

**JS**

**APRENDE  
JAVASCRIPT  
DESDE CERO  
HASTA AVANZADO**

A C A D E M I A X

## Contenido

<b>1 Introducción</b>	<b>10</b>
1.1 Bienvenida . . . . .	10
1.1.1 Libro vivo . . . . .	11
1.1.2 Alcance . . . . .	11
1.2 Prerequisitos . . . . .	12
1.3 ¿Cómo evitar bloqueos? . . . . .	12
<b>2 Primeros pasos</b>	<b>13</b>
2.1 Visión general . . . . .	13
2.1.1 ¿Qué es y porqué debes aprenderlo? . . . . .	13
2.1.2 ¿En dónde se utiliza? . . . . .	14
2.1.3 ¿Qué trabajos puedes conseguir? . . . . .	15
2.1.4 ¿Cuánto puedes ganar? . . . . .	16
2.1.5 ¿Cuales son las preguntas más comunes? . . . . .	16
2.2 Historia, evolución, y versiones . . . . .	17
2.3 Características y ventajas . . . . .	18
2.4 Diferencias con otros lenguajes de programación . . . . .	19
2.5 Configuración . . . . .	19
2.5.1 IDE . . . . .	19
2.5.2 Entorno . . . . .	20
2.6 Hola Mundo . . . . .	20
<b>3 Gramática</b>	<b>21</b>
3.1 Sintaxis . . . . .	21
3.2 Comentarios (una sola línea y multilínea) . . . . .	21
3.3 Literales y tipos de datos . . . . .	23
3.4 Operadores y expresiones . . . . .	24
3.5 Palabras clave e identificadores . . . . .	25
3.6 Sentencias, declaraciones, y tipado . . . . .	25

## Contenido

---

3.7 Bloques e indentación . . . . .	26
3.8 Estructuras de control de flujo y excepciones . . . . .	27
3.9 Conjunto de caracteres . . . . .	29
3.10 Sensibilidad de mayúsculas y minúsculas . . . . .	29
<b>4 Tipos y variables . . . . .</b>	<b>30</b>
4.1 Tipos de datos primitivos . . . . .	30
4.2 Tipos de datos no-primitivos . . . . .	31
4.3 Primitivos . . . . .	32
4.3.1 Texto . . . . .	32
4.3.2 Número . . . . .	33
4.3.3 Entero grande . . . . .	34
4.3.4 Buleano . . . . .	35
4.3.5 Indefinido . . . . .	36
4.3.6 Símbolo . . . . .	36
4.3.7 Nulo . . . . .	37
4.4 Objetos . . . . .	38
4.5 Acceder a propiedades de objetos . . . . .	38
4.6 Colecciones con llave . . . . .	40
4.6.1 Mapas . . . . .	40
4.6.2 Sets . . . . .	41
4.6.3 WeakMaps . . . . .	42
4.6.4 WeakSets . . . . .	43
4.7 Listas . . . . .	44
4.8 Acceder a elementos de listas . . . . .	45
4.9 Fechas . . . . .	46
4.10 Expresiones regulares . . . . .	46
4.11 Declaracion e inicialización (var y let) . . . . .	47
4.12 Constantes . . . . .	48
4.13 Plantillas de texto . . . . .	49
4.14 Chequear tipo . . . . .	50

## Contenido

---

4.15 Conversión de tipos de datos . . . . .	51
4.16 Coerción . . . . .	51
<b>5 Operadores . . . . .</b>	<b>53</b>
5.1 Operadores de aritméticos (+, -, *, /, %, **, //, ++, -) . . . . .	53
5.2 Operador de agrupación (( )) . . . . .	54
5.3 Operadores de texto (+, +=) . . . . .	55
5.4 Operadores de asignación (=, +=, -=, *=, /=, %=, **=, //=) . . . . .	56
5.5 Operadores comparativos (==, !=, >, <, >=, <=) . . . . .	60
5.6 Comparar por valor y por referencia . . . . .	61
5.7 Operadores lógicos (&&,   , !) . . . . .	62
5.8 Operadores de bits (&,  , ~, ^, », «, »>) . . . . .	63
5.9 Operadores unarios (delete, typeof, void) . . . . .	64
5.10 Operadores relacionales (in, instanceof) . . . . .	66
5.11 Operador spread (...) . . . . .	67
5.12 Operador de coma (,) . . . . .	68
<b>6 Estructuras de control de flujo . . . . .</b>	<b>69</b>
6.1 Condicionales . . . . .	69
6.1.1 if . . . . .	69
6.1.2 if…else . . . . .	70
6.1.3 if…elif…else . . . . .	71
6.1.4 Operador ternario (?) . . . . .	72
6.1.5 Condicionales anidados . . . . .	73
6.1.6 switch . . . . .	74
6.2 Bucles . . . . .	76
6.2.1 for . . . . .	76
6.2.2 while . . . . .	76
6.2.3 do…while . . . . .	77
6.2.4 continue . . . . .	78
6.2.5 break . . . . .	79

## Contenido

---

6.2.6	Etiquetas . . . . .	80
6.2.7	for...in . . . . .	81
6.2.8	for...of . . . . .	83
6.3	Excepciones . . . . .	84
6.3.1	Tipos de excepciones . . . . .	84
6.3.2	try...catch . . . . .	85
6.3.3	Utilizando objetos de errores . . . . .	86
6.3.4	try...catch...finally . . . . .	87
6.3.5	throw . . . . .	88
<b>7</b>	<b>Funciones</b>	<b>89</b>
7.1	Declarar y llamar funciones . . . . .	89
7.2	Parámetros y argumentos . . . . .	89
7.3	Parámetros predeterminados . . . . .	91
7.4	Parámetros de descanso (rest) . . . . .	91
7.5	El objeto arguments . . . . .	92
7.6	Retorno . . . . .	93
7.7	Declaración y expresión de funciones . . . . .	94
7.8	Funciones anidadas . . . . .	96
7.9	Ámbito global y local . . . . .	97
7.10	Devolución de llamada (Callbacks) . . . . .	99
7.11	Elevación de variables y funciones (Hoisting) . . . . .	100
7.12	Recursividad . . . . .	102
7.13	Cierres (Closures) . . . . .	103
7.14	Funciones de flecha . . . . .	104
7.15	Funciones predefinidas . . . . .	105
<b>8</b>	<b>Uso avanzado de datos</b>	<b>106</b>
8.1	Uso de números . . . . .	106
8.1.1	Números decimales . . . . .	106
8.1.2	Números binarios . . . . .	106

## Contenido

---

8.1.3	Números octales . . . . .	107
8.1.4	Números hexadecimales . . . . .	108
8.1.5	Infiniy . . . . .	109
8.1.6	NaN . . . . .	109
8.1.7	Propiedades y métodos del objeto Number . . . . .	110
8.1.8	Propiedades comunes del objeto Number . . . . .	112
8.1.9	Propiedades y métodos del objeto Math . . . . .	115
8.1.10	Propiedades y métodos del objeto BigInt . . . . .	116
8.2	Uso de fechas . . . . .	118
8.2.1	Propiedades y métodos del objeto Date . . . . .	118
8.3	Uso de texto . . . . .	119
8.3.1	Secuencias de escape . . . . .	119
8.3.2	Convertir texto a números . . . . .	120
8.3.3	Propiedades y métodos del objeto String . . . . .	121
8.4	Uso de expresiones regulares . . . . .	123
8.4.1	Crear una expresión regular . . . . .	123
8.4.2	Búsqueda con banderas en expresiones regulares . . . . .	123
8.4.3	Propiedades y métodos del objeto RegExp . . . . .	124
8.5	Uso de listas . . . . .	126
8.5.1	Insertar elementos en una lista . . . . .	126
8.5.2	Modificar elementos de listas . . . . .	127
8.5.3	Borrar elementos de listas . . . . .	128
8.5.4	Propiedades y métodos del objeto Array . . . . .	129
8.5.5	Propiedades comunes de listas . . . . .	131
8.5.6	Métodos comunes de listas . . . . .	132
8.5.7	Arreglos multidimensionales . . . . .	140
8.5.8	Arreglos tipados (Int8Array, Uint8Array) . . . . .	141
8.5.9	Array buffer . . . . .	142
8.6	Uso de objetos . . . . .	144
8.6.1	Insertar elementos en un objeto . . . . .	144

## Contenido

---

8.6.2 Desestructuración . . . . .	145
8.6.3 Modificar propiedades de objetos . . . . .	146
8.6.4 Borrar propiedades de objetos . . . . .	147
8.6.5 Crear métodos . . . . .	148
8.6.6 Propiedades y métodos del objeto Object . . . . .	149
8.6.7 Métodos comunes de objetos . . . . .	151
<b>9 Programación orientada a objetos (POO)</b>	<b>151</b>
9.1 Paradigma . . . . .	151
9.2 Función constructora . . . . .	153
9.3 Contexto . . . . .	154
9.3.1 Contexto usando this . . . . .	154
9.3.2 Vinculación (call(), apply(), bind()) . . . . .	155
9.3.3 this en funciones flecha . . . . .	158
9.4 Prototipo . . . . .	159
9.5 Herencia prototípica . . . . .	161
9.6 Clases . . . . .	162
9.7 Declaración y expresión de clases . . . . .	163
9.8 Métodos de una instancia . . . . .	165
9.9 Campos públicos . . . . .	166
9.10 Campos privados . . . . .	167
9.11 Campos estáticos . . . . .	169
9.12 Captadores (getters) y establecedores (setters) . . . . .	170
9.13 Herencia . . . . .	172
<b>10 Asincronía</b>	<b>174</b>
10.1 Temporizadores . . . . .	174
10.1.1 setTimeout() y clearTimeout() . . . . .	174
10.1.2 setInterval() y clearInterval() . . . . .	175
10.2 Promesas . . . . .	176
10.2.1 Crear una promesa . . . . .	176

## Contenido

---

10.2.2 Encadenamiento de promesas . . . . .	178
10.2.3 Propagación de errores de promesas . . . . .	179
10.2.4 Fetch API . . . . .	180
10.2.5 Multiples promesas . . . . .	180
10.2.6 Propiedades y métodos del objeto Promise . . . . .	181
10.3 async/await . . . . .	183
<b>11 Módulos</b>	<b>184</b>
11.1 Módulos y herramientas . . . . .	184
11.1.1 Historia de los módulos . . . . .	184
11.1.2 CommonJS . . . . .	186
11.1.3 Webpack . . . . .	187
11.1.4 Babel . . . . .	188
11.1.5 Extensiones .mjs vs .js . . . . .	189
11.2 Exportar . . . . .	190
11.3 Importar . . . . .	192
11.4 Exportación predeterminada . . . . .	193
11.5 Alias de importación . . . . .	194
11.6 Importar paquetes . . . . .	195
11.7 Cargar módulos dinámicamente . . . . .	196
11.8 Ámbito de un módulo . . . . .	197
<b>12 Iteradores y generadores</b>	<b>198</b>
12.1 Objetos iteradores . . . . .	198
12.2 Objetos iterables . . . . .	199
12.3 Método de iterador next() . . . . .	200
12.4 Funciones generadoras . . . . .	201
12.5 yield . . . . .	202
<b>13 Internacionalización</b>	<b>203</b>
13.1 El objeto Intl . . . . .	203

## Contenido

---

13.2 El objeto DateFormat . . . . .	204
13.3 El objeto NumberFormat . . . . .	205
13.4 El objeto Collator . . . . .	206
<b>14 Meta</b>	<b>207</b>
14.1 El objeto Proxy . . . . .	207
14.2 El objeto Reflect . . . . .	208
<b>15 Siguientes pasos</b>	<b>209</b>
15.1 Herramientas . . . . .	209
15.2 Recursos . . . . .	210
15.3 ¿Que viene después? . . . . .	211
15.4 Preguntas de entrevista . . . . .	212

# 1 Introducción

## 1.1 Bienvenida

Bienvenid@ a este libro de Academia X en donde aprenderás JavaScript práctico.

Hola, mi nombre es Xavier Reyes Ochoa y soy el autor de este libro. He trabajado como consultor en proyectos para Nintendo, Google, entre otros proyectos top-tier, trabajé como líder de un equipo de desarrollo por 3 años, y soy Ingeniero Ex-Amazon. En mis redes me conocen como Programador X y comparto videos semanalmente en YouTube desde diversas locaciones del mundo con el objetivo de guiar y motivar a mis estudiantes mientras trabajo haciendo lo que más me gusta: la programación.

En este libro vas a aprender estos temas:

- Gramática
- Tipos y variables
- Operadores
- Estructuras de control de flujo
- Funciones
- Uso avanzado de datos
- Programación orientada a objetos (POO)
- Asincronía
- Módulos
- Iteradores y generadores
- Internacionalización
- Meta

La motivación de este libro es darte todo el conocimiento técnico que necesitas para elevar la calidad de tus proyectos y al mismo tiempo puedas perseguir metas más grandes, ya sea utilizar esta tecnología para tus pasatiempos creativos, aumentar tus oportunidades laborales, o si tienes el espíritu emprendedor, incluso crear tu

## 1 INTRODUCCIÓN

---

propio negocio en línea. Confío en que este libro te dará los recursos que necesitas para que tengas éxito en este campo.

Empecemos!

### 1.1.1 Libro vivo

Esta publicación fue planeada, editada, y revisada manualmente por Xavier Reyes Ochoa. La fundación del contenido fue generada parcialmente por inteligencia artificial usando ChatGPT (Feb 13 Version) de OpenAI. Puedes ver más detalles en <https://openai.com/>

Esto es a lo que llamo un trabajo **VIVO**, esto quiere decir que será actualizado en el tiempo a medida que existan cambios en la tecnología. La versión actual es 1.0.0 editada el 18 de abril de 2023. Si tienes correcciones importantes, puedes escribirnos a nuestra sección de contacto en <https://www.academia-x.com>

### 1.1.2 Alcance

El objetivo de este libro es llenar el vacío que existe sobre esta tecnología en Español siguiendo el siguiente enfoque:

- Se revizan los temas con un enfoque práctico incluyendo ejemplos y retos.
- Se evita incluir material de relleno ya que no es eficiente.
- Se evita entrar en detalle en temas simples o avanzados no-prácticos.

Si deseas profundizar en algún tema, te dejo varios recursos populares y avanzados en la lección de recursos como el estándar de esta tecnología (que puede ser difícil de leer si recién estás empezando).

## 1 INTRODUCCIÓN

---

### 1.2 Prerequisitos

Antes de aprender JavaScript, necesitas lo siguiente:

1. Un computador: cualquier computador moderno tiene las capacidades de correr este lenguaje. Te recomiendo un monitor de escritorio o laptop ya que dispositivos móviles o ipads no son cómodos para programar.
2. Sistema operativo: conocimiento básico de cómo utilizar el sistema operativo en el que se ejecutará JavaScript (por ejemplo, Windows, MacOS, Linux). Te recomiendo tener el sistema operativo actualizado.
3. Conocimiento básico de la línea de comandos: se utiliza para ejecutar programas en JavaScript.
4. Un editor de texto: lo necesitas para escribir código de JavaScript. Los editores de texto más populares son Visual Studio Code, Sublime Text, Atom y Notepad ++.
5. Un navegador web y conexión al internet: es útil para investigar más sobre esta tecnología cuando tengas dudas. Los navegadores más populares son Google Chrome, Mozilla Firefox, Safari y Microsoft Edge. Se recomienda tener el navegador actualizado.

Si ya tienes estos requisitos, estarás en una buena posición para comenzar a aprender JavaScript y profundizar en sus características y aplicaciones.

Si todavía no tienes conocimiento sobre algunos de estos temas, te recomiendo buscar tutoriales básicos en blogs a través de Google, ver videos en YouTube, o hacer preguntas a ChatGPT. Alternativamente, puedes empezar ya e investigar en línea a medida que encuentres bloqueos entiendo los conceptos en este libro.

### 1.3 ¿Cómo evitar bloqueos?

Para sacarle el mayor provecho a este libro:

## 2 PRIMEROS PASOS

---

1. No solo leas este libro. La práctica es esencial en este campo. Practica todos los días y no pases de lección hasta que un concepto esté 100% claro.
2. No tienes que memorizarlo todo, solo tienes que saber donde están los temas para buscarlos rápidamente cuando tengas dudas.
3. Cuando tengas preguntas usa [Google](#), [StackOverFlow](#), y [ChatGPT](#)
4. Acepta que en esta carrera, mucho de tu tiempo lo vas utilizar investigando e innovando, no solo escribiendo código.
5. No tienes que aprender inglés ahora pero considera aprenderlo en un futuro porque los recursos más actualizados están en inglés y también te dará mejores oportunidades laborales.
6. Si pierdas la motivación, recuerda tus objetivos. Ninguna carrera es fácil pero ya tienes los recursos para llegar muy lejos. Te deseo lo mejor en este campo!

## 2 Primeros pasos

### 2.1 Visión general

#### 2.1.1 ¿Qué es y porqué debes aprenderlo?

JavaScript es un lenguaje de programación de alto nivel y dinámico que se utiliza en el desarrollo web para agregar interactividad y dinamismo a las páginas web. JavaScript permite a los desarrolladores crear funcionalidades en las páginas web como formularios interactivos, animaciones, juegos, aplicaciones y mucho más.

Hay varias razones por las que deberías aprender JavaScript:

1. Es ampliamente utilizado: JavaScript es uno de los lenguajes de programación más populares y ampliamente utilizados en el mundo, lo que significa que hay una gran demanda de desarrolladores JavaScript.

## 2 PRIMEROS PASOS

---

2. Es versátil: JavaScript se utiliza tanto en el lado del cliente como en el lado del servidor, lo que significa que puedes desarrollar una amplia gama de aplicaciones y proyectos con él.
3. Es fácil de aprender: JavaScript es un lenguaje de programación fácil de aprender, especialmente para aquellos que ya tienen una comprensión básica de programación.
4. Te abre puertas: Aprender JavaScript te permitirá aprovechar nuevas oportunidades de carrera y te permitirá trabajar en una amplia gama de proyectos interesantes.

En resumen, JavaScript es un lenguaje de programación importante y versátil que te permitirá desarrollar una amplia gama de aplicaciones y proyectos. Aprender JavaScript puede ser una inversión valiosa en tu futuro profesional.

### 2.1.2 ¿En dónde se utiliza?

JavaScript se utiliza en una amplia variedad de contextos y plataformas, incluyendo:

1. **Desarrollo web:** JavaScript es un lenguaje de programación clave para el desarrollo web. Se utiliza para crear funcionalidades interactivas y dinámicas en las páginas web, como formularios interactivos, animaciones, galerías de imágenes, menús desplegables, juegos y mucho más.
2. **Aplicaciones web:** JavaScript se utiliza para crear aplicaciones web complejas y avanzadas, como herramientas de productividad, aplicaciones de negocios, plataformas de e-commerce y mucho más.
3. **Aplicaciones móviles:** JavaScript se utiliza en la creación de aplicaciones móviles mediante tecnologías como React Native y PhoneGap.
4. **Internet de las cosas (IoT):** JavaScript se utiliza para desarrollar aplicaciones IoT, que permiten a los dispositivos conectados a internet interactuar entre sí.

## 2 PRIMEROS PASOS

---

5. Desarrollo de servidor: JavaScript se utiliza en el lado del servidor para crear aplicaciones y servicios web. Con Node.js, puedes crear aplicaciones y servicios web complejos con JavaScript en el lado del servidor.

En resumen, JavaScript es un lenguaje de programación versátil que se utiliza en una amplia variedad de contextos y plataformas, desde el desarrollo web hasta las aplicaciones móviles y IoT.

### 2.1.3 ¿Qué trabajos puedes conseguir?

Aprender JavaScript puede abrirte la puerta a una amplia gama de oportunidades de carrera, incluyendo:

1. Desarrollador web front-end: Los desarrolladores web front-end utilizan JavaScript para crear y mejorar la experiencia de usuario en las páginas web.
2. Desarrollador web full-stack: Los desarrolladores web full-stack utilizan JavaScript en conjunto con otros lenguajes de programación para crear aplicaciones web complejas y avanzadas.
3. Desarrollador de aplicaciones móviles: Los desarrolladores de aplicaciones móviles utilizan JavaScript para crear aplicaciones móviles y hacerlas compatibles con diferentes plataformas.
4. Desarrollador de juegos: Los desarrolladores de juegos utilizan JavaScript para crear juegos interactivos y emocionantes para la web y dispositivos móviles.
5. Desarrollador de aplicaciones IoT: Los desarrolladores de aplicaciones IoT utilizan JavaScript para crear aplicaciones que permiten a los dispositivos conectados a internet interactuar entre sí.
6. Desarrollador de inteligencia artificial y machine learning: JavaScript se utiliza para crear aplicaciones y algoritmos de inteligencia artificial y machine learning.

En resumen, aprender JavaScript puede prepararte para una amplia gama de carreras en el campo de la tecnología, desde el desarrollo web hasta las aplicaciones

## 2 PRIMEROS PASOS

---

móviles y la inteligencia artificial.

### 2.1.4 ¿Cuánto puedes ganar?

El salario que puedes ganar al utilizar JavaScript en tu trabajo depende de varios factores, incluyendo tu nivel de experiencia, el tipo de trabajo que desempeñas, la ubicación geográfica y la industria en la que trabajes.

En general, los desarrolladores web con experiencia en JavaScript pueden esperar ganar salarios competitivos en comparación con otros profesionales de la tecnología. Según Glassdoor, el salario promedio de un desarrollador web en los Estados Unidos es de aproximadamente \$75,000 al año.

Es importante tener en cuenta que estos son solo promedios y que el salario real que puedes ganar puede ser mayor o menor, dependiendo de los factores mencionados anteriormente. Además, siempre es una buena idea investigar y hacer preguntas sobre los salarios y las condiciones laborales antes de aceptar un trabajo.

### 2.1.5 ¿Cuáles son las preguntas más comunes?

Algunas de las preguntas más comunes sobre JavaScript incluyen:

1. ¿Qué es JavaScript y cómo funciona?
2. ¿Qué se puede hacer con JavaScript?
3. ¿Cómo se compara JavaScript con otros lenguajes de programación?
4. ¿Cómo se instala y configura JavaScript en una computadora?
5. ¿Cómo se utiliza JavaScript para crear aplicaciones web y móviles?
6. ¿Cómo se integra JavaScript con otros lenguajes de programación y marcos de trabajo?
7. ¿Cómo se debuggear y solucionar errores en el código de JavaScript?
8. ¿Cuáles son las mejores prácticas para escribir código JavaScript eficiente y mantenible?

## 2 PRIMEROS PASOS

---

9. ¿Cómo se puede mejorar la seguridad de las aplicaciones de JavaScript?
10. ¿Qué recursos y comunidades pueden ayudar a aprender y mejorar en JavaScript?

Estas son solo algunas de las preguntas más comunes sobre JavaScript. Hay muchas más que pueden surgir a medida que avanzas en tu aprendizaje y desarrollo de aplicaciones con este lenguaje de programación.

Al finalizar este recurso, tendrás las habilidades necesarias para responder o encontrar las respuestas a estas preguntas.

### 2.2 Historia, evolución, y versiones

JavaScript es un lenguaje de programación interpretado, dinámico y orientado a objetos. Fue creado por Brendan Eich en 1995 para la empresa Netscape Communications Corporation. Originalmente fue diseñado para ser un lenguaje de scripting para la web, pero ahora se usa ampliamente en aplicaciones web, servidores y aplicaciones móviles.

La primera versión de JavaScript fue lanzada en 1996 como parte de Netscape Navigator 2.0. Desde entonces, ha habido varias versiones de JavaScript, cada una con nuevas características y mejoras. Las versiones más recientes incluyen ECMAScript 6 (ES6) y ECMAScript 7 (ES7).

Las versiones más antiguas de JavaScript se conocen como JavaScript 1.0 y JavaScript 1.1. Estas versiones fueron reemplazadas por JavaScript 1.2, que fue lanzada en 1997. Esta versión introdujo nuevas características como el soporte para objetos, la capacidad de crear objetos personalizados y la capacidad de crear funciones.

En 1999, se lanzó JavaScript 1.3, que introdujo mejoras en la velocidad de ejecución y la capacidad de crear objetos más complejos. Esta versión también introdujo el soporte para la programación orientada a objetos.

## 2 PRIMEROS PASOS

---

En el año 2015, se lanzó ECMAScript 6 que introdujo mejoras en la velocidad de ejecución y la capacidad de crear objetos más complejos. Esta versión también introdujo el soporte para la programación orientada a objetos.

Desde ES6, cada año se presenta una nueva versión con mejoras siendo la versión 14 para el año 2023.

### 2.3 Características y ventajas

JavaScript es un lenguaje de programación interpretado, orientado a objetos y multiplataforma. Está diseñado para crear aplicaciones web interactivas y dinámicas.

#### Características:

- Es un lenguaje de programación interpretado, lo que significa que no necesita ser compilado antes de su ejecución.
- Es orientado a objetos, lo que significa que los programas se escriben usando objetos y sus propiedades.
- Es multiplataforma, lo que significa que se puede ejecutar en diferentes sistemas operativos.
- Es compatible con la mayoría de los navegadores web, lo que significa que se puede ejecutar en la mayoría de los navegadores web.

#### Ventajas:

- Es fácil de aprender y usar.
- Es un lenguaje de programación versátil, lo que significa que se puede usar para crear aplicaciones web, aplicaciones de escritorio y aplicaciones móviles.
- Es un lenguaje de programación dinámico, lo que significa que los programas se pueden cambiar en tiempo de ejecución.
- Es un lenguaje de programación seguro, lo que significa que los programas se ejecutan en un entorno aislado.

## 2 PRIMEROS PASOS

---

- Es un lenguaje de programación escalable, lo que significa que se puede usar para crear aplicaciones de gran tamaño.

### 2.4 Diferencias con otros lenguajes de programación

JavaScript es un lenguaje de programación interpretado, lo que significa que el código se ejecuta directamente sin necesidad de compilarlo. Esto lo hace único entre los lenguajes de programación, ya que la mayoría de los lenguajes de programación requieren que el código se compile antes de que se pueda ejecutar.

Otra diferencia importante entre JavaScript y otros lenguajes de programación es que JavaScript es un lenguaje orientado a objetos. Esto significa que los programadores pueden crear objetos y usarlos para almacenar y manipular datos. Esto es diferente de otros lenguajes de programación, como C, que son lenguajes estructurados y no orientados a objetos.

### 2.5 Configuración

#### 2.5.1 IDE

Los archivos de JavaScript son archivos de texto. Puedes editarlos con **editores de texto** como Notepad en Windows o Notes en MacOS pero es recomendado utilizar un **IDE** (Integrated Development Environment) que es una aplicación de edición de código más avanzado que le da colores a tu código para que sea más fácil de leer y tengas funciones de autocompletado, entre otras. Algunos IDEs populares son [Brackets](#), [Atom](#), [Sublime Text](#), [Vim](#), y [Visual Studio Code](#).

El editor recomendado para practicar el código que vamos a ver es Visual Studio Code (o VSCode) que puedes bajar desde <https://code.visualstudio.com/>

## 2 PRIMEROS PASOS

---

### 2.5.2 Entorno

Para usar JavaScript en tu computador, no necesitas instalar ningún software especial, ya que JavaScript es un lenguaje interpretado y se ejecuta en el navegador web. Sin embargo, hay algunas cosas que debes tener en cuenta:

1. Navegador web: es recomendable tener un navegador web actualizado, como Google Chrome, Mozilla Firefox, Safari, etc.
2. Editor de texto: puedes usar un editor de texto sencillo como el Bloc de notas en Windows o TextEdit en Mac, pero también puedes usar un editor de código más avanzado como Visual Studio Code, Sublime Text, etc.
3. Conocimiento básico de HTML y CSS: ya que JavaScript se utiliza para agregar dinamismo y interactividad a las páginas web, es importante tener una comprensión básica de HTML y CSS.

Una vez que tengas estos elementos, estarás list@ para escribir y ejecutar código JavaScript en tu navegador. Puedes crear un archivo HTML y agregar tu código JavaScript dentro de una etiqueta `<script>` o puedes escribir código JavaScript directamente en la consola del navegador.

## 2.6 Hola Mundo

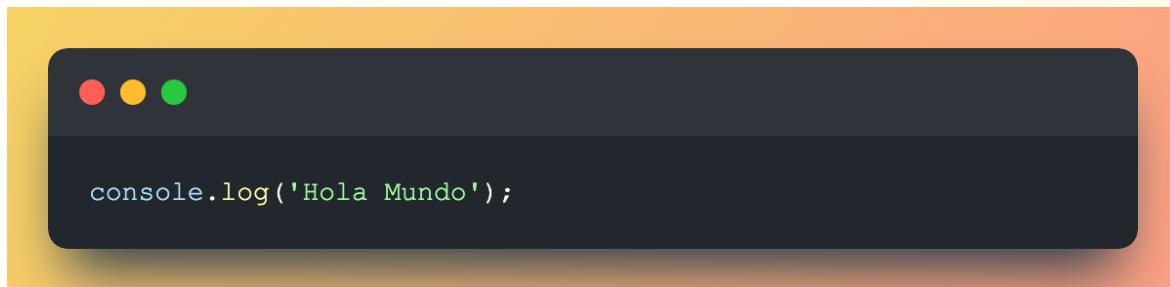
“Hola Mundo” es un ejemplo clásico que se utiliza para mostrar el funcionamiento básico de un lenguaje de programación.

Ejemplo:

En este ejemplo, se imprime el texto “Hola Mundo” en la consola.

### 3 GRAMÁTICA

---



También podrías modificar este código con tu nombre. Si es “Juan”, debería imprimir en la consola “Hola, Juan” .

Reto:

Modifica el ejemplo anterior para imprimir “Hola Universo” en la consola.

## 3 Gramática

### 3.1 Sintaxis

La sintaxis de un lenguaje de programación es la estructura de un lenguaje de programación, que incluye reglas para la construcción de programas. Estas reglas se refieren a la forma en que los elementos de un programa se relacionan entre sí, como los operadores, variables, palabras clave, etc. La sintaxis también se refiere a la forma en que los programas se escriben, como la indentación y la estructura de los bloques de código.

### 3.2 Comentarios (una sola línea y multilínea)

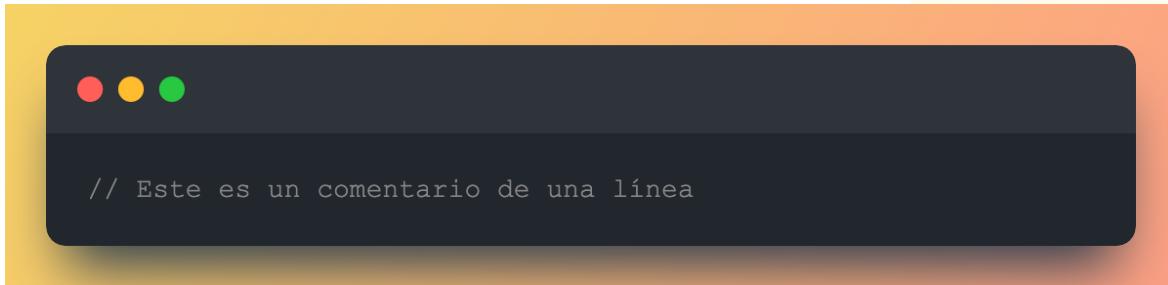
Los comentarios en JavaScript son líneas de texto que no son interpretadas como parte del código. Son utilizados para agregar información adicional o notas sobre el código que puede ser útil para otros desarrolladores o para uno mismo.

### 3 GRAMÁTICA

---

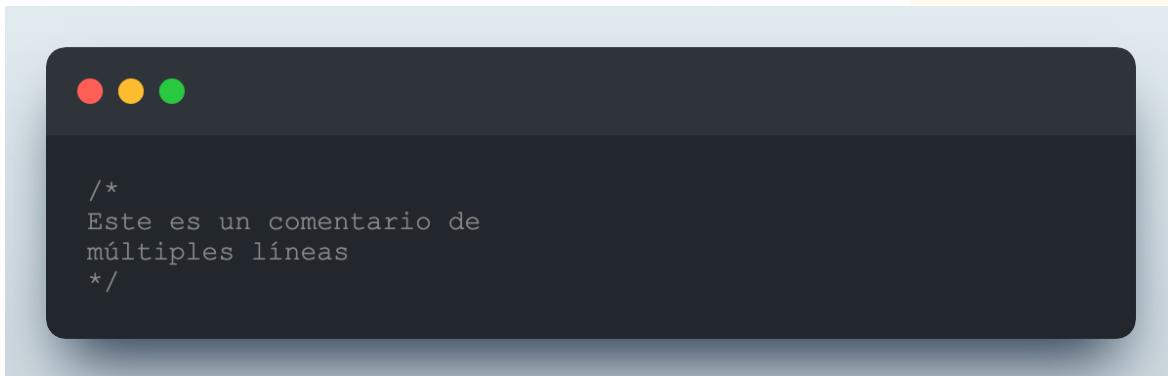
Hay dos formas de agregar comentarios en JavaScript:

1. Comentarios de una línea: Se pueden crear comentarios de una sola línea usando dos barras (//). Por ejemplo:



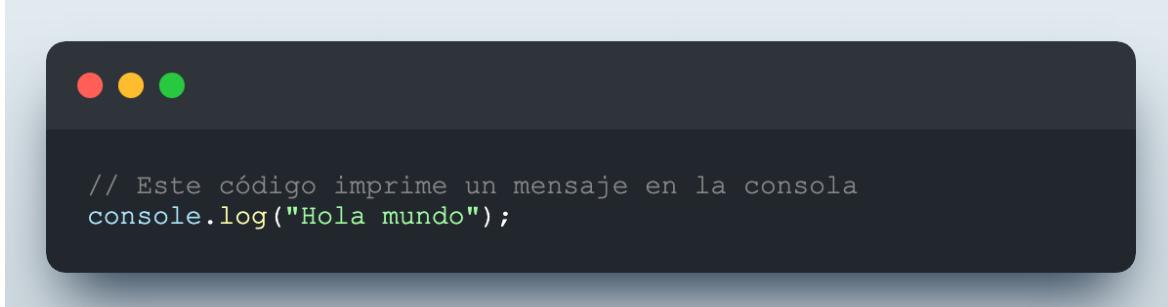
```
// Este es un comentario de una línea
```

1. Comentarios de múltiples líneas: Se pueden crear comentarios de múltiples líneas usando /\* \*/. Por ejemplo:



```
/*
Este es un comentario de
múltiples líneas
*/
```

Ejemplo:



```
// Este código imprime un mensaje en la consola
console.log("Hola mundo");
```

### 3 GRAMÁTICA

---

Reto:

Comenta este código para que ya no corra y se lea como comentario:



```
console.log("Esta línea de código ya no es necesaria.");
```

#### 3.3 Literales y tipos de datos

En JavaScript, los literales son valores que se escriben directamente en el código y no pueden ser modificados. Algunos de los tipos de literales más comunes en JavaScript son **números, texto, booleanos, objetos y arreglos**.

Ejemplo:



```
// Número  
30  
  
// Texto  
"Juan"  
  
// Booleano  
false  
  
// Arreglo o lista  
["leer", "nadar", "viajar"]
```

### 3 GRAMÁTICA

---

En este ejemplo, se utilizan literales que representan diferentes valores.

Reto:

Crea un arreglo que represente una lista de personas y muestra este arreglo en la consola.

#### 3.4 Operadores y expresiones

Los operadores y las expresiones en JavaScript son una parte importante de la programación. Una expresión es una combinación de valores, variables y operadores que se evalúan para producir un resultado. Los operadores son los símbolos que se usan para realizar operaciones matemáticas, lógicas y de comparación.

Ejemplo:

Por ejemplo, consideremos la siguiente expresión:



En esta expresión, 5 y 3 son los valores, = y + son los operadores y x es una variable. Esta expresión se evalúa como 8 y se guarda en una variable.

Reto:

Usa los operadores aritméticos para realizar las siguientes operaciones: suma, resta, multiplicación y división entre 10 y 2. Imprime los resultados de las operaciones en la consola.

### 3 GRAMÁTICA

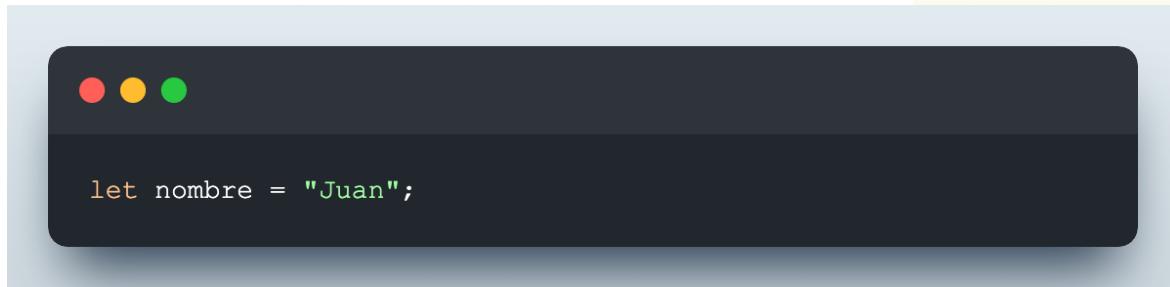
---

#### 3.5 Palabras clave e identificadores

Las palabras clave en JavaScript son términos reservados que tienen un significado específico en el lenguaje y no pueden ser utilizados como identificadores. Algunas de las palabras clave más comunes en JavaScript son **var**, **function**, **if**, **else**, **for**, **while**, **do**, **switch**, etc.

Un identificador, por otro lado, es un nombre que le das a una variable, función o objeto en JavaScript. Por ejemplo, podrías crear una variable llamada **nombre** y asignarle un valor:

Ejemplo:



```
● ● ●
let nombre = "Juan";
```

Aquí, **let** es una palabra clave al igual que **var** para crear una variable y **nombre** es un identificador que está asociado con el valor “Juan”. **var** y **let** tienen algunas diferencias que vas a aprender después.

Reto:

Escribe un programa que multiplique 3 y 2 y guarde el resultado en una variable usando **var**. Luego, imprime la variable en la consola.

#### 3.6 Sentencias, declaraciones, y tipado

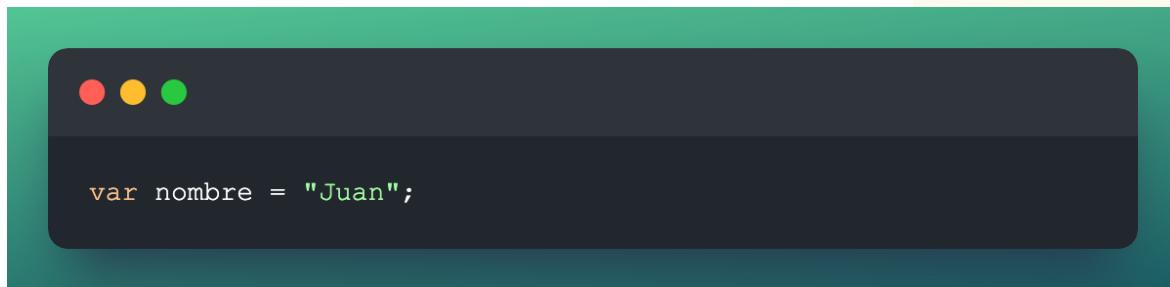
Una sentencia es una unidad de código que realiza una acción. Por ejemplo, la sentencia `console.log("Hola mundo")` imprime el texto “Hola mundo” en la pantalla.

### 3 GRAMÁTICA

---

Las declaraciones en JavaScript son sentencias (oraciones o frases) de código que se utilizan para realizar una tarea específica. Por ejemplo, una declaración puede ser la creación de una variable, la asignación de un valor a una variable, etc. Las declaraciones finalizan con punto y coma.

Ejemplo:



En este ejemplo, se ha creado una variable llamada “nombre” y se le ha asignado el valor “Juan” .

El tipado en JavaScript es el proceso de asignar un tipo de datos a una variable. Por ejemplo, la declaración `let x = 5` asigna el tipo de dato entero a la variable `x`. JavaScript no es un lenguaje tipado por lo que el tipo es inferido implícitamente. Si deseas usar JavaScript como lenguaje tipado puedes revisar otro lenguaje que se llama TypeScript en un futuro.

Reto:

Crea 3 variables diferentes (`x`, `y`, `z`) y asígnales los valores de 10, “Hola Mundo” y `true`. Luego, imprime los valores de esas variables en la consola.

### 3.7 Bloques e indentación

Los bloques en JavaScript son una forma de agrupar código para que sea más fácil de leer y mantener. Esto se logra mediante la indentación, que es el proceso de alinear el código para que sea más fácil de leer.

### 3 GRAMÁTICA

---

Los bloques se definen usando llaves {} y se usan para agrupar código relacionado. Por ejemplo:



En este ejemplo, todo el código dentro de las llaves {} se considera parte del bloque de la función saludar(). Vamos a ver funciones más adelante.

La indentación se usa para hacer que el código sea más fácil de leer. Por ejemplo, en el ejemplo anterior, todas las líneas de código dentro del bloque de la función saludar() están indentadas para indicar que forman parte del mismo bloque.

Reto:

Intenta escribir una función que imprima un saludo y una pregunta en la consola. Asegúrate de usar bloques y indentación correctamente.

### 3.8 Estructuras de control de flujo y excepciones

Las estructuras de control de flujo son una parte importante de cualquier lenguaje de programación, y JavaScript no es una excepción. Estas estructuras nos permiten controlar el flujo de ejecución de nuestro código, permitiéndonos ejecutar ciertas partes del código solo si se cumplen ciertas condiciones.

Las estructuras de control de flujo más comunes en JavaScript son:

- if: Esta estructura nos permite ejecutar un bloque de código si una condición es verdadera.

### 3 GRAMÁTICA

---

```
let edad = 18;  
  
if (edad >= 18) {  
    console.log("Eres mayor de edad");  
}
```

Las excepciones o errores son otra parte importante de JavaScript. Estas nos permiten controlar errores en nuestro código, permitiéndonos ejecutar un bloque de código si se produce un error. Esto nos permite evitar que nuestro código se detenga si se produce un error.

Las excepciones en JavaScript se manejan con la palabra clave try/catch.

```
try {  
    // Código que puede lanzar una excepción  
} catch (e) {  
    // Código que se ejecuta si se produce una excepción  
}
```

Reto:

Crea un programa que muestre un mensaje diferente dependiendo de si es mayor o menor de edad.

### 3 GRAMÁTICA

---

#### 3.9 Conjunto de caracteres

JavaScript es un lenguaje de programación que utiliza el conjunto de caracteres Unicode, que es un conjunto de caracteres amplio y universal que incluye caracteres de muchos idiomas y símbolos. Esto significa que puede trabajar con caracteres en muchos idiomas y símbolos diferentes, lo que lo hace ideal para desarrollar aplicaciones que deben ser utilizadas por personas de diferentes partes del mundo.

Ejemplo:

Por ejemplo, puedes usar la función “charAt()” para acceder a un carácter específico en una cadena:

```
var saludo = ";Hola mundo!";
console.log(saludo.charAt(0)); // ;
```

Reto:

Añade un emoji de “rostro feliz” (que es un carácter de Unicode) a la variable “happy”

#### 3.10 Sensibilidad de mayúsculas y minúsculas

JavaScript es un lenguaje de programación case-sensitive, lo que significa que distingue entre mayúsculas y minúsculas en el nombre de variables, funciones y palabras clave. Esto significa que “HolaMundo” y “holamundo” son considerados dos nombres de variables diferentes.

Ejemplo:

## 4 TIPOS Y VARIABLES

---



```
var HolaMundo = "Hola Mundo";
console.log(HolaMundo);

var holamundo = "Adiós Mundo";
console.log(holamundo);
```

En este ejemplo, se declaran dos variables con nombres diferentes debido a su sensibilidad a mayúsculas y minúsculas. Cuando se imprimen en la consola, se muestran los valores asignados a cada una de las variables.

Reto:

Crea dos variables con nombres diferentes debido a la sensibilidad a mayúsculas y minúsculas y asígnale valores diferentes 10 y 100. La primera variable es 'elefante' y la segunda es la misma palabra en mayúsculas. Imprime los valores de ambas variables en la consola.

## 4 Tipos y variables

### 4.1 Tipos de datos primitivos

Los tipos de datos primitivos en JavaScript son:

- **Números:** Estos son números enteros o decimales, como por ejemplo: 3, 3.14, -5.
- **Cadenas de texto:** Estas son secuencias de caracteres entre comillas, como por ejemplo: "Hola mundo", "Esto es una cadena de texto".

## 4 TIPOS Y VARIABLES

---

- **Booleanos:** Estos son valores lógicos que pueden ser true o false, como por ejemplo: true, false.
- **Valores nulos:** Estos son valores especiales que representan la ausencia de un valor, como por ejemplo: null.
- **Valores indefinidos:** Estos son valores especiales que representan un valor desconocido, como por ejemplo: undefined.

Ejemplo:



```
● ● ●

let numero = 3;
let texto = "Hola mundo";
let booleano = true;
let nulo = null;
let indefinido = undefined;
```

Reto:

Crea una variable llamada edad y asígnale un valor numérico de 25. Luego, crea una variable llamada nombre y asígnale un valor de cadena de texto “Juan”. Finalmente, crea una variable llamada estaRegistrado y asígnale un valor booleano true.

### 4.2 Tipos de datos no-primitivos

Los **tipos de datos no primitivos** en JavaScript son aquellos que no son números, cadenas de texto, booleanos, ni valores especiales como null o undefined. Estos tipos de datos son objetos, arreglos, funciones y expresiones regulares.

Ejemplo:

## 4 TIPOS Y VARIABLES

---

Por ejemplo, podemos crear una lista de compañías:

```
let lista = ['Apple', 'Microsoft', 'Amazon'];
```

Reto:

Crea una lista que contenga: mazanas, naranjas, y piñas.

### 4.3 Primitivos

#### 4.3.1 Texto

El texto en JavaScript es un tipo de datos primitivo y se representa mediante la utilización de comillas simples o dobles.

Ejemplo:

```
var nombre = "Juan";
var saludo = 'Hola';
```

Puedes concatenar dos o más cadenas de texto utilizando el operador de adición (+). Por ejemplo:

## 4 TIPOS Y VARIABLES

---

```
var saludoCompleto = saludo + ', ' + nombre;
console.log(saludoCompleto); // imprime "Hola, Juan"
```

Reto:

Crea una variable “a” con 33 como texto.

### 4.3.2 Número

JavaScript es un lenguaje de programación que permite trabajar con números. Estos números pueden ser enteros (como 1, 2, 3, etc.), decimales (como 0.1, 0.2, 0.3, etc.) o números en notación científica (como 1e2, 1e3, 1e4, etc.).

Ejemplo:

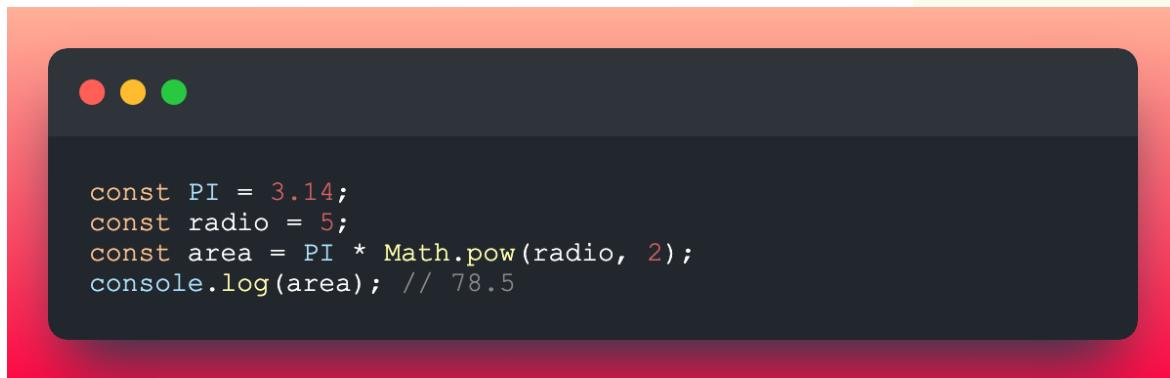
Supongamos que queremos calcular el área de un círculo con un radio de 5. Podemos usar la siguiente fórmula para calcular el área:

$$\text{Area} = \pi * \text{radio}^2$$

En JavaScript, podemos escribir el código para calcular el área de la siguiente manera:

## 4 TIPOS Y VARIABLES

---



Reto:

Escribe un programa en JavaScript que calcule el área de un rectángulo con una base de 10 y una altura de 5.

### 4.3.3 Entero grande

Entero grande en JavaScript es una forma de representar números enteros de un tamaño mayor que el que se puede representar con el tipo de datos nativo Number. Esto se logra mediante la creación de un objeto BigInt, que permite representar números enteros de hasta  $2^{53} - 1$ .

Ejemplo:

Supongamos que queremos representar el número entero  $2^{53}$ . Esto no se puede hacer con el tipo de datos nativo Number, ya que el límite es  $2^{53} - 1$ . Para representar este número, podemos usar un objeto BigInt:

## 4 TIPOS Y VARIABLES

---



```
let numero = BigInt(2) ** BigInt(53);
console.log(numero); // 9007199254740992n
```

Reto:

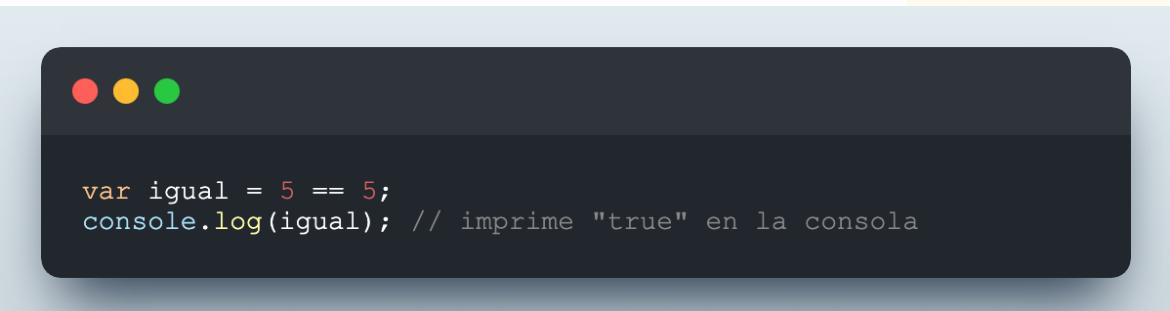
Crea un programa que devuelva el resultado de la suma de los dos números 987654321000 y 987654321001 usando BigInt.

### 4.3.4 Buleano

En JavaScript, un booleano es un tipo de dato que solo puede tener dos valores: true o false. Estos valores se utilizan para representar una condición verdadera o falsa.

Ejemplo:

El siguiente código determina si el número 5 es igual a 5 y almacena el resultado en una variable booleana:



```
var igual = 5 == 5;
console.log(igual); // imprime "true" en la consola
```

Reto:

## 4 TIPOS Y VARIABLES

---

Crea una variable ‘puedelIngresar’ que almacene en un valor booleano indicando que el usuario no está permitido de ingresar a una app.

### 4.3.5 Indefinido

En JavaScript, el valor indefinido se usa para indicar que una variable no tiene un valor asignado. Esto significa que la variable no se ha inicializado o que se ha eliminado una propiedad de un objeto.

Ejemplo:

A screenshot of a terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top. The terminal window contains the following code:

```
let x;
console.log(x); // imprime "undefined"
```

En el ejemplo anterior, la variable x no tiene un valor asignado, por lo que se imprime undefined.

Reto:

Crea una variable indefinida llamada sumar e imprime la variable que debe devolver undefined.

### 4.3.6 Símbolo

Un símbolo en JavaScript es un tipo de dato primitivo que representa un valor único e inmutable. Es decir, no puede ser cambiado. Estos símbolos se utilizan para crear identificadores únicos para propiedades de objetos y son tipo es poco utilizados en práctica. Vamos a ver más sobre objetos después.

## 4 TIPOS Y VARIABLES

---

Ejemplo:

```
// Crear un símbolo  
const miSimbolo = Symbol();  
  
// Imprimir símbolo  
console.log(miSimbolo);
```

Reto:

Crea un símbolo y una cadena de texto. Luego, imprime cada uno.

### 4.3.7 Nulo

Nulo en JavaScript es un valor especial que se utiliza para indicar que una variable no tiene un valor asignado. Esto es distinto de undefined ya que si es definido pero tiene valor nulo. Se representa con la palabra clave null.

Ejemplo:

```
let miVariable = null;  
console.log(miVariable); // Imprime null
```

Reto:

## 4 TIPOS Y VARIABLES

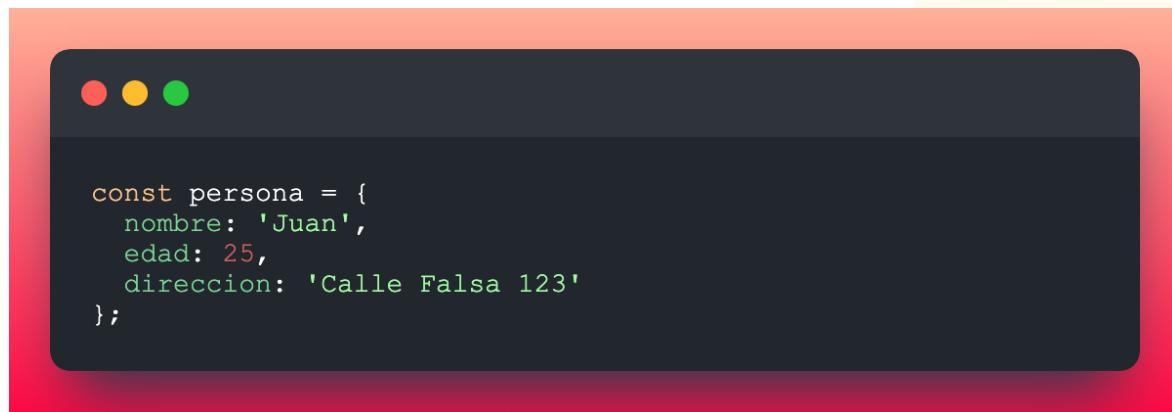
---

Crea una variable llamada miNombre y asígnale el valor null. Luego, imprime el valor de la variable en la consola.

### 4.4 Objetos

Los objetos en JavaScript son una forma de almacenar y organizar datos. Estos datos se almacenan en forma de pares clave-valor, donde la clave es una cadena de texto y el valor puede ser cualquier tipo de dato.

Un ejemplo práctico de un objeto en JavaScript es el siguiente:



```
● ● ●

const persona = {
  nombre: 'Juan',
  edad: 25,
  direccion: 'Calle Falsa 123'
};
```

Reto:

Crea un objeto en JavaScript que represente una computadora, con sus respectivas características (marca, modelo, procesador, memoria RAM, etc).

### 4.5 Acceder a propiedades de objetos

Acceder en JavaScript es una forma de obtener información de un objeto. Esto se puede hacer usando la notación de punto o la notación de corchetes.

Ejemplo:

## 4 TIPOS Y VARIABLES

---

Supongamos que tenemos un objeto llamado persona con los siguientes atributos:

```
const persona = {  
    nombre: 'Juan',  
    edad: 25,  
    direccion: {  
        calle: 'Calle Falsa 123',  
        ciudad: 'Ciudad Real'  
    }  
};
```

Podemos acceder a los atributos del objeto usando la notación de punto:

```
console.log(persona.nombre); // 'Juan'  
console.log(persona.edad); // 25  
console.log(persona.direccion.calle); // 'Calle Falsa 123'  
console.log(persona.direccion.ciudad); // 'Ciudad Real'
```

También podemos acceder a los atributos del objeto usando la notación de corchetes:

## 4 TIPOS Y VARIABLES

---



```
console.log(persona['nombre']); // 'Juan'
console.log(persona['edad']); // 25
console.log(persona['direccion']['calle']); // 'Calle Falsa
123'
console.log(persona['direccion']['ciudad']); // 'Ciudad Real'
```

Reto:

Crea un objeto llamado libro con los siguientes atributos: título, autor, año de publicación y editorial. Usa la notación de punto y la notación de corchetes para acceder a los atributos del objeto y mostrar los valores en la consola.

### 4.6 Colecciones con llave

#### 4.6.1 Mapas

Los mapas en JavaScript son una forma de representar datos en una estructura de clave-valor. Esto significa que cada elemento de datos se asigna a una clave única, y luego se puede acceder a los datos a través de la clave. Esto es útil para almacenar y recuperar datos de manera eficiente.

Ejemplo:

Supongamos que queremos almacenar los nombres de los estudiantes de una clase y sus calificaciones. Podemos usar un mapa para hacer esto de la siguiente manera:

## 4 TIPOS Y VARIABLES

---

```
const calificaciones = new Map();

calificaciones.set("El Mafla", 90);
calificaciones.set("Marcos", 85);
calificaciones.set("Jara", 80);

console.log(calificaciones.get("Jara")); // 90
```

Reto:

Crea un mapa que almacene los nombres de los estudiantes y sus edades. Luego, imprime la edad de un estudiante específico.

### 4.6.2 Sets

Los Sets en JavaScript son una estructura de datos que almacena valores únicos. Esto significa que no puede haber dos elementos iguales dentro de un Set. Esto los hace útiles para almacenar conjuntos de datos sin duplicados.

Ejemplo:

Supongamos que queremos almacenar los nombres de los estudiantes de una clase. Podemos usar un Set para almacenar los nombres de los estudiantes sin duplicados.

## 4 TIPOS Y VARIABLES

---

```
let estudiantes = new Set();

estudiantes.add("Viviana");
estudiantes.add("Verónica");
estudiantes.add("Isabel");
estudiantes.add("Viviana");

console.log(estudiantes); // Set { 'Viviana', 'Verónica',
'Isabel' }
```

Reto:

Crea un Set que almacene los nombres de los estudiantes de una clase y luego agrega los nombres de los estudiantes que faltan. Luego, imprime el Set para verificar que los nombres se hayan agregado correctamente.

### 4.6.3 WeakMaps

WeakMaps en JavaScript son una estructura de datos que se utiliza para almacenar pares clave-valor. Esta estructura de datos es similar a un Map, pero con la diferencia de que los objetos usados como claves deben ser objetos débiles. Esto significa que los objetos usados como claves no se contarán para el recuento de referencias, lo que significa que no se mantendrán en memoria. Esto hace que los WeakMaps sean útiles para evitar fugas de memoria.

Ejemplo:

Supongamos que queremos almacenar algunos datos relacionados con un usuario en un WeakMap. Podemos hacer esto de la siguiente manera:

## 4 TIPOS Y VARIABLES

---

```
let datosDeUsuario = new WeakMap();

let usuario = { nombre: 'Pedro' };

datosDeUsuario.set(usuario, { edad: 30 });

console.log(datosDeUsuario.get(usuario)); // { age: 30 }
```

En este ejemplo, creamos un WeakMap llamado datosDeUsuario y luego creamos un objeto usuario con un nombre. Luego, usamos el método set para almacenar algunos datos relacionados con el usuario en el WeakMap. Finalmente, usamos el método get para recuperar los datos almacenados.

Reto:

Crea un WeakMap y almacena algunos datos relacionados con un usuario. Luego, usa el método get para recuperar los datos almacenados.

### 4.6.4 WeakSets

WeakSets en JavaScript son una colección de objetos únicos que no se pueden iterar. Esto significa que no se pueden usar bucles para recorrerlos. Esto los hace útiles para almacenar objetos que no se quieren mantener en memoria por mucho tiempo.

Ejemplo:

Supongamos que queremos almacenar objetos de usuario en un WeakSet. Esto nos permitirá mantener una referencia a los usuarios sin mantenerlos en memoria por mucho tiempo.

## 4 TIPOS Y VARIABLES

---

```
let usuarios = new WeakSet();

let usuarios1 = {nombre: 'Sara', edad: 20};
let usuarios2 = {nombre: 'Jimmy', edad: 25};

usuarios.add(usuarios1);
usuarios.add(usuarios2);

console.log(usuarios.has(usuarios1)); // true
console.log(usuarios.has(usuarios2)); // true
```

Reto:

Crea un WeakSet que almacene objetos de usuario y luego usa el método has() para verificar si un usuario específico está en el WeakSet.

### 4.7 Listas

Las listas en JavaScript son una estructura de datos que nos permite almacenar una colección de elementos. Estos elementos pueden ser de cualquier tipo, desde números hasta objetos. También los vas a encontrar con otros nombres como arreglos, matrices, arrays, etc.

Ejemplo:

A continuación se muestra un ejemplo de una lista en JavaScript:

## 4 TIPOS Y VARIABLES

---

```
let lista = [1, 2, 3, 4, 5];
```

En este ejemplo, la lista contiene cinco elementos enteros.

Reto:

Crea una lista en JavaScript que contenga los nombres de los meses del año.

### 4.8 Acceder a elementos de listas

Acceder a elementos de listas en JavaScript es una tarea sencilla. Para acceder a un elemento de una lista, simplemente necesitamos conocer el índice del elemento que queremos acceder. El índice de un elemento de una lista comienza en 0.

Ejemplo:

```
let lista = ["manzana", "plátano", "naranja"];
console.log(lista[1]); // plátano
```

Reto:

Crea una lista con 5 elementos y accede al tercer elemento de la lista.

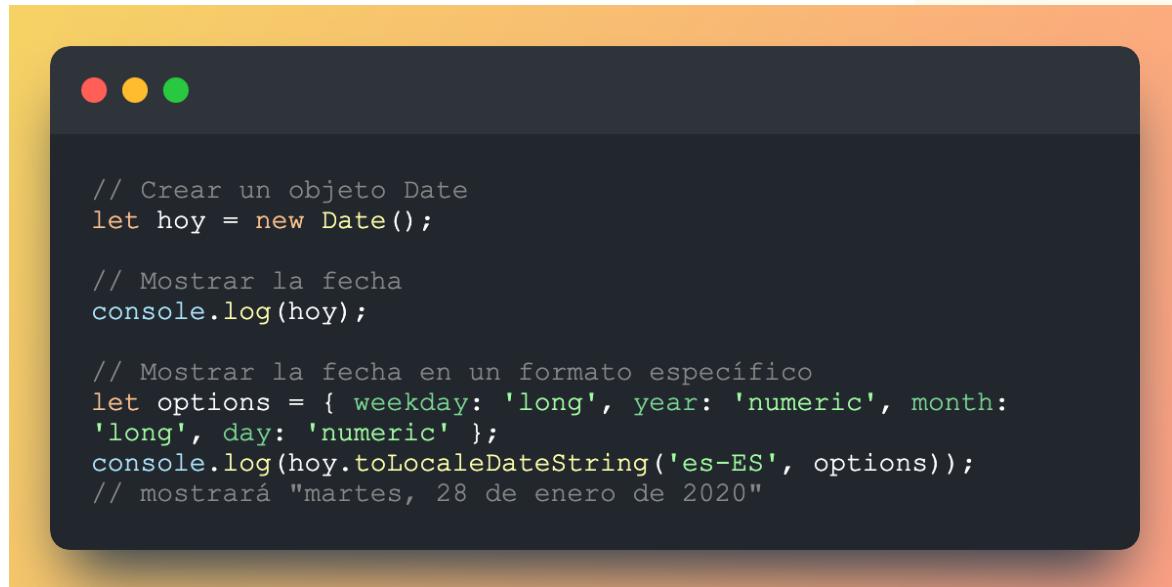
## 4 TIPOS Y VARIABLES

---

### 4.9 Fechas

JavaScript proporciona un objeto de fecha incorporado que permite trabajar con fechas y horas. El objeto Date se usa para representar una fecha, y se crea con el constructor Date().

Ejemplo:



```
// Crear un objeto Date
let hoy = new Date();

// Mostrar la fecha
console.log(hoy);

// Mostrar la fecha en un formato específico
let options = { weekday: 'long', year: 'numeric', month:
  'long', day: 'numeric' };
console.log(hoy.toLocaleDateString('es-ES', options));
// mostrará "martes, 28 de enero de 2020"
```

Reto:

Crea una fecha y cambia el formato removiendo “year: ‘numeric’ ” del ejemplo.

### 4.10 Expresiones regulares

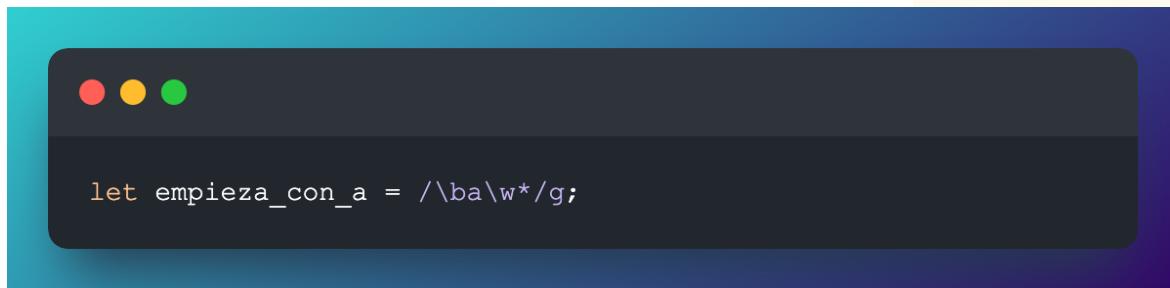
Las expresiones regulares son una secuencia de caracteres que forman un patrón de búsqueda. Estas son usadas para buscar y reemplazar patrones dentro de cadenas de texto. En JavaScript, las expresiones regulares son objetos, y se crean usando la notación literal / para abrir y cerrar la expresión o el constructor RegExp.

Ejemplo:

## 4 TIPOS Y VARIABLES

---

Para buscar todas las palabras que comienzan con la letra “a” en una cadena de texto, se puede usar la siguiente expresión regular:



Esta expresión regular buscará todas las palabras que comienzan con la letra “a” en una cadena de texto.

Las expresiones regulares requieren su propio curso ya que usan patrones bastante complejos pero miraremos algunos ejemplos más adelante.

Reto:

Crea una expresión regular para buscar todas las palabras que comienzan con la letra “b” en una cadena de texto.

### 4.11 Declaracion e inicialización (var y let)

var es una palabra clave en JavaScript que se utiliza para declarar variables. Una variable es un contenedor para almacenar datos. Esto significa que una variable puede cambiar su valor a lo largo del tiempo.

let es una palabra clave en JavaScript que se usa para declarar variables. Esta palabra clave es similar a la palabra clave var, pero con una diferencia importante: let limita el alcance de la variable a la declaración de bloque en la que se encuentra. Esto significa que la variable solo estará disponible dentro del bloque de código en el que se declaró.

Ejemplo:

## 4 TIPOS Y VARIABLES

---

```
// Usando var  
var x = 10;  
if (x > 5) {  
    var x = 20;  
    console.log(x); // 20  
}  
console.log(x); // 20  
  
// Usando let  
let y = 10;  
if (y > 5) {  
    let y = 20;  
    console.log(y); // 20  
}  
console.log(y); // 10
```

Reto:

Crea un programa que use la palabra clave let para declarar dos variables, una dentro de un bloque de código y otra fuera del bloque de código. Luego, imprime el valor de cada variable en la consola.

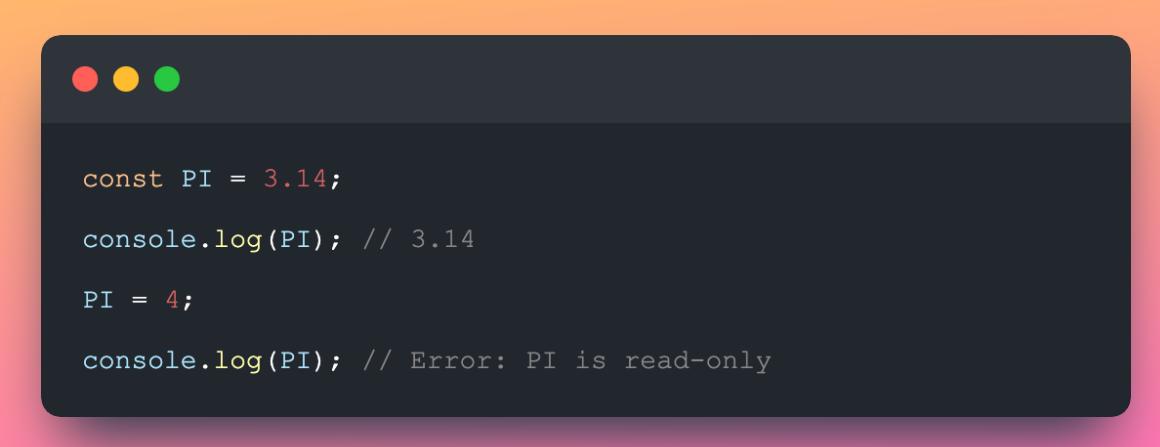
### 4.12 Constantes

Las constantes en JavaScript son variables que no pueden ser reasignadas. Esto significa que una vez que se asigna un valor a una constante, este no puede ser cambiado.

Ejemplo:

## 4 TIPOS Y VARIABLES

---



```
const PI = 3.14;
console.log(PI); // 3.14
PI = 4;
console.log(PI); // Error: PI is read-only
```

Reto:

Crea una constante llamada SALUDO y asígnale el valor "Hola Planeta". Luego, imprime el valor de la constante en la consola.

### 4.13 Plantillas de texto

Las plantillas de texto en JavaScript son una forma de crear cadenas de texto con una sintaxis especial. Estas plantillas permiten la interpolación de variables y expresiones dentro de la cadena de texto. Esto significa que podemos insertar variables y expresiones dentro de una cadena de texto sin tener que concatenar cadenas de texto.

Ejemplo:

Supongamos que queremos imprimir un saludo personalizado para un usuario. Podemos usar una plantilla de texto para hacer esto:

## 4 TIPOS Y VARIABLES

---

```
let userName = 'John';
console.log(`Hola, ${userName}! ¿Cómo estás?`);
// Imprime: Hola, John! ¿Cómo estás?
```

Reto:

Crea una plantilla de texto que imprima una frase con un nombre, un adjetivo y una comida. Por ejemplo: “John es un chico hambriento que ama la pizza” .

### 4.14 Chequear tipo

Chequear el tipo de una variable en JavaScript es una tarea común que se realiza para asegurar que los datos sean del tipo correcto. Esto se puede hacer usando la palabra `typeof` de JavaScript. Esta palabra devuelve una cadena que indica el tipo de la variable.

Ejemplo:

```
let numero = 5;
console.log(typeof numero); // Devuelve "number"
```

Reto:

Reemplaza el ejemplo usando todos los tipos de datos que has visto hasta ahora.

## 4 TIPOS Y VARIABLES

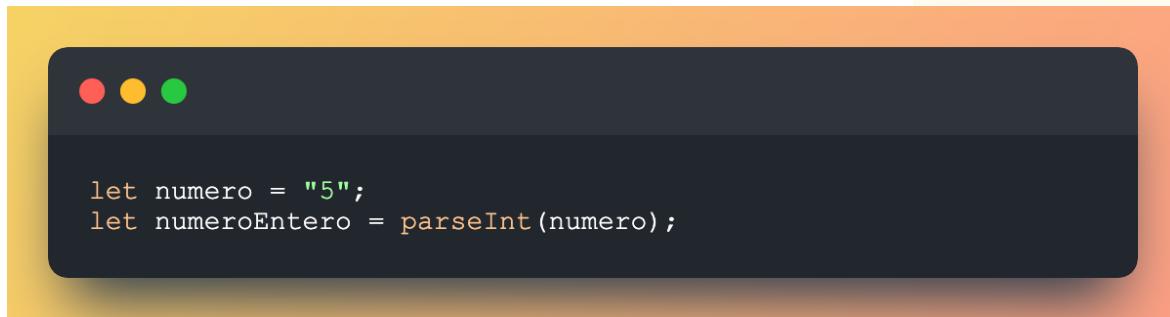
---

### 4.15 Conversión de tipos de datos

JavaScript es un lenguaje de programación dinámico, lo que significa que los tipos de datos de una variable pueden cambiar durante la ejecución del programa. Esto significa que, a veces, es necesario convertir un tipo de datos a otro para que el programa funcione correctamente.

Ejemplo:

Supongamos que tenemos una variable llamada `numero` que contiene el valor "5". Esto es una cadena de caracteres, pero necesitamos convertirlo a un número para poder realizar operaciones matemáticas con él. Para hacer esto, podemos usar la función `parseInt()`:



Ahora, `numeroEntero` contiene el valor 5, que es un número entero.

Reto:

Crea un programa que tome una cadena de caracteres que contiene un número con decimales (por ejemplo, "3.14") y conviértalo a un número con decimales (por ejemplo, 3.14).

### 4.16 Coerción

Coerción en JavaScript es una técnica de programación que se utiliza para convertir un valor de un tipo de datos a otro automáticamente. Esto se hace para asegurar

## 4 TIPOS Y VARIABLES

---

que los datos sean del tipo correcto para una operación específica. Por ejemplo, si se intenta realizar una operación matemática con una cadena de texto, JavaScript intentará convertir la cadena de texto a un número antes de realizar la operación.

Ejemplo:



```
let x = "5";
let y = 10;

let resultado = x + y;

console.log(resultado); // imprime "510"
```

En el ejemplo anterior, la variable `x` es una cadena de texto, mientras que la variable `y` es un número. Al intentar realizar la operación de suma, JavaScript realiza una coerción de la cadena de texto a un número antes de realizar la operación. Esto resulta en un resultado de 510.

En JavaScript, se dice que “todo es un objeto” ya que existe coerción detrás de escena para que los tipos primitivos tengan propiedades más avanzadas de objetos como propiedades y métodos. Esto lo veremos en práctica más adelante.

Reto:

Intenta sumar revertir el orden deel ejemplo como `y + x`. ¿Qué sucede?

## 5 Operadores

### 5.1 Operadores de aritméticos (+, -, \*, /, %, \*\*, //, ++, --)

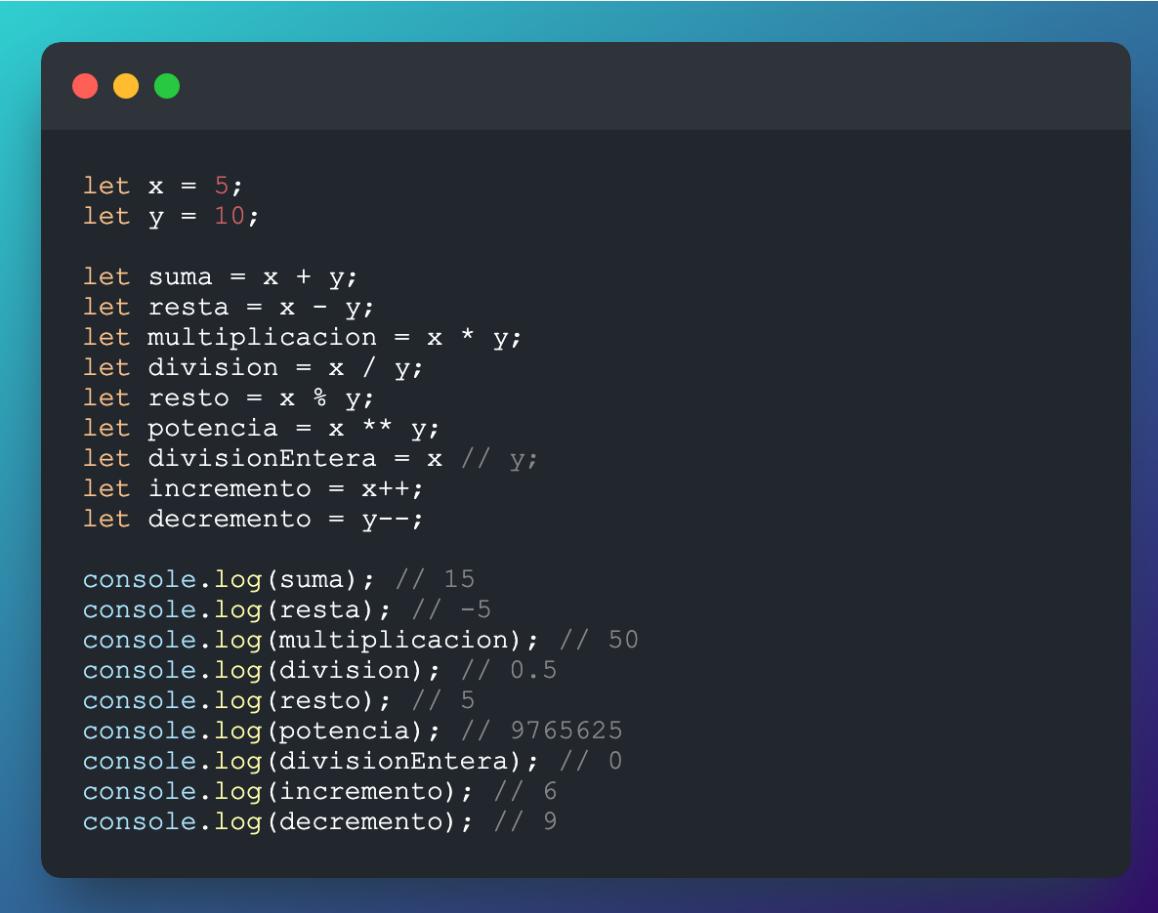
Los operadores de aritméticos en JavaScript son usados para realizar operaciones matemáticas básicas. Estos operadores incluyen:

- +: Suma dos números.
- -: Resta dos números.
- \*: Multiplica dos números.
- /: Divide dos números.
- %: Calcula el resto de una división entre dos números.
- \*\*: Calcula la potencia de un número.
- //: Realiza una división entera entre dos números.
- ++: Incrementa un número en uno.
- --: Decrementa un número en uno.

Ejemplo:

## 5 OPERADORES

---



```
let x = 5;
let y = 10;

let suma = x + y;
let resta = x - y;
let multiplicacion = x * y;
let division = x / y;
let resto = x % y;
let potencia = x ** y;
let divisionEntera = x // y;
let incremento = x++;
let decremento = y--;

console.log(suma); // 15
console.log(resta); // -5
console.log(multiplicacion); // 50
console.log(division); // 0.5
console.log(resto); // 5
console.log(potencia); // 9765625
console.log(divisionEntera); // 0
console.log(incremento); // 6
console.log(decremento); // 9
```

Reto:

Usa los operadores de aritméticos para calcular el área de un círculo con radio 5.

### 5.2 Operador de agrupación (())

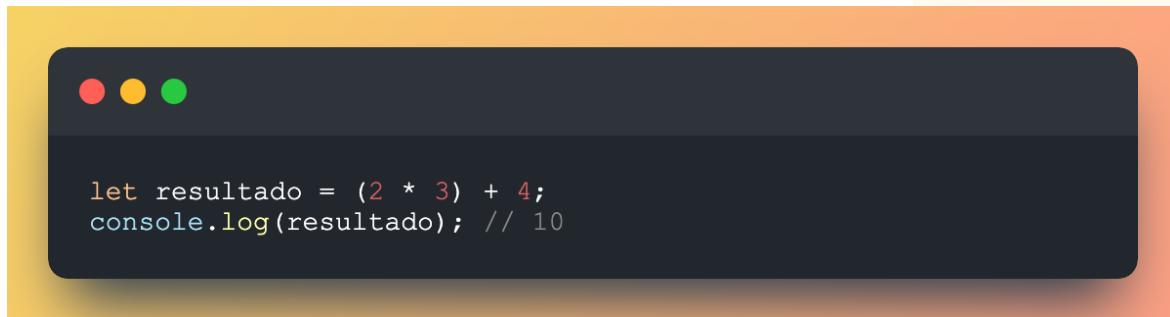
El operador de agrupación en JavaScript es un operador que se utiliza para agrupar expresiones y forzar la prioridad de una operación. Esto significa que el operador de agrupación se utiliza para asegurar que una operación se realice antes que otra.

Ejemplo:

## 5 OPERADORES

---

Por ejemplo, si queremos calcular la suma de dos números, pero queremos que se realice la multiplicación primero, podemos usar el operador de agrupación para asegurarnos de que la multiplicación se realice primero.



Reto:

Utiliza el operador de agrupación para que el resultado de la siguiente expresión sea 20:  $2 + 3 * 4$

### 5.3 Operadores de texto (+, +=)

Los operadores de texto en JavaScript nos permiten concatenar cadenas

Ejemplo:

## 5 OPERADORES

---



```
// Concatenar cadenas
let nombre = "Juan";
let apellido = "Perez";
let nombreCompleto = nombre + " " + apellido;
console.log(nombreCompleto); // Juan Perez

// Buscar y reemplazar caracteres
let frase = "Hola";
frase += " amigos";
console.log(frase); // Hola amigos
```

Reto:

Une los texto “Fatboy” y “Slim” con el operador + añadiendo un espacio entre las palabras.

### 5.4 Operadores de asignación (=, +=, -=, \*=, /=, %=, \*\*=, //=)

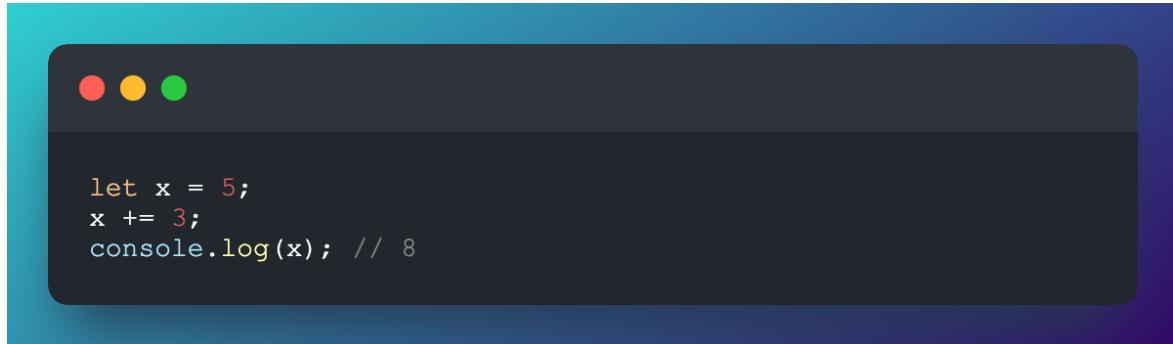
Los operadores de asignación en JavaScript son usados para asignar un valor a una variable. Estos operadores consisten en un signo igual (=) seguido de uno de los siguientes operadores:

- **+=**: Esto significa “agregar a” . Esto significa que el valor de la variable se incrementará en la cantidad especificada.

Ejemplo:

## 5 OPERADORES

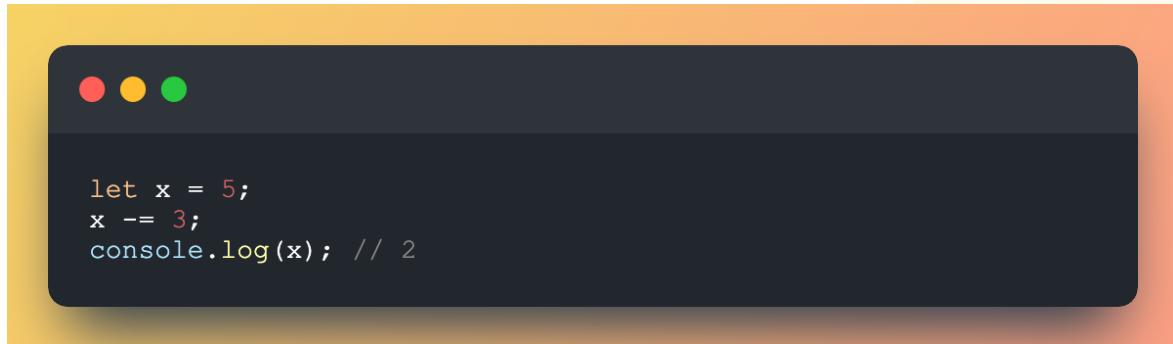
---



```
let x = 5;
x += 3;
console.log(x); // 8
```

- `-=`: Esto significa “restar de” . Esto significa que el valor de la variable se reducirá en la cantidad especificada.

Ejemplo:



```
let x = 5;
x -= 3;
console.log(x); // 2
```

- `*=`: Esto significa “multiplicar por” . Esto significa que el valor de la variable se multiplicará por la cantidad especificada.

Ejemplo:

## 5 OPERADORES

---

```
let x = 5;
x *= 3;
console.log(x); // 15
```

- /=: Esto significa “dividir entre” . Esto significa que el valor de la variable se dividirá entre la cantidad especificada.

Ejemplo:

```
let x = 15;
x /= 3;
console.log(x); // 5
```

- %=: Esto significa “módulo de” . Esto significa que el valor de la variable se calculará como el resto de la división entre la cantidad especificada.

Ejemplo:

## 5 OPERADORES

---

```
let x = 15;
x %= 3;
console.log(x); // 0
```

- \*\*=: Esto significa “elevar a la potencia” . Esto significa que el valor de la variable se elevará a la potencia especificada.

Ejemplo:

```
let x = 2;
x **= 3;
console.log(x); // 8
```

- //=: Esto significa “dividir entre y redondear hacia abajo” . Esto significa que el valor de la variable se dividirá entre la cantidad especificada y se redondeará hacia abajo.

Ejemplo:

## 5 OPERADORES

---

```
let x = 15;  
x /= 3;  
console.log(x); // 5
```

Reto:

Usa los operadores de asignación para calcular el área de un cuadrado donde el lado es igual a 5.

### 5.5 Operadores comparativos (==, !=, >, <, >=, <=)

Los operadores comparativos en JavaScript son usados para comparar dos valores y determinar si son iguales o diferentes. Estos operadores son:

- == (igual a)
- != (no igual a)
- > (mayor que)
- < (menor que)
- >= (mayor o igual que)
- <= (menor o igual que)

Ejemplo:

## 5 OPERADORES

---

```
let x = 5;
let y = 10;

console.log(x == y); // false
console.log(x != y); // true
console.log(x > y); // false
console.log(x < y); // true
console.log(x >= y); // false
console.log(x <= y); // true
```

Reto:

Escribe un programa que compare dos números y determina si el primero es mayor, menor o igual al segundo.

### 5.6 Comparar por valor y por referencia

Comparar por valor significa comparar los valores de dos variables para determinar si son iguales.

Ejemplo:

```
let x = 5;
let y = 5;

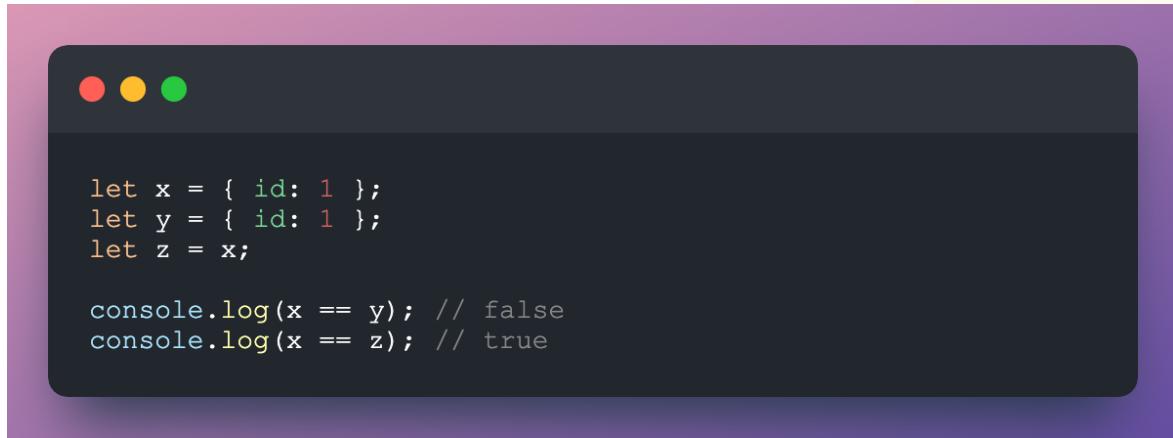
console.log(x == y); // true
```

## 5 OPERADORES

---

Las variables primitivas se pasan “por valor” y los objetos, listas, y datos no primitivos se pasan “por referencia”. Esto quiere decir que al compararlos se compara la dirección en la memoria y no su valor.

Ejemplo:



```
let x = { id: 1 };
let y = { id: 1 };
let z = x;

console.log(x == y); // false
console.log(x == z); // true
```

Esta es una de las fuentes más comunes de errores al trabajar con JavaScript en la industria de software. Una solución común es usar librerías o JSON.stringify() para convertir los objetos a texto y comparar su valor.

Reto:

Crea dos variables de tipo lista con el mismo valor y comprueba si son iguales usando el operador de igualdad (==).

### 5.7 Operadores lógicos (&&, ||, !)

Los operadores lógicos en JavaScript son && (AND), || (OR) y ! (NOT). Estos operadores se usan para realizar comparaciones entre expresiones y devolver un valor booleano (verdadero o falso).

El operador && (AND) devuelve verdadero si ambas expresiones son verdaderas.

Ejemplo:

## 5 OPERADORES

---



```
let x = 5;
let y = 10;

console.log(x > 0 && y > 0); // Devuelve true
```

Reto:

Escribe una línea de código que devuelva true si x es mayor que 0 y y es menor que 0.

### 5.8 Operadores de bits (&, |, ~, ^, », «, »>)

Los operadores de bits en JavaScript son un conjunto de operadores que permiten realizar operaciones bit a bit sobre los números enteros. Estos operadores son:

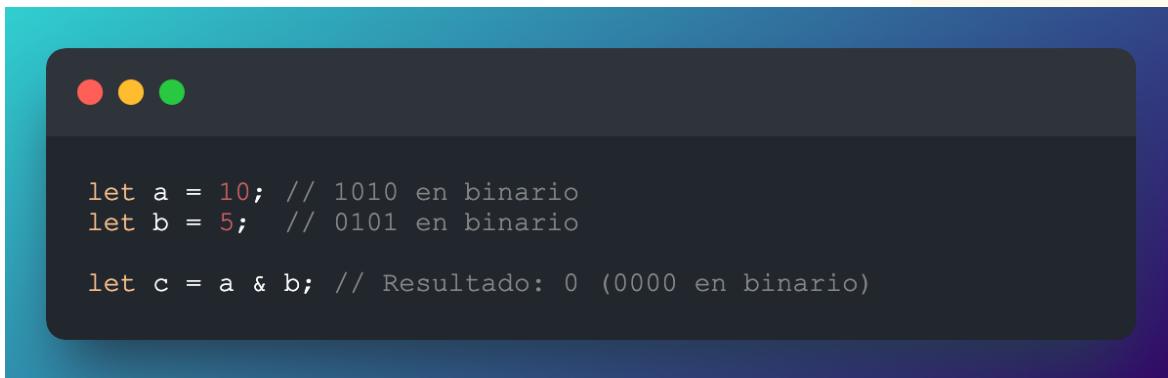
- & (AND)
- | (OR)
- ~ (NOT)
- ^ (XOR)
- >> (Desplazamiento a la derecha)
- << (Desplazamiento a la izquierda)
- >>> (Desplazamiento a la derecha sin signo)

Ejemplo:

A continuación se muestra un ejemplo de código que usa los operadores de bits para realizar una operación bit a bit:

## 5 OPERADORES

---



```
let a = 10; // 1010 en binario
let b = 5; // 0101 en binario

let c = a & b; // Resultado: 0 (0000 en binario)
```

En este ejemplo se usa el operador `&` para realizar una operación AND bit a bit entre los números `a` y `b`. El resultado de esta operación es 0.

Reto:

Para reforzar lo aprendido, el reto consiste en realizar una operación bit a bit usando el operador `|` entre los números `a` y `b` del ejemplo anterior. El resultado de esta operación debe ser 15 (1111 en binario).

### 5.9 Operadores unarios (`delete`, `typeof`, `void`)

Los operadores unarios son un tipo de operador que toma un solo operando (un argumento) para realizar una operación. En JavaScript, hay tres operadores unarios principales: `delete`, `typeof` y `void`.

El operador `delete` se utiliza para eliminar una propiedad de un objeto. Por ejemplo, si tenemos un objeto llamado `persona` con una propiedad llamada `edad`, podemos eliminar la propiedad `edad` con el siguiente código:

## 5 OPERADORES

---

```
delete persona.edad;
```

El operador `typeof` se utiliza para determinar el tipo de una variable. Por ejemplo, si tenemos una variable llamada `nombre` que contiene una cadena, podemos determinar su tipo con el siguiente código:

```
typeof nombre; // devuelve "string"
```

El operador `void` se utiliza para evaluar una expresión y devolver `undefined`. Por ejemplo, si queremos evaluar una expresión y devolver `undefined`, podemos usar el siguiente código:

```
void (nombre + " " + apellido); // devuelve undefined
```

Reto:

Intenta crear un objeto llamado `auto` con una propiedad llamada `marca` y luego eliminar la propiedad `marca` usando el operador `delete`.

## 5 OPERADORES

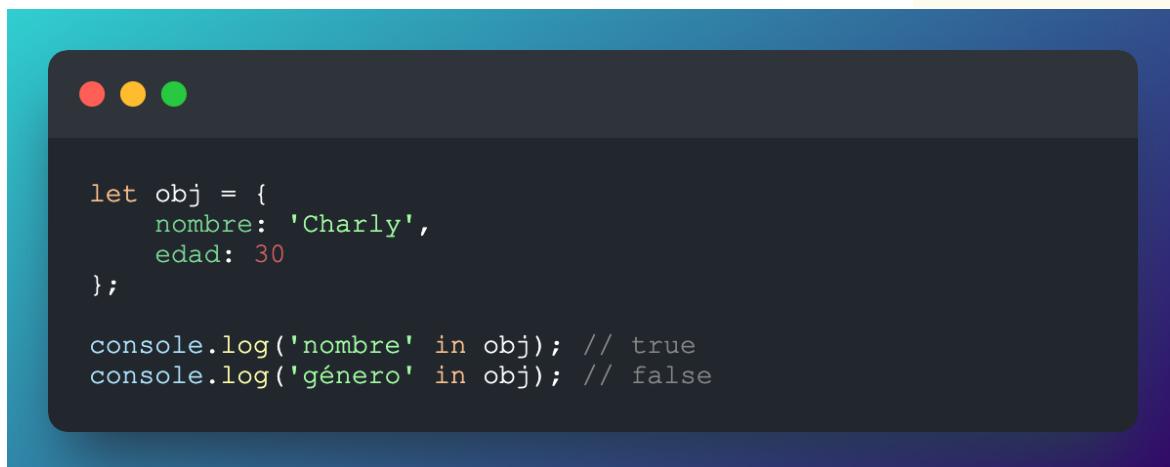
---

### 5.10 Operadores relacionales (in, instanceof)

Los operadores relacionales en JavaScript son usados para comparar dos valores y determinar si son iguales o no. Estos operadores incluyen `in` y `instanceof`.

El operador `in` se usa para determinar si una propiedad existe en un objeto. Devuelve `true` si la propiedad existe en el objeto, de lo contrario devuelve `false`.

Ejemplo:



```
● ● ●

let obj = {
    nombre: 'Charly',
    edad: 30
};

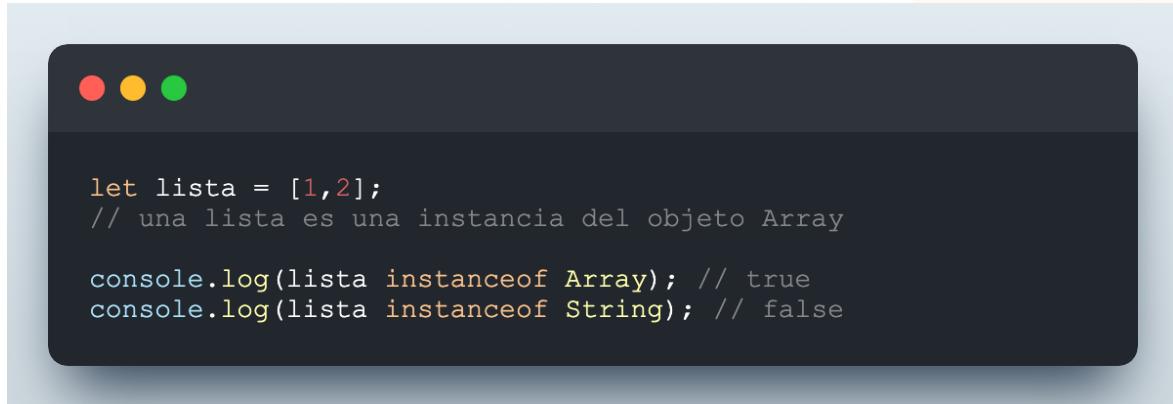
console.log('nombre' in obj); // true
console.log('género' in obj); // false
```

El operador `instanceof` se usa para determinar si un objeto es una instancia de una clase determinada. Devuelve `true` si el objeto es una instancia de la clase, de lo contrario devuelve `false`.

Ejemplo:

## 5 OPERADORES

---



A screenshot of a terminal window with three colored window control buttons (red, yellow, green) at the top. The terminal displays the following JavaScript code:

```
let lista = [1,2];
// una lista es una instancia del objeto Array

console.log(lista instanceof Array); // true
console.log(lista instanceof String); // false
```

Reto:

Usa instanceof para ver la instancia de un número.

### 5.11 Operador spread (…)

El operador spread (...) es una característica de JavaScript que permite expandir una expresión en varios argumentos o elementos. Esto significa que se puede usar para expandir una lista, un objeto o una cadena de texto en lugares donde se esperan varios argumentos o elementos.

Ejemplo:

Supongamos que tenemos una lista de números y queremos imprimir cada elemento de la lista.

## 5 OPERADORES

---

```
const numeros = [1, 2, 3, 4, 5];
console.log(...numeros);
// es equivalente a console.log(1, 2, 3, 4, 5);
```

Reto:

Usa el operador spread para crear una nueva lista que contenga los elementos de dos listas diferentes.

### 5.12 Operador de coma (,)

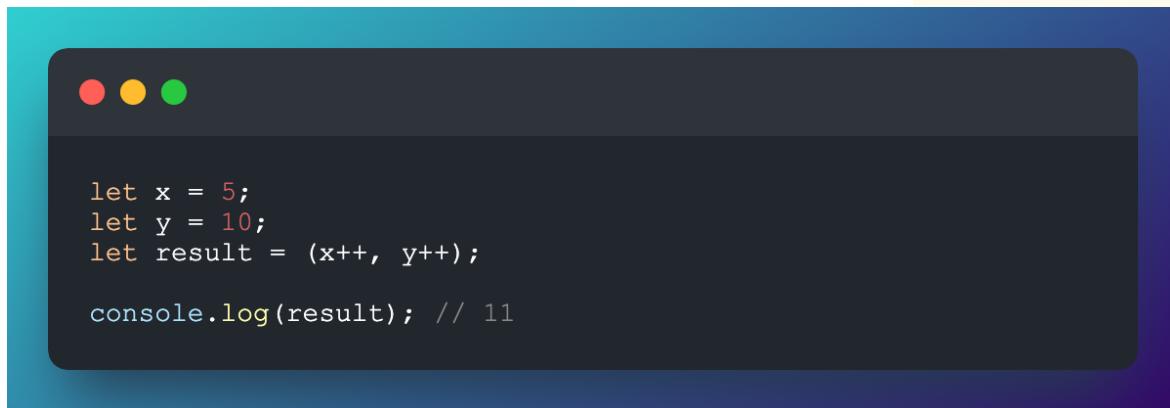
El operador de coma en JavaScript es un operador binario que se utiliza para separar expresiones individuales dentro de una sentencia. Esto significa que cada expresión se evalúa por separado y el resultado de la última expresión se devuelve como resultado de la sentencia.

Ejemplo:

Por ejemplo, el siguiente código usa el operador de coma para evaluar dos expresiones y devolver el resultado de la segunda expresión:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---



```
let x = 5;
let y = 10;
let result = (x++, y++);

console.log(result); // 11
```

X En este ejemplo, primero se evalúa la primera expresión (`x++`), que incrementa el valor de `x` en 1. Luego, se evalúa la segunda expresión (`y++`), que incrementa el valor de `y` en 1. El resultado de la segunda expresión (11) se devuelve como resultado de la sentencia.

Reto:

Escribe una sentencia que use el operador de coma para evaluar tres expresiones y devolver el resultado de la tercera expresión.

# 6 Estructuras de control de flujo

## 6.1 Condicionales

### 6.1.1 if

`if` es una estructura de control de flujo en JavaScript que permite ejecutar una sección de código si una condición es verdadera. Si la condición es falsa, se puede ejecutar otra sección de código.

Ejemplo:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---



Reto:

Escribe un código que compruebe si un número es par o impar.

### 6.1.2 if…else

if...else es una estructura de control de flujo en JavaScript que **permite a los programadores tomar decisiones basadas en condiciones**. Esta estructura se compone de dos partes: una condición (if) y una acción (else). Si la condición se evalúa como verdadera, se ejecuta la acción asociada al if; de lo contrario, se ejecuta la acción asociada al else.

Ejemplo:

Supongamos que queremos escribir un programa que imprima un mensaje diferente dependiendo de si una variable es mayor o menor que 10. Podemos hacer esto con una estructura if…else:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---

```
var num = 5;

if (num > 10) {
    console.log("El número es mayor que 10");
} else {
    console.log("El número es menor que 10");
}
```

En este ejemplo, la variable num se evalúa para ver si es mayor que 10. Si es así, se imprime el mensaje “El número es mayor que 10” ; de lo contrario, se imprime el mensaje “El número es menor que 10” .

Reto:

Escribe un programa que imprima un mensaje diferente dependiendo de si una variable es mayor, menor o igual a 10.

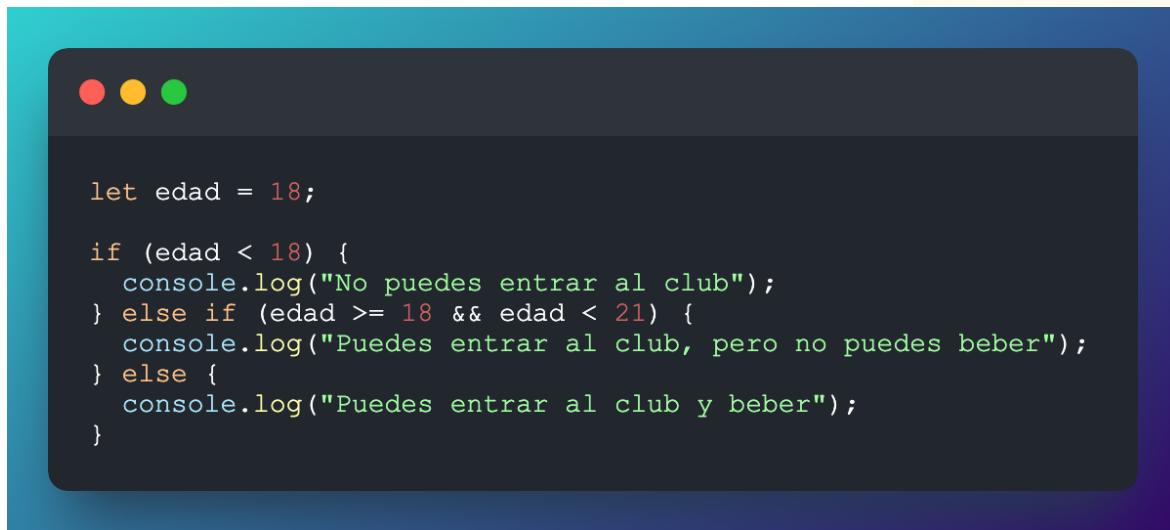
### 6.1.3 if…elif…else

if...elif...else es una estructura de control de flujo en JavaScript que permite a los programadores tomar decisiones basadas en una o más condiciones. Esta estructura se compone de tres partes: if, elif y else.

Ejemplo:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---



A screenshot of a code editor window. At the top, there are three colored circular icons: red, yellow, and green. The main area contains the following JavaScript code:

```
let edad = 18;

if (edad < 18) {
    console.log("No puedes entrar al club");
} else if (edad >= 18 && edad < 21) {
    console.log("Puedes entrar al club, pero no puedes beber");
} else {
    console.log("Puedes entrar al club y beber");
}
```

Reto:

Escribe un programa que imprima un mensaje que indique si el usuario puede jubilarse, es mayor a 80 o ambas cosas.

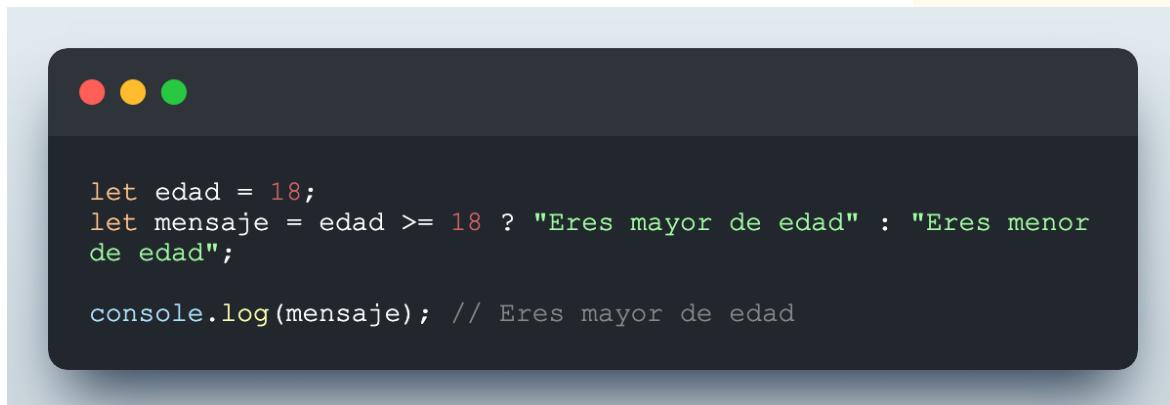
### 6.1.4 Operador ternario (?)

El operador condicional ternario en JavaScript **es una forma abreviada de escribir una condición if-else**. Esta sintaxis es útil para evaluar una expresión y asignar un valor basado en el resultado de la evaluación.

Ejemplo:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---



```
let edad = 18;
let mensaje = edad >= 18 ? "Eres mayor de edad" : "Eres menor de edad";

console.log(mensaje); // Eres mayor de edad
```

Reto:

Escribe una línea de código usando el operador ternario para asignar el valor “Es par” a la variable “mensaje” si el número “numero” es par, o “Es impar” si el número “numero” es impar.

### 6.1.5 Condicionales anidados

Los condicionales anidados en JavaScript son una forma de estructurar el código para que se ejecute de manera diferente dependiendo de una serie de condiciones. Esto se logra anidando varios condicionales dentro de uno.

Ejemplo:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---

```
if (condición1) {  
    if (condición2) {  
        // código a ejecutar si se cumplen ambas condiciones  
    } else {  
        // código a ejecutar si se cumple la condición1 pero no  
        la condición2  
    }  
} else {  
    // código a ejecutar si no se cumple la condición1  
}
```

Este código funciona pero se debe evitar ya que complica la legibilidad. En su lugar se recomienda usar cláusulas de guarda.

Reto:

Escribe un código que evalúe si un número es mayor que 10 y menor que 20. Si se cumple, imprime “El número está entre 10 y 20” . Si no se cumple, imprime “El número no está entre 10 y 20” .

### 6.1.6 switch

switch en JavaScript es una estructura de control que nos permite evaluar una expresión y ejecutar un bloque de código específico dependiendo del resultado de la evaluación. Esta estructura de control es útil cuando hay una gran cantidad de posibles resultados para una expresión.

Ejemplo:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---

```
let diaSemana = "Lunes";

switch (diaSemana) {
  case "Lunes":
    console.log("Hoy es lunes");
    break;
  case "Martes":
    console.log("Hoy es martes");
    break;
  case "Miercoles":
    console.log("Hoy es miercoles");
    break;
  default:
    console.log("No es un dia de la semana");
}
```

En este ejemplo, la variable diaSemana se evalúa en el switch y dependiendo del resultado, se ejecuta un bloque de código específico.

default es la condición en caso de que ninguna otra se cumpla.

break evita que corra las siguientes comparaciones y termina el condicional.

Reto:

Crea un programa que evalúe una variable numero y dependiendo del resultado, imprima un mensaje diferente. Si el número es mayor a 10, imprime “El número es mayor a 10” . Si el número es menor a 10, imprime “El número es menor a 10” . Si el número es igual a 10, imprime “El número es igual a 10” .

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

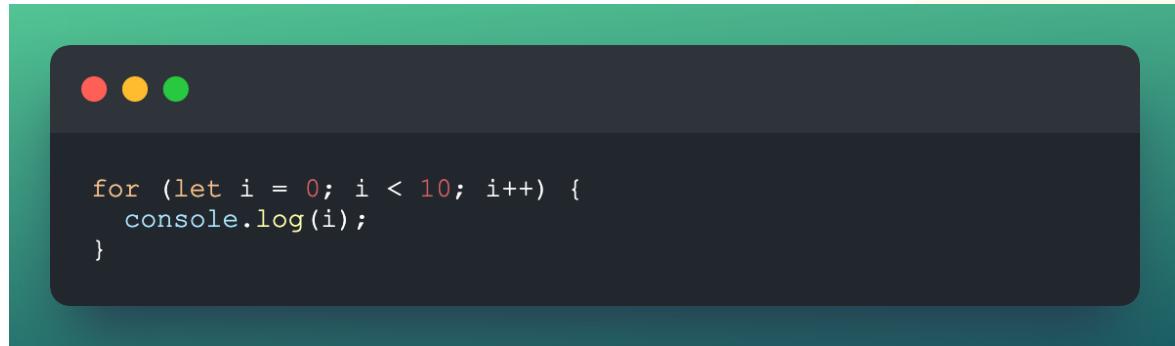
---

### 6.2 Bucles

#### 6.2.1 for

for es una estructura de control en JavaScript que se utiliza para iterar sobre una secuencia de elementos. Esta estructura de control se compone de una inicialización, una condición de parada y una actualización.

Ejemplo:



A screenshot of a terminal window on a Mac OS X system, indicated by the red, yellow, and green window control buttons at the top. The terminal window contains the following JavaScript code:

```
for (let i = 0; i < 10; i++) {
    console.log(i);
}
```

En este ejemplo, la inicialización es `let i = 0`, la condición de parada es `i < 10` y la actualización es `i++`. Esto significa que el ciclo se ejecutará hasta que `i` sea mayor que 10.

Reto:

Escribe un ciclo for que imprima los números del 1 al 10 en orden ascendente.

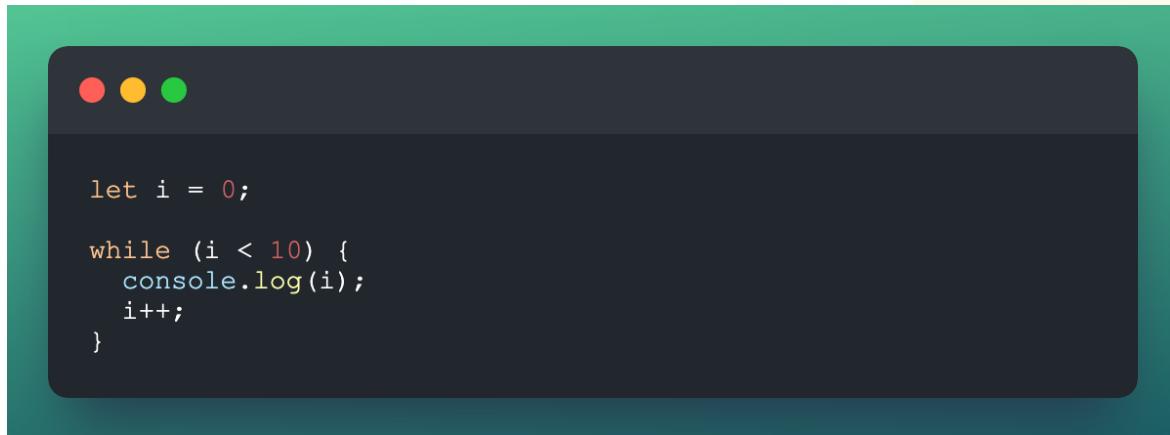
#### 6.2.2 while

while es una estructura de control de flujo en JavaScript que se usa para ejecutar una serie de instrucciones mientras se cumpla una condición. Esta condición se evalúa antes de cada iteración del bucle.

Ejemplo:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---



A screenshot of a terminal window with three colored window control buttons (red, yellow, green) at the top. The terminal window has a dark background. Inside, there is a code snippet:

```
let i = 0;
while (i < 10) {
    console.log(i);
    i++;
}
```

En el ejemplo anterior, se inicializa una variable `i` con el valor 0. Luego, se ejecuta un bucle `while` que se ejecutará mientras `i` sea menor que 10. Dentro del bucle, se imprime el valor de `i` en la consola y se incrementa en 1.

Reto:

Crea un bucle `while` que imprima los números del 1 al 10 en la consola.

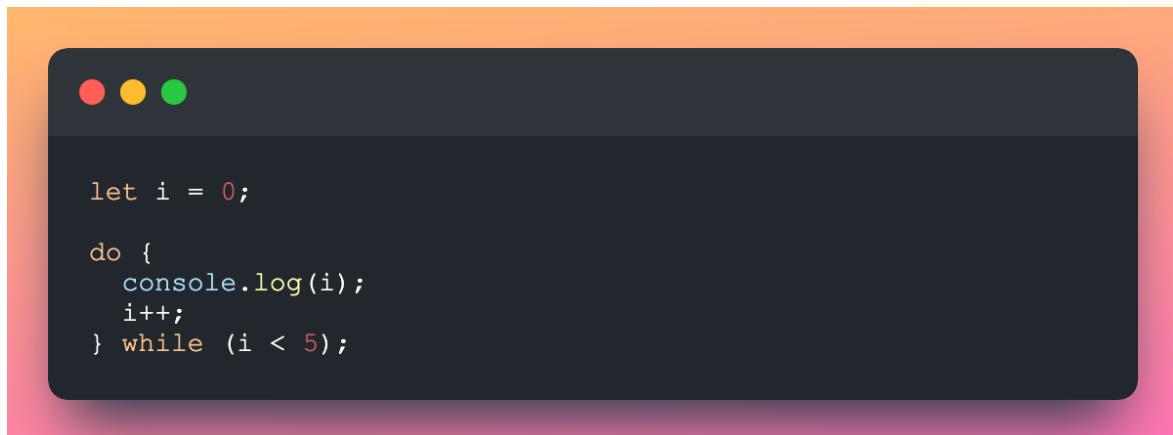
### 6.2.3 do...while

`do...while` es una estructura de control de bucle en JavaScript que ejecuta una sentencia o bloque de código una vez, y luego continúa ejecutándolo mientras se cumpla una condición. Esta estructura es útil cuando se necesita asegurar que una sentencia se ejecute al menos una vez.

Ejemplo:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---



A screenshot of a terminal window with a dark background and a light gray border. In the top left corner, there are three colored dots: red, yellow, and green. The terminal displays the following JavaScript code:

```
let i = 0;
do {
    console.log(i);
    i++;
} while (i < 5);
```

En el ejemplo anterior, la sentencia `console.log(i)` se ejecutará al menos una vez, ya que la condición `i < 5` se evalúa al final de cada iteración.

Reto:

Escribe un programa que imprima los números del 1 al 10 usando un bucle `do...while`.

### 6.2.4 continue

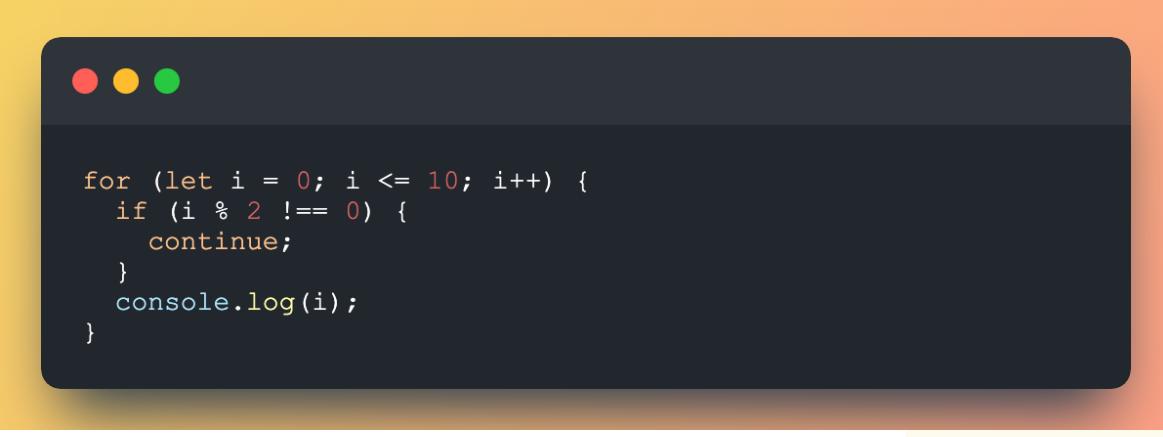
`continue` es una palabra clave en JavaScript que se usa para saltar a la siguiente iteración de un bucle. Esto significa que cuando se encuentra una instrucción `continue`, el código de la iteración actual se salta y se continúa con la siguiente iteración.

Ejemplo:

Por ejemplo, si queremos imprimir todos los números pares entre 0 y 10, podemos usar un bucle `for` con una instrucción `continue` para saltar los números impares:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---



```
for (let i = 0; i <= 10; i++) {
  if (i % 2 !== 0) {
    continue;
  }
  console.log(i);
}
```

Esto imprimirá los números pares entre 0 y 10:

0  
2  
4  
6  
8  
10

Reto:

Escribe un programa que imprima los números impares entre 0 y 10 usando un bucle for y una instrucción continue.

### 6.2.5 break

break es una palabra clave en JavaScript que se usa para salir de un bucle. Esto significa que cuando se encuentra una instrucción break, el bucle se detiene y el control se transfiere a la siguiente instrucción después del bucle.

Ejemplo:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---

A continuación se muestra un ejemplo de cómo usar break para salir de un bucle:

```
for (let i = 0; i < 10; i++) {  
  if (i === 5) {  
    break;  
  }  
  console.log(i);  
}
```

En este ejemplo, el bucle se ejecutará hasta que se encuentre la instrucción break. Cuando se encuentra, el bucle se detiene y el control se transfiere a la siguiente instrucción después del bucle.

Reto:

Escribe un programa que imprima los números del 1 al 5, excepto el 3. Utiliza la palabra clave break para salir del bucle cuando se encuentre el número 3.

### 6.2.6 Etiquetas

Una etiqueta en JavaScript es una forma de etiquetar una instrucción para que sea más fácil de identificar (no confundir con etiquetas de HTML). Esto se hace mediante una palabra “etiqueta” seguida de dos puntos y un bucle. Se puede indicar que hacer después de break o continue indicando la etiqueta.

Ejemplo:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---

```
bucle1: for (i = 0; i < 3; i++) {  
    bucle2: for (j = 0; j < 3; j++) {  
        if (i === 1 && j === 1) {  
            continue bucle2;  
        }  
        console.log(`i = ${i}, j = ${j}`);  
    }  
}
```

En este caso, al llegar a `continue`, saltará a “`bucle2`” .

Reto:

Reemplaza “`continue bucle2;`” con “`continue bucle1;`” para que veas el cambio de dirección de flujo del código.

### 6.2.7 `for...in`

`for...in` es una estructura de control de bucle en JavaScript que se utiliza para iterar sobre los elementos de un objeto. Esta estructura de control se utiliza para recorrer los elementos de un objeto y realizar una acción para cada elemento.

Ejemplo:

Supongamos que tenemos un objeto llamado `persona` con los siguientes atributos:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---

```
const persona = {  
    nombre: 'Juan',  
    edad: 25,  
    ciudad: 'Madrid'  
};
```

Podemos usar `for...in` para recorrer los atributos de este objeto y mostrar su valor en la consola:

```
for (const atributo in persona) {  
    console.log(`${atributo}: ${persona[atributo]}`);  
}
```

Esto imprimirá en la consola:

nombre: Juan  
edad: 25  
ciudad: Madrid

Reto:

Crea un objeto llamado libro con los siguientes atributos: título, autor, año de publicación y género. Usa `for...in` para recorrer los atributos del objeto y mostrar su valor en la consola.

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---

### 6.2.8 for...of

for...of es una estructura de control de bucle en JavaScript que se utiliza para iterar sobre los elementos de un objeto iterable, como una lista, una cadena o un objeto. Esta estructura de control de bucle es una forma más moderna de bucle que se introdujo en ES6.

Ejemplo:

A continuación se muestra un ejemplo de cómo usar for...of para iterar sobre los elementos de una lista:



```
let arr = [1, 2, 3, 4, 5];

for (let element of arr) {
    console.log(element);
}

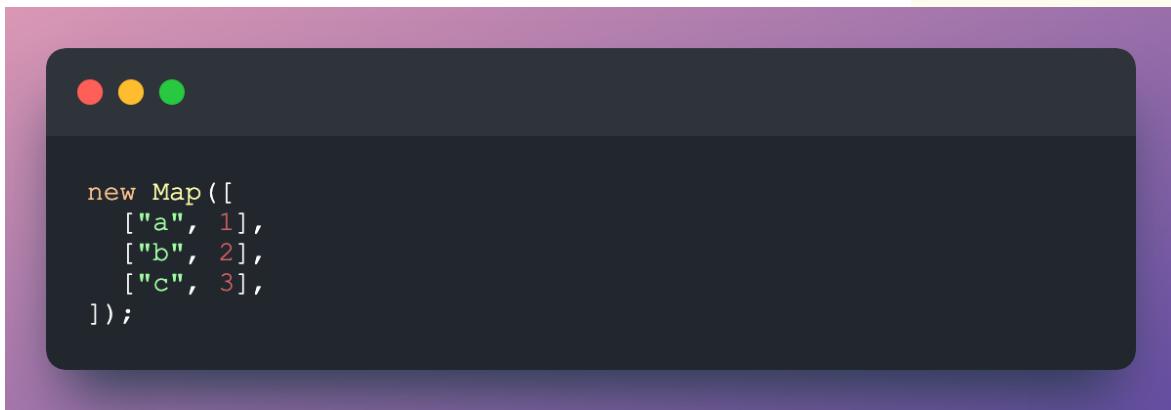
// Salida:
// 1
// 2
// 3
// 4
// 5
```

Reto:

Utiliza for...of para iterar sobre los elementos de este mapa y mostrar los valores en la consola.

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---



### 6.3 Excepciones

#### 6.3.1 Tipos de excepciones

Las excepciones en JavaScript son errores que se producen durante la ejecución de un programa. Estas excepciones pueden ser de varios tipos, entre los que se incluyen:

- **Excepciones de sintaxis:** Estas excepciones se producen cuando el código contiene errores de sintaxis.
- **Excepciones de referencia:** Estas excepciones se producen cuando el código contiene errores de referencia. Por ejemplo, si se intenta acceder a un elemento de una lista que no existe, se producirá una excepción de tiempo de ejecución.
- **Excepciones de tipo:** Estas excepciones se producen cuando se intenta realizar una operación con un tipo de datos incorrecto. Por ejemplo, si se intenta cambiar una constante.

Ejemplo:

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---



```
const str = "No cambia";
str = 5;
// Esto provocará una excepción de tipo
```

Reto:

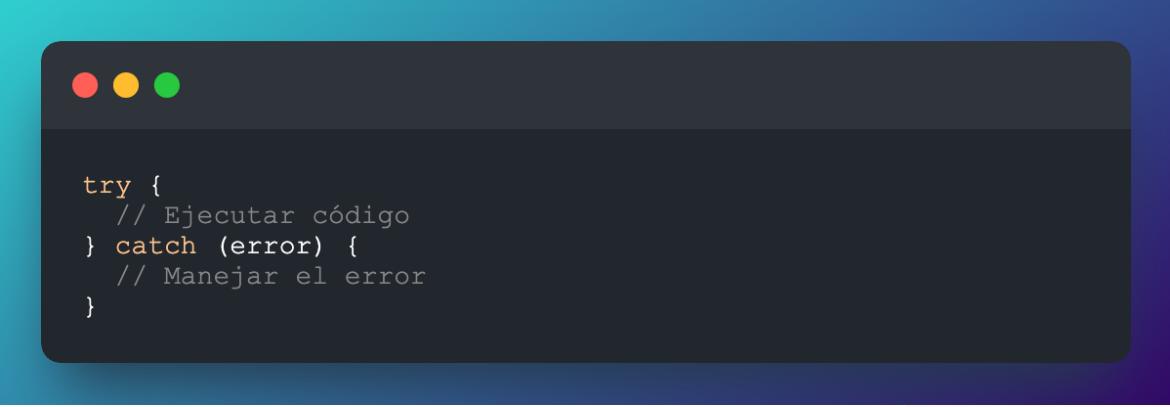
Escribe un código que intente acceder a un elemento de una lista que no existe, lo que provocará una excepción de referencia.

### 6.3.2 try...catch

La sentencia try/catch en JavaScript es una forma de controlar errores en tiempo de ejecución. Esta sentencia permite ejecutar un bloque de código y, en caso de que se produzca un error, ejecutar un bloque de código para manejar el error.

Ejemplo:

A continuación se muestra un ejemplo de cómo se puede utilizar la sentencia try/catch en JavaScript:



```
try {
    // Ejecutar código
} catch (error) {
    // Manejar el error
}
```

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---

En este ejemplo, el bloque de código dentro del try se ejecutará primero. Si se produce un error, el bloque de código dentro del catch se ejecutará para manejar el error.

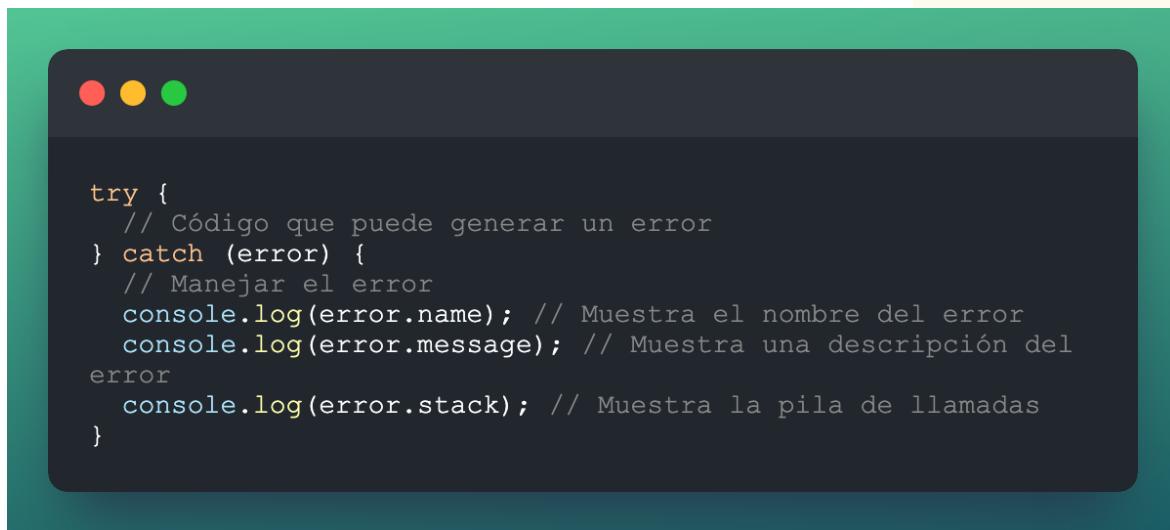
Reto:

El reto consiste en acceder a una lista. Esta función debe utilizar la sentencia try/catch para imprimir el error en caso de que la lista no exista.

### 6.3.3 Utilizando objetos de errores

Los objetos de errores en JavaScript son objetos que se utilizan para representar errores que ocurren durante la ejecución de un programa. Estos objetos contienen información sobre el error, como el nombre del error, una descripción del error, y una pila de llamadas que muestra dónde se produjo el error.

Ejemplo:



The screenshot shows a dark-themed code editor window. At the top, there are three small circular icons: red, yellow, and green. The main area contains the following JavaScript code:

```
try {
    // Código que puede generar un error
} catch (error) {
    // Manejar el error
    console.log(error.name); // Muestra el nombre del error
    console.log(error.message); // Muestra una descripción del error
    console.log(error.stack); // Muestra la pila de llamadas
}
```

Reto:

Crea código que produzca un error. Atrápalo con try...catch y revisa sus propiedades.

## 6 ESTRUCTURAS DE CONTROL DE FLUJO

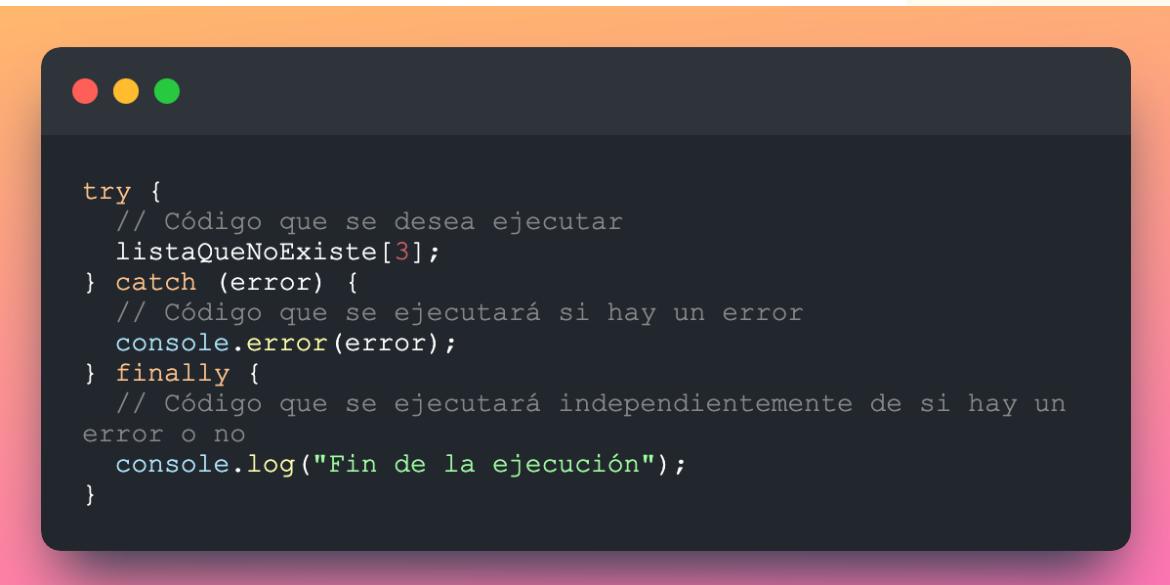
---

### 6.3.4 try...catch...finally

try...catch...finally es una estructura de control de flujo en JavaScript que se utiliza para controlar errores en el código. Esta estructura se compone de tres partes: try, catch y finally.

finally es la tercera y última parte de la estructura. Aquí se coloca el código que se ejecutará independientemente de si hay un error en el código de la sección try o no.

Ejemplo:



```
try {
    // Código que se desea ejecutar
    listaQueNoExiste[3];
} catch (error) {
    // Código que se ejecutará si hay un error
    console.error(error);
} finally {
    // Código que se ejecutará independientemente de si hay un
    // error o no
    console.log("Fin de la ejecución");
}
```

Reto:

Intenta escribir código con problemas de sintaxis y utiliza try...catch...finally para controlar los errores.

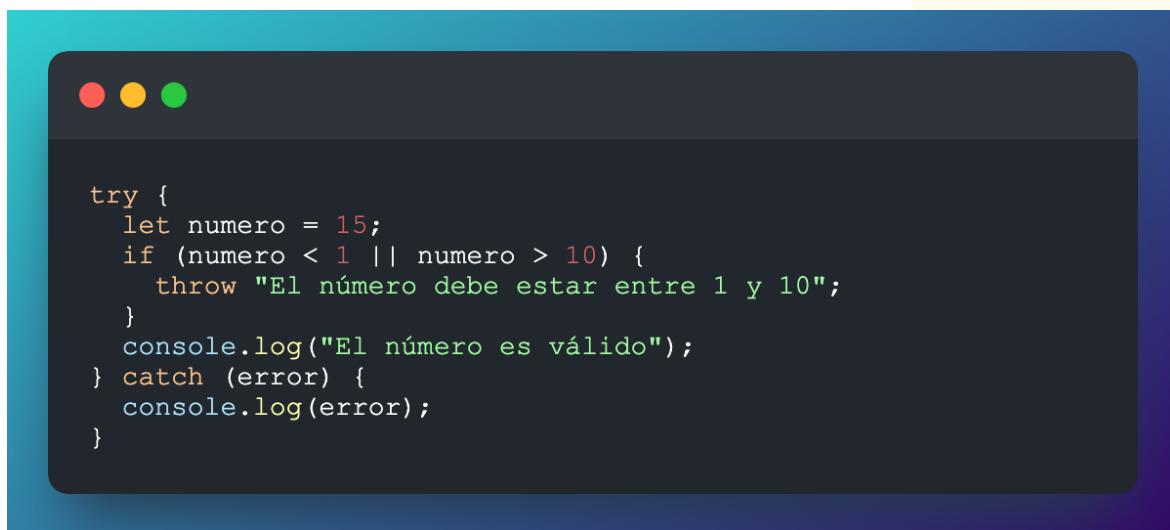
## 6 ESTRUCTURAS DE CONTROL DE FLUJO

---

### 6.3.5 throw

La sentencia `throw` en JavaScript permite lanzar una excepción, es decir, una situación anormal que interrumpe el flujo normal de un programa. Esta sentencia se utiliza para generar errores personalizados, y se debe usar dentro de un bloque `try...catch` para poder controlar el error.

Ejemplo:



A screenshot of a browser's developer tools console window. The title bar shows three colored dots (red, yellow, green). The main area contains the following JavaScript code:

```
try {
  let numero = 15;
  if (numero < 1 || numero > 10) {
    throw "El número debe estar entre 1 y 10";
  }
  console.log("El número es válido");
} catch (error) {
  console.log(error);
}
```

En el ejemplo anterior, si el número no se encuentra dentro de ese rango, se lanza una excepción con la sentencia `throw` y se muestra un mensaje de error.

Reto:

Verifica si un número es par o impar. Si el número no es un entero, lanza una excepción con la sentencia `throw`.

## 7 Funciones

### 7.1 Declarar y llamar funciones

En JavaScript, una función es un bloque de código diseñado para realizar una tarea específica. Las funciones se pueden definir para realizar cualquier tarea, desde realizar cálculos hasta mostrar mensajes en la pantalla.

Ejemplo:

Por ejemplo, aquí hay una función simple que calcula el área de un cuadrado dado su lado:



```
function areaCuadrado() {  
    let lado = 2;  
    console.log(lado * lado);  
}  
  
// llamar función  
areaCuadrado(); // 4
```

Reto:

Escribe una función que calcule el área de un círculo dado su radio.

### 7.2 Parámetros y argumentos

Los argumentos y los parámetros son conceptos relacionados en JavaScript. Los argumentos son los valores que se pasan a una función cuando se llama. Los pará-

## 7 FUNCIONES

---

metros son los nombres de variables que se usan para recibir los argumentos dentro de la función.

Ejemplo:

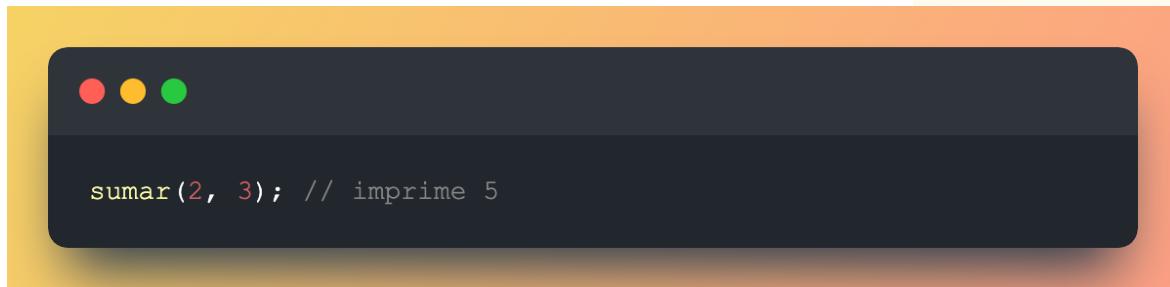
Supongamos que tenemos una función llamada `sumar()` que recibe dos parámetros: `a` y `b`.



A screenshot of a browser's developer tools console. At the top, there are three colored dots (red, yellow, green). Below them, the code for the `sumar` function is shown:

```
function sumar(a, b) {  
  console.log(a + b);  
}
```

Cuando llamamos a la función `sumar()` pasamos dos argumentos: `2` y `3`. Esto permite reutilizar la función de forma dinámica con diferentes valores.



A screenshot of a browser's developer tools console. At the top, there are three colored dots (red, yellow, green). Below them, the function is called with arguments `2` and `3`:

```
sumar(2, 3); // imprime 5
```

En este ejemplo, los parámetros son `a` y `b`, y los argumentos son `2` y `3`.

Reto:

Crea una función llamada `multiplicar()` que reciba dos parámetros e imprima el resultado de multiplicar los dos argumentos. Luego, llama a la función con dos argumentos y comprueba que el resultado sea correcto.

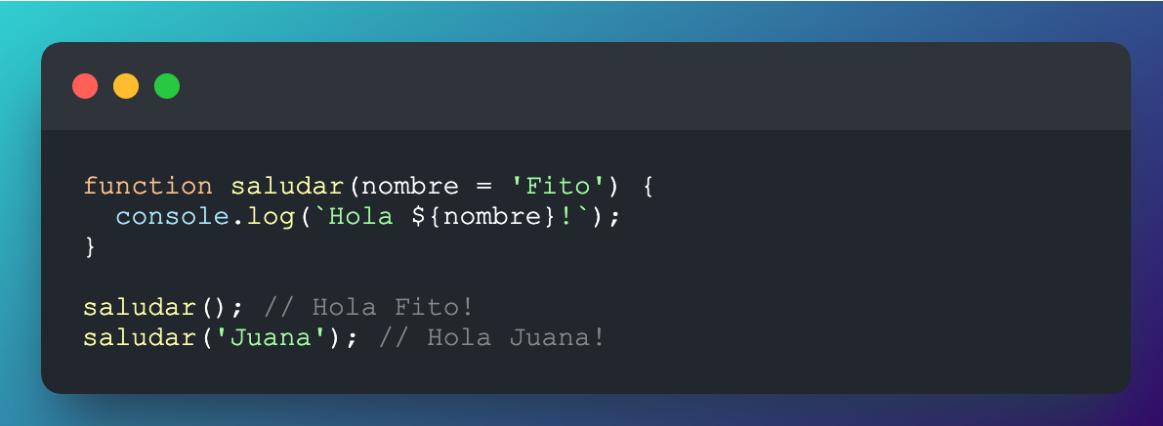
## 7 FUNCIONES

---

### 7.3 Parámetros predeterminados

Los parámetros predeterminados en JavaScript son una forma de asignar valores predeterminados a los parámetros de una función si no se proporcionan argumentos. Esto significa que si una función espera recibir un argumento, pero no se proporciona, el valor predeterminado se usará en su lugar.

Ejemplo:



A screenshot of a terminal window on a Mac OS X system. The window has red, yellow, and green status icons at the top. The terminal itself is dark-themed. It contains the following code:

```
function saludar(nombre = 'Fito') {  
    console.log(`Hola ${nombre}!`);  
}  
  
saludar(); // Hola Fito!  
saludar('Juana'); // Hola Juana!
```

The output of the code is visible below the input, showing the strings "Hola Fito!" and "Hola Juana!" respectively.

Reto:

Escribe una función llamada calcularArea que tome dos parámetros, ancho y alto, e imprima el área de un rectángulo. Si alguno de los parámetros no se proporciona, asígnale el valor predeterminado de 10.

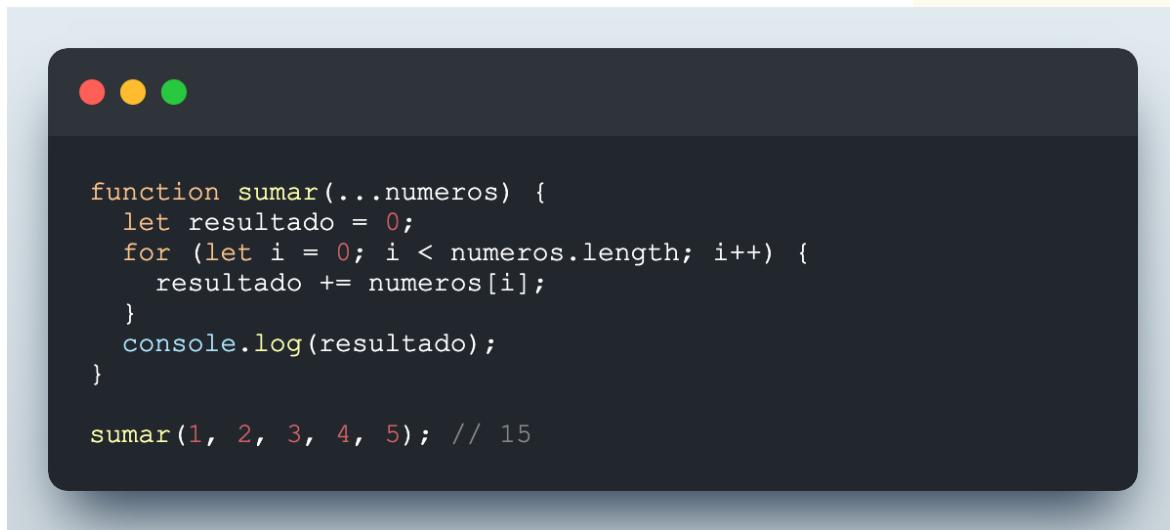
### 7.4 Parámetros de descanso (rest)

Los parámetros de descanso (rest) en JavaScript son una forma de recopilar argumentos en una función. Esto significa que, en lugar de recibir una cantidad fija de argumentos, una función puede recibir un número variable de argumentos.

Ejemplo:

## 7 FUNCIONES

---



```
function sumar(...numeros) {
  let resultado = 0;
  for (let i = 0; i < numeros.length; i++) {
    resultado += numeros[i];
  }
  console.log(resultado);
}

sumar(1, 2, 3, 4, 5); // 15
```

Reto:

Crea una función que tome una cantidad variable de cadenas de texto e imprima una cadena de texto con todas las cadenas concatenadas.

### 7.5 El objeto arguments

El objeto arguments es un objeto especial que se pasa a cada función en JavaScript. Está disponible dentro de la función como una variable local. Esto significa que puede acceder a los argumentos pasados a la función, sin importar cuántos sean.

Ejemplo:

## 7 FUNCIONES

---



A screenshot of a browser's developer tools console. At the top, there are three colored circular icons: red, yellow, and green. Below them, the console displays the following code:

```
function miFuncion(a, b) {
    console.log(arguments[0]); // a
    console.log(arguments[1]); // b
    console.log(arguments.length); // 2
}

miFuncion('Blink', '182');
```

Reto:

Crea una función que tome un número variable de argumentos y devuelva la suma de todos los argumentos pasados.

### 7.6 Retorno

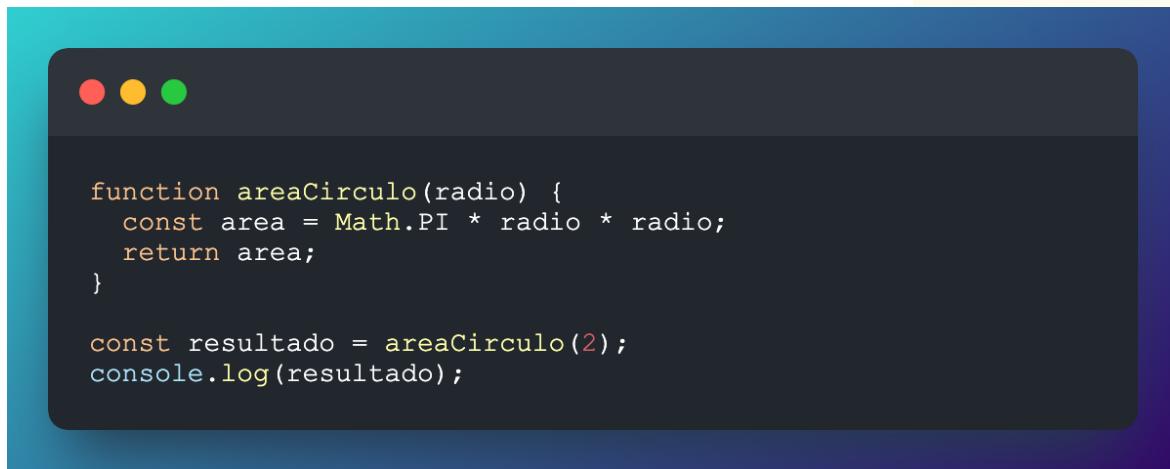
return en JavaScript es una palabra que se usa dentro de una función para devolver un valor. Esto significa que una vez que se ejecuta una función, el valor devuelto se puede usar en otra parte del código. Esto es útil para evitar la repetición de código y para ahorrar tiempo.

Ejemplo:

Por ejemplo, consideremos una función que calcula el área de un círculo.

## 7 FUNCIONES

---



A screenshot of a browser's developer tools console. The title bar shows three colored dots (red, yellow, green). The console area contains the following code:

```
function areaCirculo(radio) {  
    const area = Math.PI * radio * radio;  
    return area;  
}  
  
const resultado = areaCirculo(2);  
console.log(resultado);
```

The output of the code is displayed below the console input:

```
6.283185307179586
```

En este ejemplo, la función `areaCirculo` toma un `radio` como argumento y devuelve el área del círculo. Esto significa que una vez que se ejecuta la función, el valor devuelto se puede usar en otra parte del código.

Reto:

Escribe una función que tome dos números como argumentos y devuelva el mayor de los dos.

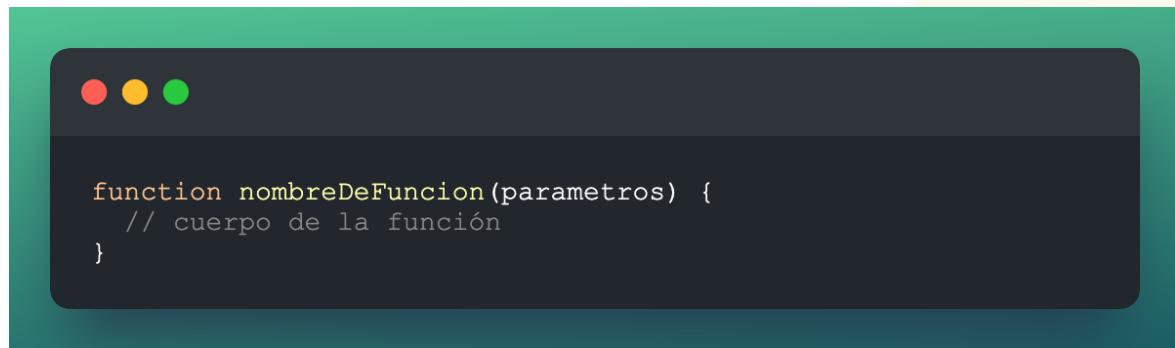
### 7.7 Declaración y expresión de funciones

Existen dos formas principales de crear funciones en JavaScript: declaración de funciones y expresión de funciones.

**Declaración de funciones** Una declaración de función es una forma de crear una función en JavaScript. Esta es la sintaxis básica para crear una función:

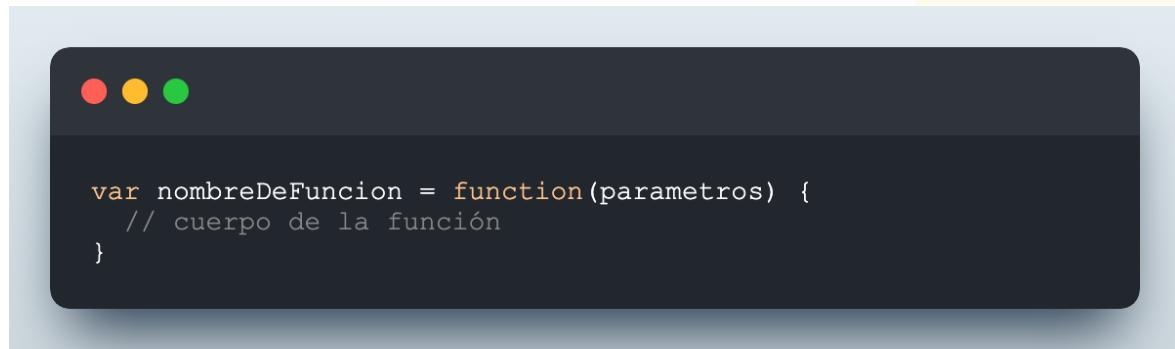
## 7 FUNCIONES

---



En esta sintaxis, `nombreDeFuncion` es el nombre de la función, `parametros` son los parámetros que se pasan a la función y el cuerpo de la función es el código que se ejecuta cuando se invoca la función.

**Expresión de funciones** Una expresión de función es otra forma de crear una función en JavaScript. Esta es la sintaxis básica para crear una función:



Ejemplo:

A continuación se muestra un ejemplo de una función declarada y una función expresada:

## 7 FUNCIONES

---

```
// Declaración de función
function saludar(nombre) {
    console.log(`Hola, ${nombre}!`);
}

// Expresión de función
var despedirse = function(nombre) {
    console.log(`Adiós, ${nombre}!`);
}
```

Reto:

Crea una función declarada como expresión llamada calcularSuma que tome dos números como parámetros y devuelva la suma de los dos números.

### 7.8 Funciones anidadas

Una función anidada es una función que se encuentra dentro de otra función. Esto significa que la función anidada solo está disponible dentro de la función que la contiene. Esto es útil para crear funciones que se usan solo dentro de otra función.

Ejemplo:

## 7 FUNCIONES

---

```
function externa() {
  let variableExt = 'externa';

  function interna() {
    let variableInt = 'interna';
    console.log(variableExt);
  }

  interna();
}

externa(); // 'externa'
```

Reto:

Intenta crear una función anidada que tome un parámetro y lo multiplique por dos.

### 7.9 Ámbito global y local

El ámbito global y local en JavaScript se refiere a la visibilidad de variables y funciones dentro de un programa. El ámbito global se refiere a la visibilidad de variables y funciones en todo el programa, mientras que el ámbito local se refiere a la visibilidad de variables y funciones solo dentro de una función.

Ejemplo:

## 7 FUNCIONES

---



```
// Ámbito global
let nombre = "Juan"; // Variable global

// Función global
function saludar() {
    console.log(`Hola ${nombre}`);
}

// Función global
function saludarAnidada() {
    // Ámbito local
    let nombre = "Pedro"; // Variable local

    // Función local
    function saludarLocal() {
        console.log(`Hola ${nombre}`);
    }

    // llamar a función local
    saludarLocal();
}

saludar(); // "Hola Juan"
saludarAnidada(); // "Hola Pedro"
console.log(nombre); // "Juan"
console.log(saludarLocal); // ReferenceError: saludarLocal is not defined
```

En el ejemplo anterior, la variable nombre es global, por lo que es visible en todo el programa. La función saludar también es global, por lo que es visible en todo el programa. La variable nombre dentro de saludarAnidada es local, por lo que solo es visible dentro de la función al igual que la función saludarLocal.

Reto:

Escribe una función que tome dos números como parámetros y devuelva la suma

## 7 FUNCIONES

---

de los dos números y un número local. Asegúrate de que la función tenga el alcance correcto para que los parámetros sean accesibles dentro de la función.

### 7.10 Devolución de llamada (Callbacks)

Una devolución de llamada (callback) en JavaScript es una función que se pasa como argumento a otra función. Esta función se ejecuta cuando la función a la que se le pasó como argumento termina de ejecutarse. Esto permite que se ejecuten ciertas acciones una vez que una función ha terminado de ejecutarse.

Ejemplo:

Un ejemplo práctico de una devolución de llamada (callback) en JavaScript es el siguiente:



```
function saludar(nombre, callback) {
  console.log(`Hola, ${nombre}`);
  callback();
}

function despedirse() {
  console.log('Adiós!');
}

saludar('Juan', despedirse);
```

En este ejemplo, la función saludar recibe como argumento el nombre de una persona y una función (la devolución de llamada). La función saludar imprime un saludo con el nombre de la persona y luego ejecuta la función que se le pasó como argumento (la devolución de llamada). En este caso, la devolución de llamada es la función despedirse, que imprime una despedida.

## 7 FUNCIONES

---

Reto:

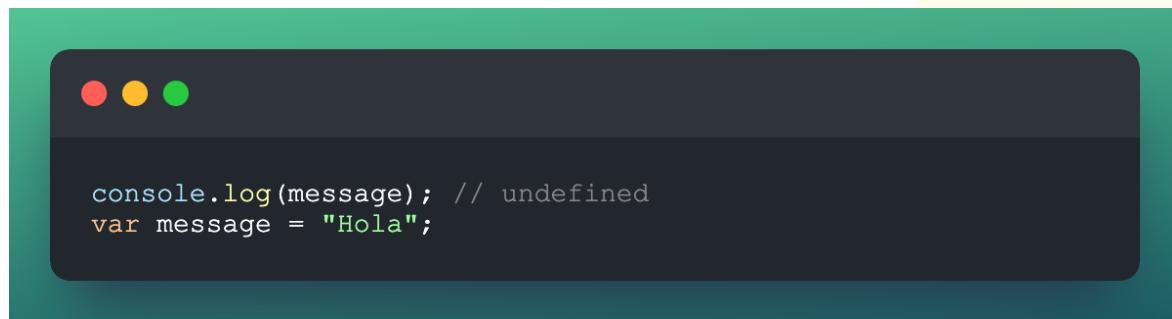
El reto consiste en crear una función que reciba como argumento un número y dos funciones (una devolución de llamada y otra devolución de llamada opcional). La función debe imprimir el número recibido como argumento y luego ejecutar la devolución de llamada. Si se recibe una segunda devolución de llamada, esta debe ejecutarse después de la primera.

### 7.11 Elevación de variables y funciones (Hoisting)

La elevación de variables y funciones es un comportamiento en JavaScript en el que las declaraciones de variables y funciones son “elevadas” al inicio del scope (alcance) en el que se encuentran, independientemente de su ubicación en el código.

Esto significa que puedes utilizar una variable o una función antes de haberla declarado en el código.

Ejemplo:

A screenshot of a terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top. The terminal displays the following code:

```
console.log(message); // undefined
var message = "Hola";
```

The output shows that the variable `message` is undefined when it is first used in the `console.log` statement, but its value is "Hola" when it is defined in the subsequent `var` statement.

En este ejemplo, aunque la variable `message` se está utilizando antes de ser declarada, no produce un error. Esto se debe a que, en realidad, la declaración de la variable se eleva al inicio del scope en el que se encuentra. Por lo tanto, el código anterior es equivalente a:

## 7 FUNCIONES

---

```
var message;  
console.log(message); // undefined  
message = "Hola";
```

Las funciones también son elevadas en JavaScript. Esto significa que puedes invocar una función antes de haberla declarado en el código.

Ejemplo:

```
saludar(); // Hola  
  
function saludar() {  
    console.log("Hola");  
}
```

En este ejemplo, la función **saludar** es invocada antes de ser declarada, pero todavía se ejecuta correctamente. Esto se debe a que la declaración de la función es elevada al inicio del scope en el que se encuentra. Por lo tanto, el código anterior es equivalente a:

## 7 FUNCIONES

---



```
function saludar() {  
    console.log("Hola");  
}  
  
saludar(); // Hola
```

Es importante tener en cuenta que, a diferencia de las variables, las funciones declaradas con la palabra clave **function** son elevadas en su totalidad, incluido su cuerpo. Mientras que las funciones declaradas como expresiones de función (**const miFuncion = function() {...}**) son elevadas solo hasta la declaración de la variable, no su cuerpo.

Reto:

Crea un pequeño programa en el que declares una función como expresión de función y luego la invoques antes de haberla declarada. Verifica que haya un error y explica por qué sucede.

### 7.12 Recursividad

La recursividad es un concepto de programación en el que una función se llama a sí misma. Esto permite a los programadores crear soluciones a problemas complejos de manera más simple.

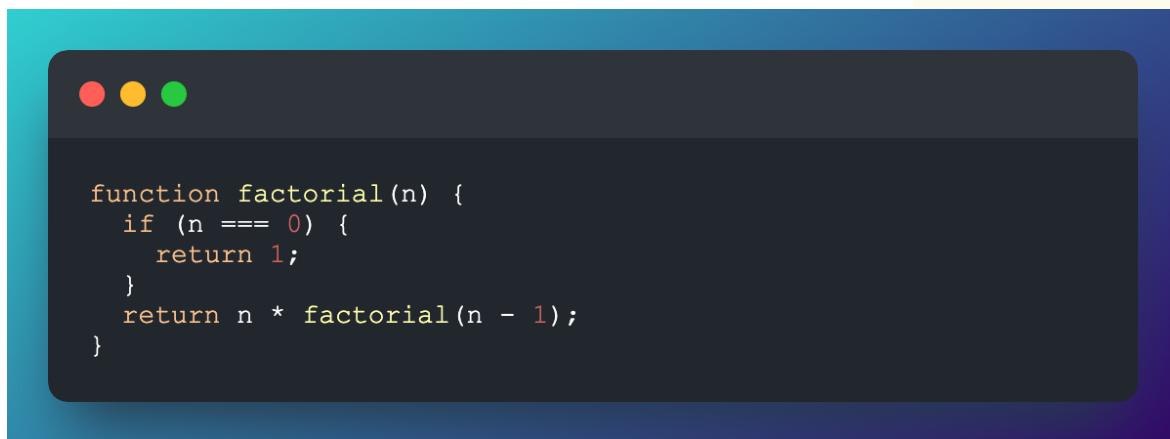
Ejemplo:

Un ejemplo de recursividad en JavaScript es una función para calcular el factorial de un número. El factorial de un número es el producto de todos los números enteros

## 7 FUNCIONES

---

positivos menores o iguales que el número dado. Por ejemplo, el factorial de 5 es  $5 * 4 * 3 * 2 * 1 = 120$ .



```
function factorial(n) {  
    if (n === 0) {  
        return 1;  
    }  
    return n * factorial(n - 1);  
}
```

Reto:

Crea una función recursiva para calcular el  $n$ -ésimo número de Fibonacci. El  $n$ -ésimo número de Fibonacci es el resultado de la suma de los dos números anteriores en la secuencia. Por ejemplo, el número de Fibonacci de 5 es 5 ( $3 + 2$ ).

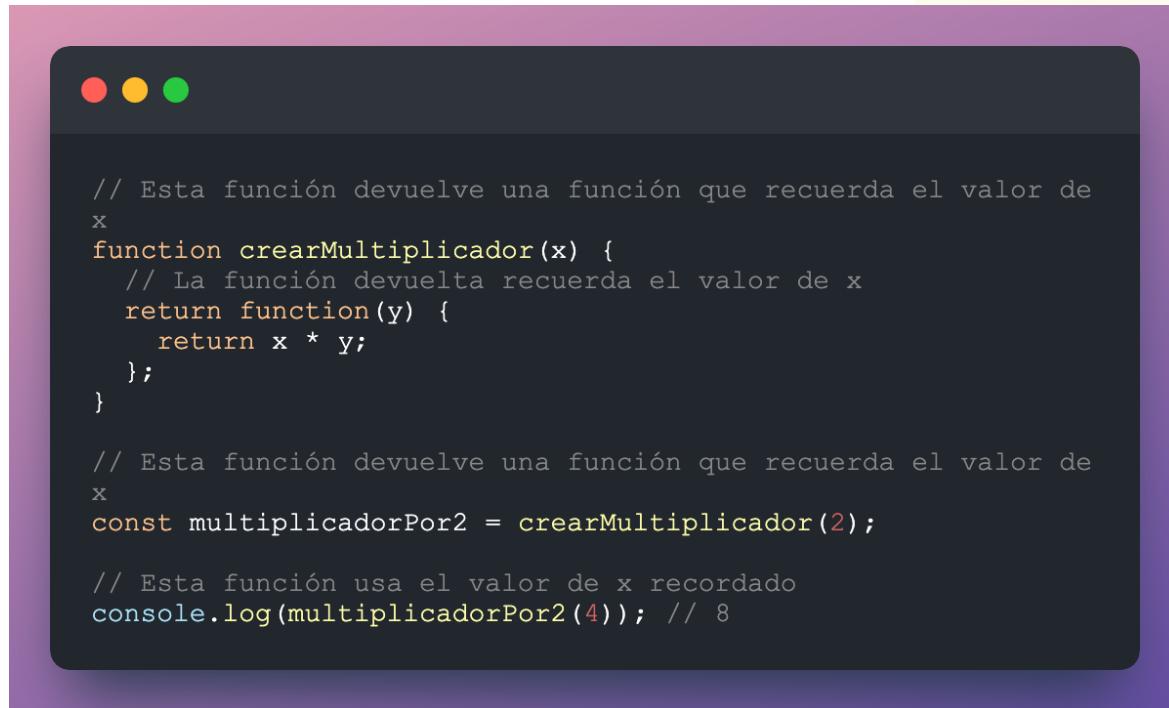
### 7.13 Cierres (Closures)

Los cierres (closures) en JavaScript son una característica de la programación funcional que permite a una función recordar y acceder a su entorno de ejecución, incluso cuando la función se ha ejecutado fuera de su entorno original. Esto significa que una función puede recordar y acceder a las variables y funciones definidas en su entorno de ejecución, incluso cuando se ejecuta fuera de él.

Ejemplo:

## 7 FUNCIONES

---



```
// Esta función devuelve una función que recuerda el valor de x
function crearMultiplicador(x) {
    // La función devuelta recuerda el valor de x
    return function(y) {
        return x * y;
    };
}

// Esta función devuelve una función que recuerda el valor de x
const multiplicadorPor2 = crearMultiplicador(2);

// Esta función usa el valor de x recordado
console.log(multiplicadorPor2(4)); // 8
```

Reto:

Crea una función que tome una cadena como argumento y devuelva una función que tome una cadena como argumento y devuelva la cadena original concatenada con la cadena pasada como argumento.

### 7.14 Funciones de flecha

Las funciones de flecha son una forma abreviada de escribir funciones en JavaScript. Estas funciones son similares a las funciones regulares, pero tienen una sintaxis más corta. Esto significa que puedes escribir una función en una sola línea. Estas funciones también son conocidas como funciones lambda, ya que se basan en la notación lambda de la programación funcional.

Ejemplo:

## 7 FUNCIONES

---

```
// Función de flecha  
const sumar = (a, b) => a + b;  
  
// Función normal  
function sumar(a, b) {  
    return a + b;  
}
```

Reto:

Crea una función de flecha que tome dos números como argumentos y devuelva su producto.

### 7.15 Funciones predefinidas

Las funciones predefinidas en JavaScript son aquellas que ya vienen incluidas en el lenguaje y que nos permiten realizar tareas comunes de forma sencilla.

Un ejemplo práctico de una función predefinida en JavaScript es la función Math.max(), que nos permite encontrar el número más grande de una lista de números. Por ejemplo, si queremos encontrar el número más grande de la lista [1, 5, 10, 3], podemos usar la función Math.max() de la siguiente manera:

```
Math.max(1, 5, 10, 3); // Devuelve 10
```

## 8 USO AVANZADO DE DATOS

---

Reto:

Utiliza la función Math.min() para encontrar el número más pequeño de la lista [2, 4, 6, 8].

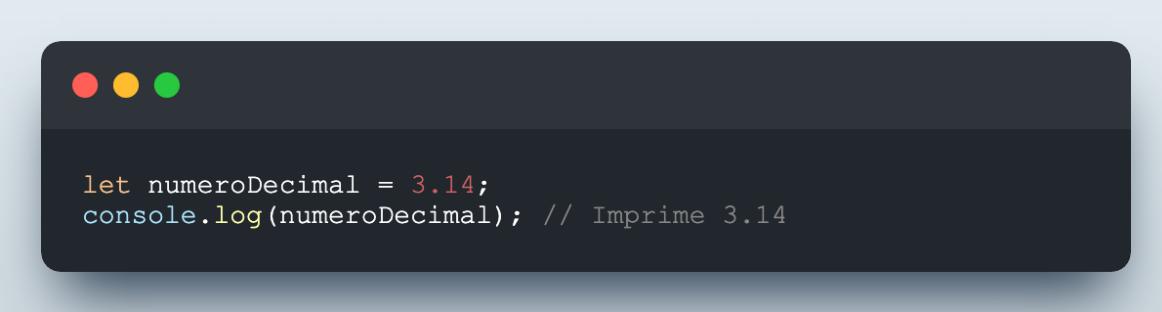
# 8 Uso avanzado de datos

## 8.1 Uso de números

### 8.1.1 Numeros decimales

Los números decimales en JavaScript son números que contienen un punto decimal. Estos números se usan para representar fracciones o números con decimales.

Ejemplo:



```
● ● ●
let numeroDecimal = 3.14;
console.log(numeroDecimal); // Imprime 3.14
```

Reto:

Crea una variable llamada numeroDecimal y asígnale un número decimal. Luego, imprime el valor de la variable en la consola.

### 8.1.2 Numeros binarios

Los números binarios son un sistema de numeración que utiliza solo dos dígitos, 0 y 1. Estos números se utilizan en computadoras para representar información, ya que

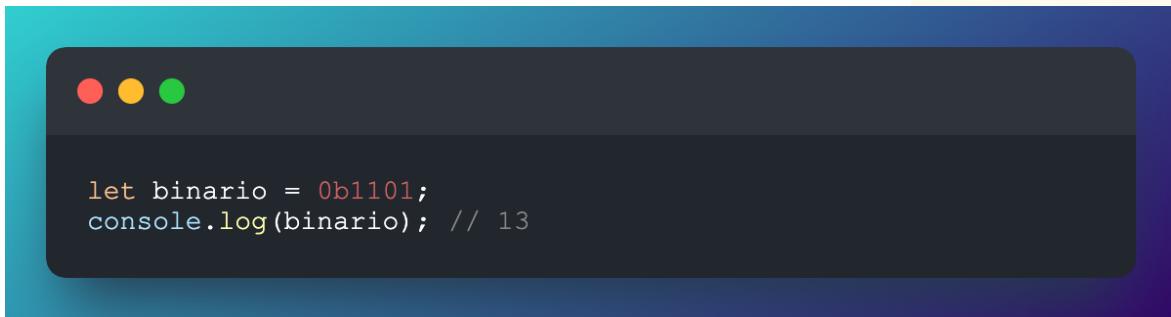
## 8 USO AVANZADO DE DATOS

---

los circuitos electrónicos de una computadora solo pueden entender dos estados: encendido o apagado. Estos dos estados se representan con 0 y 1, respectivamente.

En JavaScript, los números binarios se representan con el prefijo 0b. Por ejemplo, el número binario 1101 se representa como 0b1101.

Ejemplo:

A screenshot of a macOS terminal window. The window has a dark theme with red, yellow, and green window control buttons at the top. The main area contains the following code:

```
let binario = 0b1101;
console.log(binario); // 13
```

Reto:

Utiliza el prefijo 0b para convertir el número binario 1100 a decimal.

### 8.1.3 Números octales

Los números octales en JavaScript son una forma de representar números enteros usando una base 8. Esto significa que cada dígito en un número octal representa una potencia de 8. Los números octales se escriben con un prefijo de 0o (cero y la letra o).

Ejemplo:

## 8 USO AVANZADO DE DATOS

---

```
let octalNumber = 0o10;
console.log(octalNumber); // 8
```

Reto:

Escribe un programa que tome un número octal como entrada y lo convierta a un número decimal.

### 8.1.4 Números hexadecimales

Los números hexadecimales en JavaScript son una forma de representar números enteros usando una notación de base 16. Esta notación se usa para representar valores binarios en una forma más fácil de leer. Los números hexadecimales se escriben con un prefijo de 0x seguido de una secuencia de dígitos hexadecimales.

Ejemplo:

```
let hexNumber = 0xFF;
console.log(hexNumber); // 255
```

Reto:

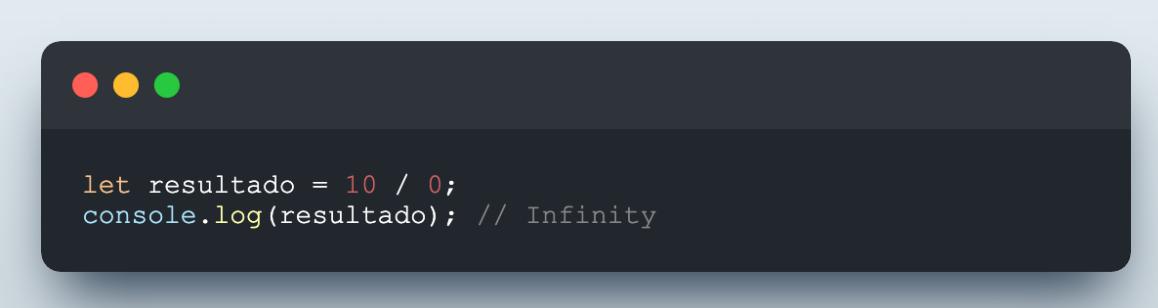
Crea una variable llamada hexNumber2 y asígnale el valor 0xA3. Luego, imprime el valor de la variable en la consola.

### 8.1.5 Infiniy

Infinity en JavaScript es un valor especial que representa un número infinito. Esto significa que no hay límite superior para el valor de Infinity.

Ejemplo:

Un ejemplo práctico de Infinity en JavaScript es cuando se usa para calcular el resultado de una división por cero. Por ejemplo, si dividimos 10 entre 0, el resultado será Infinity:



```
let resultado = 10 / 0;
console.log(resultado); // Infinity
```

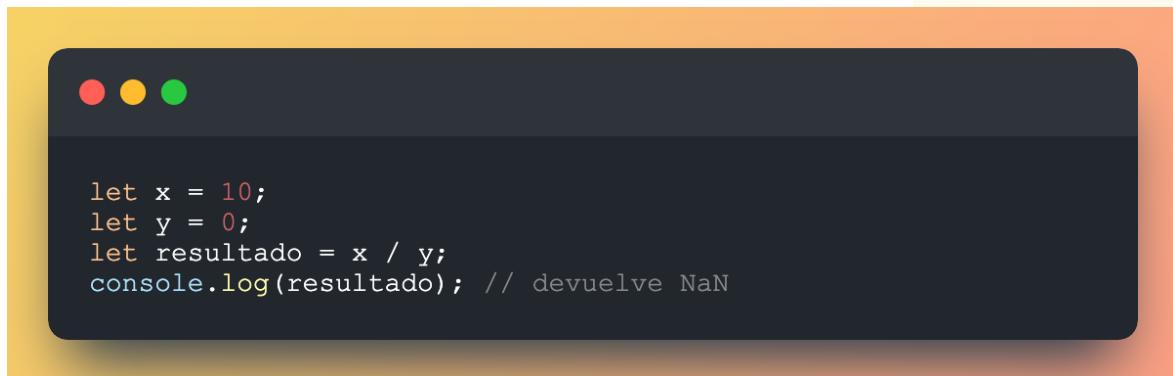
Reto:

Escribe una función que tome dos números como argumentos y devuelva el resultado de dividir el primer número entre el segundo. Si el segundo número es 0, la función debe devolver Infinity.

### 8.1.6 NaN

NaN (Not a Number) es un valor especial en JavaScript que se devuelve cuando una operación matemática no puede ser realizada. Por ejemplo, si intentamos dividir un número por cero, el resultado será NaN.

Ejemplo:



Reto:

Escribe una función llamada esNaN que tome un argumento y devuelva true si el argumento es NaN y false si no lo es.

### 8.1.7 Propiedades y métodos del objeto Number

El objeto Number en JavaScript es un objeto global que se utiliza para representar números. Tiene propiedades y métodos que permiten realizar operaciones con números.

**Propiedades** Las propiedades del objeto Number son:

- MAX\_VALUE: El número más grande que se puede representar en JavaScript.
- MIN\_VALUE: El número más pequeño que se puede representar en JavaScript.
- NEGATIVE\_INFINITY: Representa el valor infinito negativo.
- POSITIVE\_INFINITY: Representa el valor infinito positivo.
- NaN: Representa un valor no numérico.

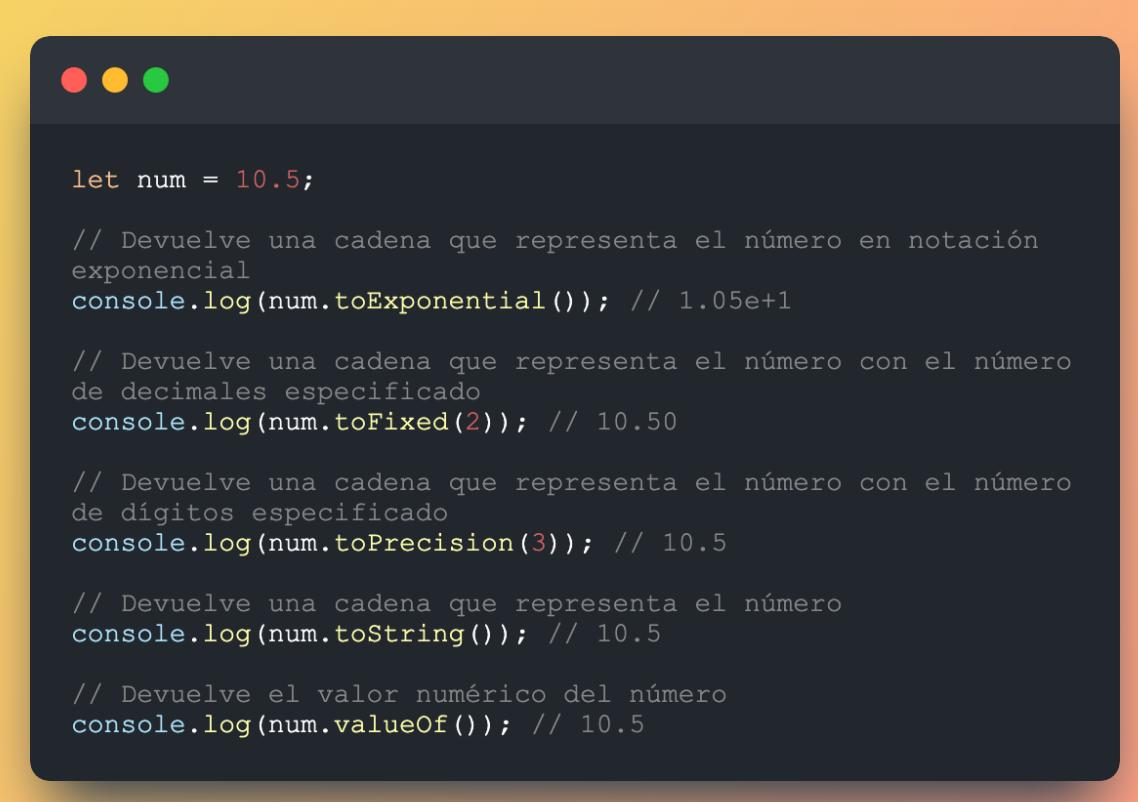
**Métodos** Los métodos del objeto Number son:

## 8 USO AVANZADO DE DATOS

---

- `toExponential()`: Devuelve una cadena que representa el número en notación exponencial.
- `toFixed()`: Devuelve una cadena que representa el número con el número de decimales especificado.
- `toPrecision()`: Devuelve una cadena que representa el número con el número de dígitos especificado.
- `toString()`: Devuelve una cadena que representa el número.
- `valueOf()`: Devuelve el valor numérico del número.

Ejemplo:



```
let num = 10.5;

// Devuelve una cadena que representa el número en notación exponencial
console.log(num.toExponential()); // 1.05e+1

// Devuelve una cadena que representa el número con el número de decimales especificado
console.log(num.toFixed(2)); // 10.50

// Devuelve una cadena que representa el número con el número de dígitos especificado
console.log(num.toPrecision(3)); // 10.5

// Devuelve una cadena que representa el número
console.log(num.toString()); // 10.5

// Devuelve el valor numérico del número
console.log(num.valueOf()); // 10.5
```

Reto:

Utiliza los métodos del objeto Number para crear una función que tome un número

## 8 USO AVANZADO DE DATOS

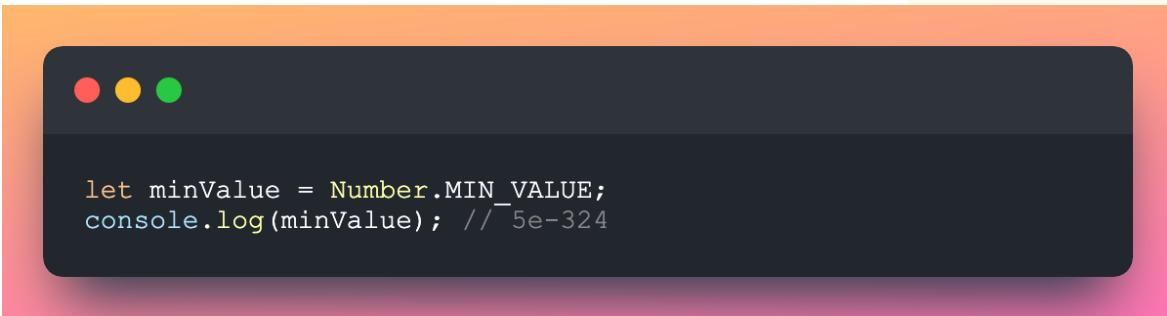
---

como parámetro y devuelva una cadena con el número formateado con dos decimales.

### 8.1.8 Propiedades comunes del objeto Number

**Number.MIN\_VALUE** Number.MIN\_VALUE es una constante de JavaScript que representa el número más pequeño representable en una variable de tipo Number. Esta constante es igual a 5e-324.

Ejemplo:



```
let minValue = Number.MIN_VALUE;
console.log(minValue); // 5e-324
```

Reto:

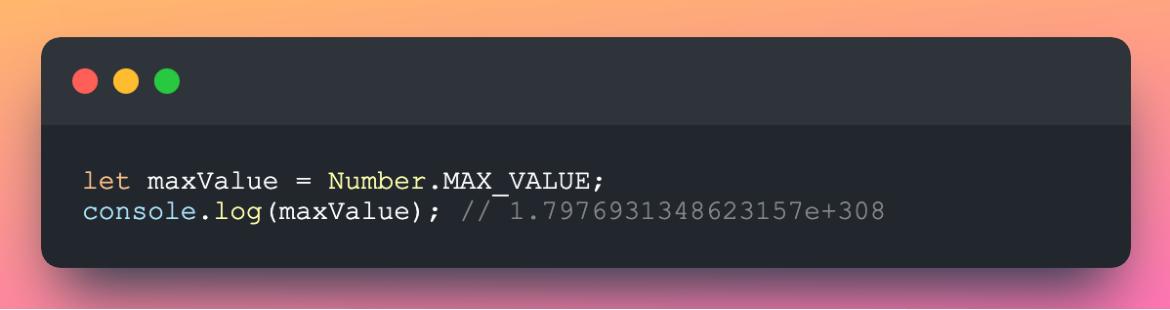
Crea una variable llamada maxValue y asígnale el valor de Number.MAX\_VALUE. Luego, imprime el valor de maxValue en la consola.

**Number.MAX\_VALUE** Number.MAX\_VALUE en JavaScript es una constante que representa el número más grande que se puede representar en JavaScript. Esta constante es igual a 1.7976931348623157e+308.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---



```
let maxValue = Number.MAX_VALUE;
console.log(maxValue); // 1.7976931348623157e+308
```

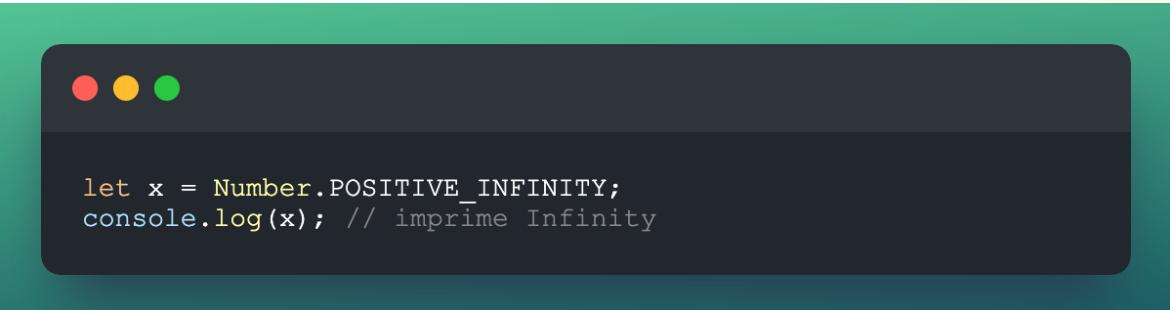
Reto:

Utiliza la constante Number.MAX\_VALUE para calcular el resultado de la siguiente operación:

1.7976931348623157e+308 + 1

**Number.POSITIVE\_INFINITY** Number.POSITIVE\_INFINITY es una propiedad de JavaScript que representa el valor infinito positivo. Esto significa que el valor es mayor que cualquier otro número.

Ejemplo:



```
let x = Number.POSITIVE_INFINITY;
console.log(x); // imprime Infinity
```

Reto:

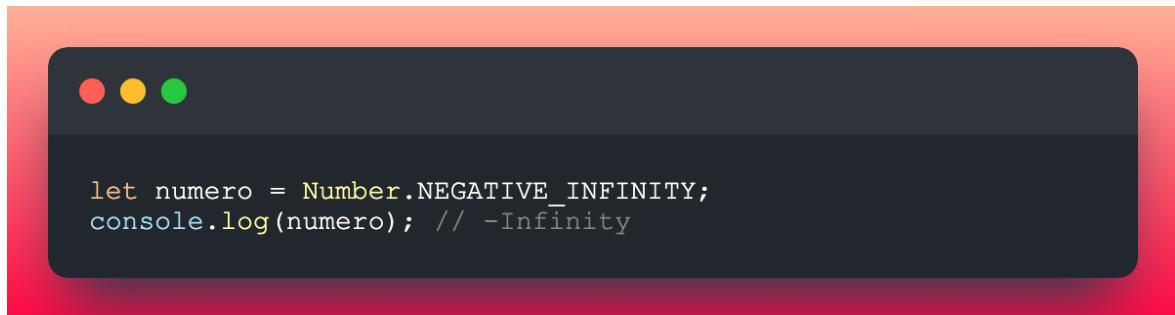
Crea una variable llamada y y asígnale el valor de Number.POSITIVE\_INFINITY. Luego, imprime el valor de y en la consola.

## 8 USO AVANZADO DE DATOS

---

**Number.NEGATIVE\_INFINITY** Number.NEGATIVE\_INFINITY es una propiedad de JavaScript que representa el número infinito negativo. Esta propiedad se utiliza para representar un valor numérico que es menor que cualquier otro número.

Ejemplo:



The screenshot shows a dark-themed browser developer tools console window. At the top, there are three colored circular icons: red, yellow, and green. Below them, the console displays the following code and its output:

```
let numero = Number.NEGATIVE_INFINITY;
console.log(numero); // -Infinity
```

Reto:

Crea una función que tome dos números como parámetros y devuelva el número más pequeño. Si alguno de los números es Number.NEGATIVE\_INFINITY, la función debe devolver Number.NEGATIVE\_INFINITY.

**Number.MAX\_SAFE\_INTEGER** Number.MAX\_SAFE\_INTEGER en JavaScript es una constante que representa el número entero más grande que se puede representar de manera segura en JavaScript. Esto significa que cualquier número mayor que esta constante puede perder precisión al ser representado. El valor de esta constante es 9007199254740991.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---

```
// Verificar si un número es mayor que
Number.MAX_SAFE_INTEGER
let num = 9007199254740992;

if (num > Number.MAX_SAFE_INTEGER) {
    console.log('El número es mayor que
Number.MAX_SAFE_INTEGER');
} else {
    console.log('El número es menor o igual que
Number.MAX_SAFE_INTEGER');
}
// Output: El número es mayor que Number.MAX_SAFE_INTEGER
```

Reto:

Crea una función que tome un número como argumento y devuelva true si el número es menor o igual que Number.MAX\_SAFE\_INTEGER y false si el número es mayor.

### 8.1.9 Propiedades y métodos del objeto Math

El objeto Math en JavaScript es un objeto global que proporciona métodos y propiedades para realizar operaciones matemáticas.

**Propiedades** Las propiedades del objeto Math son constantes que contienen valores matemáticos específicos. Algunos ejemplos de propiedades son:

- Math.PI: El número pi (3.14159…)
- Math.E: El número e (2.71828…)
- Math.SQRT2: La raíz cuadrada de 2 (1.41421…)

## 8 USO AVANZADO DE DATOS

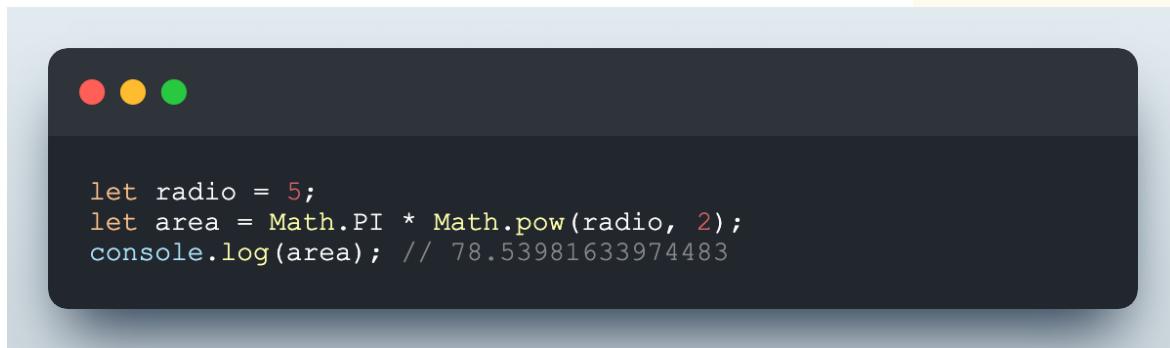
---

**Métodos** Los métodos del objeto Math son funciones que realizan operaciones matemáticas. Algunos ejemplos de métodos son:

- Math.abs(): Devuelve el valor absoluto de un número.
- Math.ceil(): Devuelve el entero más pequeño mayor o igual al número especificado.
- Math.floor(): Devuelve el entero más grande menor o igual al número especificado.

Ejemplo:

Supongamos que queremos calcular el área de un círculo con un radio de 5. Podemos usar el método Math.PI para obtener el valor de pi y el método Math.pow() para elevar el radio al cuadrado. El código para calcular el área sería el siguiente:



```
let radio = 5;
let area = Math.PI * Math.pow(radio, 2);
console.log(area); // 78.53981633974483
```

Reto:

Utiliza los métodos y propiedades del objeto Math para calcular el área de un rectángulo con un ancho de 10 y un alto de 5.

### 8.1.10 Propiedades y métodos del objeto BigInt

BigInt es un nuevo tipo de datos en JavaScript que permite representar números enteros de tamaño arbitrario. Esto significa que los números enteros más grandes que se pueden representar con el tipo de datos Number de JavaScript ya no son un problema.

## 8 USO AVANZADO DE DATOS

---

### Propiedades

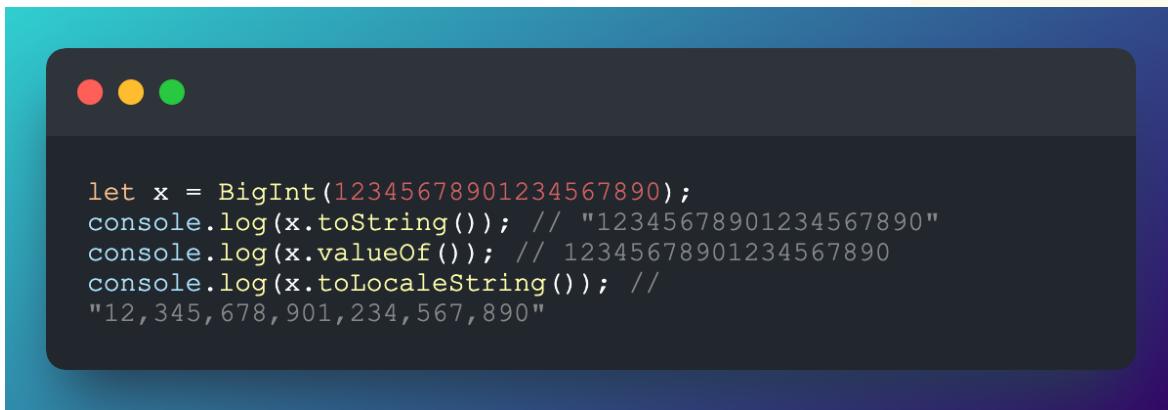
- **MAX\_SAFE\_INTEGER**: Esta propiedad es una constante que contiene el número entero más grande que se puede representar con seguridad en JavaScript.
- **MIN\_SAFE\_INTEGER**: Esta propiedad es una constante que contiene el número entero más pequeño que se puede representar con seguridad en JavaScript.

### Métodos

- **BigInt()**: Esta función se usa para convertir un número entero a un BigInt.
- **toString()**: Esta función se usa para convertir un BigInt a una cadena.
- **valueOf()**: Esta función se usa para convertir un BigInt a un número entero.
- **toLocaleString()**: Esta función se usa para convertir un BigInt a una cadena con formato local.

Ejemplo:

A continuación se muestra un ejemplo de cómo usar BigInt en JavaScript:



The screenshot shows a browser's developer tools console window. At the top, there are three colored circular icons (red, yellow, green). Below them, the console output is displayed in white text on a dark background. The code shown is:

```
let x = BigInt(12345678901234567890);
console.log(x.toString()); // "12345678901234567890"
console.log(x.valueOf()); // 12345678901234567890
console.log(x.toLocaleString()); //
"12,345,678,901,234,567,890"
```

Reto:

Intenta crear una función que tome un BigInt como argumento y devuelva el número entero más grande que se puede representar con seguridad en JavaScript.

### 8.2 Uso de fechas

#### 8.2.1 Propiedades y métodos del objeto Date

El objeto Date en JavaScript es un objeto nativo que nos permite trabajar con fechas y horas. Está diseñado para ser compatible con la mayoría de los navegadores y sistemas operativos.

**Propiedades** Las propiedades del objeto Date nos permiten obtener información sobre la fecha y hora actual. Algunas de las propiedades más comunes son:

- `getFullYear()`: Devuelve el año actual.
- `getMonth()`: Devuelve el mes actual.
- `getDate()`: Devuelve el día del mes actual.
- `getHours()`: Devuelve la hora actual.
- `getMinutes()`: Devuelve los minutos actuales.

**Métodos** Los métodos del objeto Date nos permiten realizar operaciones con fechas y horas. Algunos de los métodos más comunes son:

- `setFullYear()`: Establece el año actual.
- `setMonth()`: Establece el mes actual.
- `setDate()`: Establece el día del mes actual.
- `setHours()`: Establece la hora actual.
- `setMinutes()`: Establece los minutos actuales.

Ejemplo:

A continuación se muestra un ejemplo de cómo usar el objeto Date para obtener la fecha y hora actuales:

## 8 USO AVANZADO DE DATOS

---

```
let fecha = new Date();
let anio = fecha.getFullYear();
let mes = fecha.getMonth();
let dia = fecha.getDate();
let hora = fecha.getHours();
let minutos = fecha.getMinutes();

console.log(`Fecha y hora actuales: ${dia}/${mes}/${anio}
${hora}:${minutos}`);
```

Reto:

Utiliza el objeto Date para crear una función que devuelva la fecha y hora de una semana a partir de la fecha y hora actuales.

### 8.3 Uso de texto

#### 8.3.1 Secuencias de escape

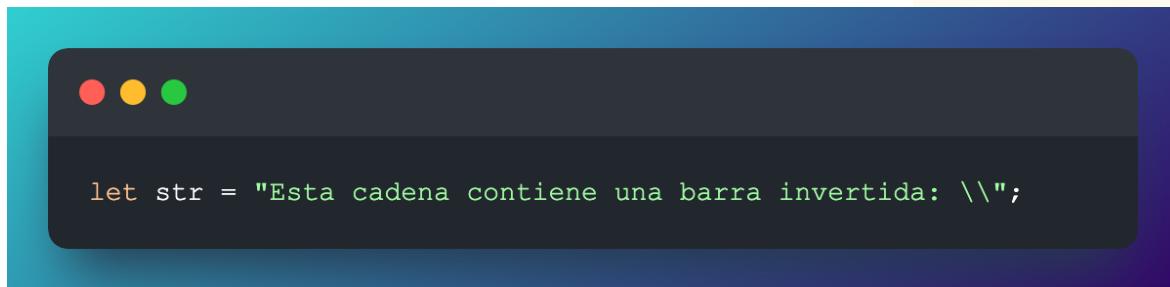
Las secuencias de escape en JavaScript son caracteres especiales que se usan para representar caracteres especiales o caracteres de control. Estas secuencias de escape se escriben como una barra invertida seguida de una letra o un número.

Ejemplo:

Por ejemplo, para mostrar una barra invertida en una cadena de texto, se usa la secuencia de escape \. Por lo tanto, la siguiente cadena de texto:

## 8 USO AVANZADO DE DATOS

---



se mostrará como:

Esta cadena contiene una barra invertida: \

Reto:

Escribe una cadena de texto que contenga una tabulación y un salto de línea. Usa la secuencia de escape adecuada para mostrar el resultado correcto.

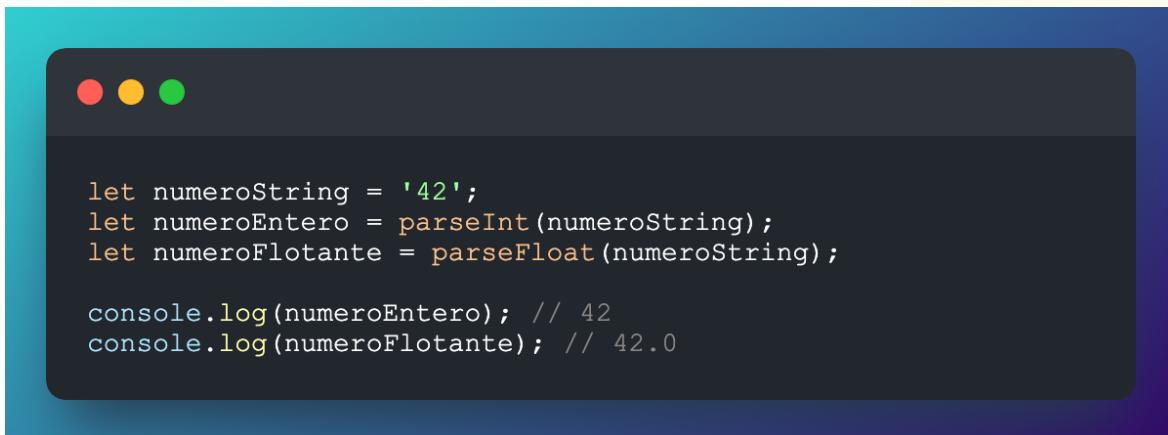
### 8.3.2 Convertir texto a números

En JavaScript, podemos convertir una cadena de texto a un número usando la función `parseInt()` o `parseFloat()`. Estas funciones toman una cadena de texto como argumento y devuelven un número entero o un número de punto flotante, respectivamente.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---



```
let numeroString = '42';
let numeroEntero = parseInt(numeroString);
let numeroFlotante = parseFloat(numeroString);

console.log(numeroEntero); // 42
console.log(numeroFlotante); // 42.0
```

Reto:

Crea una función que tome una cadena de texto como argumento y devuelva un número entero. La cadena de texto puede contener números enteros o decimales. Si la cadena de texto contiene un número decimal, la función debe devolver el número entero más cercano.

### 8.3.3 Propiedades y métodos del objeto String

El objeto String en JavaScript es un objeto global que se utiliza para representar cadenas de texto. Está compuesto por una serie de propiedades y métodos que permiten manipular y trabajar con cadenas de texto.

**Propiedades** Las propiedades del objeto String son:

- **length**: Esta propiedad devuelve la longitud de la cadena de texto.

**Métodos** Los métodos del objeto String son:

- **charAt()**: Esta función devuelve el carácter en la posición especificada de la cadena de texto.

## 8 USO AVANZADO DE DATOS

---

- `indexOf()`: Esta función devuelve la posición de la primera aparición de un carácter o subcadena en la cadena de texto.
- `replace()`: Esta función reemplaza un carácter o subcadena por otro carácter o subcadena en la cadena de texto.

Ejemplo:

Supongamos que queremos contar el número de veces que aparece una letra en una cadena de texto. Para ello, podemos utilizar el método `indexOf()` para encontrar la primera aparición de la letra y luego utilizar el método `replace()` para reemplazarla por una cadena vacía. Luego, podemos contar el número de veces que se ha reemplazado la letra para obtener el número de veces que aparece en la cadena de texto.



```
let cadena = "Hola mundo";
let letra = "o";
let contador = 0;

while (cadena.indexOf(letra) != -1) {
    contador++;
    cadena = cadena.replace(letra, "");
}

console.log(`La letra "${letra}" aparece ${contador} veces en
la cadena "${cadena}"`);
```

Reto:

Crea una función que reciba una cadena de texto y un carácter como parámetros y devuelva el número de veces que aparece el carácter en la cadena de texto.

## 8 USO AVANZADO DE DATOS

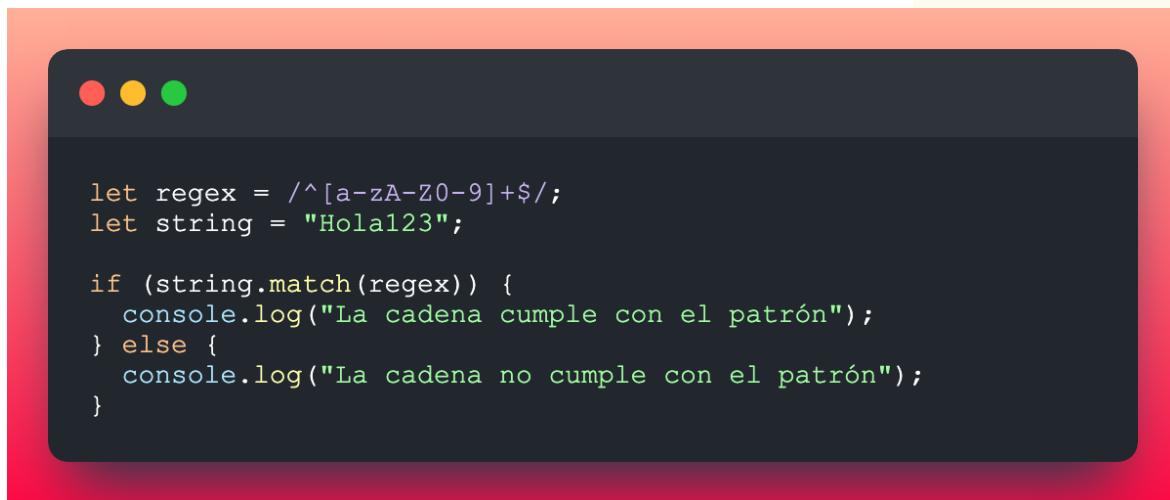
---

### 8.4 Uso de expresiones regulares

#### 8.4.1 Crear una expresión regular

Una expresión regular en JavaScript es una secuencia de caracteres que forma un patrón de búsqueda, que se utiliza para encontrar una coincidencia con una cadena de texto dada. Las expresiones regulares son tema bastante avanzado y complejo por lo que requieren su propio libro.

Ejemplo:



```
let regex = /^[a-zA-Z0-9]+$/;
let string = "Hola123";

if (string.match(regex)) {
    console.log("La cadena cumple con el patrón");
} else {
    console.log("La cadena no cumple con el patrón");
}
```

Reto:

Busca en línea la expresión regular para validar una dirección de correo electrónico e intenta decifrar lo que hace.

#### 8.4.2 Búsqueda con banderas en expresiones regulares

Las búsquedas con banderas en expresiones regulares en JavaScript son una forma de modificar el comportamiento de una expresión regular. Estas banderas se

## 8 USO AVANZADO DE DATOS

---

colocan al final de la expresión regular y se usan para cambiar el comportamiento de la búsqueda.

Las banderas más comunes son:

- i: Esta bandera hace que la búsqueda sea insensible a mayúsculas y minúsculas.
- g: Esta bandera hace que la búsqueda sea global, es decir, que busque todas las coincidencias en el texto.
- m: Esta bandera hace que la búsqueda sea multilínea, es decir, que busque coincidencias en todas las líneas del texto.

Ejemplo:

Supongamos que queremos buscar la palabra “hola” en un texto. Si usamos la expresión regular /hola/ sin banderas, sólo encontrará la primera coincidencia.

Si usamos la expresión regular /hola/g, buscará todas las coincidencias en el texto.

Reto:

Escribe una expresión regular que busque todas las palabras que empiecen por “hola” en un texto, sin importar si están en mayúsculas o minúsculas.

### 8.4.3 Propiedades y métodos del objeto RegExp

El objeto RegExp en JavaScript es una expresión regular que se utiliza para realizar búsquedas de patrones en cadenas de texto. Está formado por una serie de propiedades y métodos que permiten realizar operaciones sobre cadenas de texto.

**Propiedades** Las propiedades del objeto RegExp son:

- global: Indica si la búsqueda debe ser global o no.

## 8 USO AVANZADO DE DATOS

---

- ignoreCase: Indica si la búsqueda debe ser sensible a mayúsculas y minúsculas o no.
- multiline: Indica si la búsqueda debe ser realizada en múltiples líneas o no.
- lastIndex: Indica el índice de la última coincidencia encontrada.
- source: Devuelve la expresión regular como una cadena.

**Métodos** Los métodos del objeto RegExp son:

- exec(): Busca una coincidencia en una cadena de texto y devuelve un array con los resultados.
- test(): Busca una coincidencia en una cadena de texto y devuelve un valor booleano.

### Métodos de texto con RegExp

- match(): Busca una coincidencia en una cadena de texto y devuelve un array con los resultados.
- search(): Busca una coincidencia en una cadena de texto y devuelve el índice de la primera coincidencia encontrada.
- replace(): Reemplaza una coincidencia en una cadena de texto con una cadena de reemplazo.

Ejemplo:

Supongamos que queremos buscar todas las palabras que comienzan con la letra “a” en una cadena de texto. Para ello, podemos utilizar el siguiente código:



```
let texto = "Hola, mi nombre es Ana";
let regex = /\ba\w+/g;
let resultado = texto.match(regex);

console.log(resultado); // ["Ana"]
```

En este ejemplo, hemos creado una expresión regular con la propiedad global para buscar todas las palabras que comienzan con la letra “a” en la cadena de texto. Luego, hemos utilizado el método `match()` para buscar las coincidencias y almacenar los resultados en un array.

Reto:

Utiliza el objeto `RegExp` para buscar todas las palabras que comienzan con la letra “b” en una cadena de texto.

### 8.5 Uso de listas

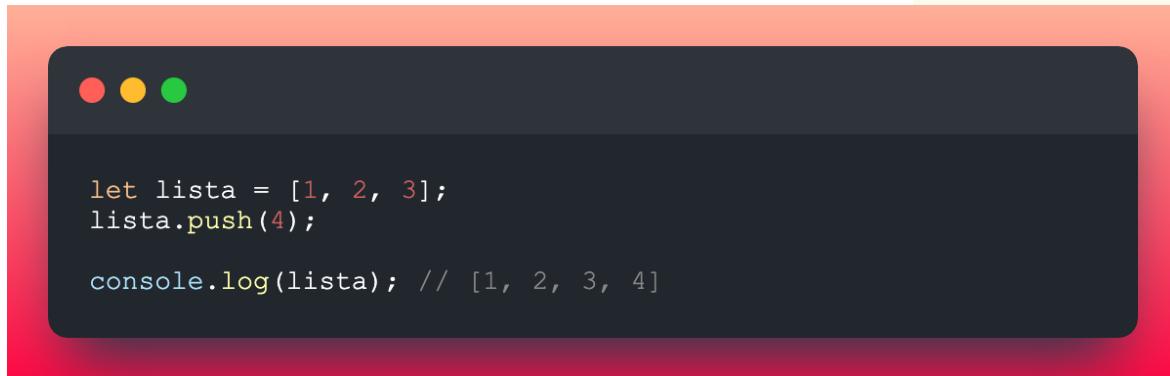
#### 8.5.1 Insertar elementos en una lista

Insertar elementos en una lista en JavaScript es una tarea sencilla. Esto se puede hacer usando el método `push()`. El método `push()` agrega un elemento al final de una lista y devuelve la nueva longitud de la lista.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---



A screenshot of a terminal window with a dark background and red, yellow, and green window control buttons at the top. The terminal displays the following JavaScript code:

```
let lista = [1, 2, 3];
lista.push(4);

console.log(lista); // [1, 2, 3, 4]
```

Reto:

Crea una lista con los números del 1 al 5 y agrega el número 6 al final de la lista usando el método push().

### 8.5.2 Modificar elementos de listas

Modificar elementos de listas en JavaScript es una tarea sencilla. Esto se puede lograr mediante el uso de los métodos push(), pop(), shift() y unshift(). Estos métodos permiten agregar, eliminar, mover y reemplazar elementos de una lista.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---

```
let lista = ["manzana", "plátano", "naranja"];  
  
// Reemplazar un elemento en el índice 0 de una lista  
lista[0] = "apple"; // ["apple", "plátano", "naranja"]  
  
// Agregar un elemento al final de la lista  
lista.push("uva"); // ["apple", "plátano", "naranja", "uva"]  
  
// Eliminar el último elemento de la lista  
lista.pop(); // ["apple", "plátano", "naranja"]  
  
// Mover el primer elemento de la lista al final  
lista.push(lista.shift()); // ["plátano", "naranja", "apple"]  
  
// Reemplazar el segundo elemento de la lista  
lista.splice(1, 1, "kiwi"); // ["plátano", "kiwi", "apple"]  
  
console.log(lista); // ["plátano", "kiwi", "apple"]
```

Reto:

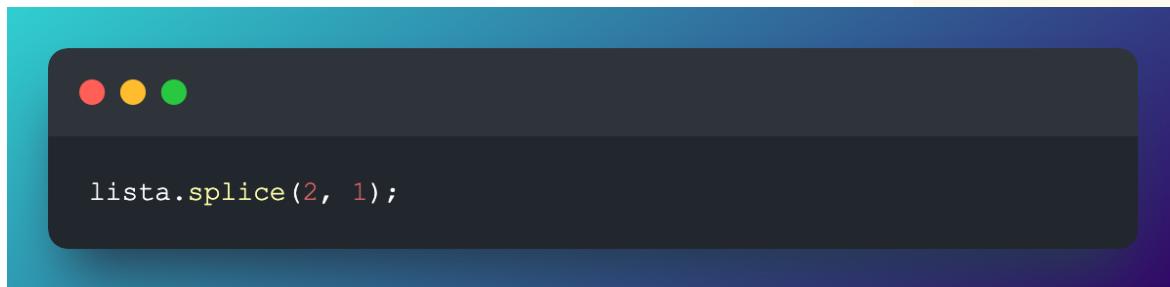
Crea una lista con 5 elementos y usa los métodos push(), pop(), shift() y unshift() para modificar los elementos de la lista.

### 8.5.3 Borrar elementos de listas

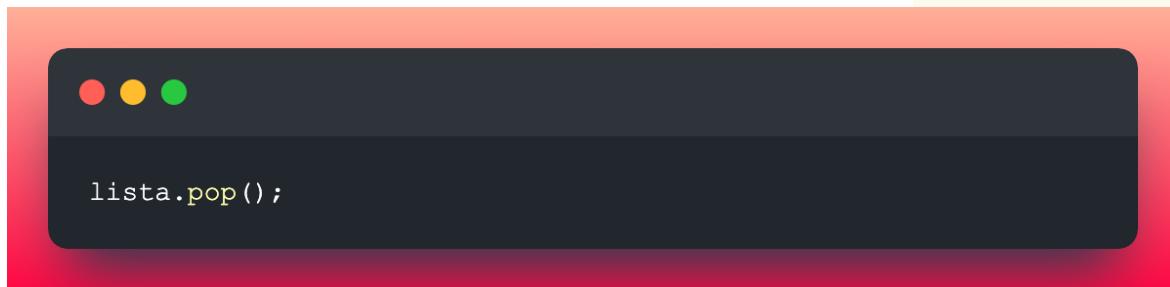
En JavaScript, hay varias formas de borrar elementos de una lista. Una forma es usar el método splice(). Este método toma dos argumentos: el índice del elemento que se desea borrar y la cantidad de elementos que se desean borrar. Por ejemplo, para borrar el tercer elemento de una lista, se usaría el siguiente código:

## 8 USO AVANZADO DE DATOS

---



Otra forma de borrar elementos de una lista es usar el método `pop()`. Este método borra el último elemento de la lista. Por ejemplo, para borrar el último elemento de una lista, se usaría el siguiente código:



Reto:

Crea una lista con 5 elementos y borra el tercer elemento usando el método `splice()`.

### 8.5.4 Propiedades y métodos del objeto Array

El objeto Array en JavaScript es un objeto global que se utiliza para almacenar y manipular datos. Está formado por una serie de propiedades y métodos que permiten trabajar con los datos de una forma más eficiente.

**Propiedades** Las propiedades del objeto Array en JavaScript son:

- `length`: Esta propiedad devuelve el número de elementos en el array.

## 8 USO AVANZADO DE DATOS

---

**Métodos** Los métodos del objeto Array en JavaScript son:

- `push()`: Este método añade uno o más elementos al final del array.
- `pop()`: Este método elimina el último elemento del array.
- `shift()`: Este método elimina el primer elemento del array.
- `unshift()`: Este método añade uno o más elementos al principio del array.
- `sort()`: Este método ordena los elementos del array.

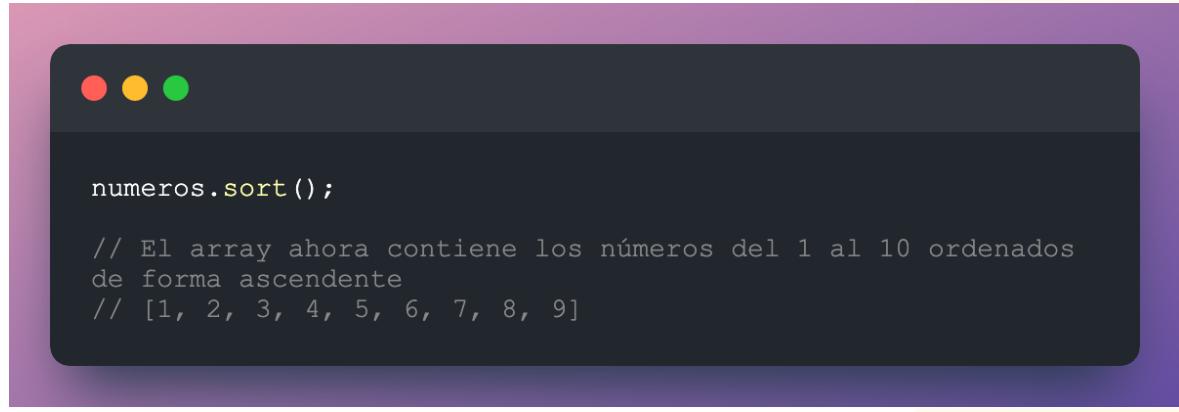
Ejemplo:

Supongamos que tenemos un array con los números del 1 al 10:



```
let numeros = [ 6, 2, 3, 4, 5, 7, 8, 1, 9];
```

Podemos usar el método `sort()` para ordenar los elementos del array de forma ascendente:



```
numeros.sort();

// El array ahora contiene los números del 1 al 10 ordenados
// de forma ascendente
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Reto:

Crea un array con los nombres de tus amigos y usa el método `sort()` para ordenarlos alfabéticamente.

## 8 USO AVANZADO DE DATOS

---

### 8.5.5 Propiedades comunes de listas

**length** length es una propiedad de los objetos de JavaScript que devuelve la longitud de una cadena, un array o un objeto. Esta propiedad es útil para determinar el número de elementos en una cadena, un array o un objeto.

Ejemplo:

Supongamos que tenemos un array llamado “colores” que contiene los siguientes elementos:

```
var colores = ["rojo", "azul", "verde", "amarillo"];
```

Podemos usar la propiedad length para determinar el número de elementos en el array:

```
var numElementos = colores.length;  
// numElementos = 4
```

Reto:

Crea una función llamada “contarElementos” que tome un array como parámetro y devuelva el número de elementos en el array.

## 8 USO AVANZADO DE DATOS

---

### 8.5.6 Métodos comunes de listas

**Métodos de lista concat() y join()** Los métodos de lista concat() y join() son dos métodos de JavaScript que se utilizan para manipular cadenas de texto. El método concat() se utiliza para unir dos o más cadenas de texto, mientras que el método join() se utiliza para unir los elementos de una matriz en una sola cadena de texto.

**Método concat()** El método concat() se utiliza para unir dos o más cadenas de texto. Acepta uno o más argumentos y devuelve una nueva cadena de texto que contiene la unión de los argumentos.

Ejemplo:



```
let str1 = "Hola";
let str2 = "Mundo";
let str3 = str1.concat(" ", str2);

console.log(str3); // "Hola Mundo"
```

Reto:

Utiliza el método concat() para unir tres cadenas de texto y guarda el resultado en una variable.

**Métodos de lista indexOf() y find()** Los métodos de lista indexOf() y find() son métodos de JavaScript que se utilizan para buscar elementos en una lista. El método indexOf() busca un elemento específico en una lista y devuelve su índice, mientras que el método find() busca un elemento que cumpla con una condición específica y devuelve el elemento.

## 8 USO AVANZADO DE DATOS

---

**indexOf()** El método indexOf() busca un elemento específico en una lista y devuelve su índice. Si el elemento no se encuentra en la lista, devuelve -1.

Ejemplo:

```
● ● ●  
let lista = [1, 2, 3, 4, 5];  
let elemento = 3;  
  
let indice = lista.indexOf(elemento);  
  
console.log(indice); // 2
```

Reto:

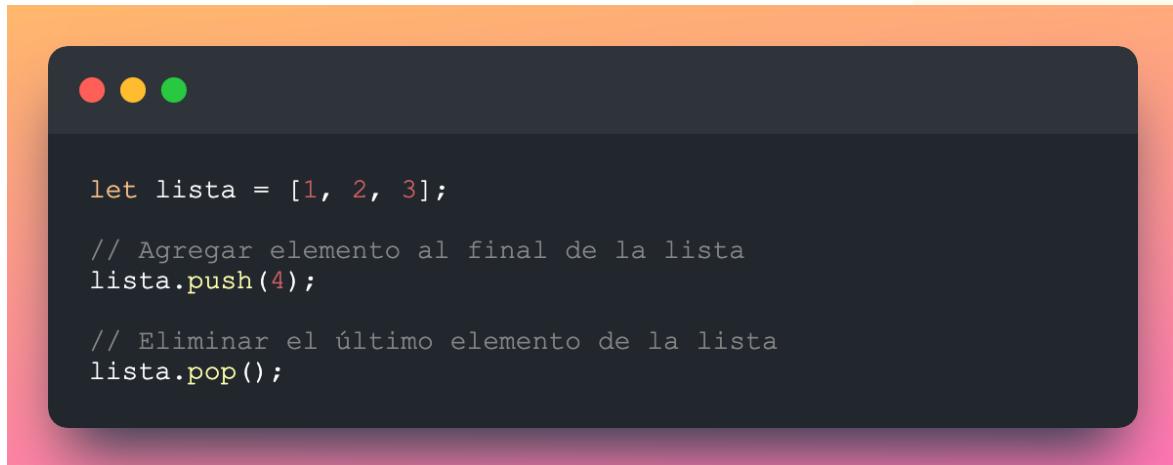
Utiliza el método indexOf() para buscar un elemento en una lista y guardar su índice en una variable.

**Método de lista push() y pop()** Los métodos push() y pop() son métodos de lista en JavaScript que se utilizan para agregar y eliminar elementos de una lista. El método push() agrega un elemento al final de la lista, mientras que el método pop() elimina el último elemento de la lista.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---



Reto:

Crea una lista con los números del 1 al 10 y usa los métodos push() y pop() para agregar y eliminar elementos de la lista.

**Método de lista shift() y unshift()** Los métodos shift() y unshift() son métodos de lista en JavaScript que se utilizan para agregar y eliminar elementos de una lista. El método shift() elimina el primer elemento de una lista y devuelve el elemento eliminado. El método unshift() agrega un elemento al principio de una lista y devuelve la nueva longitud de la lista.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---

```
let lista = [1, 2, 3, 4, 5];

// Eliminar el primer elemento de la lista
let eliminado = lista.shift();

console.log(lista); // [2, 3, 4, 5]
console.log(eliminado); // 1

// Agregar un elemento al principio de la lista
let nuevaLongitud = lista.unshift(0);

console.log(lista); // [0, 2, 3, 4, 5]
console.log(nuevaLongitud); // 5
```

Reto:

Crea una lista con los números del 1 al 5 y usa los métodos shift() y unshift() para eliminar y agregar elementos a la lista.

**Método de lista slice() y splice()** El método slice() se usa para extraer una parte de una matriz y devolver una nueva matriz. Esta nueva matriz contiene los elementos extraídos, desde el índice inicial hasta el índice final (no incluido). El método splice() se usa para agregar o eliminar elementos de una matriz.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---

```
// Usando slice()
var frutas = ["Banana", "Naranja", "Limón", "Manzana",
"Pera"];
var citrus = frutas.slice(1, 3);

console.log(citrus); // ["Naranja", "Limón"]

// Usando splice()
var frutas = ["Banana", "Naranja", "Limón", "Manzana",
"Pera"];
var eliminadas = frutas.splice(2, 2, "Uva", "Melón");

console.log(frutas); // ["Banana", "Naranja", "Uva", "Melón",
"Pera"]
console.log(eliminadas); // ["Limón", "Manzana"]
```

Reto:

Crea una matriz con los nombres de los meses del año y usa slice() para extraer los meses de verano (junio, julio y agosto) y guardalos en una nueva matriz.

**Método de lista sort() y reverse()** El método sort() en JavaScript es un método de lista que se usa para ordenar los elementos de una lista de forma ascendente. El método reverse() es un método de lista que se usa para invertir el orden de los elementos de una lista.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---

```
let lista = [3, 5, 1, 4, 2];
lista.sort();
console.log(lista); // [1, 2, 3, 4, 5]

lista.reverse();
console.log(lista); // [5, 4, 3, 2, 1]
```

Reto:

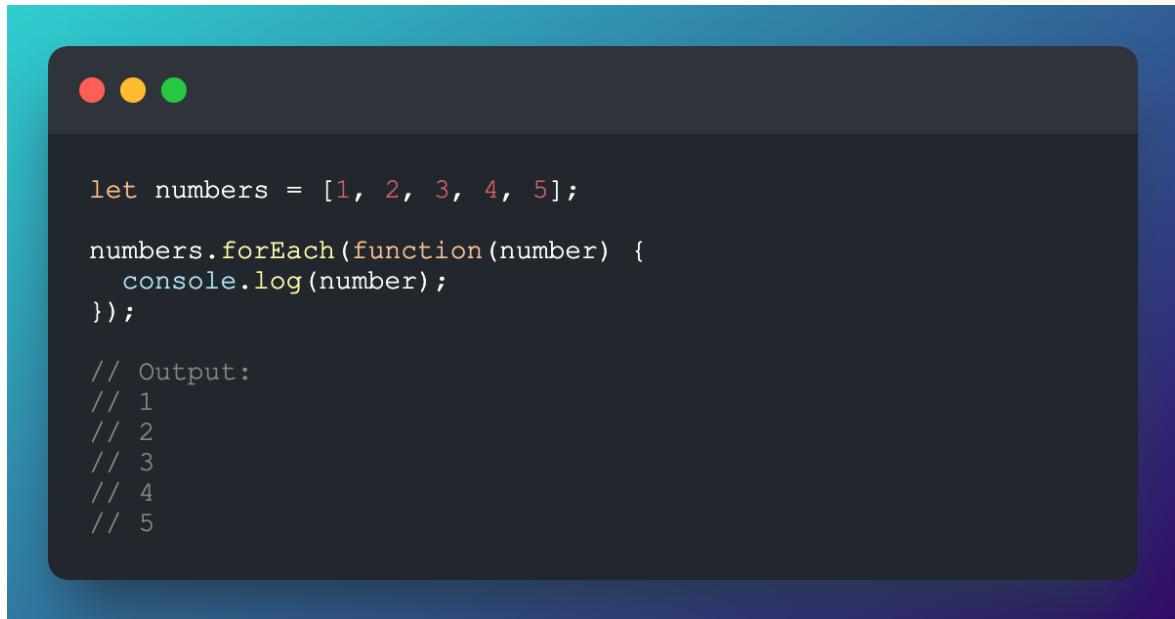
Crea una lista con números enteros y usa los métodos `sort()` y `reverse()` para ordenarla de forma ascendente y descendente.

**Método de lista `forEach()` y `map()`** El método `forEach()` se usa para iterar sobre una lista de elementos y ejecutar una función para cada elemento. Esta función se ejecuta una vez por cada elemento de la lista.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---



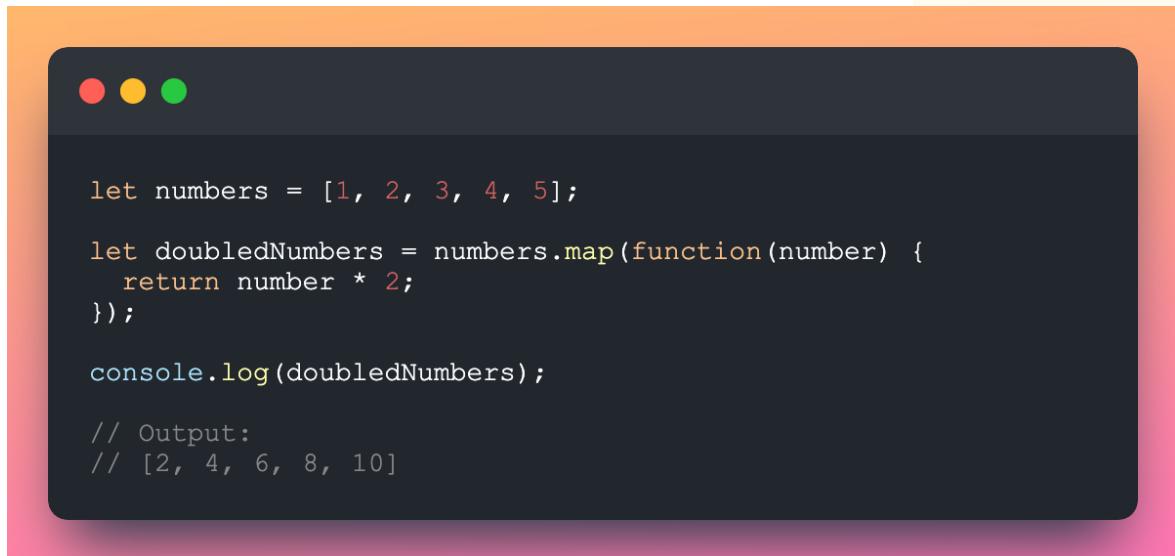
```
let numbers = [1, 2, 3, 4, 5];

numbers.forEach(function(number) {
  console.log(number);
});

// Output:
// 1
// 2
// 3
// 4
// 5
```

El método `map()` también se usa para iterar sobre una lista de elementos, pero en lugar de ejecutar una función para cada elemento, devuelve una nueva lista con los resultados de la función aplicada a cada elemento.

Ejemplo:



```
let numbers = [1, 2, 3, 4, 5];

let doubledNumbers = numbers.map(function(number) {
  return number * 2;
});

console.log(doubledNumbers);

// Output:
// [2, 4, 6, 8, 10]
```

## 8 USO AVANZADO DE DATOS

---

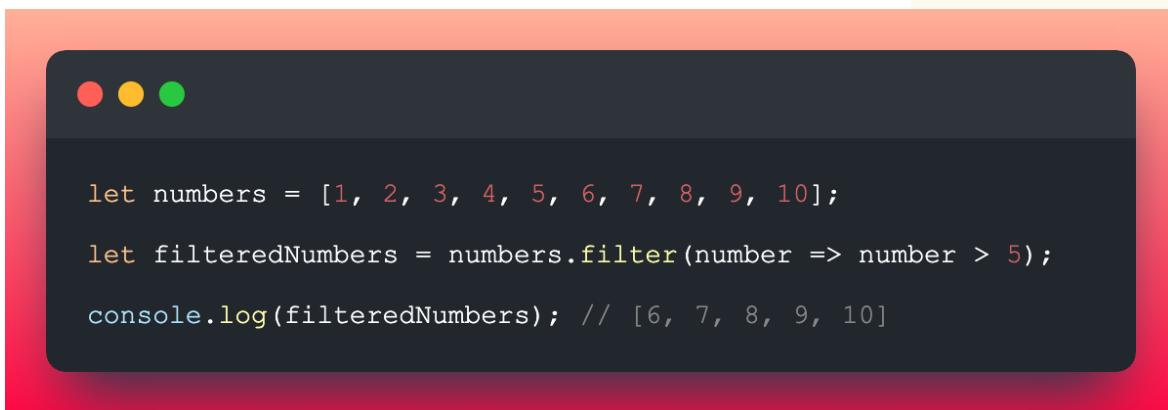
Reto:

Crea una lista de números del 1 al 10 y usa el método map() para devolver una nueva lista con los números elevados al cuadrado.

**Método de lista filter()** filter() es una función de JavaScript que se utiliza para filtrar elementos de una matriz. Esta función toma una función de devolución de llamada como argumento y devuelve una nueva matriz con todos los elementos que pasan la prueba implementada por la función de devolución de llamada.

Ejemplo:

Supongamos que tenemos una matriz de números enteros y queremos filtrar los números mayores que 5.



```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let filteredNumbers = numbers.filter(number => number > 5);
console.log(filteredNumbers); // [6, 7, 8, 9, 10]
```

Reto:

Utiliza la función filter() para filtrar los elementos de una matriz de cadenas que contengan la letra “a” .

**Método de lista reduce()** reduce() es una función de JavaScript que se utiliza para reducir un array a un solo valor. Esta función toma dos argumentos: una función de acumulación y un valor inicial (opcional). La función de acumulación toma dos argumentos: el valor acumulado y el elemento actual del array. Reduce() itera a

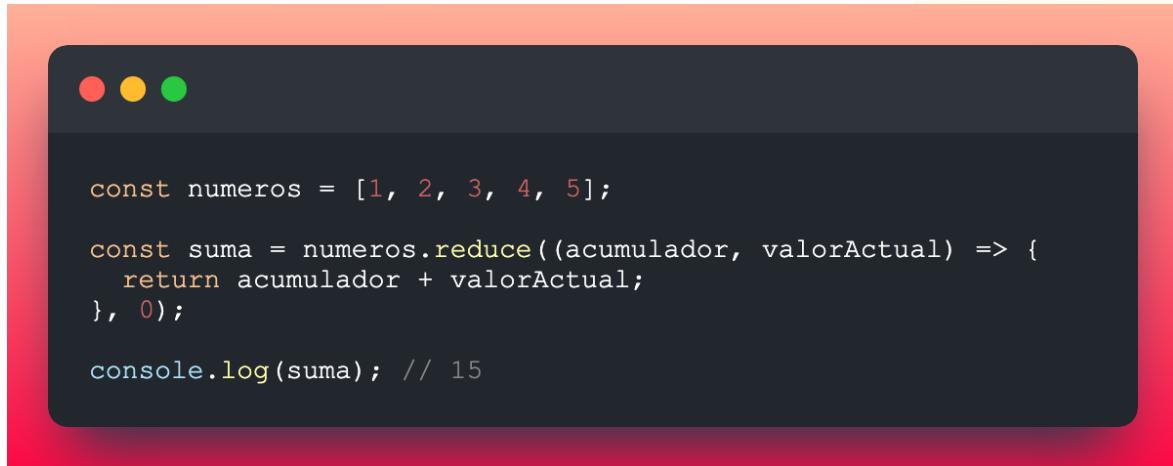
## 8 USO AVANZADO DE DATOS

---

través de los elementos del array, llamando a la función de acumulación para cada uno de ellos y devuelve un único valor.

Ejemplo:

Supongamos que tenemos un array de números y queremos calcular la suma de todos los elementos del array. Podemos usar reduce() para hacer esto:



A screenshot of a terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top. The terminal displays the following JavaScript code:

```
const numeros = [1, 2, 3, 4, 5];
const suma = numeros.reduce((acumulador, valorActual) => {
    return acumulador + valorActual;
}, 0);
console.log(suma); // 15
```

Reto:

Utiliza reduce() para calcular el promedio de los elementos de un array de números.

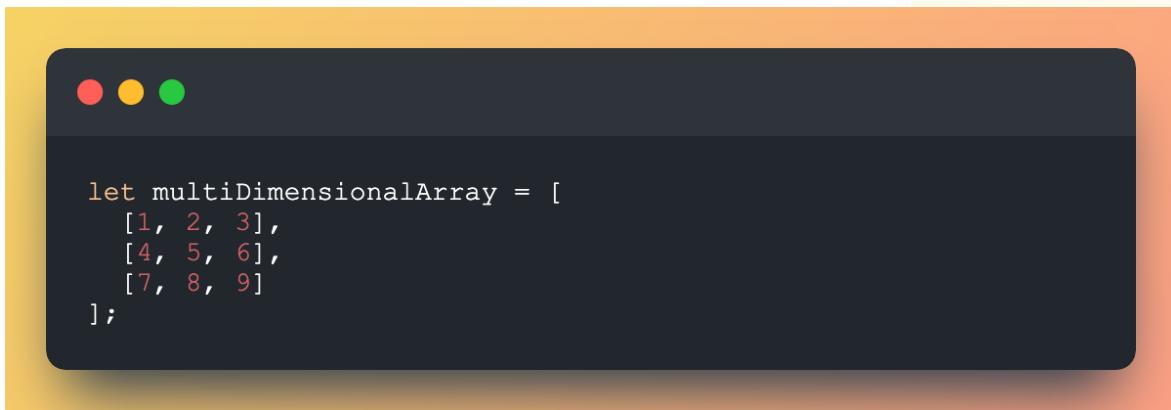
### 8.5.7 Arreglos multidimensionales

Los arreglos multidimensionales en JavaScript son arreglos que contienen otros arreglos. Esto significa que un arreglo multidimensional es un arreglo de arreglos. Esto nos permite almacenar datos en una estructura de datos jerárquica.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---



Reto:

Crea un arreglo multidimensional que contenga los nombres de tus amigos y sus edades.

### 8.5.8 Arreglos tipados (Int8Array, Uint8Array)

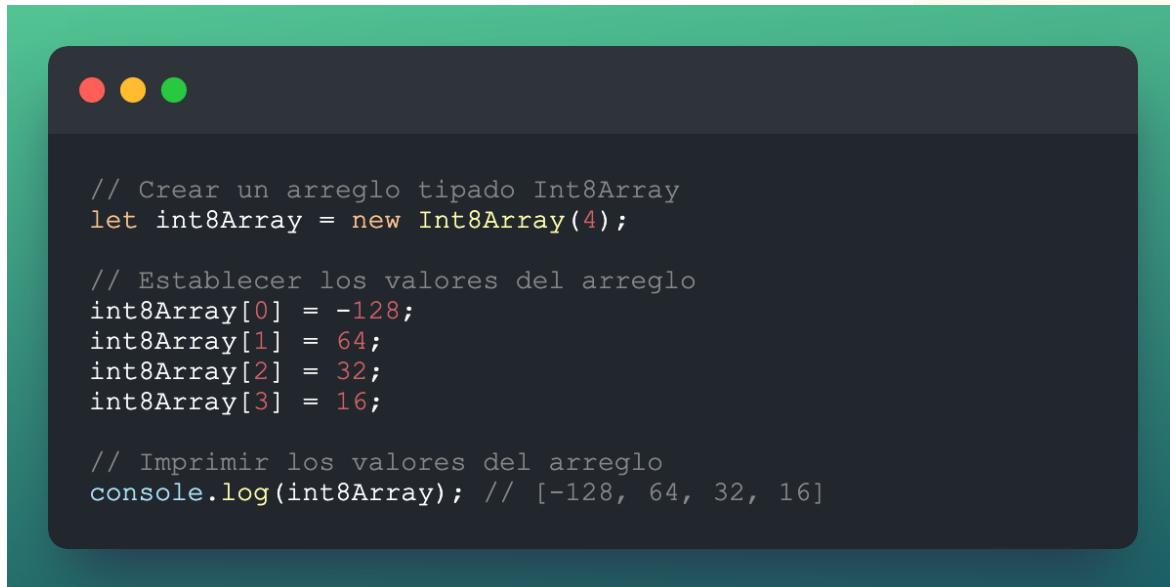
Los arreglos tipados son arreglos especiales en JavaScript que almacenan valores de un solo tipo de datos. Estos arreglos son más eficientes que los arreglos normales, ya que los valores se almacenan en un formato de datos específico. Esto significa que los arreglos tipados son más rápidos y usan menos memoria.

Existen varios tipos de arreglos tipados pero los más comunes son Int8Array y Uint8Array. Int8Array almacena valores enteros de 8 bits, mientras que Uint8Array almacena valores sin signo de 8 bits.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---



```
// Crear un arreglo tipado Int8Array
let int8Array = new Int8Array(4);

// Establecer los valores del arreglo
int8Array[0] = -128;
int8Array[1] = 64;
int8Array[2] = 32;
int8Array[3] = 16;

// Imprimir los valores del arreglo
console.log(int8Array); // [-128, 64, 32, 16]
```

Reto:

Crea un arreglo tipado Uint8Array con los valores [255, 128, 64, 32]. Luego, imprime los valores del arreglo en la consola.

### 8.5.9 Array buffer

ArrayBuffer es un objeto en JavaScript que permite almacenar datos binarios en una matriz de bytes. Esta matriz de bytes se puede usar para almacenar datos como cadenas de texto, imágenes, audio, video, etc.

Ejemplo:

A continuación se muestra un ejemplo de cómo crear un ArrayBuffer y almacenar una cadena de texto en él:

## 8 USO AVANZADO DE DATOS

---

```
// Crear un ArrayBuffer de 8 bytes
let buffer = new ArrayBuffer(8);

// Crear una vista de datos sobre el buffer
let view = new DataView(buffer);

// Escribir una cadena de texto en el buffer
view.setInt8(0, 'H'.charCodeAt(0));
view.setInt8(1, 'o'.charCodeAt(0));
view.setInt8(2, 'l'.charCodeAt(0));
view.setInt8(3, 'a'.charCodeAt(0));

// Leer la cadena de texto del buffer
let str = String.fromCharCode(view.getInt8(0)) +
          String.fromCharCode(view.getInt8(1)) +
          String.fromCharCode(view.getInt8(2)) +
          String.fromCharCode(view.getInt8(3));

console.log(str); // 'Hola'
```

Reto:

Crea una función que tome un ArrayBuffer y una cadena de texto como argumentos, y escriba la cadena de texto en el ArrayBuffer. Luego, crea otra función que tome un ArrayBuffer como argumento y devuelva la cadena de texto almacenada en el ArrayBuffer.

### 8.6 Uso de objetos

#### 8.6.1 Insertar elementos en un objeto

Insertar elementos en un objeto en JavaScript es una tarea sencilla. Para hacerlo, primero debemos crear un objeto. Esto se puede hacer usando la sintaxis de objeto literal:

```
var miObjeto = {};
```

Una vez que tenemos el objeto creado, podemos agregar elementos a él usando la sintaxis de punto:

```
miObjeto.nombre = "Juan";
miObjeto.edad = 25;
miObjeto.ciudad = "Madrid";
```

También podemos agregar elementos usando la sintaxis de corchetes:

## 8 USO AVANZADO DE DATOS

---

```
miObjeto["nombre"] = "Juan";
miObjeto["edad"] = 25;
miObjeto["ciudad"] = "Madrid";
```

Reto:

Crea un objeto con tres elementos y luego imprimelo en la consola.

### 8.6.2 Desestructuración

La desestructuración es una característica de JavaScript que nos permite extraer datos de arreglos u objetos y asignarlos a variables. Esto nos permite simplificar la sintaxis y hacer nuestro código más legible.

Ejemplo:

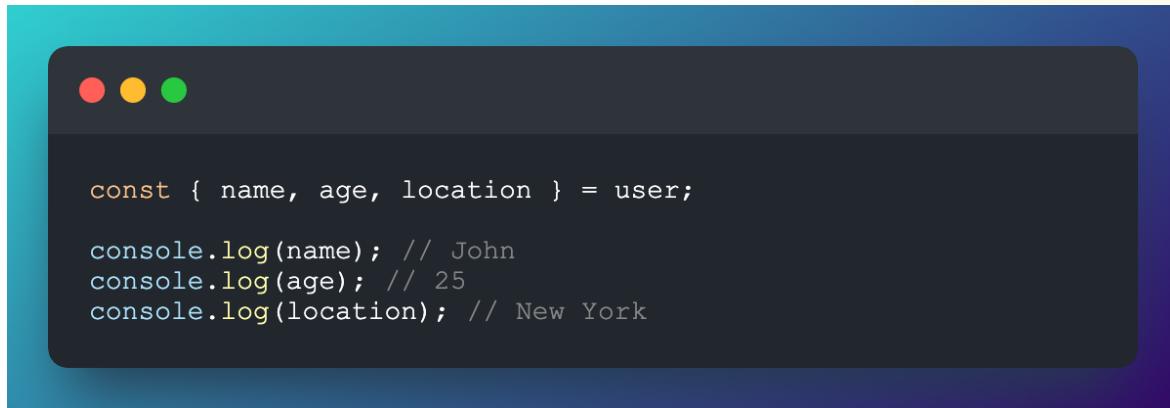
Supongamos que tenemos un objeto con información de un usuario:

```
const user = {
  name: 'John',
  age: 25,
  location: 'New York'
};
```

Podemos usar la desestructuración para extraer los datos del objeto y asignarlos a variables:

## 8 USO AVANZADO DE DATOS

---



A screenshot of a terminal window with a dark theme. At the top, there are three colored window control buttons: red, yellow, and green. The main area contains the following code:

```
const { name, age, location } = user;
console.log(name); // John
console.log(age); // 25
console.log(location); // New York
```

Reto:

Crea un objeto con información de una película y usa la desestructuración para extraer los datos y asignarlos a variables. Luego, imprime los datos en la consola.

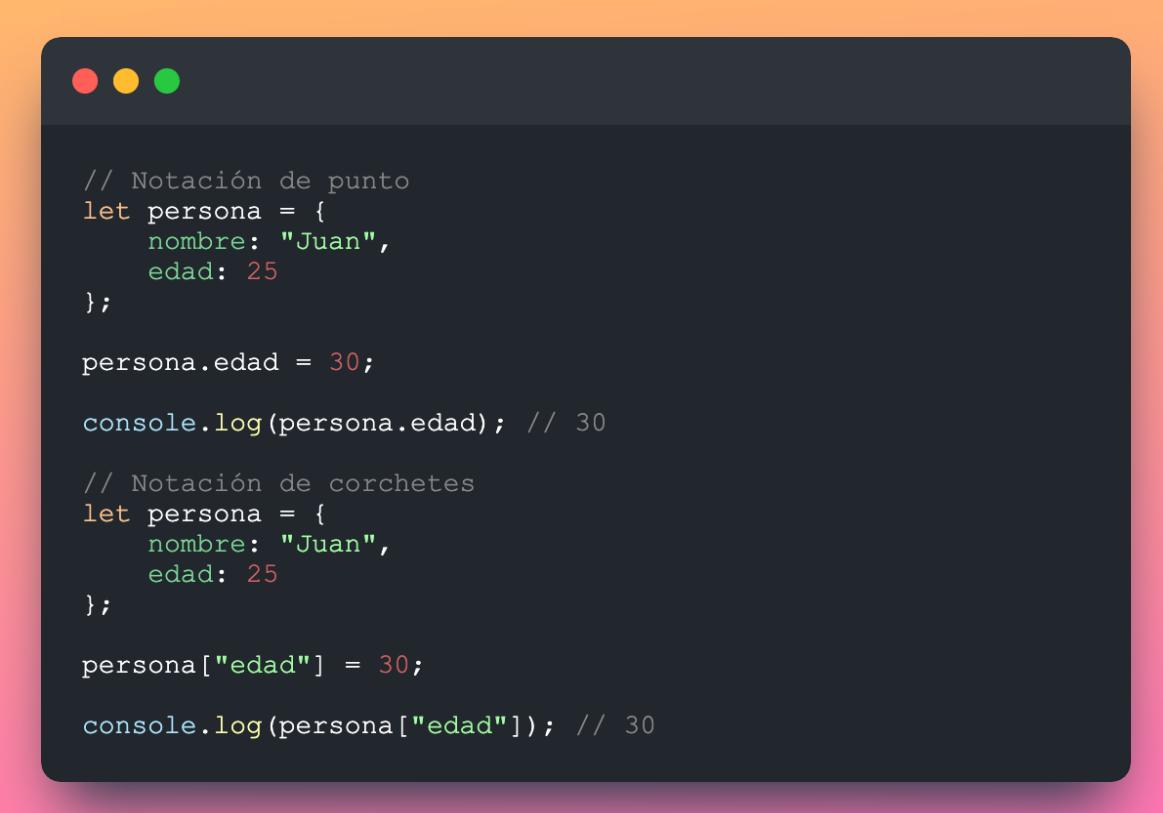
### 8.6.3 Modificar propiedades de objetos

Modificar propiedades de objetos en JavaScript es una tarea sencilla. Esto se puede hacer usando la notación de punto o la notación de corchetes.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---



The screenshot shows a code editor window with a dark theme. At the top, there are three colored circular icons: red, yellow, and green. The code itself is written in JavaScript and illustrates two ways to access and modify properties of an object:

```
// Notación de punto
let persona = {
    nombre: "Juan",
    edad: 25
};

persona.edad = 30;

console.log(persona.edad); // 30

// Notación de corchetes
let persona = {
    nombre: "Juan",
    edad: 25
};

persona["edad"] = 30;

console.log(persona["edad"]); // 30
```

Reto:

Crea un objeto con tres propiedades y modifica una de ellas usando la notación de punto y la notación de corchetes.

### 8.6.4 Borrar propiedades de objetos

En JavaScript, podemos borrar propiedades de objetos usando la palabra clave `delete`. Esta palabra clave elimina la propiedad especificada del objeto.

Ejemplo:

## 8 USO AVANZADO DE DATOS

---



```
let persona = {  
    nombre: 'Juan',  
    edad: 25  
};  
  
delete persona.edad;  
  
console.log(persona); // { nombre: 'Juan' }
```

Reto:

Crea un objeto con tres propiedades y elimina una de ellas usando la palabra clave delete.

### 8.6.5 Crear métodos

JavaScript es un lenguaje de programación orientado a objetos, lo que significa que podemos crear objetos y asignarles métodos. Un método es una función asociada a un objeto que se puede usar para realizar una tarea específica.

Ejemplo:

Por ejemplo, podemos crear un objeto llamado Persona con un método llamado saludar():

## 8 USO AVANZADO DE DATOS

---



```
let Persona = {
    nombre: 'Juan',
    edad: 25,
    saludar: function() {
        console.log(`Hola, soy ${this.nombre} y tengo
${this.edad} años.`);
    }
};

Persona.saludar(); // Hola, soy Juan y tengo 25 años.
```

La palabra `this` permite acceder a otras propiedades y métodos del objeto.

Reto:

Crea un objeto llamado `Cuenta` con dos propiedades: `saldo` y `interes`. Agrega un método llamado `calcularInteres()` que calcule el interés anual de la cuenta y lo agregue al saldo.

### 8.6.6 Propiedades y métodos del objeto Object

`Object` es uno de los objetos básicos de JavaScript. Está disponible en todos los navegadores y se usa para crear objetos y trabajar con ellos.

Las propiedades de `Object` son:

- `constructor`: Esta propiedad contiene una referencia al constructor de un objeto.
- `prototype`: Esta propiedad contiene una referencia al prototipo de un objeto.
- `__proto__`: Esta propiedad contiene una referencia al prototipo de un objeto.

## 8 USO AVANZADO DE DATOS

---

Estás propiedades tendrás más sentido en la sección de Programación Orientada a los Objetos.

Los métodos de Object son:

- `hasOwnProperty()`: Esta función devuelve un valor booleano que indica si el objeto contiene la propiedad especificada.
- `isPrototypeOf()`: Esta función devuelve un valor booleano que indica si el objeto especificado es un prototipo del objeto actual.
- `propertyIsEnumerable()`: Esta función devuelve un valor booleano que indica si la propiedad especificada es enumerable.

Ejemplo:

Supongamos que queremos comprobar si un objeto tiene una propiedad específica. Podemos usar el método `hasOwnProperty()` para hacer esto:



```
let obj = {
  name: 'John',
  age: 30
};

console.log(obj.hasOwnProperty('name')); // true
console.log(obj.hasOwnProperty('gender')) // false
```

Reto:

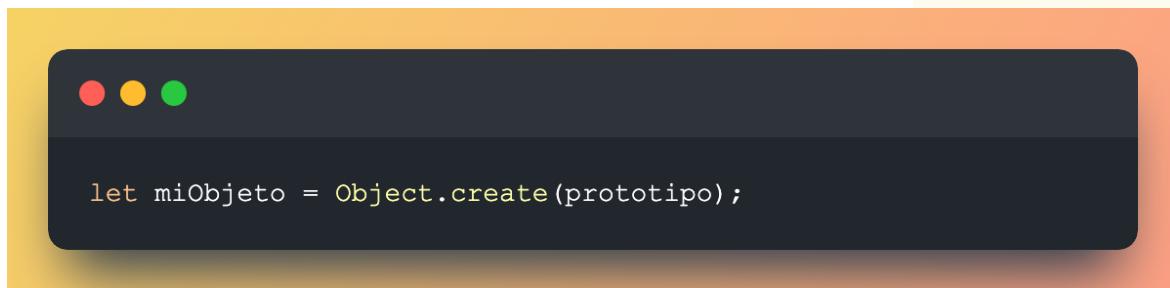
Crea un objeto con algunas propiedades y usa el método `propertyIsEnumerable()` para comprobar si una propiedad es enumerable.

### 8.6.7 Métodos comunes de objetos

**create()** `create()` es un método de JavaScript que se utiliza para crear un nuevo objeto a partir de un prototipo. Esto significa que puede crear un objeto con las mismas propiedades y métodos que el prototipo. Esto es útil para crear objetos similares sin tener que escribir todo el código desde cero.

Ejemplo:

Supongamos que queremos crear un objeto llamado `miObjeto` que tenga las mismas propiedades y métodos que el objeto prototipo. Podemos hacer esto usando el método `create()` de la siguiente manera:



Reto:

Crea un objeto llamado `miObjeto` que tenga las mismas propiedades y métodos que el objeto prototipo. Usa el método `create()` para crear el objeto.

## 9 Programación orientada a objetos (POO)

### 9.1 Paradigma

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en la creación de objetos que contienen datos y funcionalidades. Estos

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

objetos se relacionan entre sí para formar una estructura de datos compleja. JavaScript es un lenguaje de programación orientado a objetos, lo que significa que los programadores pueden crear objetos y usarlos para construir aplicaciones.

Ejemplo:

A continuación se muestra un ejemplo de una clase de objeto en JavaScript. No tienes que entenderla ahora ya que en las siguientes lecciones veremos todos los detalles.



```
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }
  saludar() {
    console.log(`Hola, soy ${this.nombre} y tengo
${this.edad} años.`);
  }
}

const personal = new Persona('Juan', 25);
personal.saludar(); // Imprime: Hola, soy Juan y tengo 25
años.
```

Reto:

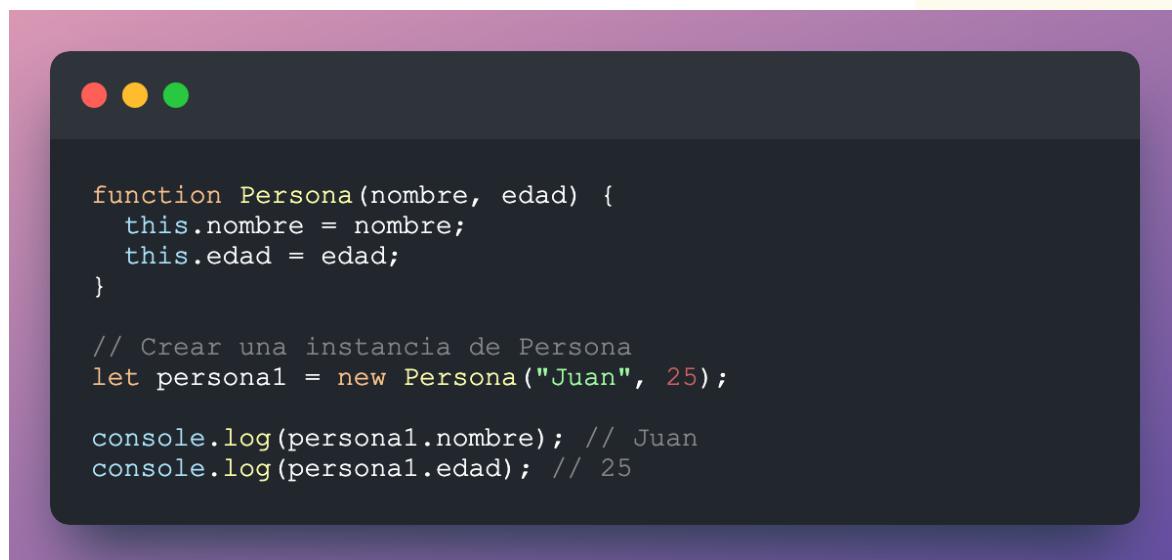
Crea un de objeto llamado Auto que tenga los atributos marca, modelo y año. Agrega un método llamado mostrarInfo que imprima en la consola la información del auto. Crea 2 objetos más para otros autos para que veas que requiere mucho código repetido que podremos simplificarlo con los conocimientos de este capítulo.

### 9.2 Función constructora

Una función constructora es una función especial en JavaScript que se usa para crear objetos. Esta función se define con la palabra clave new y se usa para crear una instancia de un objeto. Esta función se usa para crear objetos con propiedades y métodos específicos.

Ejemplo:

A continuación se muestra un ejemplo de una función constructora en JavaScript para crear un objeto Persona con propiedades nombre y edad:



```
function Persona(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
}

// Crear una instancia de Persona
let personal = new Persona("Juan", 25);

console.log(personal.nombre); // Juan
console.log(personal.edad); // 25
```

Reto:

Crea una función constructora en JavaScript para crear un objeto Auto con propiedades marca, modelo y año. Luego, crea una instancia de Auto y muestra los valores de sus propiedades en la consola.

### 9.3 Contexto

#### 9.3.1 Contexto usando this

En JavaScript, el contexto se refiere a la forma en que una función se ejecuta. El contexto de una función se determina por el valor de la palabra clave `this` dentro de la función.

Ejemplo:

Supongamos que tenemos un objeto llamado `persona` con dos propiedades: `nombre` y `edad`.



A screenshot of a terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top. The terminal shows the following code and its execution:

```
const persona = {
  nombre: 'Juan',
  edad: 25,
  saludar: function() {
    console.log(`Hola, soy ${this.nombre}!`);
  }
};

persona.saludar(); // Hola, soy Juan!
```

Ahora, supongamos que queremos crear una función que imprima el nombre y la edad de la persona. Para hacer esto, podemos usar la palabra clave `this` para acceder a las propiedades del objeto `persona` dentro de la función.

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

```
function imprimirPersona() {  
    console.log(`${this.nombre} tiene ${this.edad} años.`);  
}
```

Ahora, podemos llamar a la función imprimirPersona y pasar el objeto persona como el contexto de la función. Esto significa que el valor de this dentro de la función será el objeto persona.

```
imprimirPersona.call(persona);  
// Imprime: Juan tiene 25 años.
```

Reto:

Crea un objeto llamado libro con dos propiedades: título y autor. Luego, crea una función llamada imprimirLibro que imprima el título y el autor del libro usando la palabra clave this. Finalmente, llama a la función imprimirLibro y pasa el objeto libro como el contexto de la función.

### 9.3.2 Vinculación (call(), apply(), bind())

La vinculación es una característica de JavaScript que permite a los desarrolladores llamar a una función con un contexto de ejecución específico. Esto significa que puedes cambiar el valor de this dentro de una función. Esto se logra mediante los

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

métodos call(), apply() y bind().

**call()** El método call() permite llamar a una función con un contexto de ejecución específico. El primer argumento de call() es el contexto de ejecución, seguido de los argumentos de la función.

Ejemplo:



```
function saludar(saludo, puntuacion) {
  console.log(` ${saludo} ${this.nombre}${puntuacion}`);
}

const persona = {
  nombre: 'Xavier',
};

saludar.call(persona, 'Hola', '!');

// Salida: Hola Xavier!
```

**apply()** El método apply() funciona de manera similar al método call(), con la diferencia de que los argumentos de la función se pasan como una lista.

Ejemplo:

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

```
function saludar(saludo, puntuacion) {
    console.log(` ${saludo} ${this.nombre}${puntuacion}`);
}

const persona = {
    nombre: 'Xavier',
};

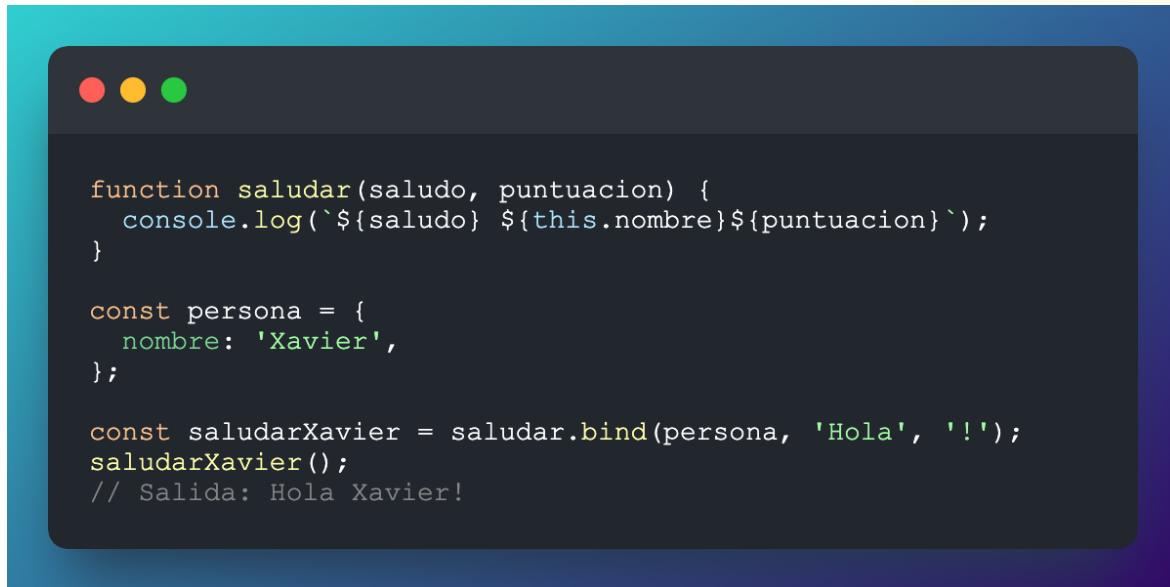
saludar.apply(persona, ['Hola', '!']);
// Salida: Hola Xavier!
```

**bind()** El método bind() crea una nueva función con el contexto de ejecución especificado. Esta nueva función puede ser llamada posteriormente con los argumentos especificados.

Ejemplo:

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---



A screenshot of a browser's developer tools console. At the top, there are three colored circular icons: red, yellow, and green. Below them, the console displays the following JavaScript code:

```
function saludar(saludo, puntuacion) {
    console.log(` ${saludo} ${this.nombre}${puntuacion}`);
}

const persona = {
    nombre: 'Xavier',
};

const saludarXavier = saludar.bind(persona, 'Hola', '!');
saludarXavier();
// Salida: Hola Xavier!
```

Reto:

Intenta crear una función que use call(), apply() y bind() para cambiar el contexto de ejecución de una función.

### 9.3.3 this en funciones flecha

This en funciones flecha en JavaScript se refiere a la referencia al contexto de la función en la que se encuentra. Esto significa que el valor de this dentro de una función flecha se determina por el contexto en el que se encuentra.

Ejemplo:

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

```
const persona = {
    nombre: 'Hilde',
    saludar: () => {
        console.log(`Hola, soy ${this.nombre}!`);
    }
};

persona.saludar(); // Hola, soy undefined!
```

En este ejemplo, el valor de this dentro de la función flecha es undefined, ya que la función no está dentro del contexto del objeto.

Reto:

Crea una función flecha que imprima el valor de this dentro de un objeto.

### 9.4 Prototipo

Los prototipos en JavaScript son una característica de la programación orientada a objetos que permite a los objetos heredar propiedades y métodos de otros objetos. Esto significa que un objeto puede heredar propiedades y métodos de otro objeto sin tener que definirlos explícitamente.

Ejemplo:

Supongamos que tenemos un objeto llamado “Persona” con las propiedades “nombre” y “edad” .

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

```
var Persona = {  
    nombre: "John",  
    edad: 25  
};
```

Ahora, si queremos crear un objeto “Estudiante” que herede las propiedades y métodos de “Persona”, podemos hacerlo usando prototype:

```
function Estudiante() {  
}  
  
Estudiante.prototype = Persona;  
  
var estudiante1 = new Estudiante();  
  
console.log(estudiante1.nombre); // John  
console.log(estudiante1.edad); // 25
```

Reto:

Crea un objeto “Profesor” que herede las propiedades y métodos de “Persona”. Agrega una propiedad “materia” al objeto “Profesor” y asignale un valor. Luego, imprime el valor de la propiedad “materia” en la consola.

### 9.5 Herencia prototípica

La Herencia Prototípica en JavaScript es una forma de herencia basada en objetos, en la que un objeto hereda las propiedades y métodos de otro objeto. Esto significa que un objeto puede heredar de otro objeto sin necesidad de definir una clase.

Ejemplo:



```
// Creamos un objeto prototipo
let animal = {
  nombre: '',
  edad: 0,
  saludar: function() {
    console.log(`Hola, soy ${this.nombre}`);
  }
};

// Creamos un objeto que hereda del prototipo
let perro = Object.create(animal);
perro.nombre = 'Fido';
perro.edad = 3;

// Llamamos al método saludar
perro.saludar(); // Hola, soy Fido
```

Reto:

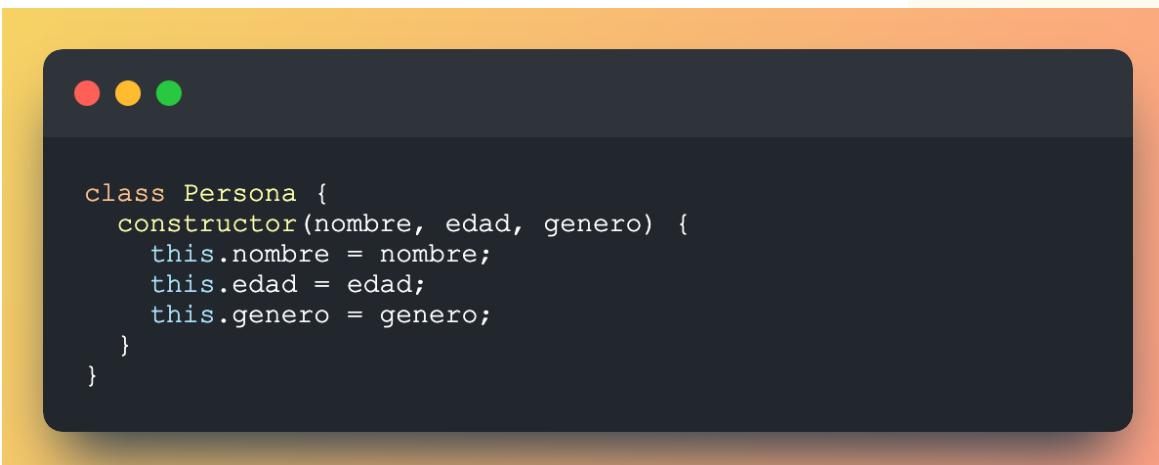
Crea un objeto prototipo llamado vehículo con las propiedades marca y modelo, y un método encender que imprima en la consola "El vehículo está encendido". Luego, crea un objeto que herede de vehículo y asigne los valores apropiados a las propiedades. Finalmente, llama al método encender para verificar que funciona.

### 9.6 Clases

Las clases en JavaScript son una forma de definir objetos con una sintaxis más clara y concisa (en lugar de las lecciones revisadas en este capítulo usando funciones). Estas clases se definen usando la palabra clave `class` y se pueden usar para crear objetos con propiedades y métodos similares.

Ejemplo:

Por ejemplo, podemos crear una clase llamada `Persona` que tenga propiedades como nombre, edad y género:



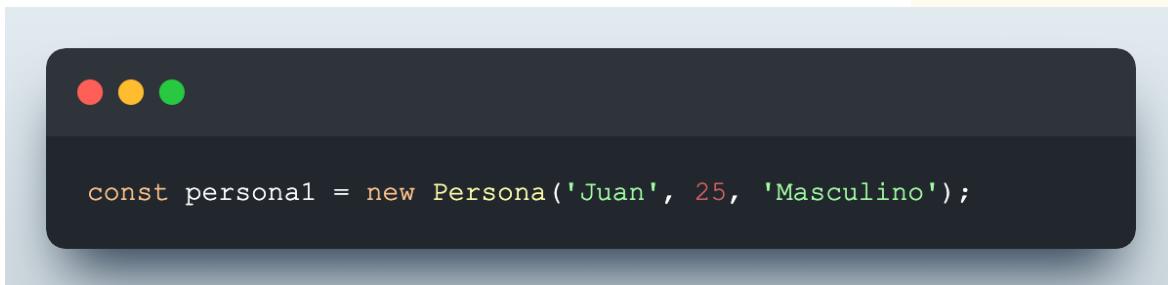
```
class Persona {
  constructor(nombre, edad, genero) {
    this.nombre = nombre;
    this.edad = edad;
    this.genero = genero;
  }
}
```

`constructor()` es una función especial en JavaScript que se usa para crear e inicializar objetos. Esta función se usa para crear una nueva instancia de un objeto, asignar valores a sus propiedades y luego devolver el objeto.

Ahora podemos crear una instancia de esta clase usando el constructor:

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---



new en JavaScript es una palabra clave que se usa para crear una nueva instancia de un objeto. Esto significa que se usa para crear un objeto a partir de una clase o constructor.

Reto:

Crea una clase llamada Auto que tenga propiedades como marca, modelo y año. Luego, crea una instancia de esta clase usando el constructor.

### 9.7 Declaración y expresión de clases

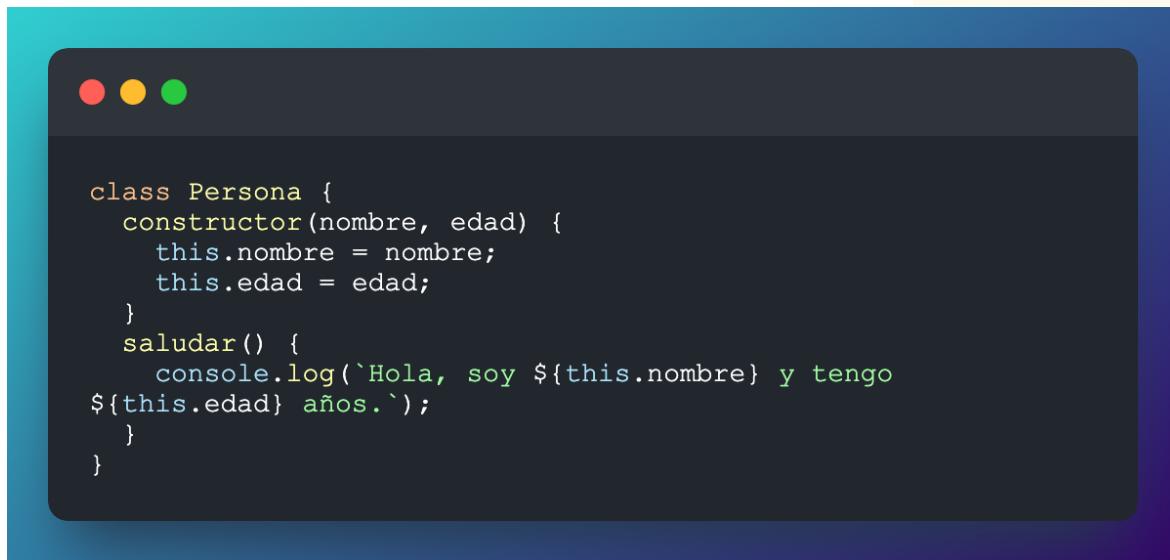
Las declaraciones de clases en JavaScript son una forma de definir una clase y sus miembros. Estas declaraciones se usan para crear una nueva clase con propiedades y métodos. Por otro lado, las expresiones de clases son una forma de definir una clase y sus miembros en una sola línea. Estas expresiones se usan para crear una nueva clase con propiedades y métodos en una sola línea.

Ejemplo:

Declaración de Clase:

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---



```
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }
  saludar() {
    console.log(`Hola, soy ${this.nombre} y tengo
${this.edad} años.`);
  }
}
```

Expresión de Clase:



```
const Persona = class {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }
  saludar() {
    console.log(`Hola, soy ${this.nombre} y tengo
${this.edad} años.`);
  }
}
```

Para estos ejemplos, la elevación o hoisting de clases es similar a la elevación de funciones y variables respectivamente.

Reto:

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

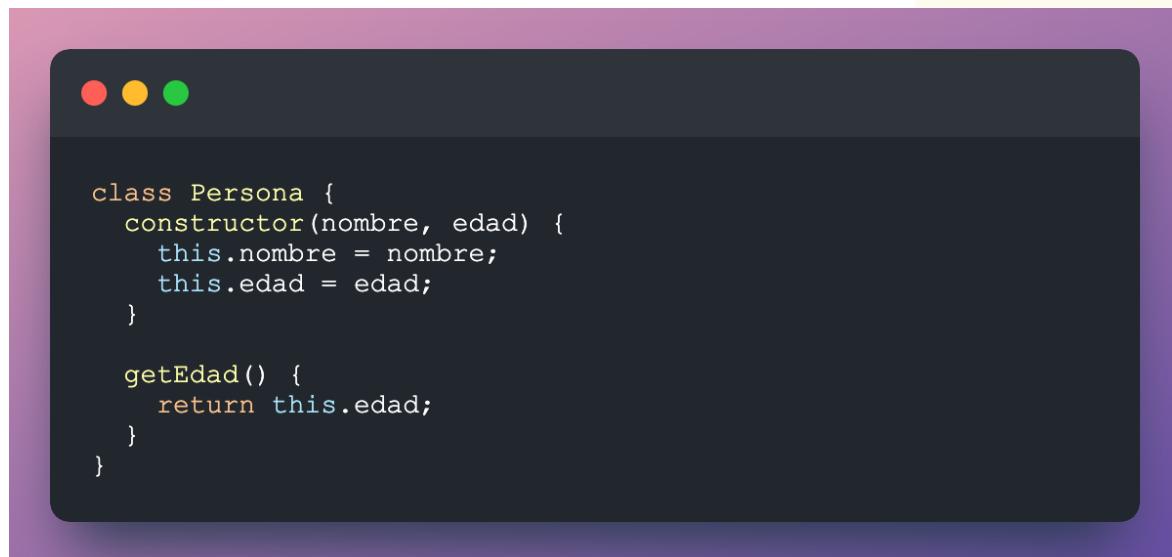
Crea una clase llamada Vehiculo con dos propiedades: marca y modelo. Agrega un método llamado mostrarInfo que imprima en consola la marca y el modelo del vehículo.

### 9.8 Métodos de una instancia

Los métodos de una instancia en JavaScript son funciones que se asocian a un objeto y que se pueden invocar para realizar una acción específica. Estos métodos se definen dentro de la clase a la que pertenece el objeto.

Ejemplo:

Supongamos que tenemos una clase llamada Persona con los atributos nombre y edad. Si queremos agregar un método para obtener la edad de una persona, podemos definir el siguiente método dentro de la clase Persona:



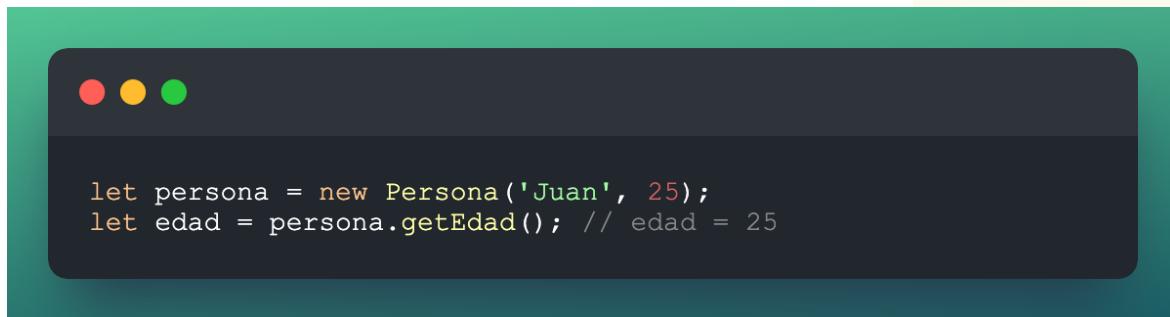
```
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }

  getEdad() {
    return this.edad;
  }
}
```

Ahora, podemos crear una instancia de la clase Persona y usar el método getEdad() para obtener la edad de la persona:

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---



```
let persona = new Persona('Juan', 25);
let edad = persona.getEdad(); // edad = 25
```

Reto:

Crea una clase llamada CuentaBancaria con los atributos nombre y saldo. Agrega un método llamado depositar() que reciba una cantidad como parámetro y aumente el saldo de la cuenta en esa cantidad.

### 9.9 Campos públicos

Los campos públicos en JavaScript son variables o métodos que se pueden acceder desde cualquier parte del código. Esto significa que no hay necesidad de crear una instancia de una clase para acceder a estos campos.

Ejemplo:

```
// Definimos una clase con un campo público
class Persona {
  constructor(nombre) {
    this.nombre = nombre;
  }
}

// Creamos una instancia de la clase
const personal1 = new Persona('Juan');

// Accedemos al campo público desde fuera de la clase
console.log(personal1.nombre); // 'Juan'
```

Reto:

Crea una clase llamada Vehículo con un campo público llamado ruedas y asígnale el valor 4. Luego, crea una instancia de la clase y accede al campo público para imprimir su valor en la consola.

### 9.10 Campos privados

Los campos privados en JavaScript son aquellos que no se pueden acceder desde fuera de la clase o objeto. Esto significa que los campos privados no se pueden modificar desde fuera de la clase o objeto. Se aplican a propiedades y métodos nombrándolos con el símbolo #.

Ejemplo:

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

```
class Persona {  
    // Campo privado  
    #salario;  
  
    constructor(nombre, edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
        this.#salario = 0;  
    }  
    // Método para obtener el salario  
    obtenerSalario() {  
        return this.#salario;  
    }  
    // Método para aumentar el salario  
    aumentarSalario(monto) {  
        this.#salario += monto;  
    }  
  
    // Método privado  
    #metodoPrivado() {  
        console.log("Privado")  
    }  
}  
  
let persona = new Persona("Juan", 25);  
  
// Intentar acceder al campo privado  
console.log(persona.#salario); // SyntaxError  
  
// Intentar modificar el campo privado  
persona.#salario = 1000;  
console.log(persona.#salario); // SyntaxError  
  
// Usar el método para obtener el salario  
console.log(persona.obtenerSalario()); // 0  
  
// Usar el método para aumentar el salario  
persona.aumentarSalario(1000);  
console.log(persona.obtenerSalario()); // 1000  
console.log(persona.#metodoPrivado()); // SyntaxError
```

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

Reto:

Crea una clase llamada CuentaBancaria que tenga dos campos privados: saldo y interes. Agrega un método llamado calcularInteres que calcule el interes y lo agregue al saldo.

### 9.11 Campos estáticos

Las propiedades estáticas en JavaScript son aquellas que se definen directamente en la clase y no en una instancia de la misma. Estas propiedades son compartidas por todas las instancias de la clase, lo que significa que cualquier cambio realizado en una instancia se verá reflejado en todas las demás.

Los métodos estáticos suelen ser funciones de utilidad, como funciones para crear o clonar objetos, mientras que las propiedades estáticas son útiles para cachés, configuración fija o cualquier otro dato que no necesite replicar entre instancias.

Ejemplo:



A screenshot of a code editor window titled "index.js". The code defines a class "Persona" with a static variable "numeroDePersonas" initialized to 0. The constructor takes a name as a parameter and sets it to "this.nombre". It also increments the static variable "numeroDePersonas". Two instances of "Persona" are created: "persona1" with name "Juan" and "persona2" with name "Pedro". Finally, "console.log(Persona.numeroDePersonas)" is called, which outputs 2. The code editor has a dark theme with syntax highlighting for keywords and comments.

```
class Persona {
  static numeroDePersonas = 0;
  constructor(nombre) {
    this.nombre = nombre;
    Persona.numeroDePersonas++;
  }
}

let persona1 = new Persona("Juan");
let persona2 = new Persona("Pedro");

console.log(Persona.numeroDePersonas); // 2
```

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

Reto:

Crea una clase llamada Vehiculo que tenga una propiedad estática llamada numeroDeVehiculos que se incremente cada vez que se crea una nueva instancia de la clase.

### 9.12 Captadores (getters) y establecedores (setters)

Los getters y setters son funciones especiales en JavaScript que nos permiten obtener y establecer valores de un objeto. Estas funciones nos permiten controlar el acceso a los datos de un objeto, permitiendo que los datos sean leídos y escritos de forma segura.

Ejemplo:

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

```
class Persona {  
    #nombre;  
    #edad;  
  
    constructor(nombre, edad) {  
        this.#nombre = nombre;  
        this.#edad = edad;  
    }  
    // Getter  
    get nombre() {  
        return this.#nombre;  
    }  
    // Setter  
    set nombre(nombre) {  
        console.log("Puedo validar nombre");  
        this.#nombre = nombre;  
    }  
}  
  
const persona = new Persona('Juan', 25);  
console.log(persona.nombre); // Juan  
persona.nombre = 'Pedro'; // Imprime: Puedo validar nombre  
console.log(persona.nombre); // Pedro
```

En este caso accedemos a las propiedades de forma indirecta en lugar de acceder directamente a `#nombre` y `#edad` y también se puede incluir lógica adicional al acceder y cambiar propiedades como validaciones.

Reto:

Crea una clase llamada Libro con los atributos título, autor y editorial. Agrega los getters y setters necesarios para leer y escribir los valores de los atributos.

### 9.13 Herencia

La herencia en JavaScript es un mecanismo que permite a los objetos heredar propiedades y métodos de otros objetos. Esto significa que un objeto puede heredar las propiedades y métodos de otro objeto, lo que le permite compartir ciertas características con otros objetos.

Ejemplo:

Por ejemplo, podemos crear una clase Vehículo que contenga las propiedades y métodos comunes a todos los vehículos, como la velocidad, el color, etc. Luego, podemos crear clases específicas para cada tipo de vehículo, como Coche, Moto y Bicicleta, que hereden las propiedades y métodos de la clase Vehículo.

## 9 PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

---

```
class Vehiculo {  
    constructor(velocidad, color) {  
        this.velocidad = velocidad;  
        this.color = color;  
    }  
  
    acelerar() {  
        this.velocidad += 10;  
    }  
  
    frenar() {  
        this.velocidad -= 10;  
    }  
}  
  
class Coche extends Vehiculo {  
    constructor(velocidad, color, marca) {  
        super(velocidad, color);  
        this.marca = marca;  
    }  
  
    acelerar() {  
        this.velocidad += 20;  
    }  
}  
  
const miCoche = new Coche(0, 'rojo', 'Fiat');  
miCoche.acelerar();  
console.log(miCoche.velocidad); // 20
```

extends es una palabra clave para heredar propiedades y métodos de una clase a otra.

super() es una palabra clave que se usa para llamar al constructor de la clase padre y pasar argumentos. Esto significa que se puede acceder a los métodos y propiedades de la clase padre desde la clase hija.

Reto:

Crea una clase Persona que tenga las propiedades nombre, edad y genero, y los métodos saludar() y cumpleaños(). Luego, crea una clase Estudiante que herede de Persona y tenga las propiedades curso y promedio, y el método estudiar().

# 10 Asincronía

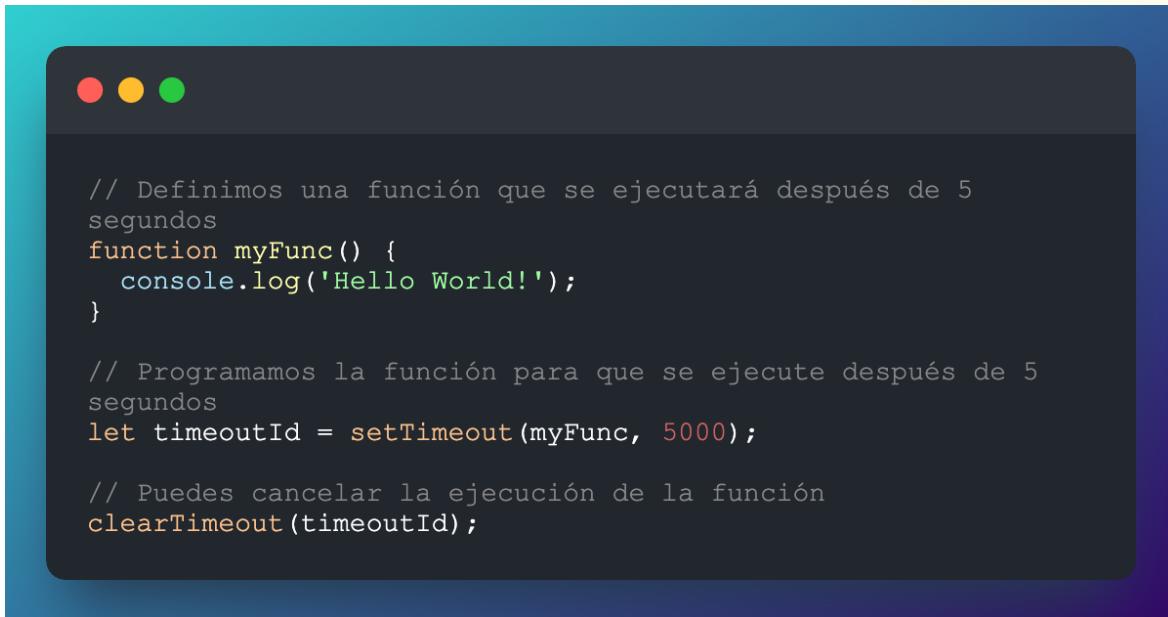
## 10.1 Temporizadores

### 10.1.1 setTimeout() y clearTimeout()

**setTimeout()** es una función disponible en ambientes de JavaScript que se utiliza para programar una función para que se ejecute después de un tiempo específico. Esta función toma dos parámetros: una función a ejecutar y un tiempo en milisegundos.

**clearTimeout()** es una función disponible en ambientes de JavaScript que se utiliza para cancelar una función programada con setTimeout(). Esta función toma un parámetro: el identificador devuelto por setTimeout().

Ejemplo:



```
// Definimos una función que se ejecutará después de 5 segundos
function myFunc() {
    console.log('Hello World!');
}

// Programamos la función para que se ejecute después de 5 segundos
let timeoutId = setTimeout(myFunc, 5000);

// Puedes cancelar la ejecución de la función
clearTimeout(timeoutId);
```

Reto:

Escribe un código que muestre un mensaje en la consola después de 10 segundos. Luego, cancela la ejecución del mensaje.

### 10.1.2 setInterval() y clearInterval()

**setInterval()** es una función disponible en ambientes de JavaScript que se utiliza para ejecutar una función o un bloque de código repetidamente, con un intervalo de tiempo específico. Esta función toma dos parámetros: una función o un bloque de código, y un intervalo de tiempo en milisegundos.

**clearInterval()** es una función disponible en ambientes de JavaScript que se utiliza para detener la ejecución de una función o un bloque de código que se está ejecutando repetidamente con setInterval(). Esta función toma un parámetro: el identificador devuelto por setInterval().

Ejemplo:

```
// Definimos una función que se ejecutará cada segundo
function saludar() {
    console.log("Hola!");
}

// Definimos una variable para almacenar el identificador
// devuelto por setInterval()
let intervalId = setInterval(saludar, 1000);

// Puedes cancelar la ejecución de la función
clearInterval(intervalId);
```

Reto:

Crea una función que imprima un número en la consola cada segundo, desde el número 1 hasta el número 10. Utiliza `setInterval()` y `clearInterval()` para lograrlo.

## 10.2 Promesas

### 10.2.1 Crear una promesa

Una promesa en JavaScript es un objeto que representa el resultado de una operación asíncrona. Esto significa que una promesa puede ser resuelta con un valor o rechazada con una excepción y puede demorar poco o largo tiempo de cumplirse.

Las garantías de promesas en JavaScript nos permiten asegurar que una promesa se cumplirá. Esto significa que si una promesa no se cumple, se ejecutará una función de fallo para manejar el error.

Ejemplo:

## 10 ASÍNCRONÍA

---

```
// la promesa toma un callback con los argumentos resolve y  
reject  
const miPromesa = new Promise((resolve, reject) => {  
    const numeroAleatorio = Math.random();  
    if (numeroAleatorio > 0.5) {  
        // el callback resolve() indica una resolución positiva  
        resolve(numeroAleatorio);  
    } else {  
        // el callback reject() indica un rechazo o error  
        reject(numeroAleatorio);  
    }  
});  
  
// llamamos la promesa  
miPromesa  
    .then(numeroAleatorio => {  
        console.log(`El número aleatorio es mayor a 0.5:  
${numeroAleatorio}`);  
    })  
    .catch(numeroAleatorio => {  
        console.log(`El número aleatorio es menor a 0.5:  
${numeroAleatorio}`);  
    });  
  
console.log("Nota que este console.log() imprime antes que la  
promesa se cumpla. Esto es asíncronía!");
```

`then()` es un callback que indica la acción a realizar si la resolución es positiva. Recibe datos de la promesa cumplida como parámetro.

`catch()` es un callback que atrapa el error en caso de un rechazo en la promesa. Recibe un error como parámetro.

Reto:

Crea una promesa que resuelva una operación asíncrona que devuelva un número

## 10 ASÍNCRONÍA

---

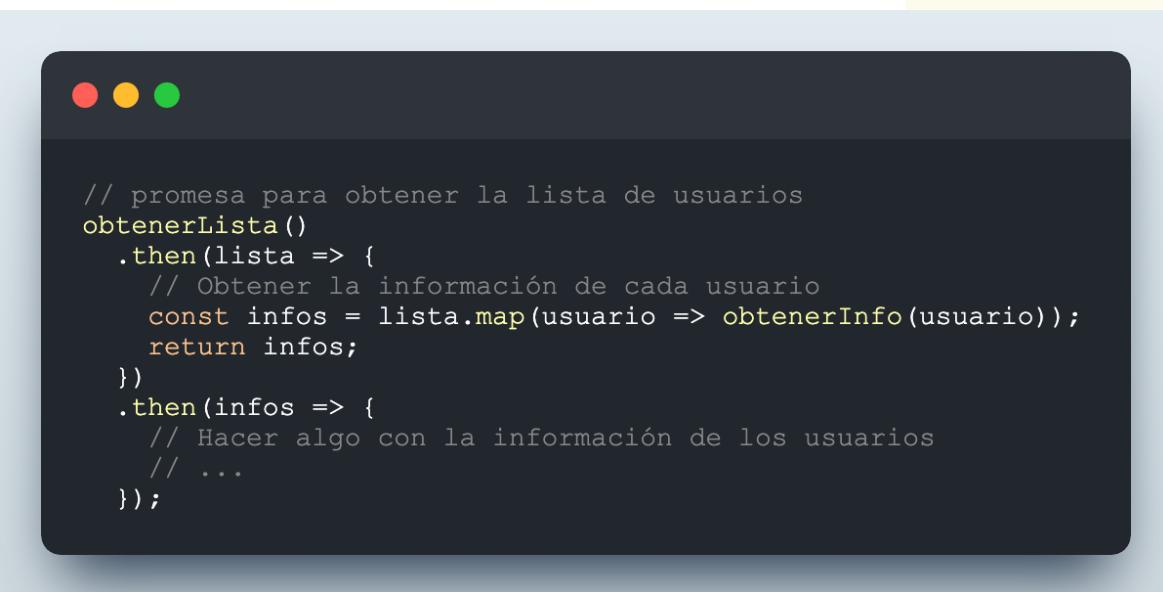
aleatorio entre 0 y 1 lance un error si el número es mayor a 0.3.

### 10.2.2 Encadenamiento de promesas

El encadenamiento de promesas en JavaScript es una técnica que permite ejecutar una serie de tareas asíncronas de forma secuencial. Esto significa que cada promesa se ejecuta una vez que la promesa anterior se ha completado. Esto permite a los desarrolladores escribir código asíncrono de forma más sencilla y mantenible.

Ejemplo:

Supongamos que queremos obtener una lista de usuarios de una base de datos y luego obtener la información de cada usuario. Esto se puede lograr con el encadenamiento de promesas de la siguiente manera:



```
// promesa para obtener la lista de usuarios
obtenerLista()
  .then(lista => {
    // Obtener la información de cada usuario
    const infos = lista.map(usuario => obtenerInfo(usuario));
    return infos;
  })
  .then(infos => {
    // Hacer algo con la información de los usuarios
    // ...
  });

```

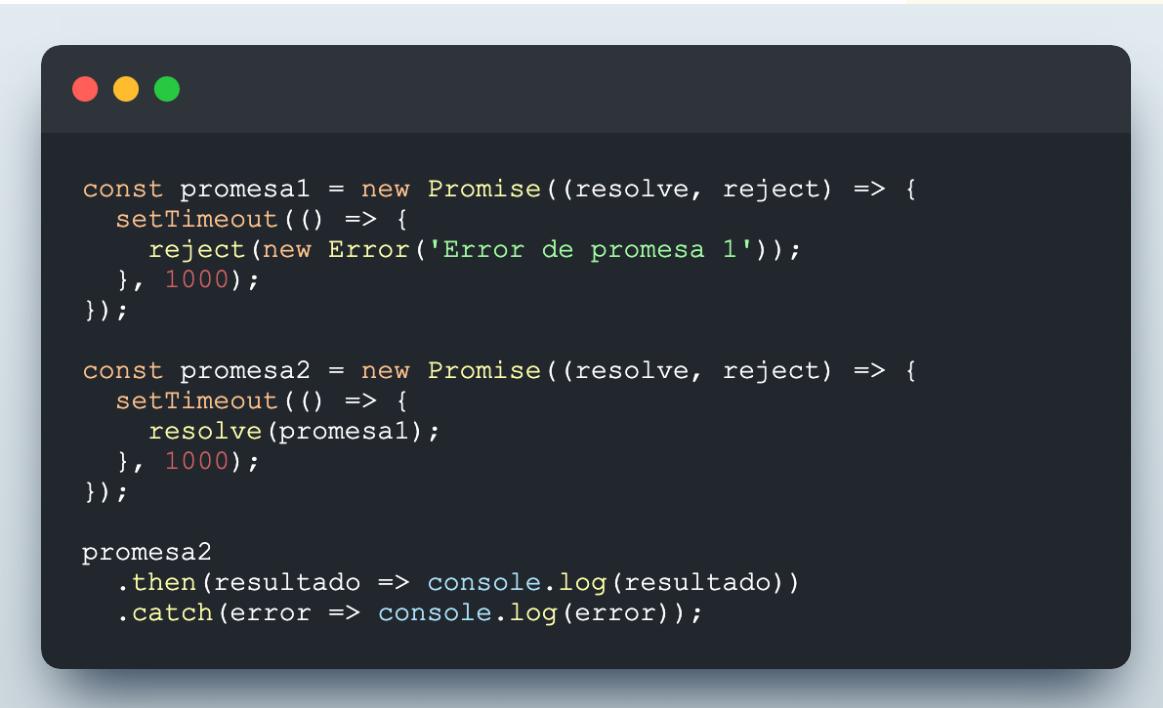
Reto:

Crea una función que tome una lista de números como argumento y devuelva la suma de todos los números usando encadenamiento de promesas.

### 10.2.3 Propagación de errores de promesas

La propagación de errores de promesas en JavaScript es una forma de manejar errores en promesas. Esto significa que si una promesa se rechaza, el error se propagará a la promesa que la llamó.

Ejemplo:



```
const promesa1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    reject(new Error('Error de promesa 1'));
  }, 1000);
});

const promesa2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(promesa1);
  }, 1000);
});

promesa2
  .then(resultado => console.log(resultado))
  .catch(error => console.log(error));
```

En el ejemplo anterior, la promesa2 se resolverá con el resultado de la promesa1. Si la promesa1 se rechaza, el error se propagará a la promesa2 y se mostrará en el bloque catch.

Reto:

Crea una promesa que se rechace después de 5 segundos y otra promesa que se resuelva con el resultado de la primera promesa. Luego, maneja el error en el bloque catch.

### 10.2.4 Fetch API

Fetch es una API (interfaz de programa de aplicación) disponible en ambientes de JavaScript que nos permite realizar peticiones HTTP a un servidor. Fetch es una promesa que permite indicar una URL y obtener datos de una API remota. Esta API nos permite realizar peticiones de tipo GET, POST, PUT, DELETE, etc.

Ejemplo:



```
fetch('https://ejemplo.com/api/datos')
  .then(respuesta => respuesta.json())
  .then(datos => console.log(datos))
  .catch(error => console.error(error))
```

respuesta.json() entrega los datos del pedido sin metadatos.

Reto:

Crea una petición fetch para obtener los datos de una API externa y mostrar los resultados en la consola.

### 10.2.5 Múltiples promesas

Las promesas en JavaScript son objetos que representan el resultado de una operación asíncrona. Esto significa que una promesa puede tener uno de tres estados: pendiente, satisfecha o rechazada. Las promesas múltiples son una forma de manejar varias promesas al mismo tiempo. Esto significa que puedes ejecutar varias promesas al mismo tiempo y recibir una sola respuesta cuando todas las promesas se hayan completado.

## 10 ASÍNCRONÍA

---

Ejemplo:

Supongamos que queremos obtener los datos de dos APIs diferentes. Podemos usar promesas múltiples para hacer esto de la siguiente manera:

```
const api1 = fetch('https://ejemplo.com/api1/datos');
const api2 = fetch('https://ejemplo.com/api2/datos');

Promise.all([api1, api2])
  .then(respuestas => {
    // aquí podemos procesar los datos de ambas APIs
  })
  .catch(error => {
    // aquí podemos manejar cualquier error que ocurra
});
```

Reto:

Crea una función que use promesas múltiples para obtener los datos de tres APIs diferentes. La función debe devolver una promesa que se resuelva con los datos de las tres APIs.

### 10.2.6 Propiedades y métodos del objeto Promise

Promise es un objeto que representa la terminación o el fracaso eventual de una operación asíncrona. Está diseñado para permitir que los flujos de control asíncronos sean expresados de manera más clara y concisa.

Las propiedades y métodos del objeto Promise son los siguientes:

#### Propiedades

- `Promise.prototype`: Esta propiedad es un objeto que contiene los métodos que se pueden usar para manipular una promesa.

### Métodos

- `Promise.all()`: Esta función toma una matriz de promesas y devuelve una promesa que se resuelve cuando todas las promesas de la matriz se han resuelto.
- `Promise.race()`: Esta función toma una lista de promesas y devuelve una promesa que se resuelve o rechaza tan pronto como una de las promesas de la lista se resuelve o rechaza.
- `Promise.reject()`: Esta función devuelve una promesa rechazada con un valor especificado.
- `Promise.resolve()`: Esta función devuelve una promesa resuelta con un valor especificado.

Ejemplo:

A continuación se muestra un ejemplo de código que usa los métodos `Promise.all()` y `Promise.race()` para resolver una matriz de promesas:



```
const promesa1 = Promise.resolve(3);
const promesa2 = 42;
const promesa3 = new Promise((resolver, rechazar) => {
    setTimeout(resolver, 100, 'foo');
});

Promise.all([promesa1, promesa2, promesa3]).then(valores => {
    console.log(valores);
});

Promise.race([promesa1, promesa2, promesa3]).then(valor => {
    console.log(valor);
});
```

Reto:

Intenta crear una promesa que se resuelva después de 5 segundos y luego imprime un mensaje en la consola.

### 10.3 `async/await`

Async/Await es una característica de JavaScript que permite escribir código asíncrono de manera síncrona. Esto significa que puedes escribir código asíncrono como si fuera código síncrono, lo que hace que sea mucho más fácil de leer y mantener.

Ejemplo:



En este ejemplo, la función `obtenerDatos` es una función asíncrona que usa `fetch` para obtener datos de una URL y luego imprimirlas en la consola. La palabra clave `await` se usa para esperar a que la promesa devuelta por `fetch` se resuelva antes de continuar con el código.

Reto:

Escribe una función asíncrona que use `fetch` para obtener una lista de usuarios de una URL y luego imprima el nombre de cada usuario en la consola.

# 11 Módulos

## 11.1 Módulos y herramientas

### 11.1.1 Historia de los módulos

Los módulos en JavaScript son una forma de organizar el código para que sea más fácil de mantener y compartir. Esta característica fue introducida en ECMAScript 6 (ES6) en 2015, y desde entonces se ha convertido en una parte integral de la programación en JavaScript.

## 11 MÓDULOS

---

Los módulos permiten a los desarrolladores dividir su código en múltiples archivos, lo que facilita la organización y la reutilización del código. Esto también permite a los desarrolladores compartir código entre proyectos, lo que ahorra tiempo y esfuerzo.

Han existido varios sistemas y tipos de sintaxis para importar y exportar módulos en la historia de JavaScript. Los más populares son módulos de JavaScript, CommonJS (usados en Node.js), AMD y Requirejs (que son menos utilizados y considerados legacy).

Ejemplo:

Supongamos que estamos creando una aplicación web que necesita una función para calcular el área de un círculo. En lugar de escribir la función en el mismo archivo, podemos crear un módulo separado para la función y luego importarlo en el archivo principal.



The image shows a terminal window with two code snippets. The top snippet, 'area.js', contains a function that calculates the area of a circle. The bottom snippet, 'main.js', imports this function and uses it to calculate the area of a circle with a radius of 5, logging the result to the console.

```
// area.js
export function calcularArea(radius) {
    return Math.PI * radius * radius;
}

// main.js
import { calcularArea } from './area.js';

const area = calcularArea(5);
console.log(area); // 78.53981633974483
```

## 11 MÓDULOS

---

Reto:

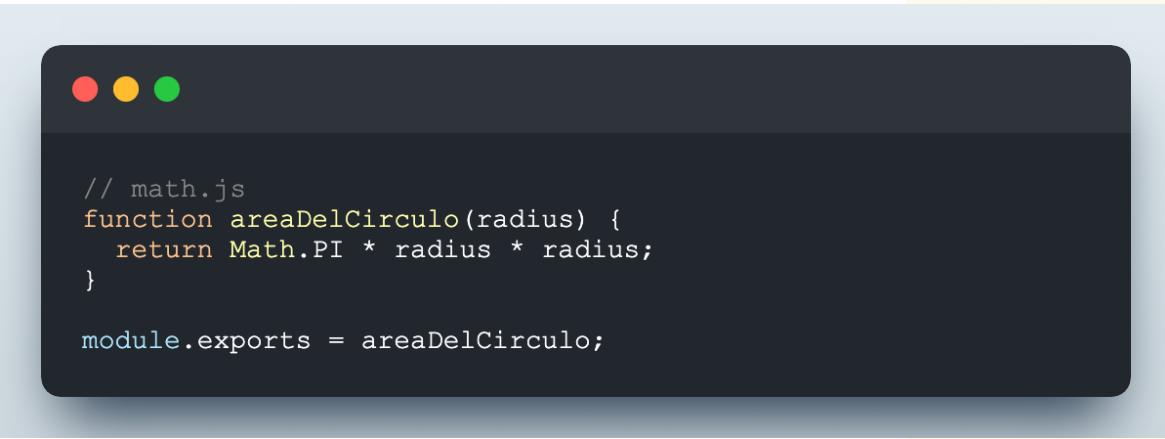
Crea un módulo que contenga una función para calcular el área de un rectángulo. Luego, importa el módulo en un archivo principal y usa la función para calcular el área de un rectángulo con lados de 5 y 10.

### 11.1.2 CommonJS

CommonJS es una especificación de módulos para JavaScript que permite a los desarrolladores crear y compartir código entre diferentes proyectos. Se utiliza principalmente en Node.js que es un ambiente para correr JavaScript en el backend.

Ejemplo:

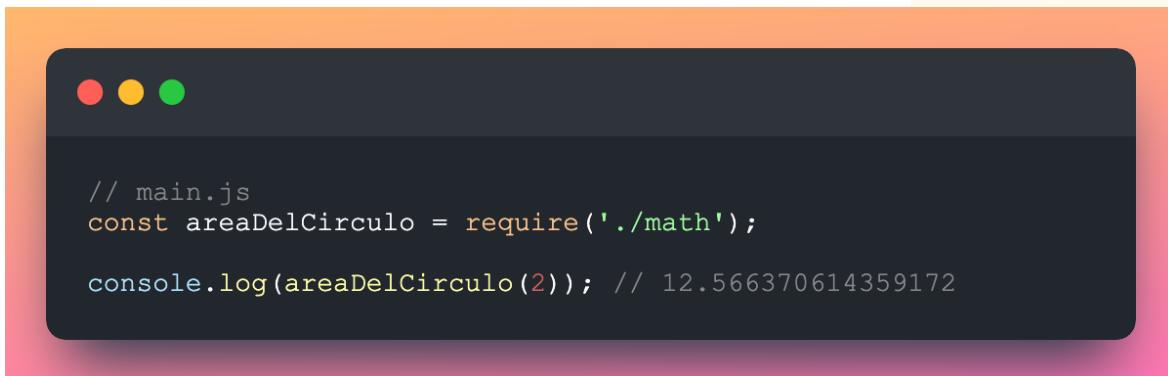
Supongamos que tenemos un archivo llamado math.js que contiene una función para calcular el área de un círculo.



```
// math.js
function areaDelCirculo(radius) {
    return Math.PI * radius * radius;
}

module.exports = areaDelCirculo;
```

Ahora, en otro archivo, podemos importar la función areaDelCirculo usando CommonJS.



A screenshot of a terminal window with a dark theme. The window title bar has three colored dots (red, yellow, green). The main area of the terminal contains the following code:

```
// main.js
const areaDelCirculo = require('./math');

console.log(areaDelCirculo(2)); // 12.566370614359172
```

Reto:

Crea un archivo llamado math.js que contenga una función para calcular el área de un rectángulo. Luego, en otro archivo, importa la función areaDelRectangulo usando CommonJS y usa la función para calcular el área de un rectángulo con un ancho de 4 y un alto de 5. Corre el código en Node.js.

### 11.1.3 Webpack

Webpack es una herramienta de empaquetado de JavaScript que se usa para compilar y empaquetar archivos o módulos de JavaScript, HTML, CSS y otros recursos en un solo archivo. Esto permite a los desarrolladores crear aplicaciones web más rápidas y escalables.

Ejemplo:

Supongamos que estamos creando una aplicación web que usa HTML, CSS y JavaScript. Usando Webpack, podemos empaquetar todos estos archivos en un solo archivo JavaScript. Esto significa que en lugar de tener que cargar cada archivo por separado, solo necesitamos cargar el archivo empaquetado. Esto acelera el tiempo de carga de la aplicación y mejora la experiencia del usuario. Webpack es una tecnología bastante amplia que requiere su propia documentación.

Reto:

## 11 MÓDULOS

---

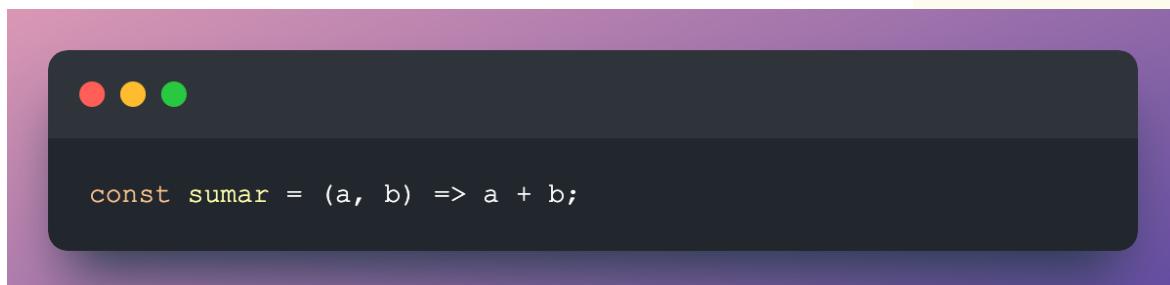
Revisa la documentación de Webpack y empaca algunos archivos de JavaScript en uno solo.

### 11.1.4 Babel

Babel es un compilador de JavaScript que convierte código JavaScript moderno en código JavaScript compatible con versiones anteriores de los navegadores. Esto significa que puedes usar características de JavaScript modernas, como clases, funciones de flecha, etc., y Babel se encargará de convertirlo en código que los navegadores antiguos puedan entender. Esta herramienta puede ser útil para cambiar el tipo de módulos utilizados en JavaScript. Babel es una tecnología bastante amplia que requiere su propia documentación.

Ejemplo:

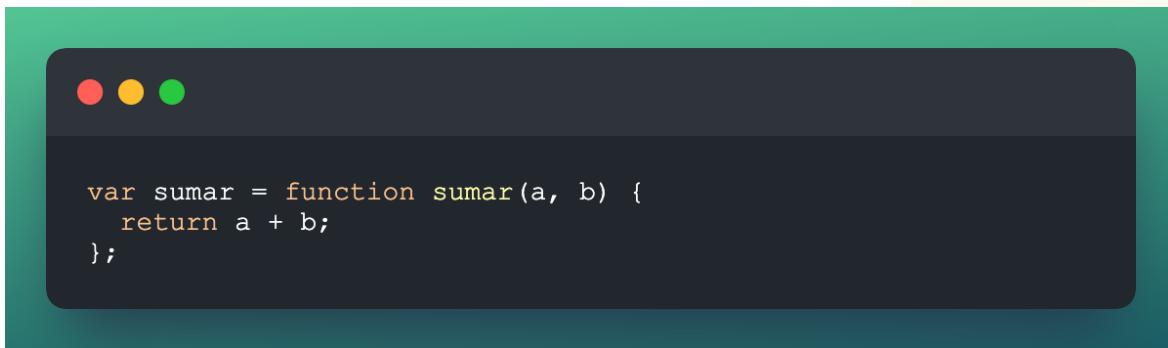
Por ejemplo, si quisieramos usar una función de flecha para crear una función que sume dos números, podríamos escribir el siguiente código:



Sin embargo, los navegadores antiguos no entienden las funciones de flecha. Por lo tanto, si queremos que nuestro código funcione en todos los navegadores, necesitamos usar Babel para convertir nuestro código en una versión compatible con navegadores antiguos. El código convertido por Babel se vería así:

## 11 MÓDULOS

---



```
var sumar = function sumar(a, b) {
    return a + b;
};
```

Reto:

Revisa la documentación de Babel y convierte un tipo de módulo a otro.

### 11.1.5 Extensiones .mjs vs .js

Las extensiones .mjs y .js son dos tipos de archivos de JavaScript. La extensión .mjs se usa para archivos de JavaScript modulares, mientras que la extensión .js se usa para archivos de JavaScript estándar. La extensión .mjs es necesaria en Node.js para indicar que se usen módulos de JavaScript, de otra manera utiliza módulos CommonJS.

Un ejemplo práctico de la diferencia entre .mjs y .js es el siguiente:

#### Archivo .mjs

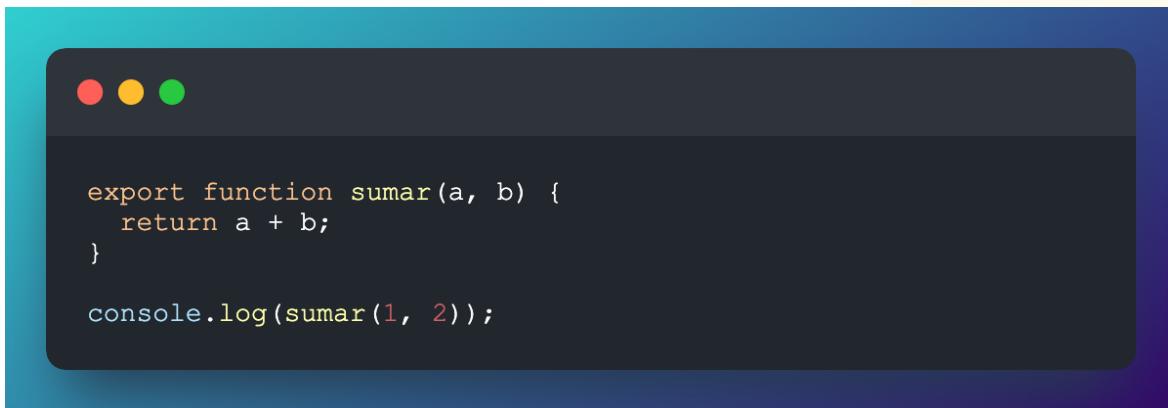


```
import { sumar } from './math.js';
console.log(sumar(1, 2));
```

#### Archivo .js

## 11 MÓDULOS

---



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

console.log(sumar(1, 2));
```

En el ejemplo anterior, el archivo .mjs importa la función sumar desde el archivo math.js, mientras que el archivo .js define la función sumar directamente en el archivo.

Reto:

Crea un archivo .mjs que importe la función sumar desde un archivo .js y luego use la función sumar para imprimir el resultado de la suma de dos números.

### 11.2 Exportar

Exportar en JavaScript es una forma de compartir código entre archivos. Esto significa que un archivo puede exportar variables, funciones y objetos para que otros archivos los puedan usar.

Ejemplo:

Supongamos que tenemos un archivo llamado math.js que contiene una función para calcular el área de un círculo:

## 11 MÓDULOS

---

```
function areaCirculo(radio) {  
    return Math.PI * radio * radio;  
}
```

Para que otros archivos puedan usar esta función, necesitamos exportarla. Esto se hace usando la palabra clave `export`:

```
export function areaCirculo(radio) {  
    return Math.PI * radio * radio;  
}
```

Ahora, en cualquier otro archivo, podemos importar la función usando la palabra clave `import`:

```
import { areaCirculo } from './math.js';  
  
console.log(areaCirculo(2)); // 12.566370614359172
```

Reto:

Crea un archivo llamado `math.js` que exporte una función llamada `sumar` que tome

## 11 MÓDULOS

---

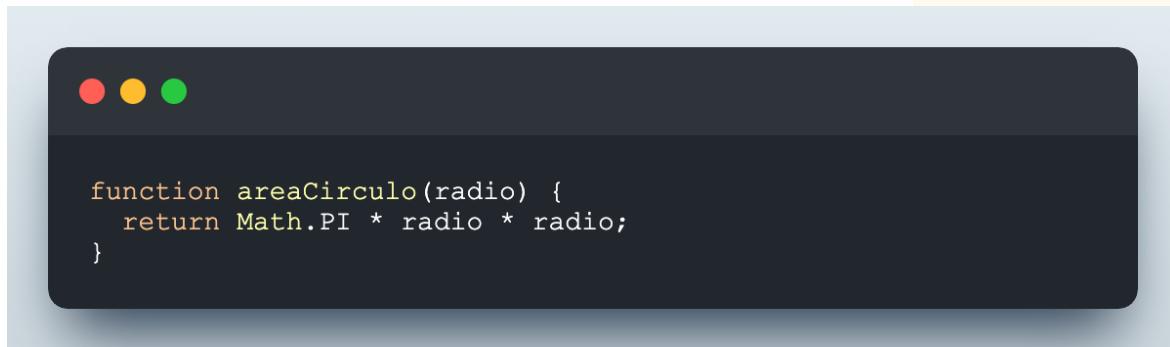
dos números como argumentos y devuelva la suma de ellos. Luego, crea otro archivo llamado app.js que importe la función sumar y la use para imprimir la suma de dos números en la consola.

### 11.3 Importar

Importar en JavaScript es una forma de incluir código externo en un archivo JavaScript. Esto permite a los desarrolladores reutilizar código y ahorrar tiempo al escribir código.

Ejemplo:

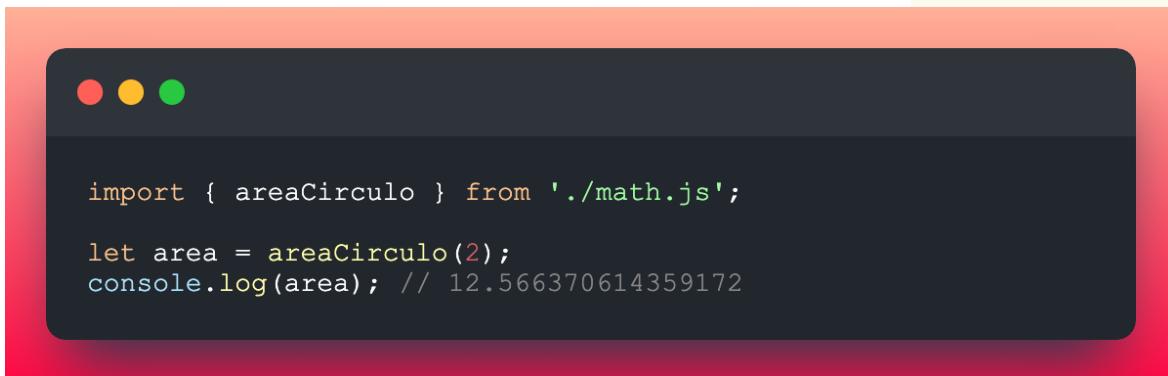
Por ejemplo, supongamos que tenemos un archivo llamado math.js que contiene una función para calcular el área de un círculo:



A screenshot of a terminal window with a dark theme. In the top-left corner, there are three small colored circles: red, yellow, and green. The main area of the terminal contains the following code:

```
function areaCirculo(radio) {
    return Math.PI * radio * radio;
}
```

Ahora, si queremos usar esta función en otro archivo, podemos importarla usando la palabra clave import:



```
import { areaCirculo } from './math.js';

let area = areaCirculo(2);
console.log(area); // 12.566370614359172
```

Reto:

Crea un archivo llamado math.js que contenga una función para calcular el área de un rectángulo. Luego, importa la función en otro archivo y usala para calcular el área de un rectángulo con lados de 3 y 4.

### 11.4 Exportación predeterminada

Exportación predeterminada en JavaScript es una característica de ES6 que permite a los desarrolladores exportar una sola declaración o valor de un módulo. Esto significa que un desarrollador puede exportar una sola clase, función, objeto o variable desde un archivo JavaScript. Esto es útil para evitar la necesidad de escribir muchas líneas de código para exportar varios elementos desde un archivo.

Ejemplo:

## 11 MÓDULOS

---

```
// Archivo: miModulo.js
export default class MiClase {
  constructor() {
    this.nombre = 'MiClase';
  }
}
```

```
// Archivo: main.js
import MiClase from './miModulo';

const miClase = new MiClase();
console.log(miClase.nombre); // MiClase
```

Reto:

Crea un archivo JavaScript llamado miModulo.js que exporte una clase llamada MiClase. Luego, crea un archivo main.js que importe la clase MiClase y cree una instancia de la misma. Por último, imprime el nombre de la clase en la consola.

### 11.5 Alias de importación

Los alias de importación en JavaScript nos permiten importar módulos y asignarles un nombre más corto para su uso. Esto nos permite ahorrar tiempo y mejorar la legibilidad del código.

## 11 MÓDULOS

---

Ejemplo:

```
import { obtenerDatos as obtenerDatosDeAPI } from './api';
obtenerDatosDeAPI();
```

Reto:

Intenta importar un módulo y asignarle un alias para su uso.

### 11.6 Importar paquetes

Importar paquetes en JavaScript es una forma de incluir código externo en nuestro proyecto. Esto nos permite aprovechar la funcionalidad de otros desarrolladores sin tener que escribir todo el código desde cero. Estos paquetes creados por compañías o desarrolladores se pueden instalar usando herramientas como npm (que viene con Node.js) o yarn.

Ejemplo:

Por ejemplo, si queremos usar la librería de React para construir una aplicación web, podemos importarla en nuestro proyecto de la siguiente manera:



```
// instalamos el paquete con npm en la terminal
// con el comando 'npm i react'
import React from 'react';
```

Reto:

Instala un paquete con npm. Puedes visitar el sitio web de npm <https://www.npmjs.com/> y ver la documentación en instrucciones de instalación de cientos de librerías.

### 11.7 Cargar módulos dinámicamente

Cargar módulos dinámicamente en JavaScript es una técnica que permite cargar contenido de forma asíncrona. Esto significa que los usuarios pueden cargar módulos en cualquier parte del código.

Ejemplo:



```
import('modulo.js')
  .then((modulo) => {
    modulo.sumar(2,3);
  })
  .catch((error) => {
    console.log(error.message);
});
```

Reto:

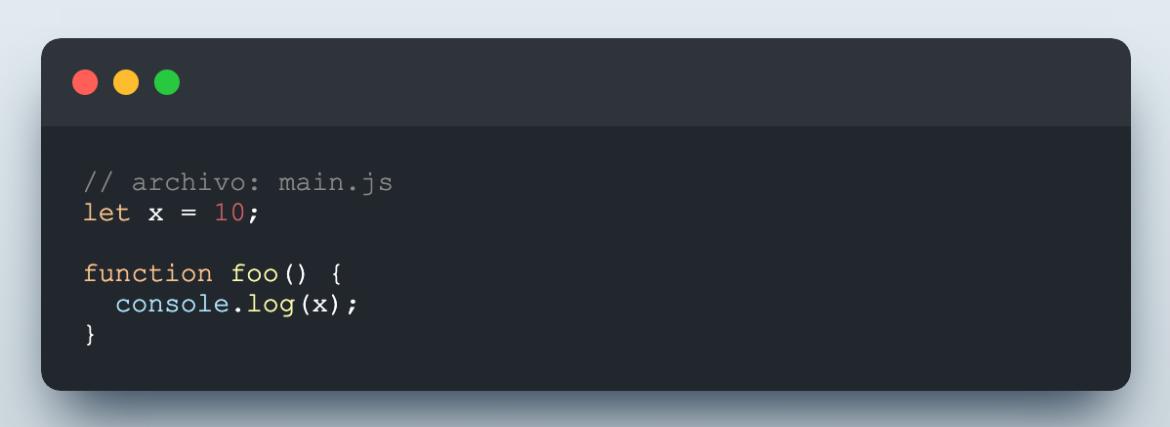
Importa un módulo que has creado previamente de forma dinámica.

### 11.8 Ámbito de un módulo

El ámbito de un módulo en JavaScript es el contexto en el que se ejecuta el código. Esto significa que el ámbito determina qué variables y funciones están disponibles para su uso. El ámbito de un módulo se establece cuando se crea el módulo y se mantiene a lo largo de la ejecución del código.

Ejemplo:

Por ejemplo, considera el siguiente código:

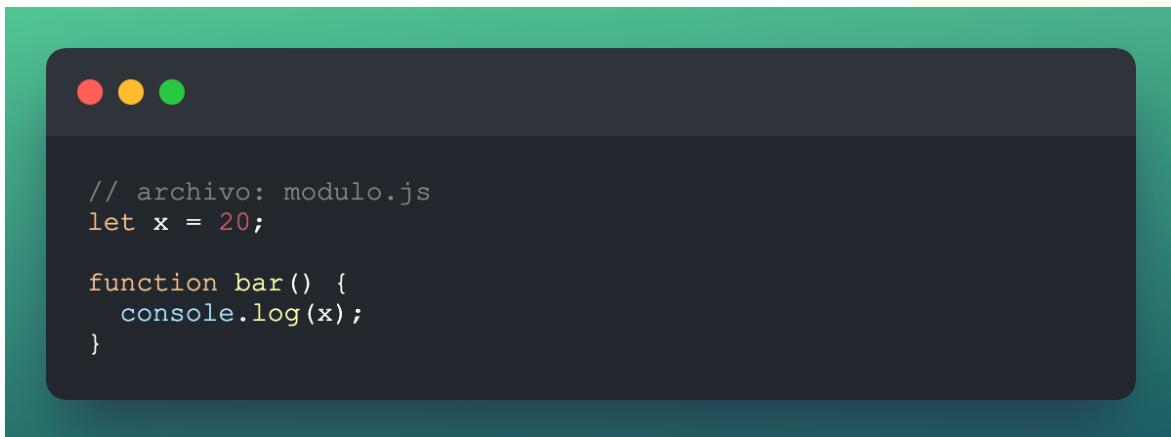


```
// archivo: main.js
let x = 10;

function foo() {
    console.log(x);
}
```

## 12 ITERADORES Y GENERADORES

---



```
// archivo: modulo.js
let x = 20;

function bar() {
    console.log(x);
}
```

En este ejemplo, el ámbito de main.js es diferente al ámbito de modulo.js. Esto significa que la variable x en main.js tendrá un valor de 10, mientras que la variable x en modulo.js tendrá un valor de 20. Esto significa que cuando se llama a la función foo desde main.js, imprimirá el valor de 10, mientras que cuando se llama a la función bar desde modulo.js, imprimirá el valor de 20.

Reto:

Escribe un programa que declare una variable x con un valor de 10 en un archivo main.js y otra variable x con un valor de 20 en un archivo modulo.js. Luego, escribe una función foo en main.js que imprima el valor de x y otra función bar en modulo.js que imprima el valor de x. Finalmente, llama a ambas funciones desde main.js para verificar que el ámbito de cada archivo se mantiene.

# 12 Iteradores y generadores

## 12.1 Objetos iteradores

Los objetos iteradores en JavaScript son objetos que permiten iterar sobre un conjunto de elementos. Estos objetos tienen un método llamado next() que devuelve el siguiente elemento en la secuencia.

## 12 ITERADORES Y GENERADORES

---

Un ejemplo práctico de un objeto iterador en JavaScript es el siguiente:

```
let iterador = {
  elementos: [1, 2, 3],
  indice: 0,
  next() {
    if (this.indice < this.elementos.length) {
      return {
        value: this.elementos[this.indice++],
        done: false
      };
    } else {
      return { done: true };
    }
  }
};
```

En este ejemplo vemos que hemos creado el objeto iterador manualmente pero JavaScript viene con objetos iteradores que se generan a partir de un objeto iterable como una lista. Veremos más detalles en las siguientes lecciones.

Reto:

Crea un objeto iterador que itere sobre una lista de textos y devuelva cada elemento de la lista como un objeto con dos propiedades: una para el índice y otra para el valor.

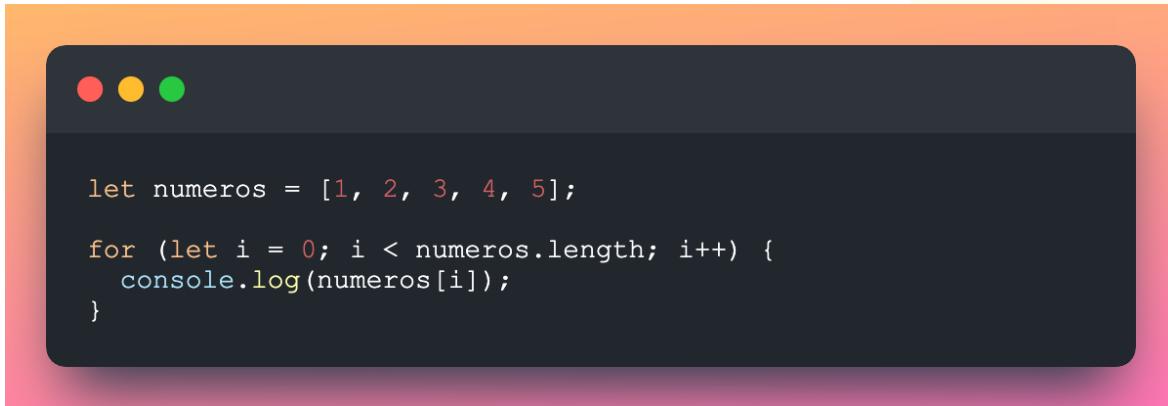
### 12.2 Objetos iterables

Los objetos iterables en JavaScript son aquellos que se pueden recorrer con un bucle. Esto significa que se pueden usar para recorrer una colección de elementos, como un array o un objeto.

## 12 ITERADORES Y GENERADORES

---

Un ejemplo práctico de un objeto iterable en JavaScript es un array. Por ejemplo, podemos usar un bucle for para recorrer un array de números y mostrar cada uno de ellos por pantalla:



```
let numeros = [1, 2, 3, 4, 5];

for (let i = 0; i < numeros.length; i++) {
    console.log(numeros[i]);
}
```

Reto:

Crea un array con los nombres de tus amigos y usa un bucle for para mostrar cada uno de ellos por pantalla.

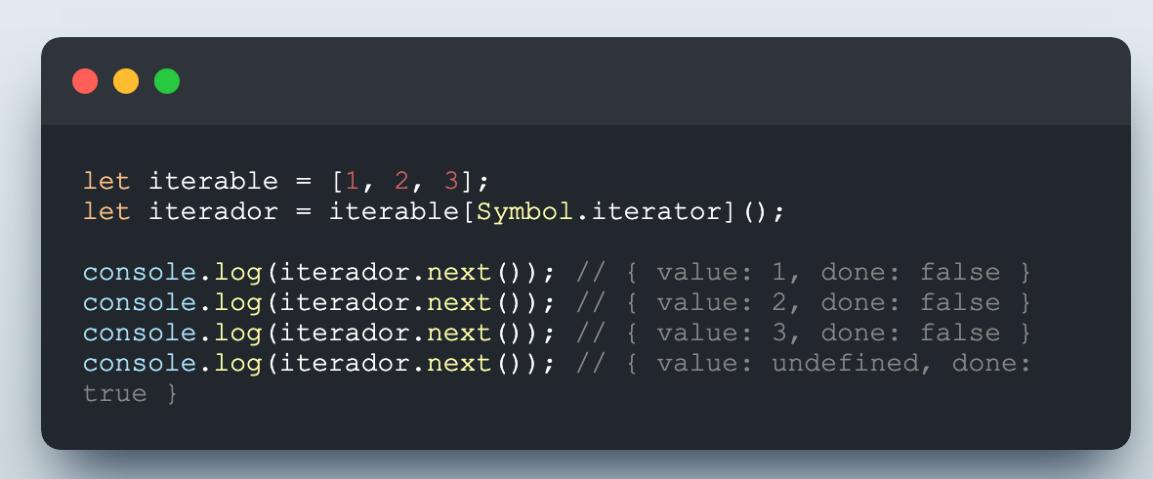
### 12.3 Método de iterador next()

**next()** en JavaScript es una función que se utiliza para iterar sobre un objeto iterable. Esto significa que se puede usar para recorrer una secuencia de elementos, como una lista, un texto o un objeto.

Ejemplo:

## 12 ITERADORES Y GENERADORES

---



```
let iterable = [1, 2, 3];
let iterador = iterable[Symbol.iterator]();

console.log(iterador.next()); // { value: 1, done: false }
console.log(iterador.next()); // { value: 2, done: false }
console.log(iterador.next()); // { value: 3, done: false }
console.log(iterador.next()); // { value: undefined, done: true }
```

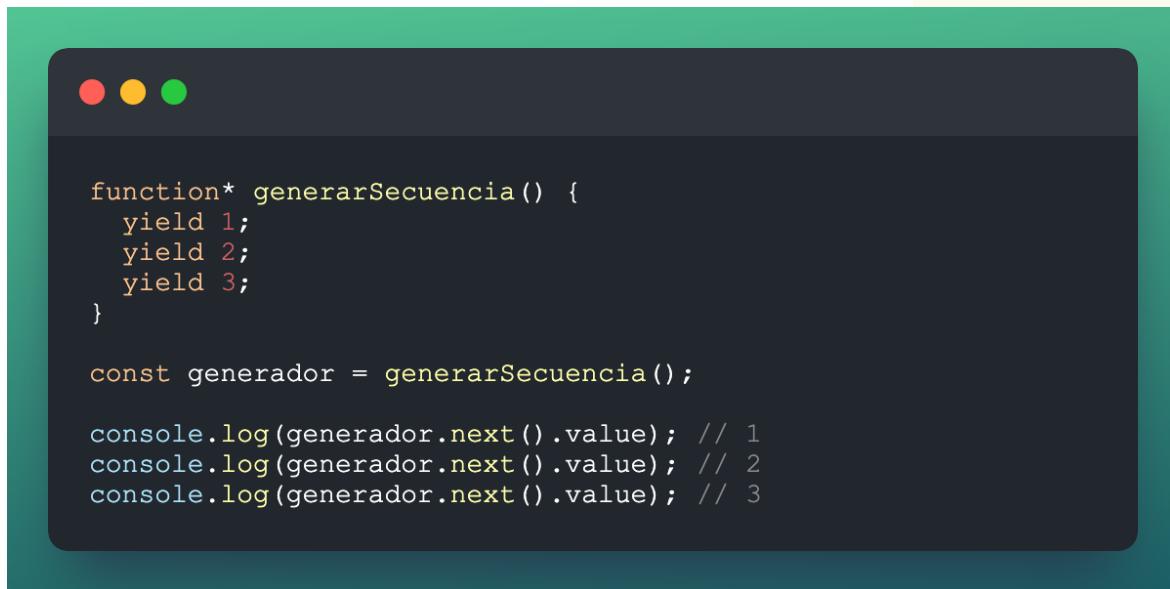
Reto:

Crea una función que use la función **next()** para recorrer una matriz de números y devolver la suma de todos los elementos.

### 12.4 Funciones generadoras

Una función generadora en JavaScript es una función especial que devuelve un objeto iterable que puede ser usado para generar una secuencia de valores. Esto significa que, en lugar de devolver un único valor, una función generadora devuelve una secuencia de valores a medida que se va ejecutando.

Ejemplo:



A screenshot of a terminal window with three colored window control buttons (red, yellow, green) at the top. The terminal shows the following code:

```
function* generarSecuencia() {
    yield 1;
    yield 2;
    yield 3;
}

const generador = generarSecuencia();

console.log(generador.next().value); // 1
console.log(generador.next().value); // 2
console.log(generador.next().value); // 3
```

Reto:

Crea una función generadora que devuelva una secuencia de números del 1 al 10. Luego, usa el objeto generador para imprimir cada número en la consola.

### 12.5 yield

**yield** es una palabra clave en JavaScript que se usa para crear generadores. Un generador es una función especial que puede devolver múltiples valores a lo largo del tiempo. Esto significa que un generador puede “recordar” su estado entre llamadas, lo que le permite devolver valores diferentes cada vez que se llama.

Ejemplo:

A continuación se muestra un ejemplo de un generador que devuelve los números pares entre 0 y 10:

## 13 INTERNACIONALIZACIÓN

---

```
function* obtenerPares() {
  for (let i = 0; i <= 10; i++) {
    if (i % 2 === 0) {
      yield i;
    }
  }
}

const pares = obtenerPares();

console.log(pares.next().value); // 0
console.log(pares.next().value); // 2
console.log(pares.next().value); // 4
console.log(pares.next().value); // 6
console.log(pares.next().value); // 8
console.log(pares.next().value); // 10
```

Reto:

Crea un generador que devuelva los números impares entre 0 y 10.

# 13 Internacionalización

## 13.1 El objeto Intl

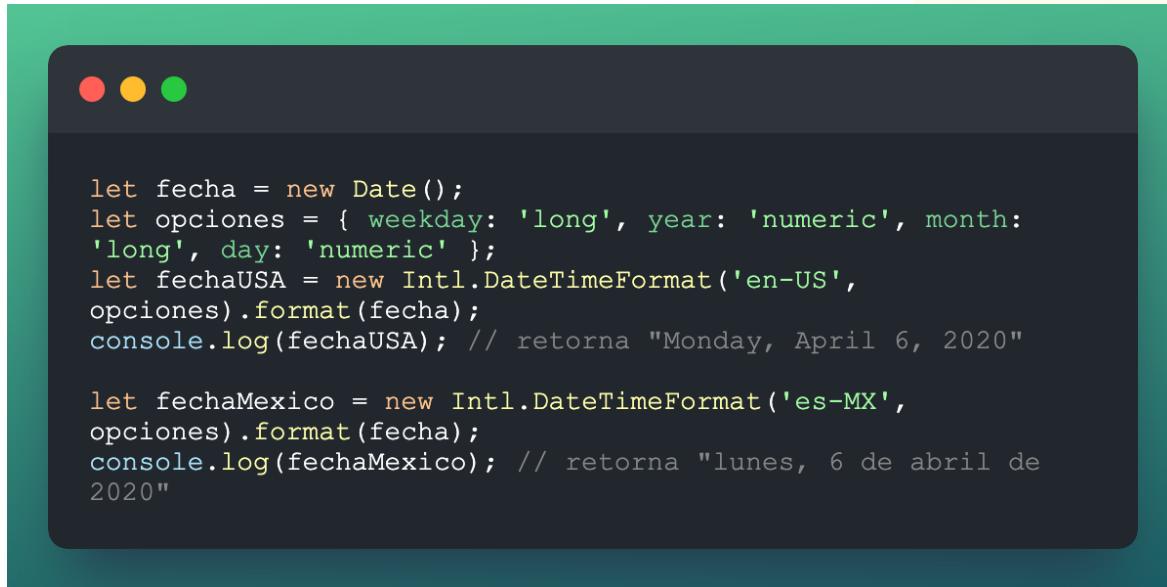
El objeto Intl (Internationalization) es una API de JavaScript que proporciona soporte para la internacionalización de aplicaciones. Esta API proporciona una variedad de funciones para manipular y formatear cadenas, números, fechas y horas, entre otros.

Ejemplo:

## 13 INTERNACIONALIZACIÓN

---

Por ejemplo, podemos usar el objeto Intl para formatear una fecha en un formato específico. Por ejemplo, para formatear una fecha en el formato de fecha larga de Estados Unidos, podemos usar el siguiente código:



```
let fecha = new Date();
let opciones = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
let fechaUSA = new Intl.DateTimeFormat('en-US', opciones).format(fecha);
console.log(fechaUSA); // retorna "Monday, April 6, 2020"

let fechaMexico = new Intl.DateTimeFormat('es-MX', opciones).format(fecha);
console.log(fechaMexico); // retorna "lunes, 6 de abril de 2020"
```

Reto:

Utilizando el objeto Intl, escribe una función que tome una fecha como parámetro y devuelva una cadena con la fecha formateada en el formato de fecha corta de España.

### 13.2 El objeto DateTimeFormat

El objeto DateTimeFormat es una interfaz de programación de aplicaciones (API) que proporciona una forma de formatear y analizar cadenas de fecha y hora en JavaScript. Esta API permite a los desarrolladores crear cadenas de fecha y hora personalizadas para mostrar a los usuarios, así como analizar cadenas de fecha y hora para obtener información sobre la fecha y la hora.

Ejemplo:

## 13 INTERNACIONALIZACIÓN

---

A continuación se muestra un ejemplo de cómo usar el objeto `DateFormatter` para formatear una fecha y hora en una cadena de texto:



```
let fecha = new Date();
let opciones = { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric' };
let formateador = new Intl.DateTimeFormat('es-ES', opciones);
let fechaTexto = formateador.format(fecha);

console.log(fechaTexto); // imprime "domingo, 28 de junio de 2020"
```

Reto:

Utilizando el objeto `DateFormatter`, crea una función que tome una fecha y hora como parámetro y devuelva una cadena de texto con la fecha y hora formateadas en el formato “dd/mm/aaaa hh:mm” .

### 13.3 El objeto `NumberFormat`

El objeto `NumberFormat` en JavaScript es una herramienta para formatear números y monedas. Esta herramienta permite a los desarrolladores especificar el formato de los números, como el número de decimales, el separador de miles, el símbolo de moneda, etc. Esto permite a los desarrolladores mostrar los números de manera más legible para los usuarios.

Ejemplo:

Supongamos que queremos formatear el número 1234567.89 para mostrarlo con dos decimales, un separador de miles y el símbolo de la moneda dólar. Esto se puede hacer usando el objeto `NumberFormat` de la siguiente manera:

## 13 INTERNACIONALIZACIÓN

---

```
const formato = new Intl.NumberFormat('en-US', {  
    style: 'currency',  
    currency: 'USD',  
    minimumFractionDigits: 2  
});  
  
const numero = formato.format(1234567.89);  
  
console.log(numero); // $1,234,567.89
```

Reto:

Utiliza el objeto NumberFormat para formatear el número 1234567.89 para mostrarlo con tres decimales, un separador de miles y el símbolo de la moneda euro.

### 13.4 El objeto Collator

El objeto Collator en JavaScript es una herramienta para comparar y ordenar cadenas de caracteres en diferentes lenguajes y configuraciones regionales. Esto significa que puede usarse para comparar cadenas de caracteres en diferentes idiomas y configuraciones regionales, como el español, el inglés, el francés, el alemán, etc.

Ejemplo:

Por ejemplo, si queremos ordenar una lista de palabras en español, podemos usar el objeto Collator para hacerlo. Por ejemplo, si tenemos la lista de palabras ['casa', 'caballo', 'cocodrilo', 'cocina'], podemos usar el objeto Collator para ordenarla de la siguiente manera:

```
let lista = ['casa', 'caballo', 'cocodrilo', 'cocina'];
let colador = new Intl.Collator('es');
lista.sort(colador.compare);
console.log(lista); // ['caballo', 'casa', 'cocina',
'cocodrilo']
```

Reto:

Crea una función que tome una lista de palabras en español como parámetro y devuelva la lista ordenada usando el objeto Collator.

## 14 Meta

### 14.1 El objeto Proxy

El objeto Proxy en JavaScript es una herramienta que nos permite interceptar y manipular cualquier operación realizada sobre un objeto. Esto significa que podemos controlar el comportamiento de un objeto cuando se realizan operaciones como lectura, escritura, enumeración, invocación, etc.

Ejemplo:

Por ejemplo, podemos usar un objeto Proxy para controlar el comportamiento de un objeto cuando se intenta leer una propiedad. Por ejemplo, podemos usar un Proxy para devolver un valor predeterminado si la propiedad no existe:

```
const obj = {  
    nombre: 'Juan',  
    edad: 30  
};  
  
const proxy = new Proxy(obj, {  
    get(objetivo, propiedad) {  
        if (propiedad in objetivo) {  
            return objetivo[propiedad];  
        } else {  
            return 'Valor predeterminado';  
        }  
    }  
});  
  
console.log(proxy.nombre); // 'Juan'  
console.log(proxy.genero); // 'Valor predeterminado'
```

Reto:

Crea un objeto Proxy que controle el comportamiento de un objeto cuando se intenta escribir una propiedad. El Proxy debe validar que el valor de la propiedad sea un número entero y, si no es así, debe lanzar un error.

### 14.2 El objeto Reflect

El objeto Reflect es una API de JavaScript que proporciona métodos para interactuar con los objetos de JavaScript. Esta API proporciona una forma de interactuar con los objetos de JavaScript de una manera más sencilla y segura. Esto significa que los desarrolladores pueden usar el objeto Reflect para realizar tareas como definir propiedades, definir métodos, establecer y obtener valores, etc.

## 15 SIGUIENTES PASOS

---

Ejemplo:

Por ejemplo, el siguiente código muestra cómo usar el objeto Reflect para definir una propiedad en un objeto:



```
● ● ●

let obj = {};
Reflect.defineProperty(obj, 'nombre', {
  value: 'Juan',
  writable: true
});

console.log(obj.nombre); // 'Juan'
```

Reto:

Crea una función que use el objeto Reflect para definir una propiedad en un objeto y luego imprima el valor de la propiedad.

# 15 Siguientes pasos

## 15.1 Herramientas

Aprende las siguientes herramientas:

1. **Node.js**: Es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome. Proporciona una plataforma para construir aplicaciones de servidor y aplicaciones de línea de comandos con JavaScript.
2. **React.js**: Es una biblioteca de JavaScript de código abierto para crear interfaces de usuario. Está diseñado para ayudar a los desarrolladores a construir

aplicaciones web y móviles escalables.

3. **Angular:** Esta es una alternativa a React. Es un marco de JavaScript de código abierto para desarrollar aplicaciones web. Está diseñado para ayudar a los desarrolladores a construir aplicaciones web modernas y escalables.
4. **TypeScript:** Es un superconjunto de JavaScript que proporciona una sintaxis más estructurada y una mejor administración de errores. Está diseñado para ayudar a los desarrolladores a escribir código JavaScript más robusto y escalable.

### 15.2 Recursos

1. MDN Web Docs: Es una de las fuentes de referencia más completas para JavaScript. Ofrece documentación detallada sobre la sintaxis, los conceptos y las API de JavaScript. También ofrece tutoriales y ejemplos para ayudar a los desarrolladores a aprender y usar JavaScript. <https://developer.mozilla.org/es/docs/Web/JavaScript>
2. W3Schools: Es una de las fuentes de referencia más populares para JavaScript. Ofrece tutoriales, ejemplos y referencias para ayudar a los desarrolladores a aprender y usar JavaScript. <https://www.w3schools.com/js/>
3. Stack Overflow: Es una de las comunidades de desarrolladores más grandes en línea. Ofrece una gran cantidad de preguntas y respuestas sobre JavaScript, así como una sección de discusión para discutir problemas y soluciones relacionadas con JavaScript. <https://stackoverflow.com/questions/tagged/javascript>
4. Academia X: Ofrece una variedad de cursos interactivos para ayudar a los desarrolladores a aprender y usar JavaScript. <https://www.academia-x.com>

### 15.3 ¿Que viene después?

Estos son los siguientes pasos recomendados:

1. Practicar JavaScript: Una vez que hayas aprendido los conceptos básicos de JavaScript, es importante practicar para mejorar tus habilidades. Puedes hacer esto escribiendo código en un editor de texto y ejecutándolo en un navegador web.
2. Aprender un marco de JavaScript: Un marco de JavaScript es un conjunto de herramientas y librerías que te ayudan a desarrollar aplicaciones web más rápido. Algunos de los marcos más populares son React, Angular y Vue.
3. Aprender una biblioteca de JavaScript: Una biblioteca de JavaScript es un conjunto de funciones y herramientas que te ayudan a realizar tareas comunes de desarrollo web. Algunas de las bibliotecas más populares son jQuery, Lodash y Underscore.
4. Aprender una herramienta de desarrollo de JavaScript: Una herramienta de desarrollo de JavaScript es una herramienta que te ayuda a escribir, depurar y optimizar tu código. Algunas de las herramientas más populares son Webpack, Babel y Gulp.
5. Aprender una base de datos relacional: Una base de datos relacional es una base de datos que almacena datos en tablas relacionadas entre sí. Algunas de las bases de datos relacionales más populares son MySQL, PostgreSQL y SQLite.
6. Aprender una base de datos no relacional: Una base de datos no relacional es una base de datos que almacena datos en formato de documentos. Algunas de las bases de datos no relacionales más populares son MongoDB, CouchDB y Redis.

## 15.4 Preguntas de entrevista

1. ¿Qué es JavaScript?
  - JavaScript es un lenguaje de programación interpretado, orientado a objetos, que se utiliza principalmente para crear contenido interactivo en páginas web.
2. ¿Qué es una variable en JavaScript?
  - Una variable es un contenedor para almacenar datos. En JavaScript, una variable se declara usando la palabra clave “var” .
3. ¿Cómo se declara una variable en JavaScript?
  - Una variable se declara usando la palabra clave “var” seguida del nombre de la variable y, opcionalmente, el valor que se le asignará.
4. ¿Qué es una función en JavaScript?
  - Una función es un bloque de código que se puede ejecutar cuando se le llama. Las funciones se utilizan para realizar tareas específicas y se pueden reutilizar en diferentes partes de un programa.
5. ¿Cómo se define una función en JavaScript?
  - Una función se define usando la palabra clave “function” seguida del nombre de la función y los parámetros que se pasarán a la función.
6. ¿Qué es un objeto en JavaScript?
  - Un objeto es una colección de propiedades y métodos relacionados. Los objetos se utilizan para representar entidades reales o abstractas en un programa.
7. ¿Cómo se crea un objeto en JavaScript?
  - Un objeto se crea usando la palabra clave “new” seguida del nombre del constructor del objeto. El constructor es una función especial que se utiliza para inicializar un objeto.

## 15 SIGUIENTES PASOS

---

8. ¿Qué es una cadena en JavaScript?

- Una cadena es una secuencia de caracteres. En JavaScript, las cadenas se escriben entre comillas simples o dobles.

9. ¿Qué es una matriz en JavaScript?

- Una matriz es una colección ordenada de valores. En JavaScript, las matrices se escriben entre corchetes.

10. ¿Qué es un bucle en JavaScript?

- Un bucle es una estructura de control que se utiliza para ejecutar un bloque de código repetidamente. En JavaScript, los bucles se escriben usando la palabra clave “for” .

11. ¿Qué es una condición en JavaScript?

- Una condición es una expresión que se evalúa como verdadera o falsa. En JavaScript, las condiciones se escriben usando la palabra clave “if” .

12. ¿Qué es una expresión regular en JavaScript?

- Una expresión regular es un patrón de caracteres que se utiliza para buscar y reemplazar cadenas de texto. En JavaScript, las expresiones regulares se escriben entre barras.

13. ¿Qué es una clase en JavaScript?

- Una clase es una plantilla para crear objetos. En JavaScript, las clases se definen usando la palabra clave “class” .

14. ¿Qué es una excepción en JavaScript?

- Una excepción es un error que se produce durante la ejecución de un programa. En JavaScript, las excepciones se manejan usando la palabra clave “try”

.

## 15 SIGUIENTES PASOS

---

15. ¿Qué es una etiqueta en JavaScript?

- Una etiqueta es una palabra clave que se utiliza para marcar una sección de código. En JavaScript, las etiquetas se escriben entre corchetes.

16. ¿Qué es una propiedad en JavaScript?

- Una propiedad es una característica de un objeto. En JavaScript, las propiedades se definen usando la palabra clave “this” .

17. ¿Qué es un método en JavaScript?

- Un método es una función asociada a un objeto. En JavaScript, los métodos se definen usando la palabra clave “this” .

18. ¿Qué es una sentencia en JavaScript?

- Una sentencia es una instrucción que se ejecuta cuando se alcanza un punto específico en un programa. En JavaScript, las sentencias se escriben usando la palabra clave “return” .

19. ¿Qué es una declaración en JavaScript?

- Una declaración es una instrucción que se ejecuta cuando se alcanza un punto específico en un programa. En JavaScript, las declaraciones se escriben usando la palabra clave “var” .

20. ¿Qué es una expresión en JavaScript?

- Una expresión es una combinación de valores, variables y operadores que se evalúa como un valor. En JavaScript, las expresiones se escriben usando la palabra clave “return” .

21. ¿Qué es un operador en JavaScript?

- Un operador es un símbolo que se utiliza para realizar una operación matemática o lógica. En JavaScript, los operadores se escriben usando la palabra clave “operator” .

## 15 SIGUIENTES PASOS

---

22. ¿Qué es una sentencia de control en JavaScript?

- Una sentencia de control es una instrucción que se utiliza para controlar el flujo de un programa. En JavaScript, las sentencias de control se escriben usando la palabra clave “switch” .

23. ¿Qué es una sentencia de iteración en JavaScript?

- Una sentencia de iteración es una instrucción que se utiliza para ejecutar un bloque de código repetidamente. En JavaScript, las sentencias de iteración se escriben usando la palabra clave “for” .

24. ¿Qué es una sentencia de salto en JavaScript?

- Una sentencia de salto es una instrucción que se utiliza para saltar a una parte específica de un programa. En JavaScript, las sentencias de salto se escriben usando la palabra clave “break” .

25. ¿Qué es una sentencia de etiqueta en JavaScript?

- Una sentencia de etiqueta es una instrucción que se utiliza para marcar una sección de código. En JavaScript, las sentencias de etiqueta se escriben usando la palabra clave “label” .

26. ¿Qué es una sentencia de continuación en JavaScript?

- Una sentencia de continuación es una instrucción que se utiliza para saltar a la siguiente iteración de un bucle. En JavaScript, las sentencias de continuación se escriben usando la palabra clave “continue” .

27. ¿Qué es una sentencia de asignación en JavaScript?

- Una sentencia de asignación es una instrucción que se utiliza para asignar un valor a una variable. En JavaScript, las sentencias de asignación se escriben usando la palabra clave “=” .

28. ¿Qué es una sentencia de declaración en JavaScript?

## 15 SIGUIENTES PASOS

---

- Una sentencia de declaración es una instrucción que se utiliza para declarar una variable. En JavaScript, las sentencias de declaración se escriben usando la palabra clave “var” .

29. ¿Qué es una sentencia de función en JavaScript?

- Una sentencia de función es una instrucción que se utiliza para definir una función. En JavaScript, las sentencias de función se escriben usando la palabra clave “function” .

30. ¿Qué es una sentencia de bloque en JavaScript?

- Una sentencia de bloque es una instrucción que se utiliza para agrupar un conjunto de sentencias. En JavaScript, las sentencias de bloque se escriben usando la palabra clave “block” .

31. ¿Qué es una sentencia de etiqueta en JavaScript?

- Una sentencia de etiqueta es una instrucción que se utiliza para marcar una sección de código. En JavaScript, las sentencias de etiqueta se escriben usando la palabra clave “label” .

32. ¿Qué es una sentencia de excepción en JavaScript?

- Una sentencia de excepción es una instrucción que se utiliza para manejar errores. En JavaScript, las sentencias de excepción se escriben usando la palabra clave “try” .

33. ¿Qué es una sentencia de clase en JavaScript?

- Una sentencia de clase es una instrucción que se utiliza para definir una clase. En JavaScript, las sentencias de clase se escriben usando la palabra clave “class” .

34. ¿Qué es una sentencia de objeto en JavaScript?

## 15 SIGUIENTES PASOS

---

- Una sentencia de objeto es una instrucción que se utiliza para crear un objeto. En JavaScript, las sentencias de objeto se escriben usando la palabra clave “new” .

35. ¿Qué es una sentencia de importación en JavaScript?

- Una sentencia de importación es una instrucción que se utiliza para importar código de otro archivo. En JavaScript, las sentencias de importación se escriben usando la palabra clave “import” .

36. ¿Qué es una sentencia de exportación en JavaScript?

- Una sentencia de exportación es una instrucción que se utiliza para exportar código a otro archivo. En JavaScript, las sentencias de exportación se escriben usando la palabra clave “export” .

37. ¿Qué es una sentencia de declaración de módulo en JavaScript?

- Una sentencia de declaración de módulo es una instrucción que se utiliza para definir un módulo. En JavaScript, las sentencias de declaración de módulo se escriben usando la palabra clave “module” .

38. ¿Qué es una sentencia de declaración de paquete en JavaScript?

- Una sentencia de declaración de paquete es una instrucción que se utiliza para definir un paquete. En JavaScript, las sentencias de declaración de paquete se escriben usando la palabra clave “package” .

39. ¿Qué es una sentencia de declaración de interfaz en JavaScript?

- Una sentencia de declaración de interfaz es una instrucción que se utiliza para definir una interfaz. En JavaScript, las sentencias de declaración de interfaz se escriben usando la palabra clave “interface” .

40. ¿Qué es una sentencia de declaración de enumeración en JavaScript?

## 15 SIGUIENTES PASOS

---

- Una sentencia de declaración de enumeración es una instrucción que se utiliza para definir una enumeración. En JavaScript, las sentencias de declaración de enumeración se escriben usando la palabra clave “enum” .

41. ¿Qué es una sentencia de declaración de tipo en JavaScript?

- Una sentencia de declaración de tipo es una instrucción que se utiliza para definir un tipo. En JavaScript, las sentencias de declaración de tipo se escriben usando la palabra clave “type” .

42. ¿Qué es una sentencia de declaración de constante en JavaScript?

- Una sentencia de declaración de constante es una instrucción que se utiliza para definir una constante. En JavaScript, las sentencias de declaración de constante se escriben usando la palabra clave “const” .

43. ¿Qué es una sentencia de declaración de variable en JavaScript?

- Una sentencia de declaración de variable es una instrucción que se utiliza para definir una variable. En JavaScript, las sentencias de declaración de variable se escriben usando la palabra clave “var” .

44. ¿Qué es una sentencia de declaración de función en JavaScript?

- Una sentencia de declaración de función es una instrucción que se utiliza para definir una función. En JavaScript, las sentencias de declaración de función se escriben usando la palabra clave “function” .

45. ¿Qué es una sentencia de declaración de clase en JavaScript?

- Una sentencia de declaración de clase es una instrucción que se utiliza para definir una clase. En JavaScript, las sentencias de declaración de clase se escriben usando la palabra clave “class” .

46. ¿Qué es una sentencia de declaración de objeto en JavaScript?

## 15 SIGUIENTES PASOS

---

- Una sentencia de declaración de objeto es una instrucción que se utiliza para definir un objeto. En JavaScript, las sentencias de declaración de objeto se escriben usando la palabra clave “object” .

47. ¿Qué es una sentencia de declaración de método en JavaScript?

- Una sentencia de declaración de método es una instrucción que se utiliza para definir un método. En JavaScript, las sentencias de declaración de método se escriben usando la palabra clave “method” .

48. ¿Qué es una sentencia de declaración de propiedad en JavaScript?

- Una sentencia de declaración de propiedad es una instrucción que se utiliza para definir una propiedad. En JavaScript, las sentencias de declaración de propiedad se escriben usando la palabra clave “property” .

49. ¿Qué es una sentencia de declaración de evento en JavaScript?

- Una sentencia de declaración de evento es una instrucción que se utiliza para definir un evento. En JavaScript, las sentencias de declaración de evento se escriben usando la palabra clave “event” .

50. ¿Qué es una sentencia de declaración de excepción en JavaScript?

- Una sentencia de declaración de excepción es una instrucción que se utiliza para definir una excepción. En JavaScript, las sentencias de declaración de excepción se escriben usando la palabra clave “exception” .

51. ¿Qué es una sentencia de declaración de interfaz en JavaScript?

- Una sentencia de declaración de interfaz es una instrucción que se utiliza para definir una interfaz. En JavaScript, las sentencias de declaración de interfaz se escriben usando la palabra clave “interface” .