se muestra en el código de abajo:



```
1 console.log(`Ulagos se ubica en ${ciudad}`);
```

La expresión que se encuentra dentro de `${...}` se evalúa y el resultado pasa a formar parte de la cadena.

**¡Esto solamente se puede hacer con los backticks!**

**¡Las otras comillas no tienen esta capacidad de incrustación!**

# OTRA MANERAS DE IMPRIMIR

## VARIABLES DENTRO DE STRINGS

Además del uso de backticks también se puede utilizar los siguientes métodos:



```
1 //Utilizando el operador + de concatenación
2 console.log('Mi nombre es ' + nombre);
```

Utilizando el operador **+** de concatenación



```
1 //Operador con Comas
2 console.log('Mi nombre es ', nombre);
```

Utilizando las **comas** para separar el texto de las variables




```
1 //Utilizando el Método concat
2 console.log("Mi nombre es ".concat(nombre, "y vivo en ", ciudad));
```

Utilizando el método **.concat**



# LARGO DEL STRING


La propiedad **'length'** nos permite devolver la longitud de una cadena:



```
1 let txt = "Hola Mundo!";  
2 let longitud = txt.length;
```

## ACCEDIENDO A CARACTERES

Si deseamos acceder a un carácter en específico dentro de una cadena de texto, podemos utilizar el método **charAt()**, el cual nos devuelve el carácter en la posición especificada:



```
1 let lenguaje = "JavaScript";  
2 let primerCaracter = lenguaje.charAt(0);
```

# MÉTODO PARA SUBCADENAS

**substring()** es un método para extraer una subcadena de texto de una cadena más grande, dadas las posiciones inicial y final dentro de la cadena.

Similar al método **substring()** tenemos el método **slice()** que se utiliza para crear una nueva cadena que es una subcadena de la cadena original. Toma dos argumentos: el índice de inicio y el índice de finalización. El índice de finalización puede ser omitido para tomar desde el índice de inicio hasta el final de la cadena.

También podemos manejar índices negativos, como se muestra en el ejemplo:



```
1 let frase = "¡Hola!, ¿Cómo estás?";  
2 let subcadena = frase.substring(0, 6);
```



```
1 const txt2 = "Universidad de Los Lagos";  
2 const otracadena = txt2.slice(0, -12);
```

# REEMPLAZANDO UNA SUBCADENA

El método **replace()** permite reemplazar una subcadena por otra en la cadena original. Permite utilizar tanto cadenas de texto simples como expresiones regulares (**patrones**) para buscar y reemplazar texto en una cadena.



```
1 let texto3 = "Estamos en la Universidad de Los Lagos";  
2 let newTxt = texto3.replace("Universidad de Los Lagos", "Ulagos");
```

## EL MÉTODO SPLIT

El método **split()** divide una cadena en un array de subcadenas basadas en un delimitador. Es decir, es empleado para dividir una cadena, lo que nos entregará como resultado un arreglo que no afecta la cadena original.



```
1 const txt4 = "Manzana,Naranja,Uva";  
2 const frutas = txt4.split(",");
```

# MÁS MÉTODOS PARA STRINGS

<\>

**IndexOf()**

Este método, devuelve la primera posición de una subcadena en la cadena original. Si no se encuentra, devuelve -1.

<\>

**toUpperCase() toLowerCase()**

El primer método convierten una cadena a mayúsculas y **toLowerCase()** a minúsculas.

<\>

**trim()**

Este método permite eliminar los espacios en blanco al principio y al final de la cadena.

# BOOLEANOS





```

1  let V = true;
2  let F = false;
3
4  if (V) {
5      console.log("Es Verdadero");
6  } else {
7      console.log("Es Falso");
8  }
9

```

# BOOLEANS

El tipo boolean tiene sólo dos valores posibles: **true** y **false**.

Este tipo se utiliza comúnmente para almacenar valores de sí/no: true significa **"sí, correcto, verdadero"**, y false significa **"no, incorrecto, falso"**.

En el ejemplo de la izquierda,



**NULL**





```
1 let dato = null;
2 if (dato === null) {
3     console.log('¡OK!');
4 }
```

Por ejemplo, podemos utilizar directamente el literal `null` para comprobar este valor es el que contiene una determinada variable o es el retorno de una función.

# NULL

A la hora de desarrollar un código, no siempre deseamos otorgar valores a nuestras variables. Para ello, existen los valores de tipo primitivo **NaN**, **undefined** y **null**.

Este último tipo de valor primitivo, **Null** es una palabra clave que **nos permite definir un valor nulo o vacío de manera intencional**. Es decir, en vez de utilizar otros tipos de valores, como podrían ser un número o un string, llenamos el “espacio” de código con un valor vacío de significado.





**UNDEFINED**



# UNDEFINED

En simples palabras **undefined** es cuando una variable ha quedado sin definir tiene un valor especial que lo denominamos como **undefined**.

En primer lugar, **undefined** es un tipo de dato y por lo tanto se puede consultar con **typeof**.

Se debe tener en cuenta que, **undefined** no es una palabra reservada de Javascript y por lo tanto podemos definir una variable con ese nombre, lo cual puede ser problemático en algunos casos.



```
1 let institución;  
2 console.log(institución) //arroja como valor undefined
```



# OBJECT



## EJEMPLO OBJECT

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. Por ejemplo, un usuario. Un usuario es un objeto con propiedades. Un usuario tiene un nombre, una edad, un peso, un email, etc. Del mismo modo, los objetos de JavaScript pueden tener propiedades que definan sus características.

```
1 let users = {  
2   name: "Mateo",  
3   age: 30,  
4   "correo electronico": "mateo@gmail.com"  
5 };
```

# JS

## EJERCICIO 1

ESTE EJERCICIO SE INCLUYE EN EL AVANCE DE PORTAFOLIO

Realizar un programa en JavaScript con lo aprendido en clases que permita ingresar un número por pantalla y saber si es un número par o número impar.

JS

## EJERCICIO 2

### ESTE EJERCICIO SE INCLUYE EN EL AVANCE DE PORTAFOLIO

Crear un algoritmo capaz de solicitar 3 notas de un alumno por pantalla y obtener el promedio ponderado. Este promedio ponderado es de la siguiente forma:

Nota 1 = 40%

Nota 2 = 30%

Nota 3 = 30%

Si el estudiante obtiene un promedio inferior a 3.95 se debe imprimir por pantalla que ha reprobado la asignatura, si obtuvo un promedio entre 3.95 a 4.94 el estudiante va a examen. Si la nota es igual o superior a 4.95 el alumno se exime de la asignatura.

## 03. OPERADORES

# OPERADORES ARITMÉTICOS

Nombre	Operador	Descripción
Suma	<code>a + b</code>	Suma el valor de <code>a</code> al valor de <code>b</code> .
Resta	<code>a - b</code>	Resta el valor de <code>b</code> al valor de <code>a</code> .
Multiplicación	<code>a * b</code>	Multiplica el valor de <code>a</code> por el valor de <code>b</code> .
División	<code>a / b</code>	Divide el valor de <code>a</code> entre el valor de <code>b</code> .
Módulo	<code>a % b</code>	Devuelve el resto de la división de <code>a</code> entre <code>b</code> .
Exponenciación	<code>a ** b</code>	Eleva <code>a</code> a la potencia de <code>b</code> , es decir, <code>a<sup>b</sup></code> . Equivalente a <code>Math.pow(a, b)</code> .

Nota: Para mayor información se mostrará ejemplos prácticos de código en clases



# OPERADORES DE ASIGNACIÓN

Nombre	Operador	Descripción
Suma y asignación	<code>a += b</code>	Es equivalente a <code>a = a + b</code> .
Resta y asignación	<code>a -= b</code>	Es equivalente a <code>a = a - b</code> .
Multiplicación y asignación	<code>a *= b</code>	Es equivalente a <code>a = a * b</code> .
División y asignación	<code>a /= b</code>	Es equivalente a <code>a = a / b</code> .
Módulo y asignación	<code>a %= b</code>	Es equivalente a <code>a = a % b</code> .
Exponenciación y asignación	<code>a **= b</code>	Es equivalente a <code>a = a ** b</code> .

Nota: Para mayor información se mostrará ejemplos prácticos de código en clases

# OPERADORES UNARIOS

Nombre	Operador	Descripción
Incremento	<code>a++</code>	Usa el valor de <code>a</code> y luego lo incrementa. También llamado <b>postincremento</b> .
Decremento	<code>a--</code>	Usa el valor de <code>a</code> y luego lo decrementa. También llamado <b>postdecremento</b> .
Incremento previo	<code>++a</code>	Incrementa el valor de <code>a</code> y luego lo usa. También llamado <b>preincremento</b> .
Decremento previo	<code>--a</code>	Decrementa el valor de <code>a</code> y luego lo usa. También llamado <b>predecremento</b> .
Resta unaria	<code>-a</code>	Cambia de signo (niega) a <code>a</code> .

Nota: Para mayor información se mostrará ejemplos prácticos de código en clases

# OPERADORES DE COMPARACIÓN

Nombre	Operador	Descripción
Operador de igualdad =	<code>a = b</code>	Comprueba si el <b>valor</b> de <code>a</code> es igual al de <code>b</code> . <b>No comprueba tipo de dato.</b>
Operador de desigualdad $\neq$	<code>a <math>\neq</math> b</code>	Comprueba si el <b>valor</b> de <code>a</code> no es igual al de <code>b</code> . <b>No comprueba tipo de dato.</b>
Operador mayor que >	<code>a &gt; b</code>	Comprueba si el valor de <code>a</code> es mayor que el de <code>b</code> .
Operador mayor/igual que $\geq$	<code>a <math>\geq</math> b</code>	Comprueba si el valor de <code>a</code> es mayor o igual que el de <code>b</code> .
Operador menor que <	<code>a &lt; b</code>	Comprueba si el valor de <code>a</code> es menor que el de <code>b</code> .
Operador menor/igual que $\leq$	<code>a <math>\leq</math> b</code>	Comprueba si el valor de <code>a</code> es menor o igual que el de <code>b</code> .
Operador de identidad $\equiv$	<code>a <math>\equiv</math> b</code>	Comprueba si el <b>valor y el tipo de dato</b> de <code>a</code> es igual al de <code>b</code> .
Operador no idéntico $\neq$	<code>a <math>\neq</math> b</code>	Comprueba si el <b>valor y el tipo de dato</b> de <code>a</code> no es igual al de <code>b</code> .

Nota: Para mayor información se mostrará ejemplos prácticos de código en clases

# **04. CONDICIONALES Y CICLOS**

# IF - ELSE



```
1 let year = prompt('¿En qué año se creo JavaScript', '');
2
3 if (year === 1995) {
4     alert('¡Correcto, en el año 1995!');
5 } else {
6     alert('Respuesta Incorrecta!');
7 }
```

Como en la mayoría de los lenguajes de programación, la sentencia **if( ... )** evalúa la condición en los paréntesis, y si el resultado es verdadero (true), ejecuta un bloque de código.

La sentencia if puede contener un bloque **else** (“**si no**”, “**en caso contrario**”) opcional. Este bloque se ejecutará cuando la condición sea falsa.

En el ejemplo de la izquierda, se muestra un código donde se combina una sentencia **if** y una **else**.

## ELSE IF

Cuando deseamos probar más de una condición, podemos utilizar la cláusula **else if** que nos permite hacer lo que se muestra en el código de ejemplo.

```
1 let edad = prompt('¿A que edad puedes tener licencia?', '');
2
3 if (edad >= 18) {
4     alert('Correcto, a partir de los 18 años!');
5 } else if (edad == 17) {
6     alert('Correcto, pero no puede manejar solo un menor de 17 años!');
7 } else {
8     alert('Respuesta Incorrecta! No puede ser menor de 17 años');
9 }
```

# OPERADOR TERNARIO



```
1  let newPrompt = prompt('¿A qué edad puedes tener licencia?', '');
2
3  const mensaje =
4      newPrompt ≥ 18
5          ? 'Correcto, a partir de los 18 años!'
6          : 'Respuesta Incorrecta! No puede ser menor de 18 años';
7
8  alert(mensaje);
```



## OPERADOR TERNARIO

El operador ternario lo podemos entender como una **estructura compacta** para hacer condicionales de una forma más fácil.

Consiste en una expresión que se evaluará y, dependiendo de si dicha evaluación fue positiva o negativa devolverá una u otra cosa.



JS

## EJERCICIO 3

ESTE EJERCICIO SE INCLUYE EN EL AVANCE DE PORTAFOLIO

Construir un programa permita calcular e imprimir el resultado de la siguiente sumatoria:

$$S = 500 + 456 + 510 + 454 + 520 + 452 + \dots + 800$$

# JS

## EJERCICIO 4

### ESTE EJERCICIO SE INCLUYE EN EL AVANCE DE PORTAFOLIO

Tres empleados de una fábrica trabajan en dos turnos: diurno y nocturno (10 hrs cada uno). Se busca calcular el pago diario y el total semanal de cada empleado de acuerdo con los siguientes puntos:

- a) La tarifa del turno diurno por hora es de 12000 CLP.
- b) La tarifa del turno nocturno por hora es de 16000 CLP.
- c) Los domingos la tarifa se incrementa en 2000 CLP el turno diurno y 3000 CLP el turno nocturno.

Además considerar:

- a) El primer empleado trabaja Lunes, Martes y Miércoles en turnos nocturnos, Jueves y Viernes en turnos diurnos.
- b) El segundo empleado trabaja Martes, Miércoles y Jueves turnos nocturnos, y también el día domingo en turno diurno.
- c) El tercer empleado trabaja Miércoles, Jueves y Viernes diurno, Sábado y Domingo en turnos nocturnos.

Guardar la información de cada empleado en un objeto literal, con el pago diario que debe recibir cada empleado y el total de la semana.