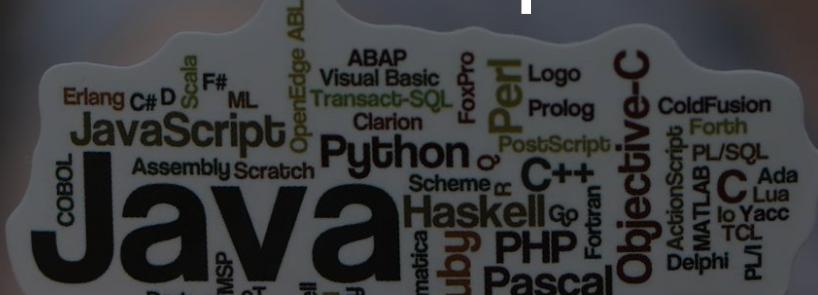


Introducción a JavaScript

Nivelación



Introducción a JavaScript



En la compleja y dinámica esfera del desarrollo web, entender las bases fundamentales es un primer paso esencial. JavaScript, como uno de los pilares de la **programación en la web**, merece una atención particular por parte de quienes desean iniciar su camino en este campo.

En este fastbook, nos centraremos en los **conceptos básicos y elementales de JavaScript**, proporcionando una introducción clara y accesible para principiantes. Exploramos la estructura y sintaxis que hacen de JavaScript una herramienta tan versátil y popular, sin adentrarnos en las áreas más avanzadas y complejas. Desde variables y tipos de datos hasta funciones simples y control de flujo, este fastbook ofrecerá una guía práctica y cercana para aquellos que buscan establecer una base sólida en programación web.

En definitiva, este fastbook será tu aliado perfecto en este emocionante viaje. La comprensión de estos conceptos iniciales te preparará para futuras exploraciones y crecimiento en el siempre cambiante mundo de la tecnología web.

Autor: Robert Da Corte

 **Configurando nuestro entorno**

 **Introducción a JavaScript**

 **Sentencias y expresiones**

 **Variables y tipos de datos**

 **Operadores**

 **Estructuras de control**

 **Funciones**

 **Objetos**

 **Arrays**

 **Document model object (DOM)**

 **Eventos**

 **Qué hemos aprendido**

Configurando nuestro entorno



Qualentum Lab

Antes de comenzar necesitamos instalar y configurar un par de aplicaciones para poder desarrollar en JavaScript, uno es un editor de texto plano llamado **Visual Studio Code** y el otro es un programa llamado **Node.js**

Instalación de Visual Studio Code (VSCode)

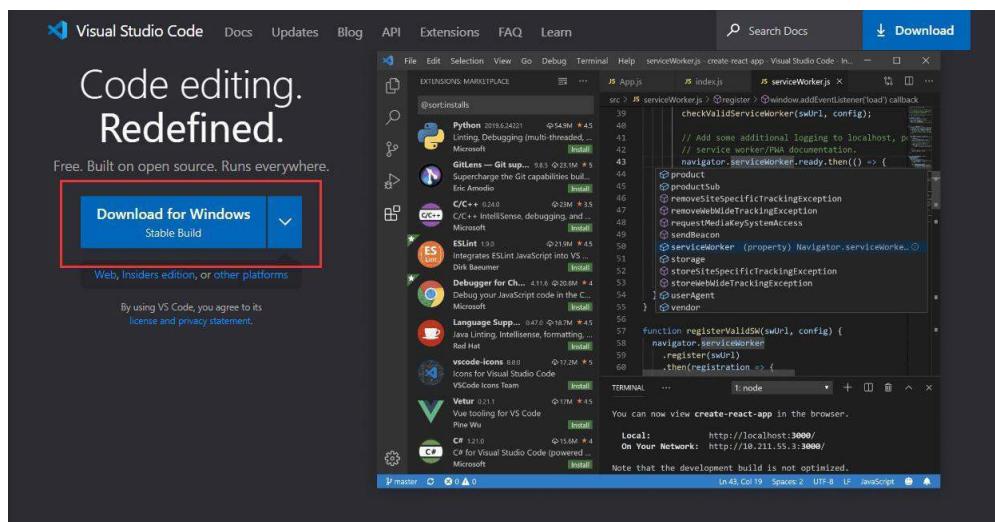
- **Para Windows**

1

Ve al sitio oficial de VSCode en Visual Studio Code.

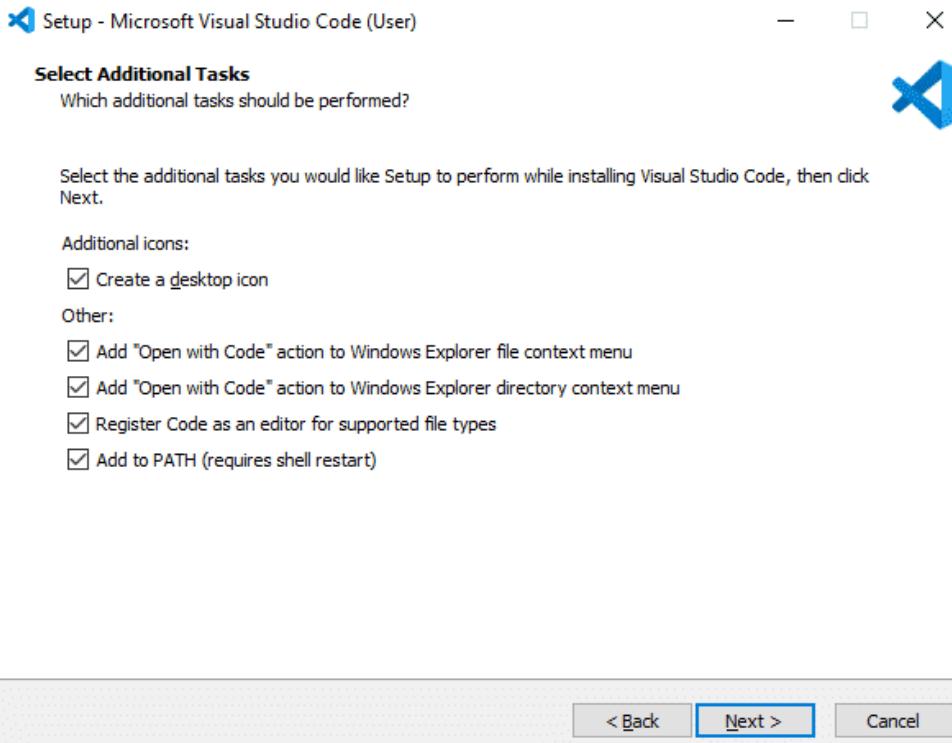
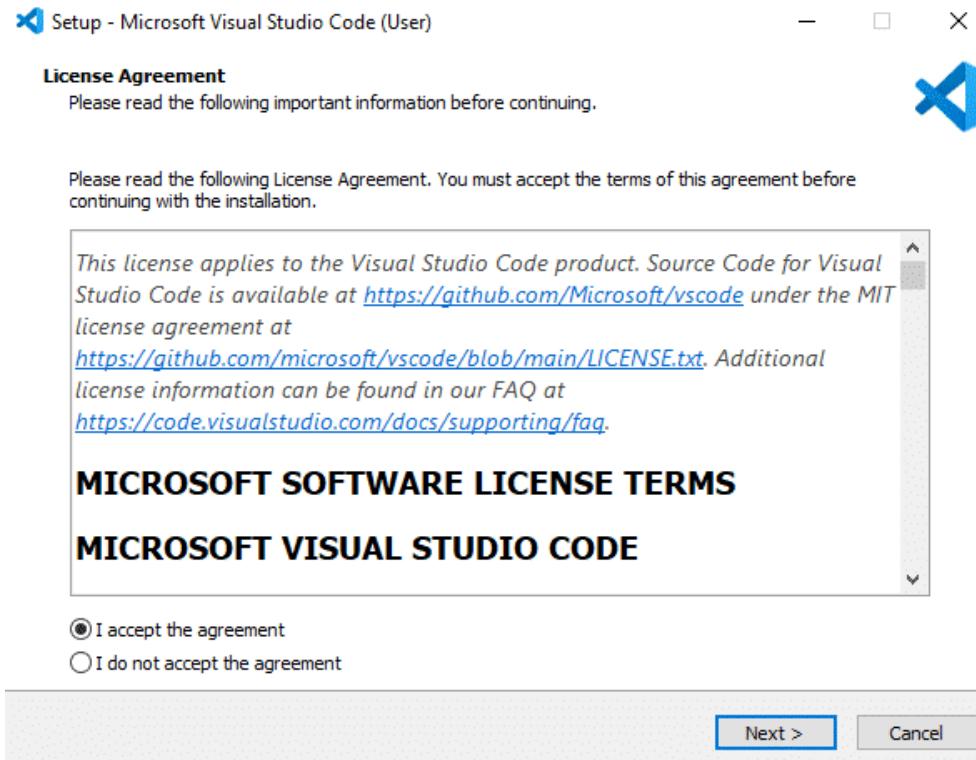
2

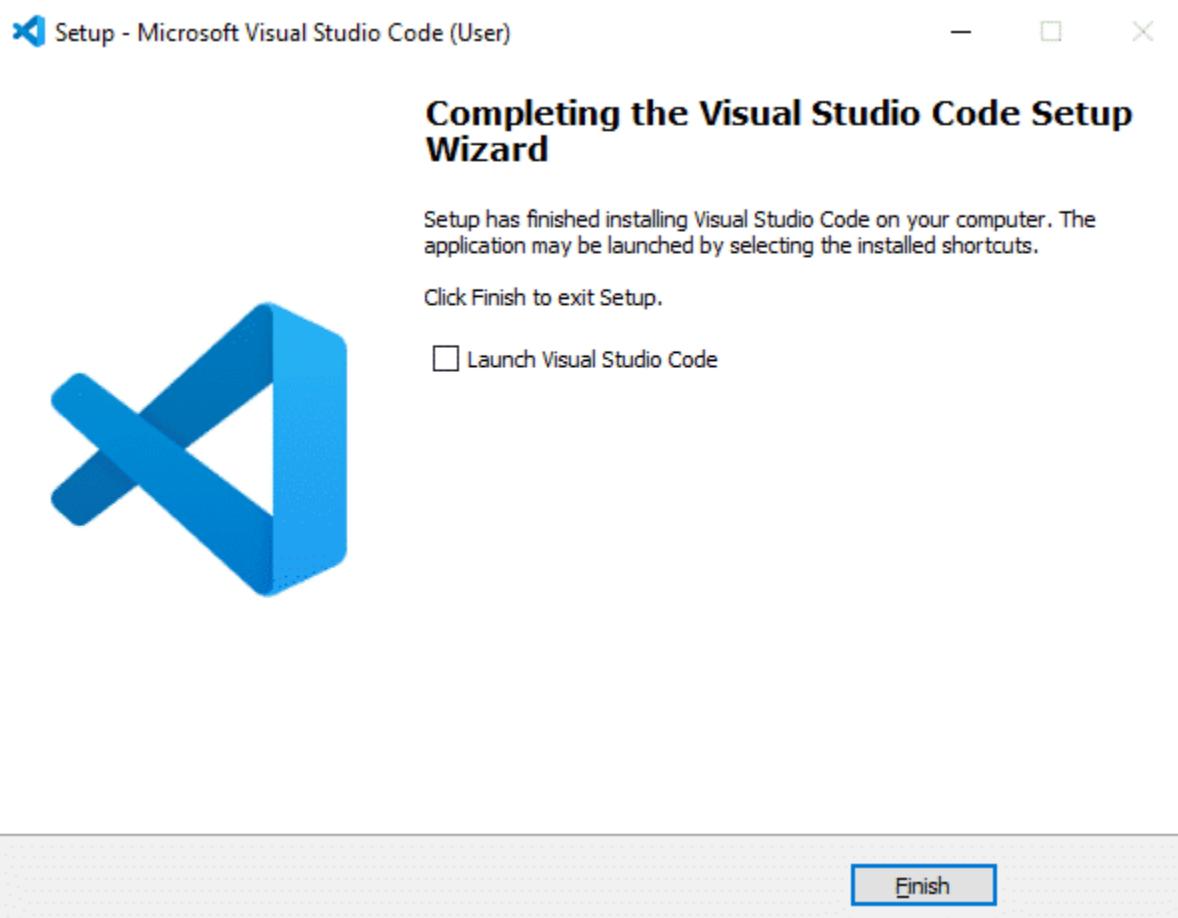
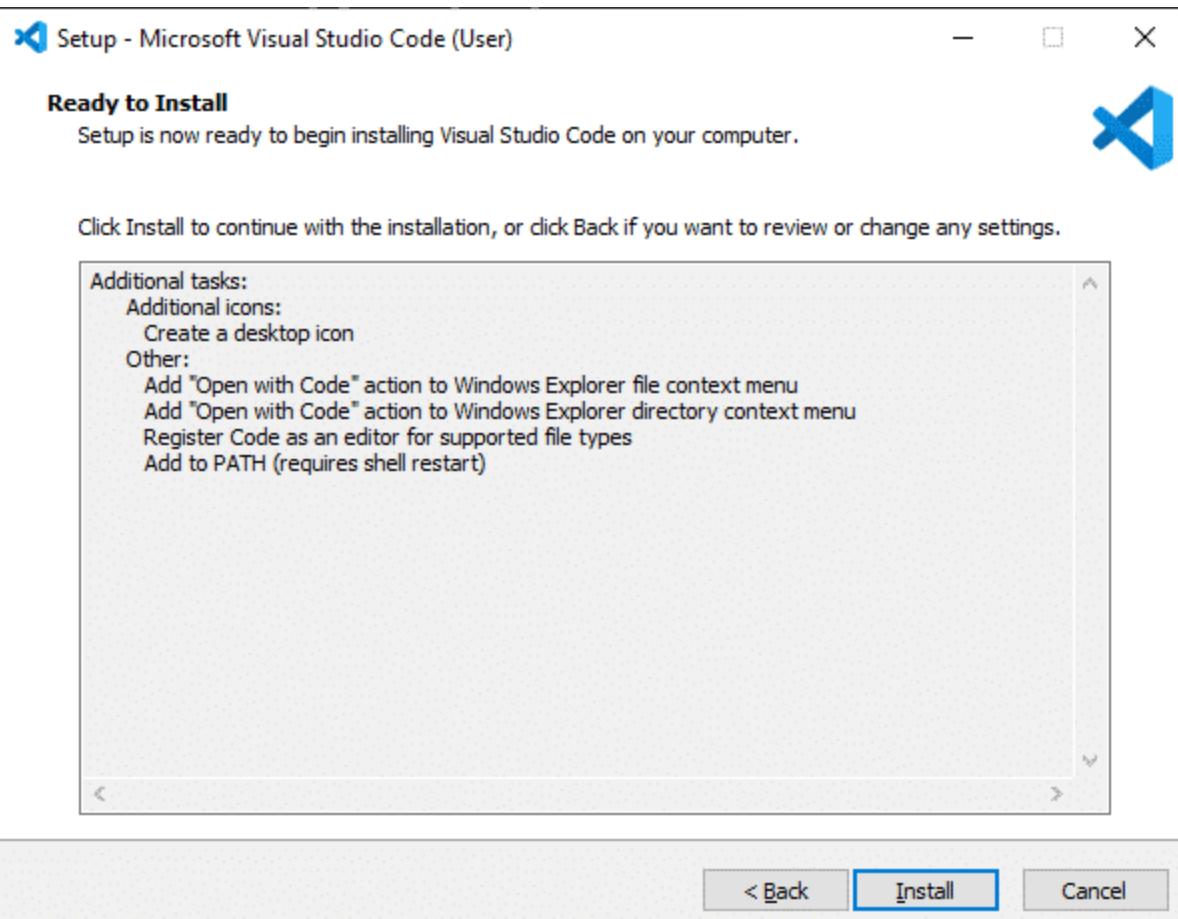
Haz clic en el botón 'Download for Windows'.



3

Una vez descargado, abre el archivo .exe y sigue las instrucciones del instalador.





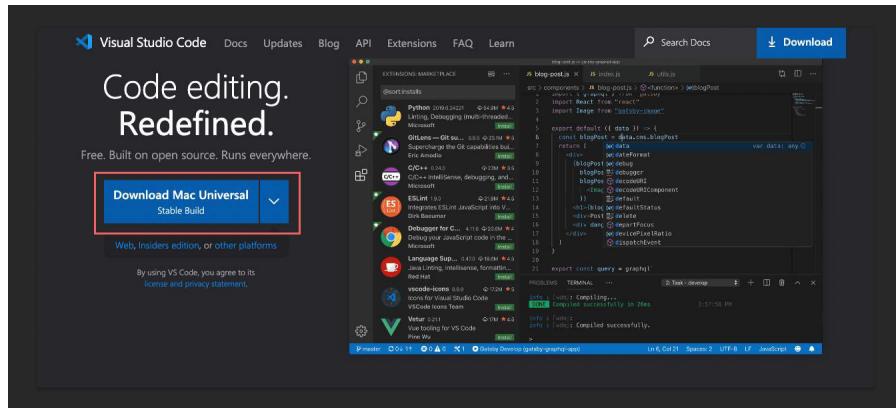
- **Para macOS**

1

Ve al sitio oficial de VSCode en Visual Studio Code.

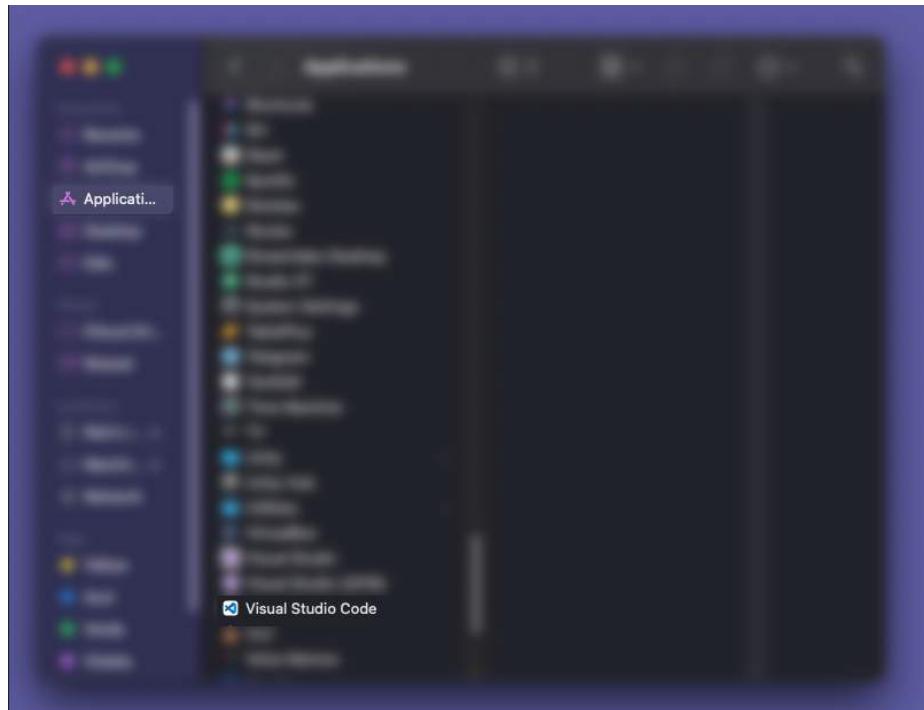
2

Haz clic en el botón 'Download Mac Universal'.



3

Una vez descargado, abre el archivo .zip y arrastra la aplicación a tu carpeta de Aplicaciones.



Instalación de Node.js

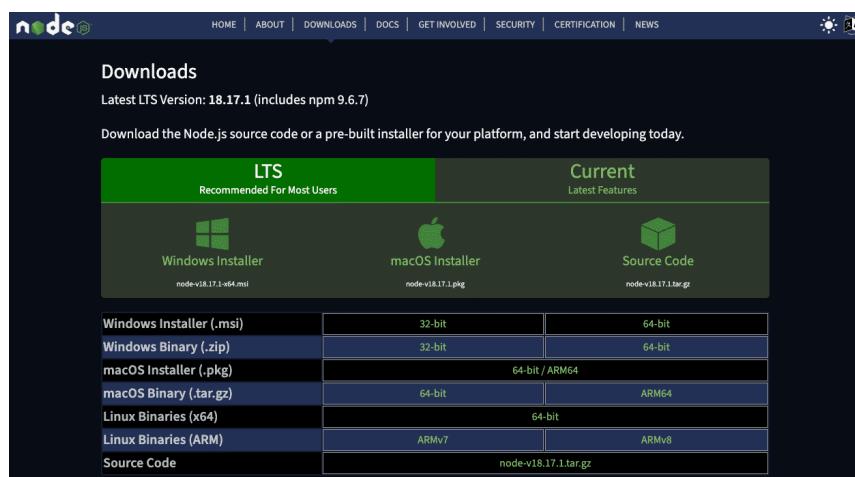
- **Para Windows**

1

Ve al sitio oficial de Node.js

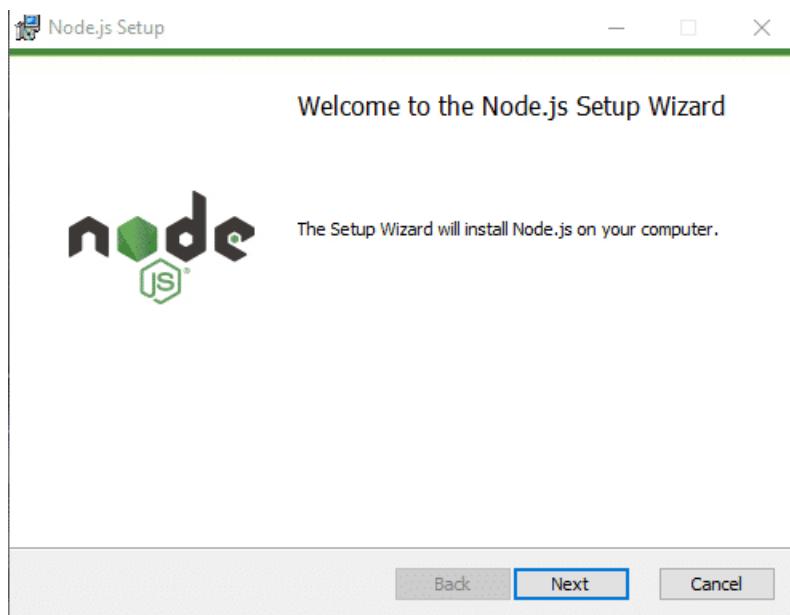
2

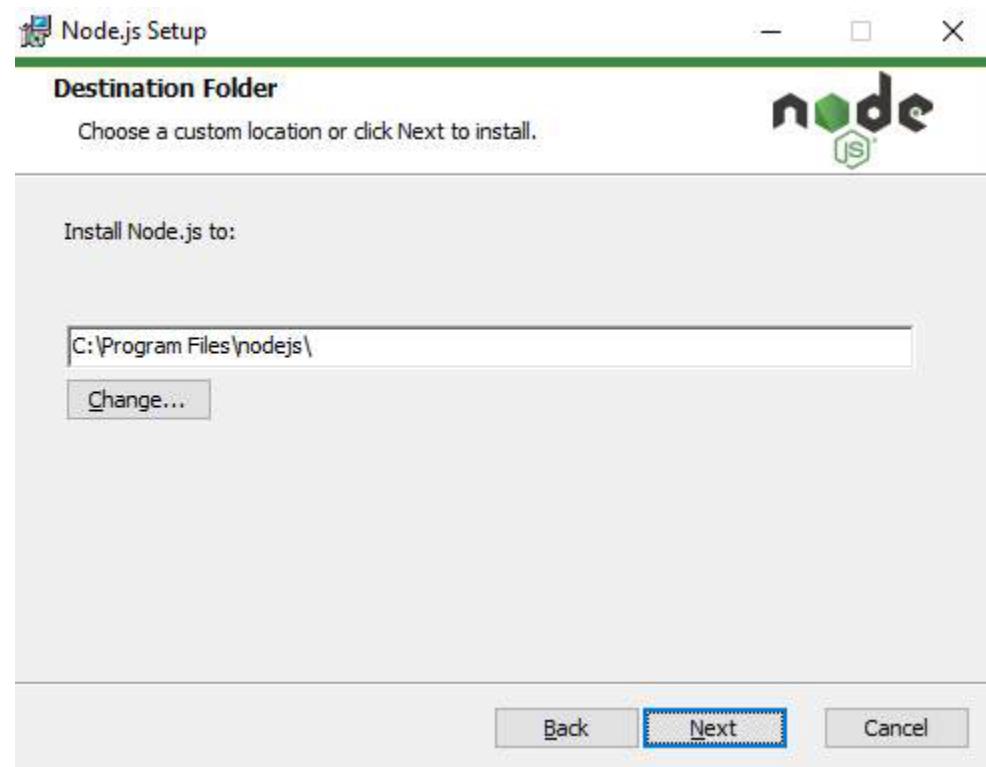
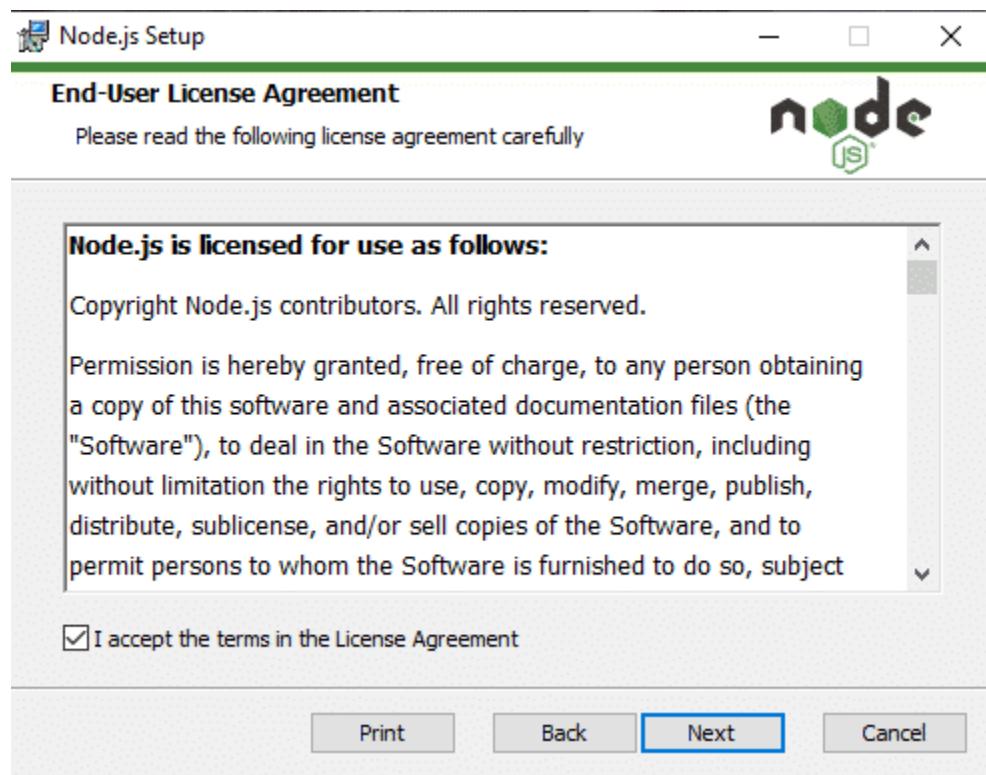
Descarga la versión **LTS (Long Term Support)** haciendo clic en el enlace de Windows Installer msi.

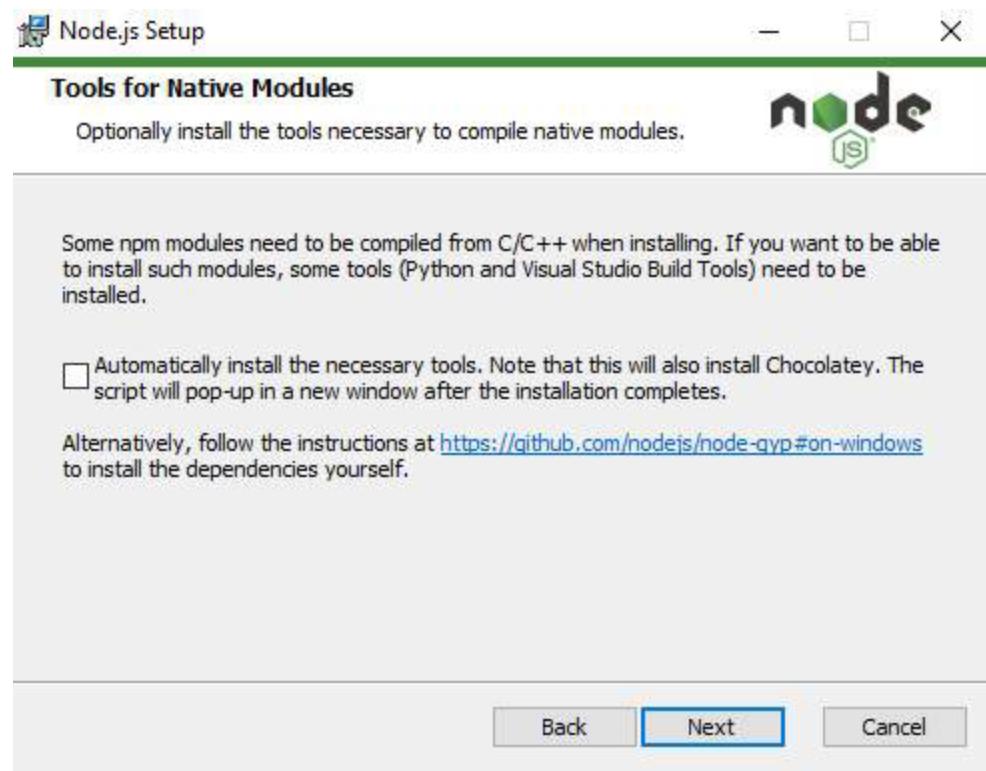
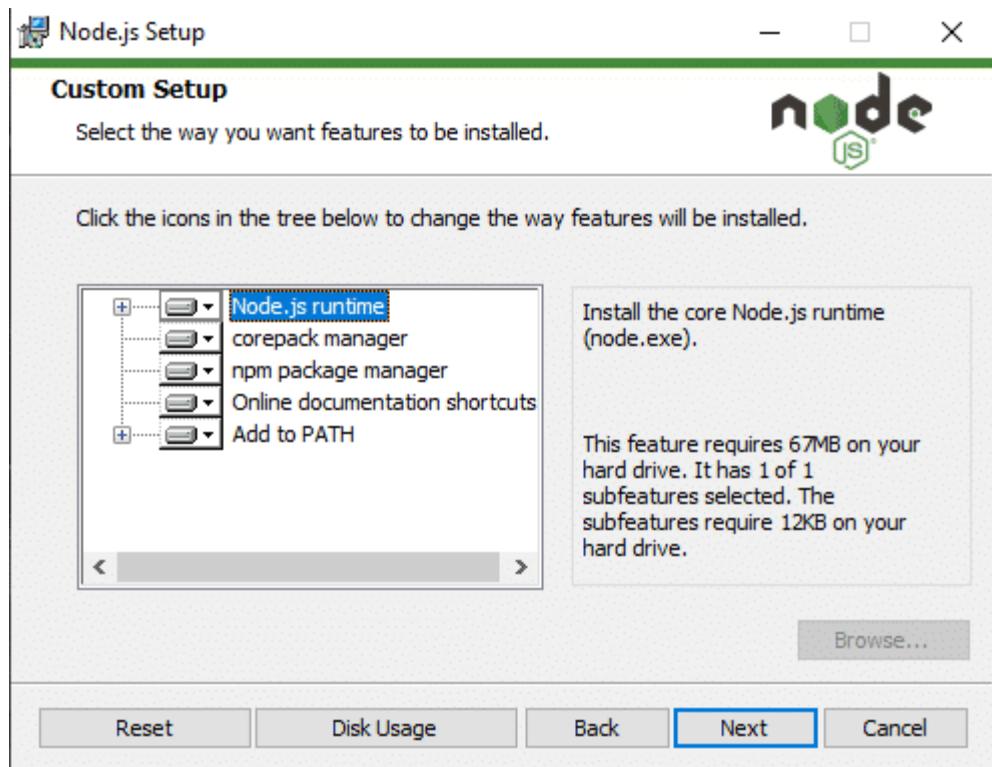


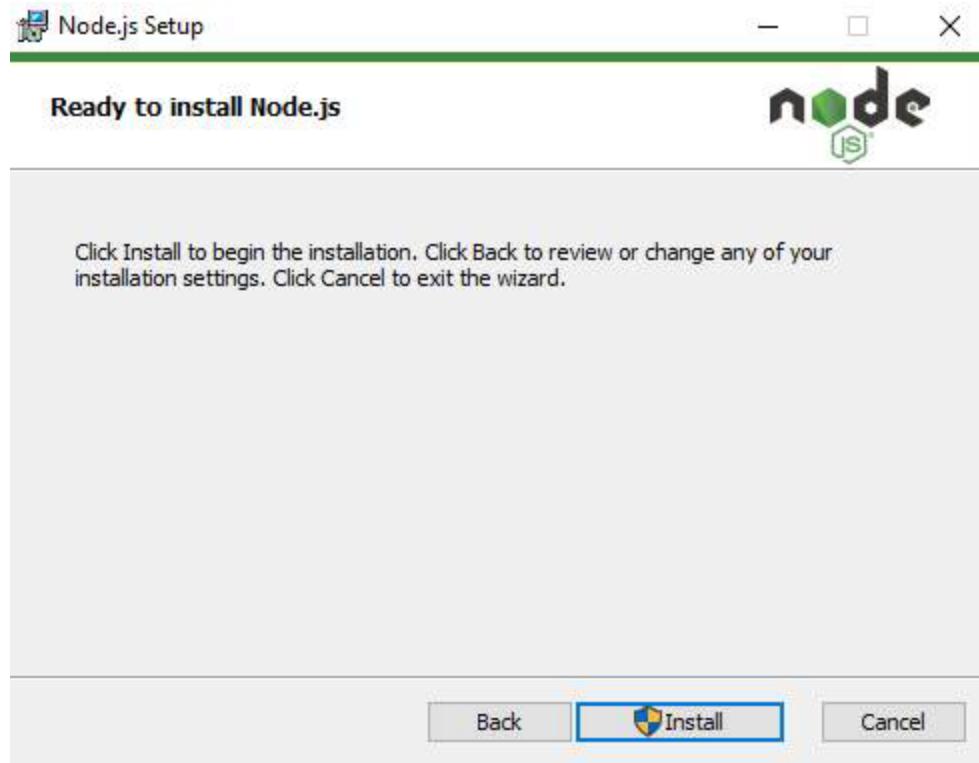
3

Ejecuta el archivo .msi descargado y sigue las instrucciones del instalador.









- **Para macOS**

1

Ve al sitio oficial de Node.js

A screenshot of the Node.js official website at nodejs.org/en. The page has a dark header with the Node.js logo and navigation links for HOME, ABOUT, DOWNLOADS, DOCS, GET INVOLVED, SECURITY, CERTIFICATION, and NEWS. Below the header, a banner states "Node.js® is an open-source, cross-platform JavaScript runtime environment." A section titled "Download for macOS" offers two options: "18.17.1 LTS" (Recommended For Most Users) and "20.5.1 Current" (Latest Features). At the bottom, links for "Other Downloads", "Changelog", and "API Docs" are provided for both versions.

2

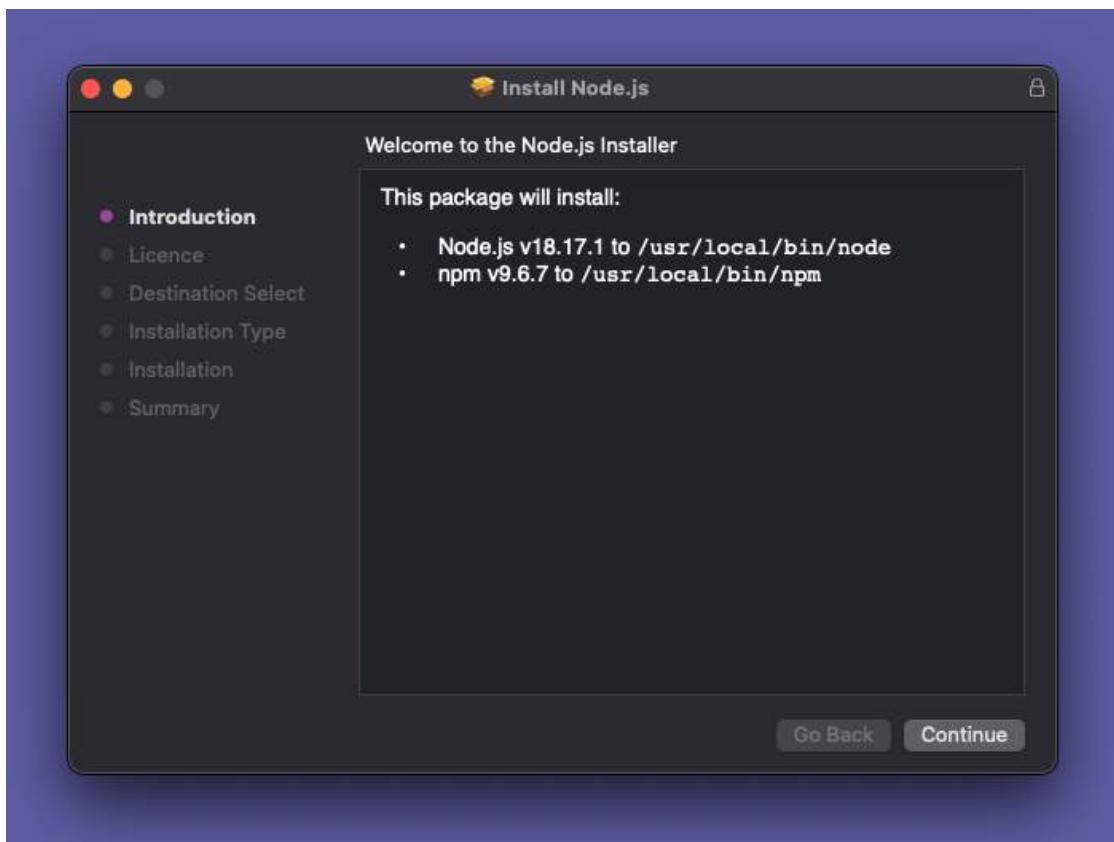
Descarga la versión LTS haciendo clic en el enlace de pkg de macOS.

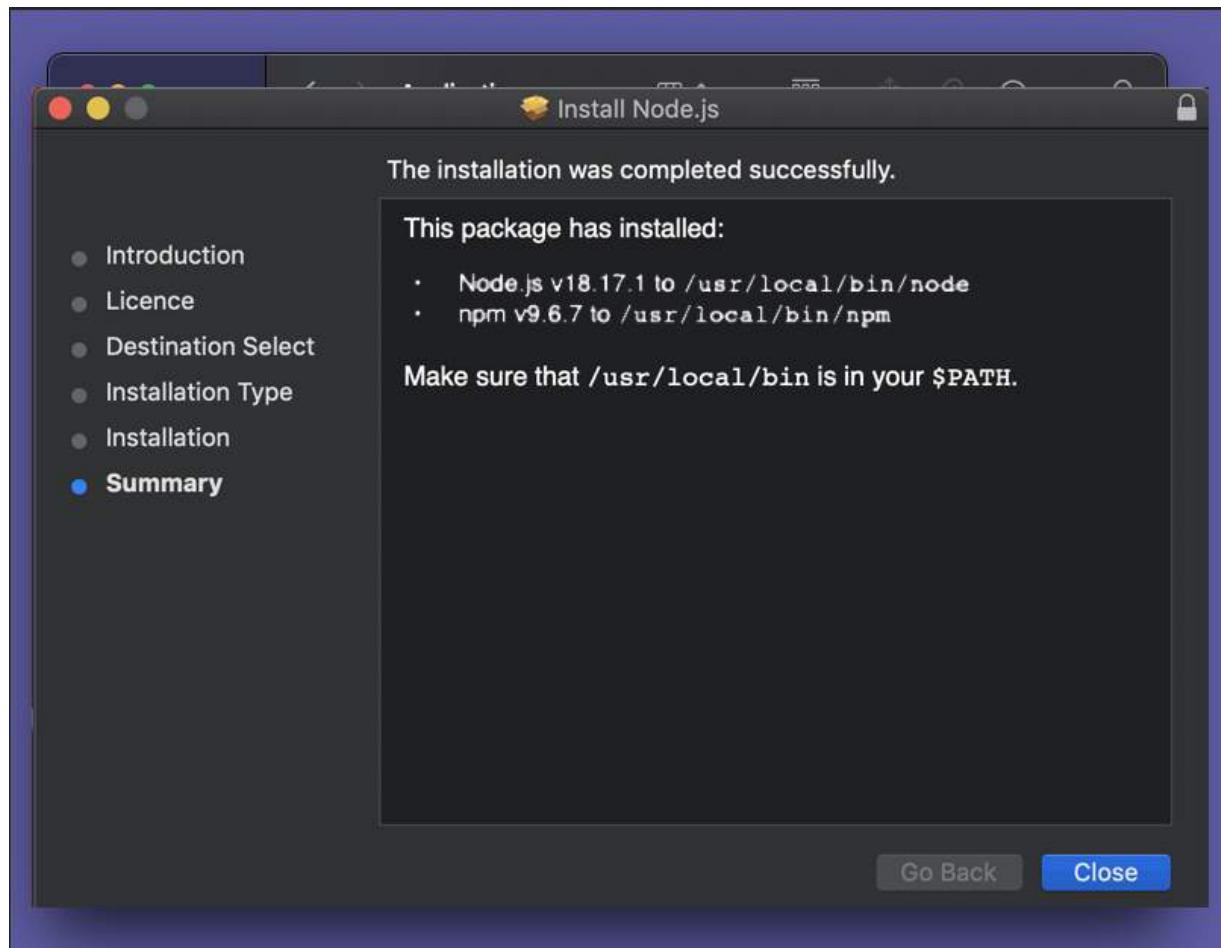
The screenshot shows the Node.js Downloads page. At the top, it says "Latest LTS Version: 18.17.1 (includes npm 9.6.7)". Below that, a message encourages users to "Download the Node.js source code or a pre-built installer for your platform, and start developing today." The page is divided into two main sections: "LTS" (Recommended For Most Users) and "Current" (Latest Features). Under "LTS", there are links for "Windows Installer (.msi)" and "macOS Installer (.pkg)". Under "Current", there are links for "Source Code" and "macOS Binary (.tar.gz)". A large table below lists all available download options, categorized by platform (Windows, macOS, Linux) and bitness (32-bit, 64-bit, ARM64, ARMv7, ARMv8). The "macOS Binary (.tar.gz)" link is highlighted.

	32-bit	64-bit
Windows Installer (.msi)		
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)		64-bit / ARM64
macOS Binary (.tar.gz)	64-bit	ARM64
Linux Binaries (x64)		64-bit
Linux Binaries (ARM)	ARMv7	ARMv8
Source Code		node-v18.17.1.tar.gz

3

Ejecuta el archivo .pkg y sigue las instrucciones del instalador.





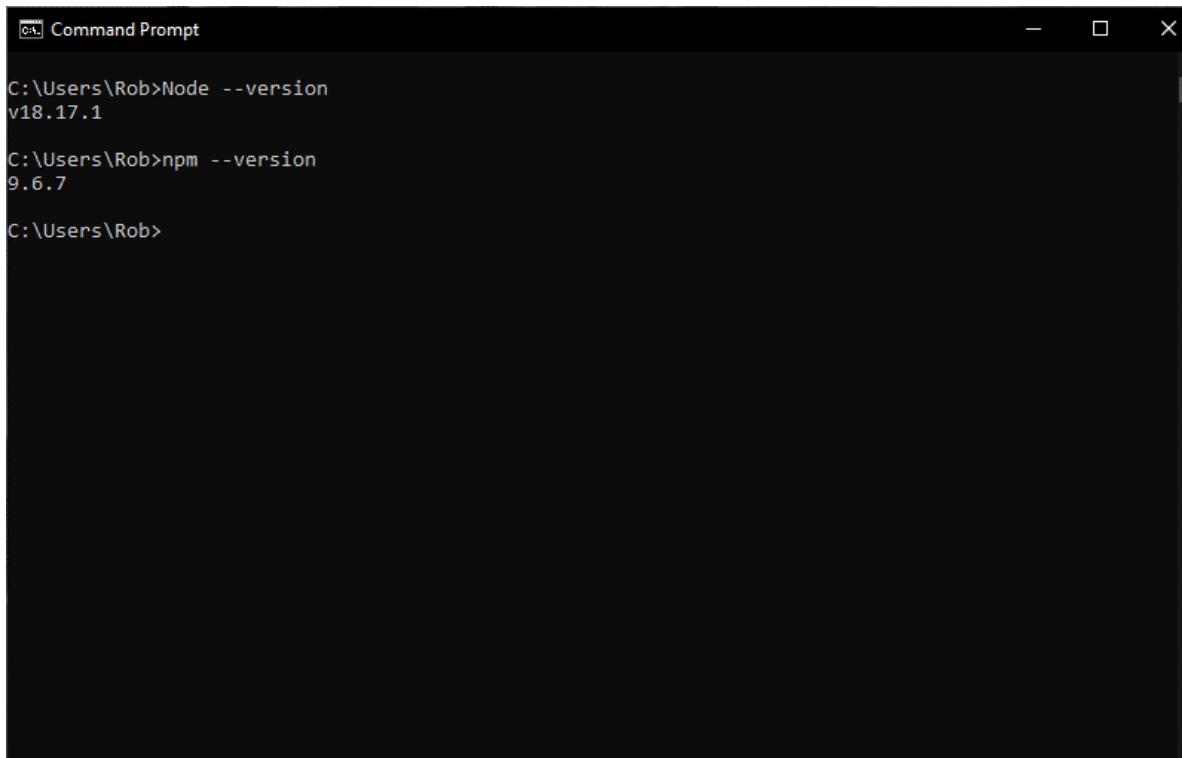
Verificación

Una vez instalado Node.js y npm (gestor de paquetes de Node), puedes verificar si la instalación fue exitosa. Abre una terminal o línea de comandos y escribe:

En Windows:

Node --version: esto mostrará la versión de node que tienes instalado en tu ordenador. Asegúrate de que sean dos dash (--) seguida de la palabra **version** sin espacios ni acentos.

npm --version: esto mostrará la versión de npm que tienes instalado en tu ordenador. Asegúrate de que sean dos *dash* (--) seguida de la palabra **version** sin espacios ni acentos.



```
Command Prompt

C:\Users\Rob>Node --version
v18.17.1

C:\Users\Rob>npm --version
9.6.7

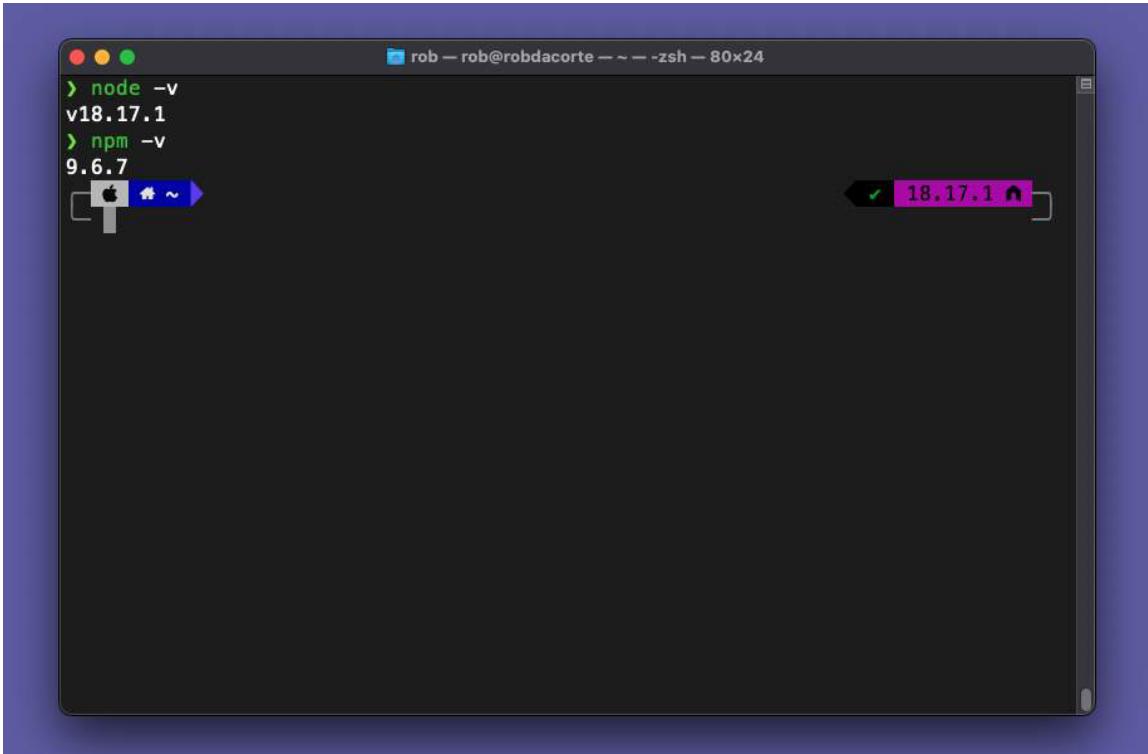
C:\Users\Rob>
```

Esto te mostrará las versiones de Node.js y npm, respectivamente.

En macOS:

node -v: esto mostrará la versión de node que tienes instalado en tu ordenador.

npm -v: esto mostrará la versión de npm que tienes instalado en tu ordenador.



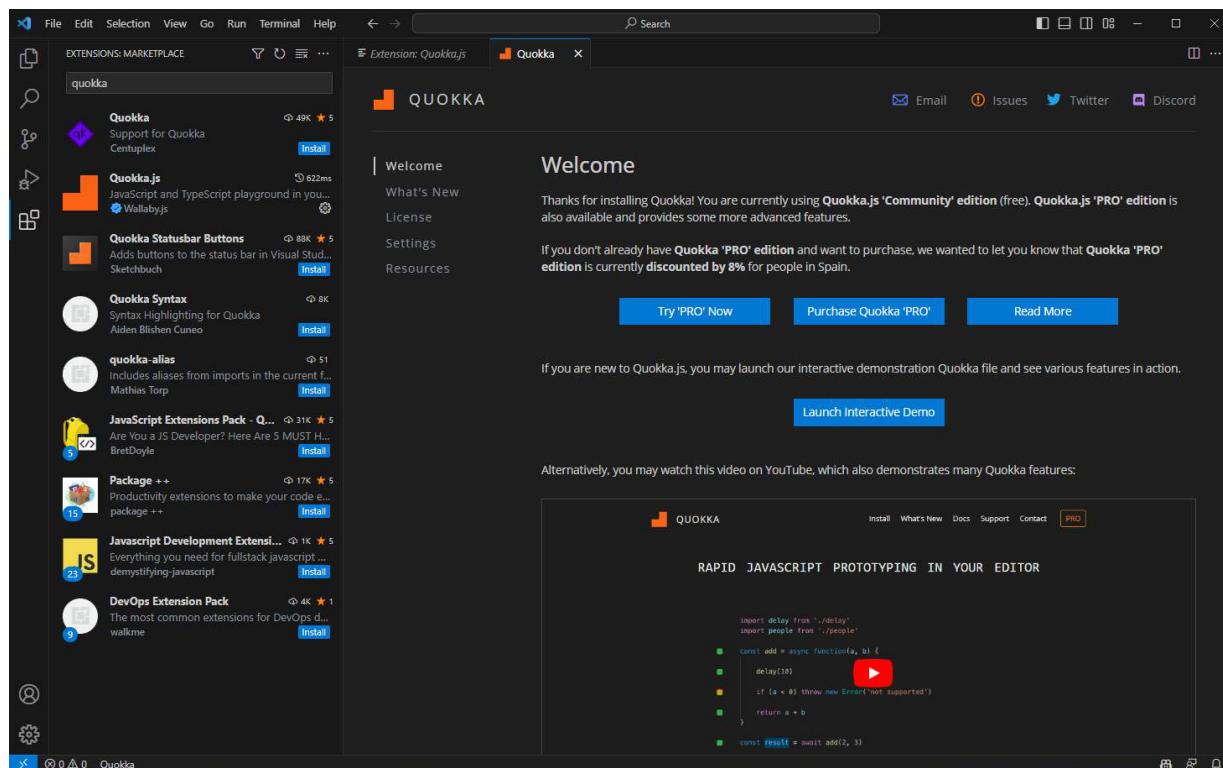
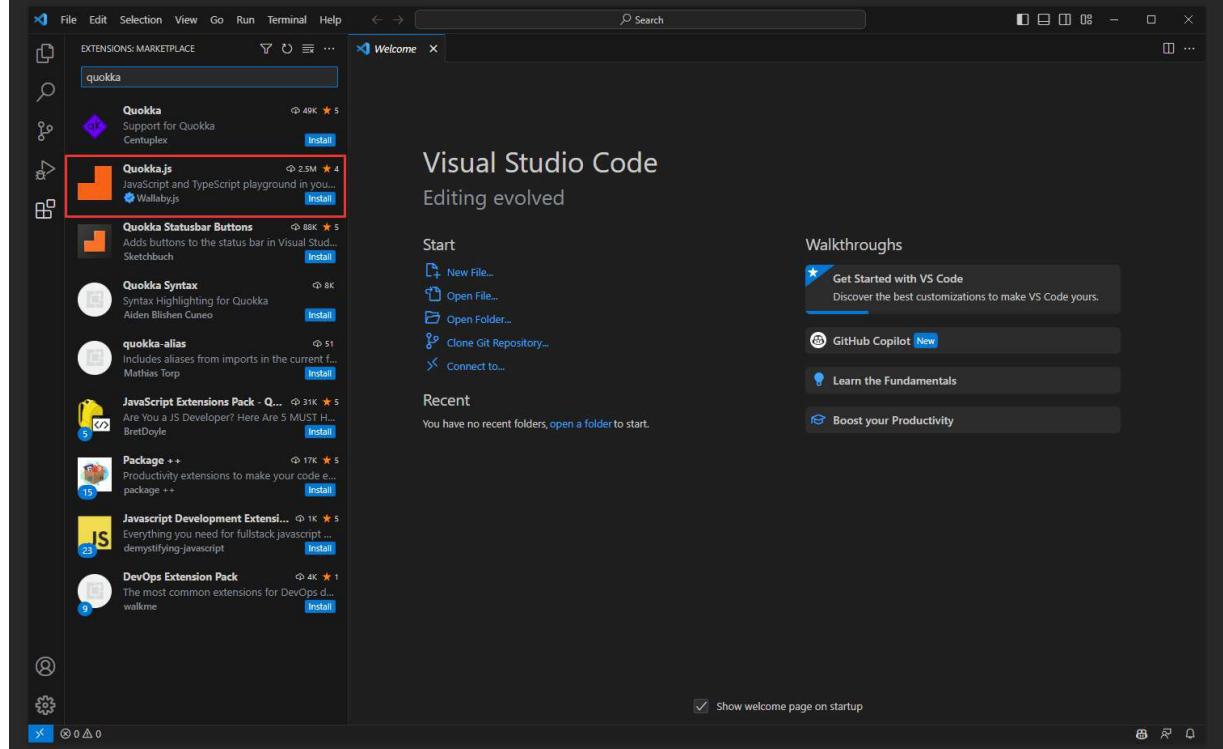
A screenshot of a macOS terminal window titled "rob — rob@robdacorte ~ zsh 80x24". The window shows the command "node -v" followed by "v18.17.1" and the command "npm -v" followed by "9.6.7". The status bar at the bottom right of the terminal window displays "18.17.1".

Esto te mostrará las versiones de Node.js y npm, respectivamente.

Extensiones

Los ejemplos que se muestran en este fastbook se realizaron usando una extensión para vscode, la llamada **Quokka**. Quokka permite usar vscode como un playground de JavaScript lo que nos servirá para ver cómo se ejecuta nuestro código simulando una consola de navegador.

Para instalarla solo tenemos que buscar en la sección de extensiones.

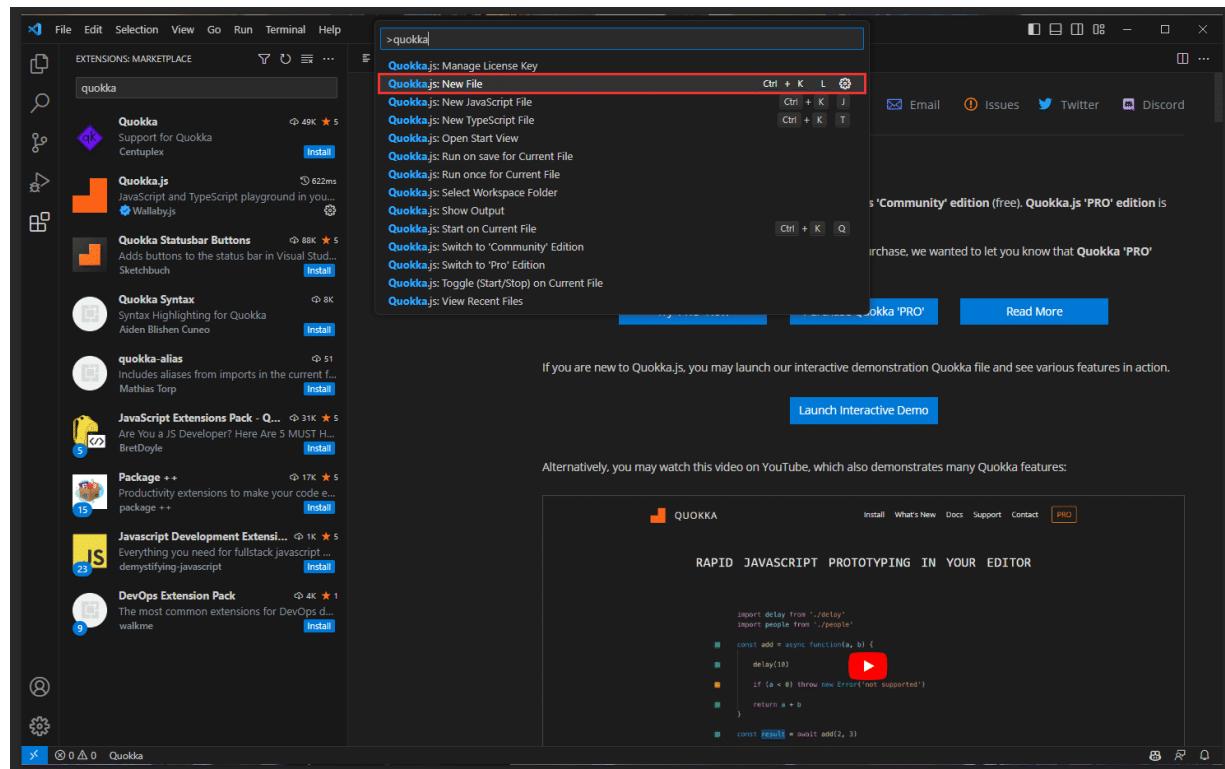


Una vez instalada podremos crear un fichero Quokka usando el atajo de teclado:

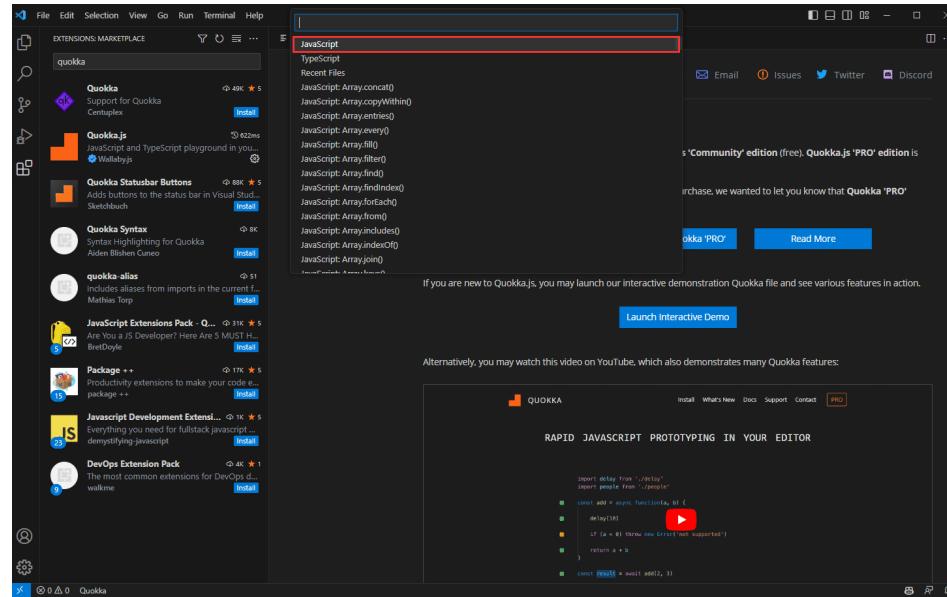
windows : CTRL + SHIFT + P

mac : CMD + SHIFT + P

Esto mostrará un *prompt*, en el que escribiremos *quokka*, y seleccionaremos **Quokka: New File**.



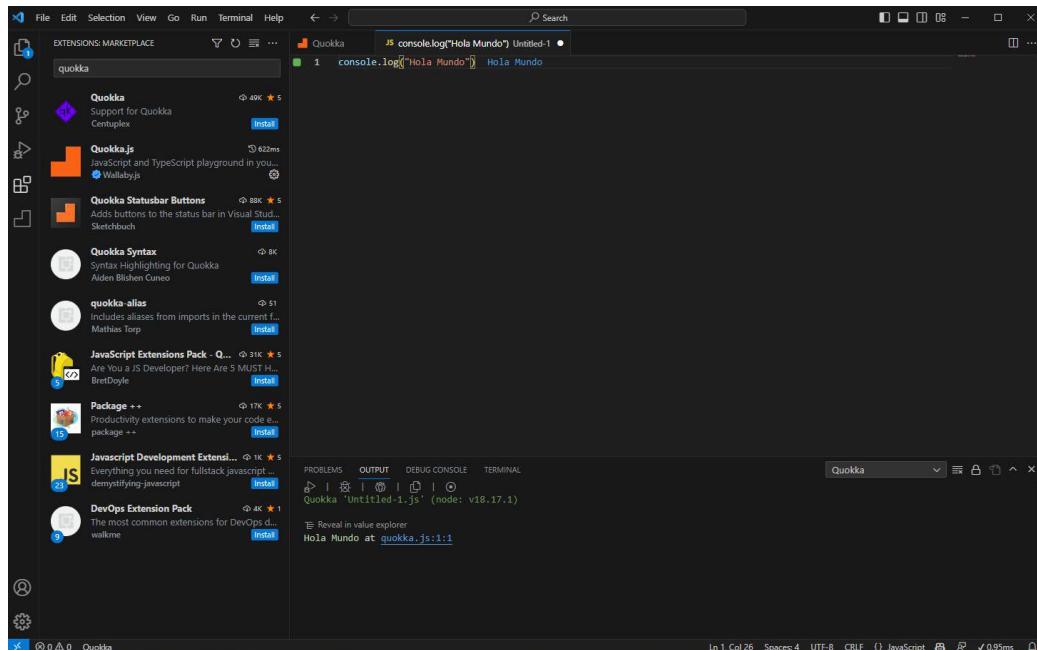
Abrirá un segundo prompt para seleccionar el lenguaje en el que querremos nuestro playground y seleccionaremos JavaScript en este caso.



Para comprobar que todo ha ido bien, solo tienes que añadir tu primera sentencia de JavaScript

```
console.log("Hola Mundo")
```

Y deberás tener un resultado como este que muestra la imagen.



Introducción a JavaScript



1

Breve historia y evolución del lenguaje

JavaScript fue creado por Brendan Eich, quien trabajaba para Netscape Communications Corporation a mediados de la década de 1990. En septiembre de 1995, el lenguaje se lanzó con el nombre de '**LiveScript**' como parte del navegador Netscape Navigator 2.0. Poco después, Netscape se asoció con Sun Microsystems y, debido a la popularidad del lenguaje de programación Java en ese momento, decidieron renombrarlo a '**JavaScript**' para capitalizar la tendencia.

Para estandarizar JavaScript y permitir que otros navegadores lo implementasen, Netscape presentó el lenguaje ante Ecma International (European Computer Manufacturers Association) en 1996. El resultado fue el estándar ECMA-262, publicado en junio de 1997, que definió la especificación del lenguaje **ECMAScript**. A partir de entonces, JavaScript es una implementación del estándar ECMAScript.

En los años siguientes, Microsoft lanzó su propio lenguaje de programación llamado **JScript**, que era prácticamente idéntico a JavaScript, pero con algunos cambios para evitar problemas legales con Netscape. Esto llevó a ciertas incompatibilidades entre los dos lenguajes y a una guerra de navegadores, donde cada uno trataba de implementar funcionalidades exclusivas.

En diciembre del 99, **ECMAScript 3 (ES3)** es lanzado como la tercera edición del estándar incorporando importantes mejoras y correcciones a JavaScript, siendo esta edición la que se mantendrá durante toda la década de los 2000.

Desde el 2009 hasta el día de hoy, JavaScript ha estado en constante evolución recibiendo en 2015 una de las actualizaciones más importantes en la historia de JavaScript, la conocida como **ECMAScript 2015 o ES6**. Introdujo nuevas características como las declaraciones de variables let y const, las funciones de flecha, clases, destructuring, y muchas otras mejoras que hicieron que el código fuera más conciso y fácil de mantener.

Es en esta versión en la que nos enfocaremos en este fastbook, ya que representa la forma como hoy en día se siguen desarrollando los proyectos en el mundo empresarial.

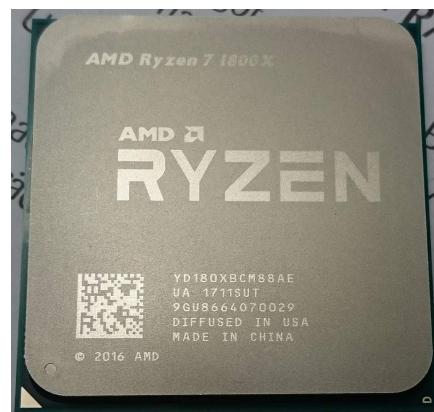
2

¿Qué es JavaScript?

JavaScript es un **lenguaje de programación de alto nivel, interpretado y multiparadigma**. Fue creado originalmente para interactuar con elementos dinámicos en páginas web en el browser Netscape y se ha convertido en un lenguaje esencial para el desarrollo web moderno.

Ahora bien, con lo definido antes, realmente no nos aclara todo, ya que nos responde con más terminología desconocida, tales como lenguaje de programación de alto nivel, interpretado y multiparadigma. A lo largo de este fastbook te encontrarás con mucho vocabulario técnico, no te preocupes porque verás que son solo *big words* y tecnicismos para términos que realmente son bastante simples. Por tanto, veamos lo primero, el lenguaje de programación de alto nivel, para poder entender a qué se refiere, así podremos entender un poco el funcionamiento de un ordenador sin profundizar demasiado.

La unidad central de procesamiento (Central Process Unit o **CPU**) es lo que podríamos llamar el cerebro de tu ordenador, internamente está compuesto por millones de componentes electrónicos llamados transistores y la forma en la que operan es dejando pasar o no electricidad a través de ellos, teniendo así, estados de encendido y apagado. Estos estados tienen sus equivalentes digitalmente siendo el número 1 su estado encendido y el número 0 su estado apagado. Estos dos dígitos conforman lo que se conoce como **sistema binario** y, a nivel informático, las instrucciones recibidas en este sistema se le conoce como **lenguaje máquina**.



Fuente: Wikipedia. CPU AMD Ryzen 7 1800X / Autor: Michael Wolf

Lenguaje de alto nivel

Cuando hacemos uso del ordenador, lo que sucede internamente es que tenemos un programa que se encarga de *interpretar* todo lo que hacemos y lo termina *traduciendo* a instrucciones de ceros y unos que son los que el ordenador entiende; y de esta forma, *encenderá* o *apagará* transistores acordes a la instrucción suministrada.

Dado que esta forma de comunicación con el ordenador implica una barrera de idioma muy elevada para cualquier persona, disponemos de múltiples lenguajes que, dependiendo de su sintaxis, entrarán en las categorías de alto nivel, que serán más fáciles de interpretar por personas, o de bajo nivel, que son lenguajes que se acercan más al lenguaje máquina. En ambos casos, pasan por un proceso de *traducción* a lenguaje máquina que tardaran más o menos dependiendo de cómo es realizada la *traducción*. Dichas traducciones pueden ser realizadas a través de un proceso de interpretación o compilación y de esta forma conectamos con el segundo término, **interpretado**.

Lenguaje interpretado

Un lenguaje interpretado es aquel que traduce sus instrucciones a lenguaje máquina a través de un intérprete en tiempo real. En el caso de JavaScript, estos intérpretes se encuentran integrados en los *browsers* (navegadores) tales como Firefox, Chrome, Safari o Edge, que usas para navegar por internet. Te puedes imaginar que asistes a una charla y que el orador está hablando en ruso y alguien por un pinganillo te va traduciendo en directo.

A diferencia de estos, tenemos los lenguajes compilados que son aquellos en los que el proceso de traducción no se hace en directo, sino que generan un archivo nuevo con las traducciones hechas que es el que se ejecutará en el ordenador. Un ejemplo fácil de ver son los ficheros con extensión .exe para el sistema operativo windows, son ficheros compilados e internamente tienen instrucciones en lenguaje máquina.

Lenguaje multiparadigma

Y ya solo nos queda el término **multiparadigma**, el cual hace referencia a que soporta múltiples estilos de programación, porque como en la vida misma, existen múltiples formas de hacer las cosas, equipos a los que ser fanáticos, etc. En el caso de la programación es cómo generar el código y organizarlo, cosas que aprenderás con el paso del tiempo.

JavaScript es un lenguaje que tiene una gramática para que un humano la pueda entender (alto nivel), que sus instrucciones son *traducidas* en directo por medio de un intérprete (interpretado) y que permite múltiples formas de generar y organizar el código de las aplicaciones que construyamos (multiparadigma).

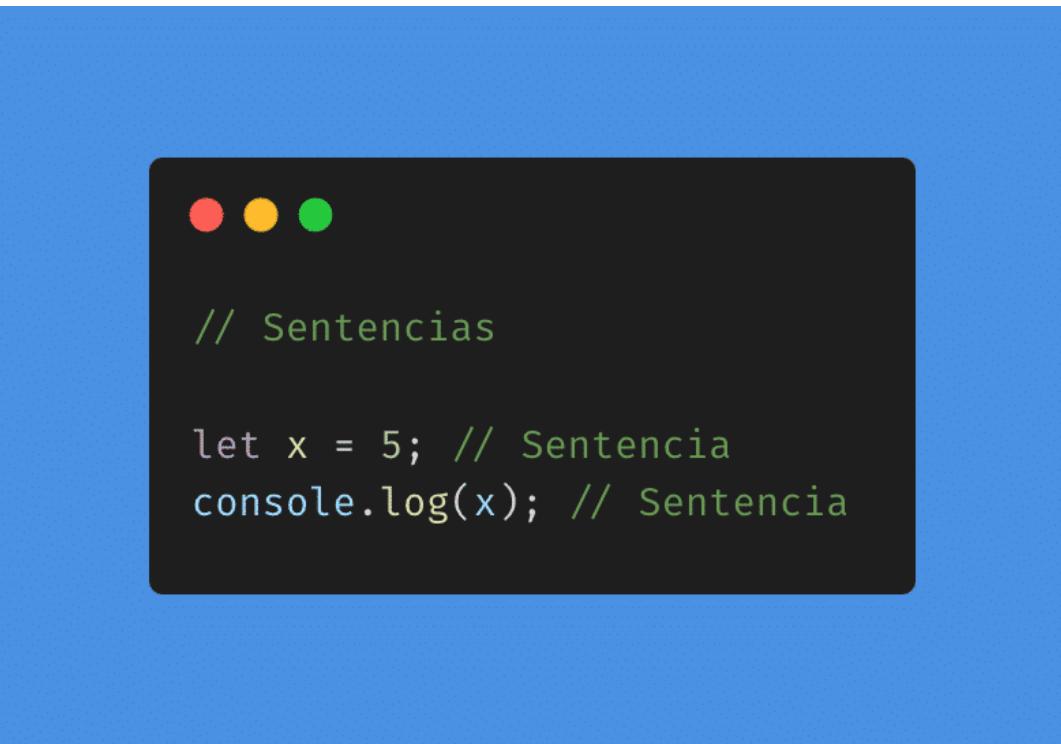
Sentencias y expresiones



Sentencia

Una sentencia en JavaScript es una instrucción que se le da al motor de JavaScript para que la ejecute. Las sentencias son los bloques de construcción de cualquier programa y determinan el flujo y la lógica de un script.

Por lo general, las sentencias en JavaScript se terminan con un punto y coma (;), aunque no siempre es obligatorio gracias a la **inserción automática de punto y coma** de JavaScript. Sin embargo, es una buena práctica incluirlo para evitar posibles errores.



A screenshot of a terminal window with a dark background and three colored dots (red, yellow, green) at the top. Below them, the text of a JavaScript script is displayed:

```
// Sentencias

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

Expresiones

Una expresión en JavaScript es cualquier fragmento de código que evalúa un valor. Las expresiones son fundamentales en la programación ya que son la base para crear y manipular datos. En JavaScript, debido a su naturaleza dinámica y flexible, casi cualquier fragmento de código que produce un valor puede considerarse una expresión.

Aunque las expresiones son similares a las sentencias, hay una distinción. Una expresión es cualquier fragmento de código que evalúa un valor. Las expresiones pueden ser parte de sentencias. Por ejemplo, en la sentencia ***let z = x + y;***, el fragmento ***x + y*** es una expresión.

Variables y tipos de datos



Imagina que una variable es como un vaso que utilizamos para contener diferentes cosas. Este vaso tiene la capacidad de almacenar diferentes tipos de elementos, como agua, zumo, monedas o cualquier otra cosa que necesitemos.

En JavaScript, una variable es similar a ese vaso, pero en lugar de contener líquidos u objetos físicos, almacena datos o valores que podemos utilizar en nuestro código. Estos valores pueden ser números, cadenas de texto, booleanos, objetos u otros tipos de datos.

```
// vamos a definir una variable age / let's define an age variable
let age = 30;
```

En este caso, el vaso (la variable `age`) contiene el valor 30, que representa la edad de una persona.

Luego, podemos utilizar esa variable para realizar diferentes acciones, como mostrar la edad en un mensaje o realizar cálculos:

```
// vamos a definir una variable age / let's define an age variable
let age = 30;
let message = "Tienes " + age + " años."
console.log(message) // Tienes 30 años.
```

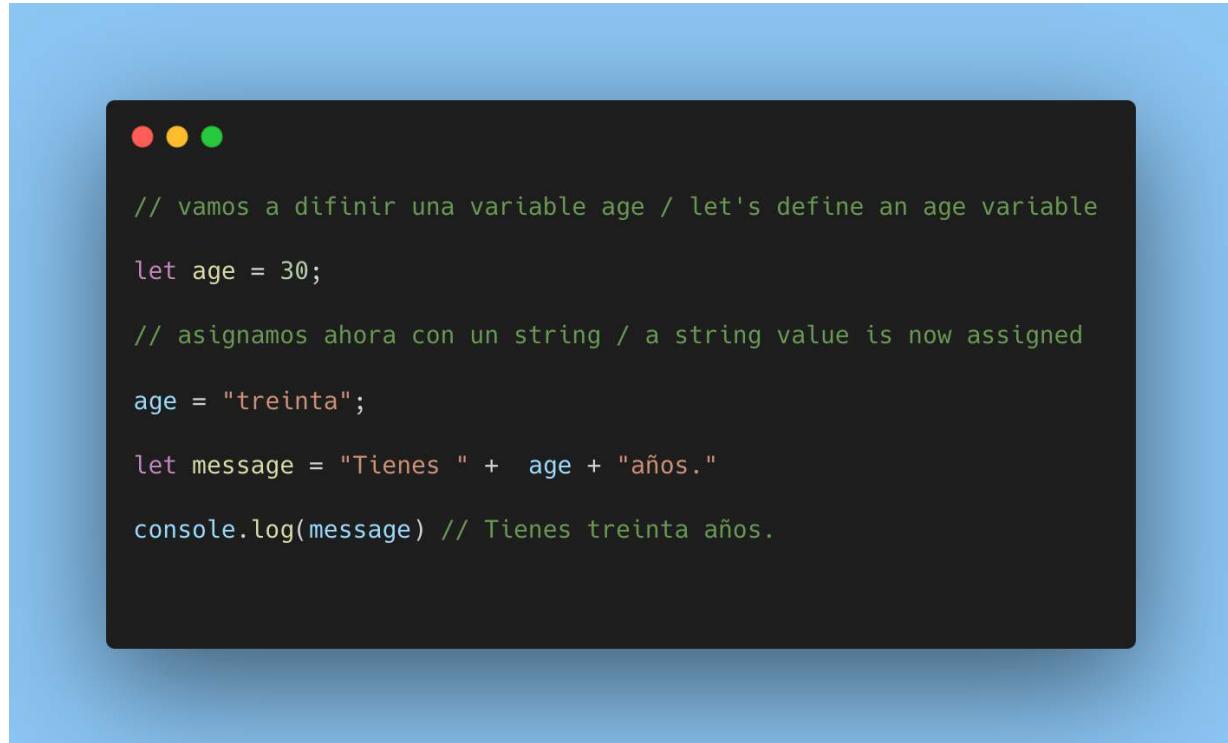
console.log() es una función que muestra en consola el valor que se le pasa como parámetro. Profundizaremos en esto más adelante en el apartado dedicado a las funciones.

Declaración de variables (**var**, **let** y **const**)

En JavaScript disponemos de 3 palabras reservadas para definir una variable;

- var**: era la única forma de declarar variables en JavaScript antes de la introducción de **let** y **const** en ES6. Su uso está poco a poco siendo descontinuado, ya que introduce efectos secundarios en cómo el código debería de funcionar.
- let**: introducida en ES6, es la palabra reservada que viene a sustituir más directamente a la anteriormente mencionada **var**. Una variable declarada con **let** permite cambiar el valor a lo largo de la ejecución de un programa. También viene a resolver problemas que generaba **var** como el *block scope* y *hoisting* que veremos más adelante.
- const**: su comportamiento es similar al de **let** con la única diferencia que es utilizada para declarar constantes, es decir, variables que no deberían de cambiar su valor a lo largo del ciclo de vida del programa.

Un punto adicional a resaltar de las variables en JavaScript es que, al igual que en el ejemplo del vaso, una misma variable (si está declarada con let o var) puede contener diferentes tipos de datos durante la vida del programa, esta característica lo hace entrar en una categoría adicional de lenguajes: **lenguaje de tipado dinámico**.



The screenshot shows 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:

```
// vamos a definir una variable age / let's define an age variable
let age = 30;

// asignamos ahora con un string / a string value is now assigned
age = "treinta";

let message = "Tienes " + age + "años."

console.log(message) // Tienes treinta años.
```

Si bien el fastbook está completamente en español, es bueno que te acostumbres a ver el código en inglés (nombres de variables, funciones, etc.), ya que es el idioma estándar en el sector. Las excepciones son los mensajes para mostrar al usuario, pues estarán en el idioma que nosotros definamos.

Tipos de datos

Los tipos de datos se refieren a las diferentes categorías de datos que pueden ser manipuladas dentro de ese lenguaje. Estos tipos definen qué valores puede tomar una variable y qué operaciones se pueden realizar con ella. Los tipos de datos son fundamentales en la programación, ya que ayudan a garantizar que los datos sean manipulados de manera adecuada y coherente.

En JavaScript existen seis tipos de **datos primitivos**:

1

number: se utiliza para representar números, ya sean enteros o de punto flotante (decimales). Nos sirve para hacer cálculos matemáticos y manipular valores numéricos.

2

symbol: es uno de los tipos primitivos introducidos en ECMAScript 6 (ES6) que se utiliza para crear identificadores únicos e inmutables. Los símbolos son utilizados principalmente como propiedades únicas de objetos para evitar colisiones en nombres de propiedades y para crear propiedades 'privadas' que no son accesibles directamente desde el exterior del objeto. Los usaremos en labs más avanzados dentro del programa.

3

string: se utiliza para representar texto. Un **string** es una secuencia de caracteres que puede incluir letras, números, símbolos y espacios en blanco.

4

boolean: es usado para representar valores de verdad, es decir, valores que son verdaderos (true) o falsos (false). Mayormente se usan para validar condiciones que se tengan que cumplir o no.

5

BigInt: este tipo de dato BigInt se introdujo en ECMAScript 2020 para representar números enteros arbitrariamente grandes. Antes de la introducción de BigInt, los números en JavaScript eran representados utilizando el tipo **number**, que tiene limitaciones en términos de tamaño debido a su representación en decimales.

6

undefined: es un valor especial y un tipo de dato primitivo que se utiliza para representar la ausencia de un valor definido. Es uno de los valores fundamentales en el lenguaje y se usa para indicar que una variable no tiene asignado un valor o que una propiedad de un objeto no está definida.

Existe un tipo adicional llamado *null* que es un primitivo especial, ya que puede funcionar como valor o como objeto (dependerá del contexto), y se utiliza para representar la intencional ausencia de un valor. A diferencia de **undefined**, que se utiliza para indicar que una variable o propiedad no ha sido inicializada, *null* se usa para asignar explícitamente la ausencia de un valor a una variable o propiedad.

Después tenemos las estructuras que son formas más complejas de organizar y almacenar datos en un programa. Pueden contener múltiples valores y, a menudo, se utilizan para representar conjuntos o colecciones de información, como es el caso de los objetos y los arrays de los que hablaremos al final de este fastbook.



Puedes investigar más sobre este tema siguiendo la documentación oficial de [Mozilla](#).

Conversión de tipos y coerción

En JavaScript, la conversión de tipos ocurre cuando un tipo de dato es pasado a otro tipo y esto puede ocurrir de manera implícita o explícita. Por otro lado, tenemos la coerción, que es la conversión automática de valores de un tipo de dato a otro. JavaScript puede realizar conversiones automáticas de tipos en ciertas situaciones. Por ejemplo:

```
● ● ●

// Coerción

let ageString = '25'; // Cadena
let ageInt = Number(numString); // Conversión a número
console.log(numInt); // Resultado: 25 (número)

let num = 42; // Número
let str = String(num); // Conversión a cadena
console.log(str); // Resultado: "42" (cadena)

let isValid = true; // Booleano
let numIsValid = Number(value); // Conversión a número
console.log(numValue); // Resultado: 1 (verdadero es equivalente a 1)

let stringValue = 'Hola'; // Cadena
let boolValue = Boolean(stringValue); // Conversión a booleano
console.log(boolValue); // Resultado: true (una cadena no vacía se considera verdadera)
```

Es importante tener en cuenta las conversiones de tipos, ya que pueden afectar al resultado de operaciones y comparaciones en el código. Por tanto, se debe estar atento a cómo JavaScript maneja las conversiones de tipos para evitar comportamientos inesperados.

Operadores



En programación, los operadores son símbolos especiales que se utilizan para realizar operaciones en variables y valores. En JavaScript, los operadores juegan un papel crucial al permitir realizar cálculos matemáticos, comparaciones y manipulaciones de datos. En este fastbook hablaremos de los operadores más utilizados en el día a día del lenguaje.

Operadores aritméticos

Estos operadores se utilizan para realizar cálculos matemáticos en números.



Suma + : se utiliza para sumar dos números o para concatenar cadenas de texto.



Resta - : se utiliza para restar un número de otro.



Multiplicación * : se utiliza para multiplicar dos números.



División / : se utiliza para dividir un número por otro.



Módulo o resto % : devuelve el residuo de la división entre dos números.

```
// Operadores aritméticos

const posX = 10;
const posY = 5;
let add = posX + posY; // 15
let subtract = posX - posY; // 5
let multiply = posY * posX; // 50
let divide = posX / posY; // 2
let remainder = posX % posY; // 0

// El operador de suma tambien sirve para concatenar strings

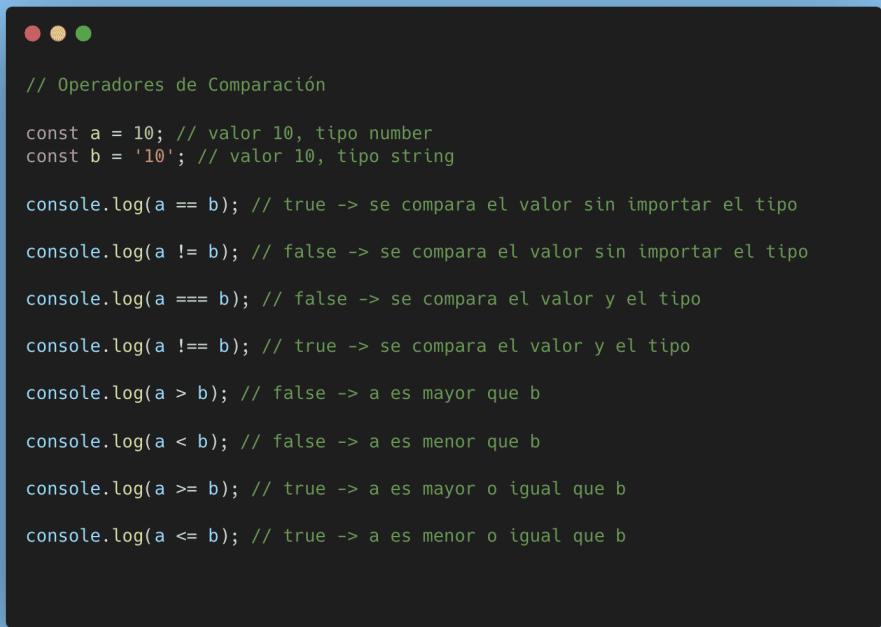
const hello = 'Hola';
const world = 'Mundo';

console.log('¡' + hello + ' ' + world + '!') // ¡Hola Mundo!
```

Operadores de comparación

Estos operadores se utilizan para comparar dos valores y devolver un valor booleano (*true* o *false*).

- Igual ==:** compara si dos valores son iguales.
- Diferente !=:** compara si dos valores no son iguales.
- Estrictamente igual ===:** compara si dos valores son iguales y del mismo tipo.
- Estrictamente no igual !==:** compara si dos valores no son iguales o no son del mismo tipo.
- Mayor que >:** compara si un valor es mayor que otro valor.
- Menor que <:** compara si un valor es menor que otro valor.
- Mayor o igual que >=:** compara si un valor es mayor o igual que otro valor.
- Menor o igual que <=:** compara si un valor es menor o igual que otro valor.



```
// Operadores de Comparación

const a = 10; // valor 10, tipo number
const b = '10'; // valor 10, tipo string

console.log(a == b); // true -> se compara el valor sin importar el tipo
console.log(a != b); // false -> se compara el valor sin importar el tipo
console.log(a === b); // false -> se compara el valor y el tipo
console.log(a !== b); // true -> se compara el valor y el tipo
console.log(a > b); // false -> a es mayor que b
console.log(a < b); // false -> a es menor que b
console.log(a >= b); // true -> a es mayor o igual que b
console.log(a <= b); // true -> a es menor o igual que b
```

Operadores lógicos

Estos operadores se utilizan para combinar valores booleanos y realizar operaciones lógicas.



AND lógico && : devuelve *true* si ambos operandos son *true*.



OR lógico || : devuelve *true* si al menos uno de los operadores es *true*.



NOT lógico !: niega el valor de un operando, convirtiendo *true* en *false* y viceversa.

```
● ● ●
// Operadores Lógicos

const minimumAgeToDrive = 18

const firstPerson = {
  age: 30,
  hasDrivingLicense: false
}

const secondPerson = {
  age: 18,
  hasDrivingLicense: true
}

//Ejemplo 1: retornara false porque no cumple ambas partes, la primera parte es true pero la segunda parte es false
const canFirstPersonDrive = firstPerson.age >= minimumAgeToDrive && firstPerson.hasDrivingLicense // false
console.log("Tiene permitido conducir la primera persona: " + canFirstPersonDrive) //Tiene permitido conducir la primera persona: false

//Ejemplo 2: retornara true porque ambas partes cumplen
const canSecondPersonDrive = secondPerson.age >= minimumAgeToDrive && secondPerson.hasDrivingLicense // true
console.log("Tiene permitido conducir la segunda persona: " + canSecondPersonDrive) //Tiene permitido conducir la segunda persona: true

//Ejemplo 3: retornara true porque uno de los dos puede conducir
const canSomeoneDrive = canFirstPersonDrive || canSecondPersonDrive // true
console.log("Alguno puede conducir: " + canSomeoneDrive) //Alguno puede conducir: true

//Ejemplo 4: En este caso cambiamos el mensaje a mostrar y cambiamos el resultado para que este acorde al mensaje
console.log("Nadie puede conducir: " + !canSomeoneDrive) // Nadie puede conducir: false
```

Operadores de asignación

Se utilizan para asignar valores a variables. Existen múltiples operadores de asignación, pero los más utilizados son los siguientes:

- Asignación =**: asigna el valor de la derecha a la variable de la izquierda.
- Asignación de adición +=**: suma el operando a la derecha del signo de igualdad y lo asigna a la variable.
- Asignación de resta -=**: resta el operador a la derecha del signo de igualdad y lo asigna a la variable.
- Asignación de multiplicación *=**: multiplica el operador a la derecha del signo de igualdad y lo asigna a la variable.
- Asignación de división /=**: divide el operador a la derecha del signo de igualdad y lo asigna a la variable.

```
// Operadores de asignación

const age = 20; // asigna el valor 20 a la variable age

let counter = 5; // Inicialización
counter += 3; // Equivalente a: counter = counter + 3;

let greeting = "Hello"; // Inicialización
greeting += ", mundo"; // Equivalente a: greeting = greeting + ", mundo";

let total = 100; // Inicialización
total -= 20; // Equivalente a: total = total - 20;

let product = 10; // Inicialización
product *= 2; // Equivalente a: product = product * 2;

let division = 50; // Inicialización
division /= 5; // Equivalente a: division = division / 5;
```

Cabe destacar que estos operadores no sirven con todos los tipos de datos y pueden tener comportamientos diferentes dependiendo del tipo de dato usado.



Puedes profundizar y descubrir otros operadores en la documentación oficial de [Mozilla](#).

Estructuras de control



Qualentum Lab

Las estructuras de control son bloques de código que permiten tomar decisiones y controlar el flujo de ejecución en un programa. En JavaScript, las estructuras de control son fundamentales para crear lógica, ejecutar acciones condicionales y repetir tareas.

Condicionales

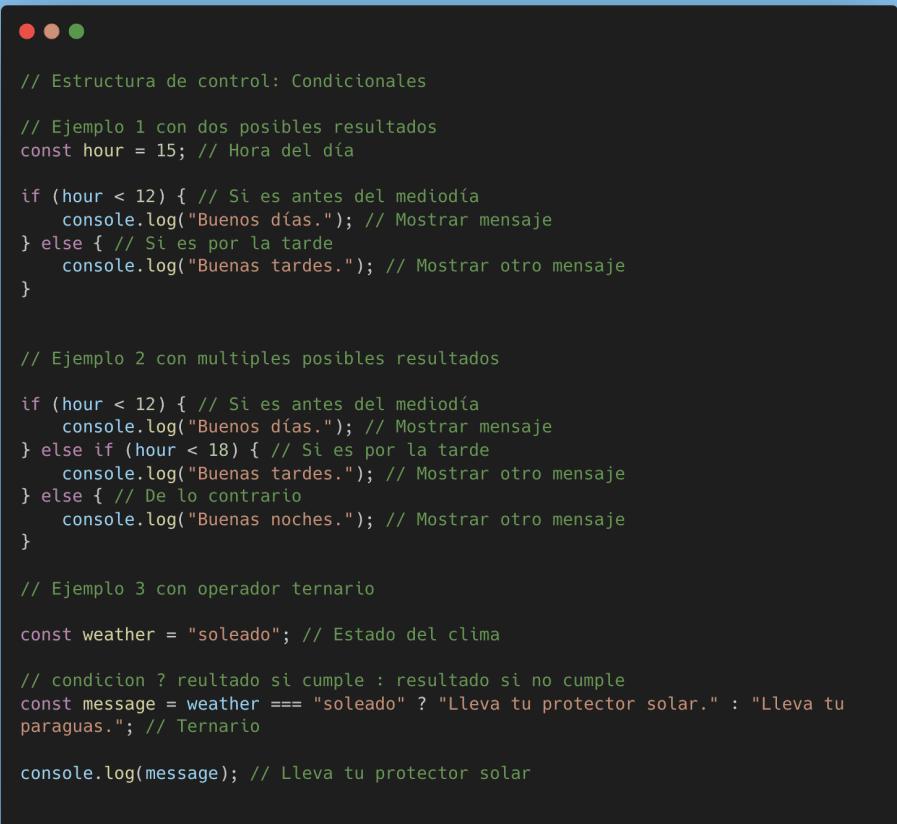
Estas estructuras permiten que un programa ejecute diferentes acciones según si una condición se cumple o no. En esencia, las estructuras de control condicionales permiten que un programa tome diferentes caminos en función de las circunstancias. Permiten que los programas respondan de manera dinámica a diferentes situaciones y datos de entrada, lo que es fundamental para crear lógica y funcionalidad en aplicaciones y sistemas.

Las estructuras de control condicionales más comunes son las siguientes.

1

if-else

Permite ejecutar un bloque de código (*branch*) si se cumple una condición y, si no se cumple, ejecuta otra *branch*. Esta estructura puede evaluar múltiples condiciones haciendo uso de la palabra reservada '**else if**' como se muestra en uno de los ejemplos a continuación.



```
// Estructura de control: Condicionales

// Ejemplo 1 con dos posibles resultados
const hour = 15; // Hora del día

if (hour < 12) { // Si es antes del mediodía
    console.log("Buenos días."); // Mostrar mensaje
} else { // Si es por la tarde
    console.log("Buenas tardes."); // Mostrar otro mensaje
}

// Ejemplo 2 con multiples posibles resultados

if (hour < 12) { // Si es antes del mediodía
    console.log("Buenos días."); // Mostrar mensaje
} else if (hour < 18) { // Si es por la tarde
    console.log("Buenas tardes."); // Mostrar otro mensaje
} else { // De lo contrario
    console.log("Buenas noches."); // Mostrar otro mensaje
}

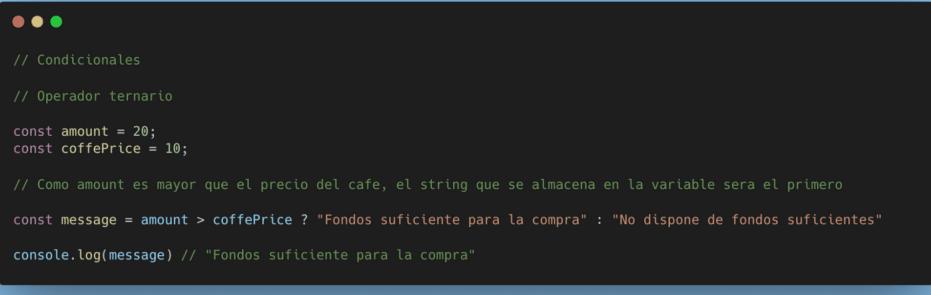
// Ejemplo 3 con operador ternario

const weather = "soleado"; // Estado del clima

// condicion ? resultado si cumple : resultado si no cumple
const message = weather === "soleado" ? "Lleva tu protector solar." : "Lleva tu paraguas." // Ternario

console.log(message); // Lleva tu protector solar
```

También existe un operador llamado **ternario** que permite realizar una evaluación condicional en una única línea de código. Es una forma concisa de escribir una expresión condicional que devuelve un valor basado en una condición booleana. Está compuesta por una condición que, seguida por el signo **?**, ejecutará la expresión si la condición se cumple; y seguido del signo **:** ejecutará la expresión si la condición no se cumple, tal como se muestra a continuación:



```
// Condicionales
// Operador ternario

const amount = 20;
const coffeePrice = 10;

// Como amount es mayor que el precio del cafe, el string que se almacena en la variable sera el primero
const message = amount > coffeePrice ? "Fondos suficiente para la compra" : "No dispone de fondos suficientes"
console.log(message) // "Fondos suficiente para la compra"
```

2

switch

Permite evaluar una variable en diferentes casos y ejecutar una *branch* correspondiente al caso que coincide con el valor de la variable. Dispone de tres palabras reservadas: **case** para evaluar si la evaluación de la variable cumple el valor de su branch; **break** de uso obligatorio para salir del flujo de control una vez evaluada la variable; y **default** que sirve para establecer un comportamiento predeterminado, en caso de que no exista alguno definido para la condición evaluada.

```
// Estructura de control: Condicionales
// Switch:

// Ejemplo 1

let dayOfTheWeek = "Martes"; // Día de la semana

switch (dayOfTheWeek) {
    case "Lunes":
        console.log("Es el primer día de la semana.");
        break;
    case "Martes":
        console.log("Es el segundo día de la semana."); // Este es el resultado obtenido
        break; // Uso de break para salir del switch
    case "Miércoles":
        console.log("Es el tercer día de la semana.");
        break;
    case "Jueves":
        console.log("Es el cuarto día de la semana.");
        break;
    case "Viernes":
        console.log("Es el quinto día de la semana.");
        break;
    default: // Valor por defecto
        console.log("Es fin de semana.");
}

// Ejemplo 2:

dayOfTheWeek = "Domingo"; // Día de la semana

switch (dayOfTheWeek) {
    case "Lunes":
        console.log("Es el primer día de la semana.");
        break;
    case "Martes":
        console.log("Es el segundo día de la semana.");
        break;
    case "Miércoles":
        console.log("Es el tercer día de la semana.");
        break;
    case "Jueves":
        console.log("Es el cuarto día de la semana.");
        break;
    case "Viernes":
        console.log("Es el quinto día de la semana.");
        break;
    default: // Valor por defecto
        console.log("Es fin de semana."); // Este es el resultado obtenido
}
```

Bucles

Permiten repetir una serie de instrucciones múltiples veces, mientras se cumpla una condición específica. Estas estructuras son fundamentales para automatizar tareas repetitivas y realizar procesamientos en conjunto.

Existen **tres tipos** principales de estructuras de control de bucle.

1

while

Este bucle ejecuta un bloque de código siempre que se cumpla una condición dada. La condición se verifica antes de cada iteración y, si es verdadera, el bucle continúa ejecutándose.

```
● ● ●

// Estructura de control: Bucles

// While:

let count = 1; // Contador

while (count <= 5) {
    console.log("Número: " + count); // Muestra el número actual
    count++; // Incrementa el contador en 1
}
```

2

for

Es útil cuando se necesita iterar sobre una secuencia de números conocida. Se compone de tres partes: una inicialización, una condición y una expresión de actualización. El bloque de código se ejecuta mientras la condición se cumpla.

```
● ● ●  
// Estructura de control: Bucles  
// For:  
  
for (let i = 1; i <= 5; i++) {  
    console.log("Número: " + i); // Muestra el número actual  
}
```

3

do-while

Similar al bucle *while*, el **bucle do-while** ejecuta un bloque de código al menos una vez antes de verificar la condición. Luego, continúa ejecutando el bloque siempre que la condición sea verdadera.

```
● ● ●  
// Estructura de control: Bucles  
// Do-While:  
  
let count = 1; // Contador  
  
do {  
    console.log("Número: " + count); // Muestra el número actual  
    count++; // Incrementa el contador en 1  
} while (count <= 5);
```

Funciones



Qualentum Lab

Las funciones son bloques de código reutilizables que pueden recibir datos de entrada (argumentos), realizar operaciones y devolver un resultado (retorno). Son una parte esencial para organizar y modularizar el código.

Declaración y llamada de funciones

Una función se declara utilizando la palabra reservada **function**, seguida por un nombre que identifica a la función, paréntesis que pueden contener parámetros y un bloque de código que define las operaciones de la función.

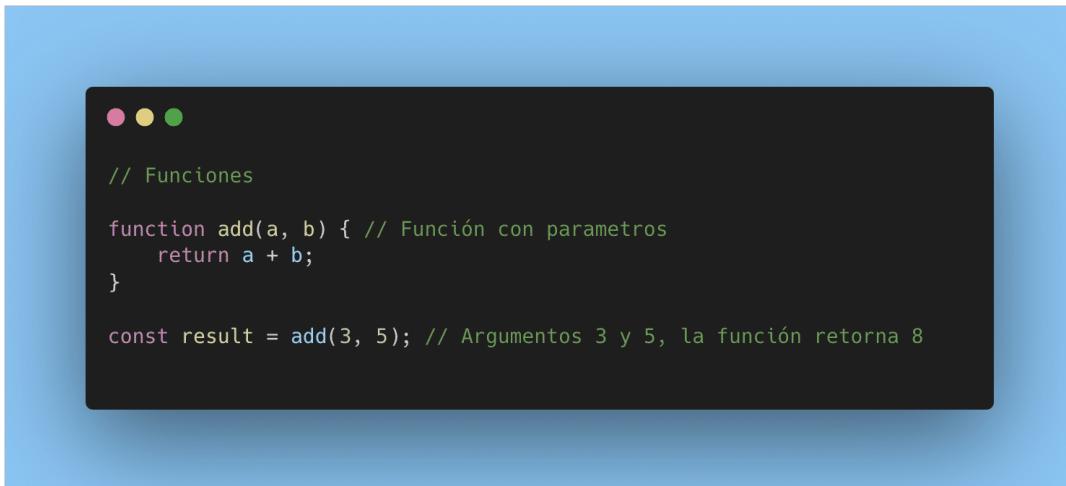
```
// Funciones

// Declaración de una función
function sayHi(name) {
    console.log("¡Hola, " + name + "!");
}

// Llamada a la función
sayHi("Juan"); // Resultado: ¡Hola, Juan!
```

Parámetros y argumentos

Los parámetros son variables que se utilizan para recibir valores dentro de una función. Los argumentos son los valores reales que se pasan a la función cuando se la llama.

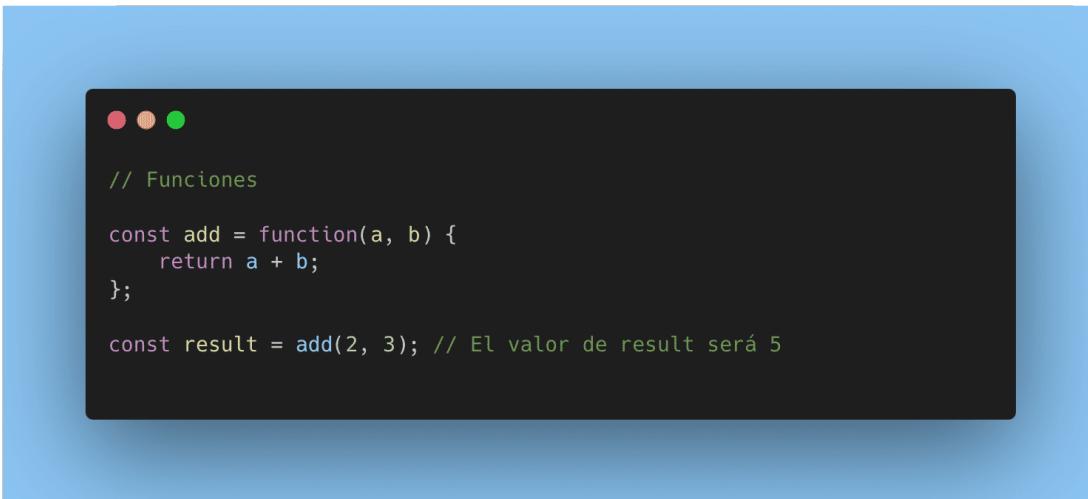


```
// Funciones

function add(a, b) { // Función con parametros
    return a + b;
}

const result = add(3, 5); // Argumentos 3 y 5, la función retorna 8
```

Las funciones también pueden ser anónimas, lo que significa que no tienen un nombre específico. Se pueden asignar a variables o utilizarse directamente en otras funciones.



```
// Funciones

const add = function(a, b) {
    return a + b;
};

const result = add(2, 3); // El valor de result será 5
```

Nota que en los ejemplos anteriores se usa otra palabra reservada **return** para hacer que la función devuelva un valor en donde está siendo ejecutada, con lo que una función puede o no retornar valores, dependiendo de la lógica definida en ella y si tiene o no **return**.

Funciones de flecha (arrow functions)

Las *arrow functions*, o funciones flecha, son una adición más reciente al lenguaje JavaScript, introducidas en ES6. Estas funciones ofrecen una sintaxis más concisa para escribir funciones. Las *arrow functions* tienen la siguiente sintaxis.

```
// Arrow function

// Sintaxis sin recibir parametros
const showHelloWorld = () => console.log("Hola Mundo")

// En el caso de que solo recibamos un parametro, la podremos escribir con o sin parentesis
const showAddInConsole = result => console.log(result);

// Recibiendo 2 o más es obligatorio el uso de parentesis
const add = (x, y) => x + y;

// Si la función contiene mas de un sentencia, tendrá que tener cuerpo

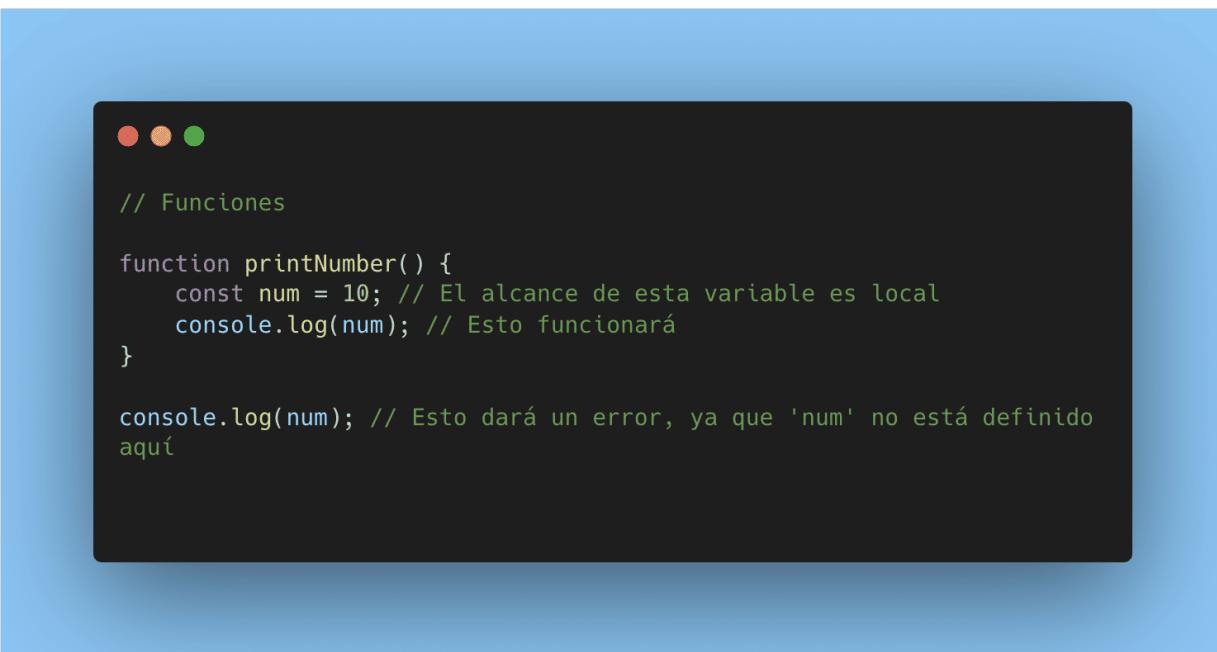
const addAndMultiplyBy = (value, multiplier) => {
    const result = value + multiplier;
    return result * multiplier;
}

showAddInConsole(addAndMultiplyBy(add(5,3),2)) // 20
```

De momento familiarízate con la sintaxis de las *arrow functions*, ya que a lo largo del curso explicaremos cuál es el problema que vienen a solucionar.

Alcance de las variables (scope)

El alcance de una variable se refiere a dónde en el código una variable es accesible y utilizada. Las variables declaradas dentro de una función solo son visibles dentro de esa función, esto se conoce como **alcance local**.



```
// Funciones

function printNumber() {
    const num = 10; // El alcance de esta variable es local
    console.log(num); // Esto funcionará
}

console.log(num); // Esto dará un error, ya que 'num' no está definido
aquí
```

Aquí retomamos el concepto de *hoisting*, al que hicimos referencia en el apartado de variables, que no es más que una característica especial que tiene JavaScript a la hora de declarar variables. Una forma resumida y fácil de ver cómo funciona el *hoisting* es como un proceso de prelectura del código, por parte del intérprete de JavaScript, para encontrar dónde se están declarando variables y funciones en el código: 'sube' la declaración a principio del contexto, ya sea global o de bloque, y después pasaría el proceso de inicialización en el que se le da el valor que contiene dicha variable.

Dependiendo de si una variable ha sido declarada con **var** o con **const** y **let**, el *hoisting* tendrá un comportamiento diferente. Para aclarar la diferencia con **var** lo veremos mejor con el siguiente ejemplo.

```
// Funciones

function hoistingTest() {
    const condition = true;

    // Tratamos de mostrar el valor de una variable con var que no esta declarado en este bloque
    console.log(functionScope) // undefined

    if(condition){
        var functionScope = 20; // Se inicializa la variable
        console.log(functionScope) // 20
    }
}
```

El intérprete de JavaScript lo que hará es lo siguiente:

```
// Funciones

function hoistingTest() {
    const condition = true;

    var functionScope; // Variable elevada por hoisting

    // Mostrará undefined porque el hoisting ha subido la declaración más no la inicialización
    console.log(functionScope) // undefined

    if(condition) {
        functionScope = 20; // Se inicializa la variable
        console.log(functionScope) // 20
    }
}
```

En el caso de **const** o **let**, el scope que poseen es de bloque y no de función y, en este caso, no sufren del efecto de elevación del *hoisting*.

```
// Funciones

function hoistingTest() {
    const condition = true;

    console.log(blockScope) // ReferenceError: blockScope is not defined

    if(condition){
        const blockScope = 20; // Se inicializa la variable
        console.log(blockScope) // 20
    }
}
```

Estando en el contexto o bloque donde se define la variable, si intentamos mostrar el valor de esta antes de su inicialización, veremos que el error es diferente.

```
// Funciones

function hoistingTest() {
    const condition = true;

    if(condition){
        console.log(blockScope) // ReferenceError: Cannot access 'blockScope' before initialization
        const blockScope = 20; // Se inicializa la variable
        console.log(blockScope) // 20
    }
}
```

Esto es porque no está permitido acceder a una variable antes de su inicialización. A este intervalo de tiempo en el que se encuentran desde el inicio del bloque hasta la declaración de la variable se le conoce como [temporal dead zone](#).

Objetos



Qualentum Lab

Los objetos son estructuras fundamentales en JavaScript que permiten agrupar datos y funcionalidades relacionadas en una única entidad. Cada objeto puede contener propiedades (variables) y métodos (funciones) que trabajan juntos para representar un concepto o entidad del mundo real en el código.

Creación y manipulación de objetos

En JavaScript, los objetos pueden ser creados de diferentes maneras, pero una de las más comunes es utilizando la sintaxis de llaves '{}' para definir un objeto y, luego, asignarle propiedades y métodos. Permiten organizar datos y funcionalidad en una estructura coherente. Los objetos pueden ser creados directamente o utilizando funciones constructoras o clases para crear instancias similares.

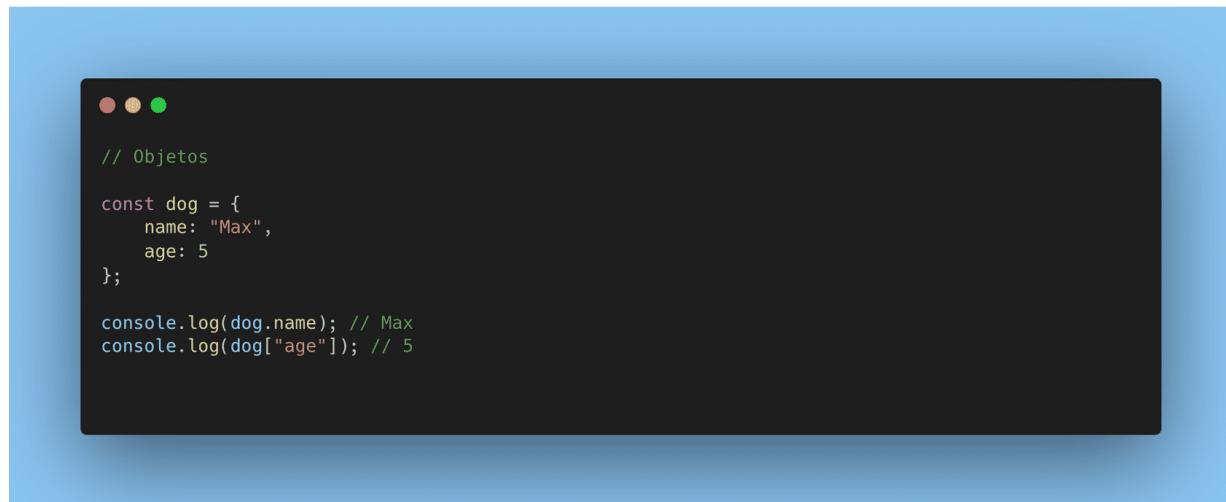
```
// Objetos

// Creación de un objeto person
const person = {
    name: "Juan",
    age: 30,
    sayHi: function() {
        console.log("¡Hola! Mi nombre es " + this.nombre + " y tengo " + this.edad + " años.");
    }
};

// Acceso a las propiedades y métodos del objeto
console.log(person.nombre); // Resultado: Juan
console.log(person.edad); // Resultado: 30
person.sayHi(); // Resultado: ¡Hola! Mi nombre es Juan y tengo 30 años.
```

Acceso a propiedades y métodos

Las propiedades representan los datos, mientras que los métodos representan las acciones que pueden realizarse con esos datos. Las propiedades de un objeto pueden ser accedidas utilizando la notación de punto `objeto.propiedad` o la notación de corchetes `objeto["propiedad"]`.



```
// Objetos

const dog = {
    name: "Max",
    age: 5
};

console.log(dog.name); // Max
console.log(dog["age"]); // 5
```

Los métodos también pueden ser accedidos y llamados de manera similar.



```
// Objetos

const dog = {
    name: "Max",
    age: 5,
    bark: function() {
        console.log("Guau guau")
    }
};

console.log(dog.bark()); // Guau guau
console.log(dog["bark"]()); // Guau guau
```

Arrays



Los arrays, también conocidos como **arreglos**, son estructuras de datos en JavaScript que permiten almacenar y organizar múltiples elementos relacionados en una sola variable. Cada elemento en un arreglo tiene un índice que indica su posición dentro del arreglo. Podemos imaginar los arrays como un organizador de pastillas en donde cada bloque contiene un valor, pero forman parte de una misma estructura.



Los arreglos pueden ser creados bien utilizando corchetes '[]', bien a través del constructor del objeto global '**Array**' de JavaScript y separando los elementos con comas. Se puede acceder a los elementos de un array utilizando su índice dentro de los corchetes, siendo 0 el primer índice de un *array*.

```
● ● ●

// Arrays

const fruits = ["manzana", "pera", "platano", "fresa"];

const moreFruits = new Array("manzana", "pera", "platano", "fresa");

// Los arrays empiezan con indice 0

console.log(fruits[0]); // manzana

console.log(fruits[1]); // pera

console.log(moreFruits[2]); // platano
```

Manipulación de arreglos

Los arreglos en JavaScript tienen una variedad de métodos incorporados para realizar operaciones, como agregar elementos, eliminar elementos, modificar elementos y más. A continuación, veremos algunos de los métodos más usados en JavaScript.



```
// Arrays

const fruits = ["manzana", "pera", "platano", "fresa"];

// 1. Pop: Elimina el último elemento del array
fruits.pop();

console.log(fruits); // ['manzana', 'pera', 'platano']

// 2. Push: Agrega un elemento al final del array
fruits.push("naranja");

console.log(fruits); // ['manzana', 'pera', 'platano', 'naranja']

// 3. Shift: Elimina el primer elemento del array
fruits.shift();

console.log(fruits); // ['pera', 'platano', 'naranja']

// 4. Unshift: Agrega un elemento al inicio del array
fruits.unshift("mango");

console.log(fruits); // ['mango', 'pera', 'platano', 'naranja']

// 5. IndexOf: Busca un elemento en el array y devuelve su índice
const position = fruits.indexOf("pera");

console.log(position); // 1

// 6. Slice: Devuelve un nuevo array con los elementos seleccionados
const newFruits = fruits.slice(1, 3);

console.log(newFruits); // ['pera', 'platano']

// 7. Splice: Elimina o reemplaza elementos del array
fruits.splice(1, 2);

console.log(fruits); // ['mango', 'naranja']

// 8. Concat: Une dos o más arrays
const vegetables = ["tomate", "cebolla", "lechuga"];

const food = fruits.concat(vegetables);

console.log(food); // ['mango', 'naranja', 'tomate', 'cebolla', 'lechuga']
```

Existen muchos más métodos, pero tienen un uso más avanzado, los cuales veremos a lo largo del curso.

Document model object (DOM)

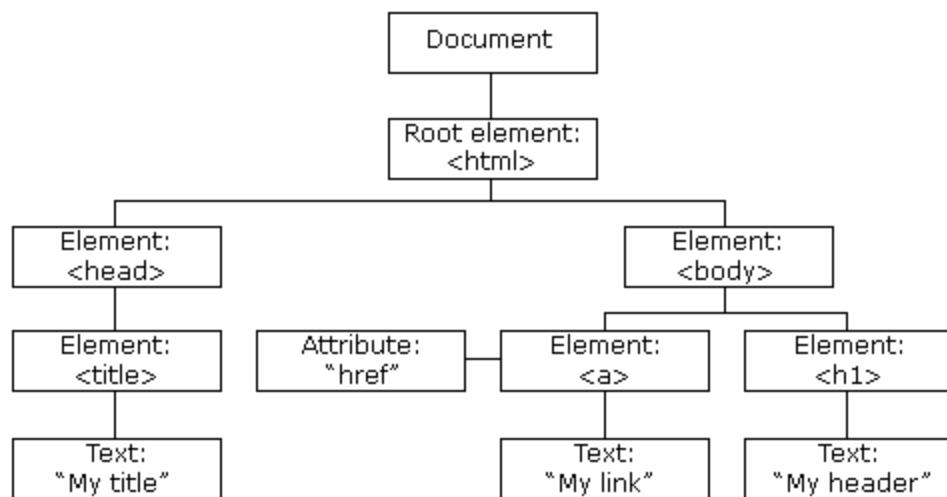


Qualentum Lab

El DOM es una **representación estructurada y programática** de un documento web.

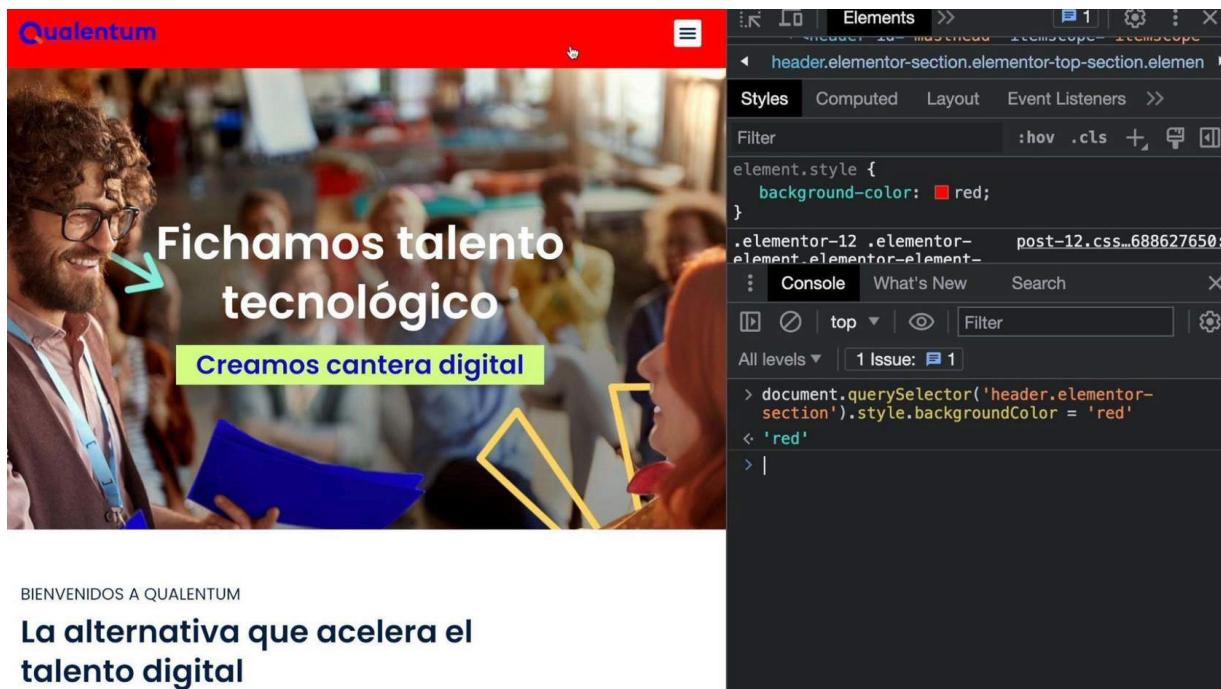
Esencialmente, el DOM convierte todo el contenido de una página web en objetos que pueden ser manipulados y gestionados usando lenguajes de programación, en su mayoría, mediante JavaScript.

El DOM representa una página web en forma de un **árbol de nodos**. Estos nodos pueden ser elementos HTML, atributos, texto dentro de los elementos, y mucho más. Por ejemplo, el elemento <body> de una página web es un nodo, y todos los elementos dentro de <body>, como <p>, <div>, <a>, etc., son nodos hijos de ese nodo <body>.



Una vez que una página web se ha cargado en el navegador puedes modificar elementos, atributos, y estilos; añadir o eliminar nodos; y responder a eventos, todo en tiempo real. Estos cambios se reflejarán inmediatamente en la visualización de la página web.

El DOM brinda a los desarrolladores la capacidad de cambiar el contenido, la estructura, y el estilo de una página web sin tener que recargarla. Por ejemplo, cuando llenas un formulario en una página y ves un mensaje de error sin que la página se recargue, es probable que JavaScript esté manipulando el DOM para mostrar ese mensaje.



Aunque el DOM es comúnmente accedido y manipulado usando JavaScript, es importante destacar que el DOM en sí es independiente de cualquier lenguaje de programación. Está diseñado para ser accesible por diversos lenguajes, aunque en el contexto de navegadores web, JavaScript es el lenguaje predominante.

JavaScript y el DOM

Como mencionamos anteriormente, en los navegadores, JavaScript dispone de una serie de objetos para la manipulación del DOM entre los que se encuentran.

document

Representa el propio documento en el que se encuentra el script. Es una entrada principal al contenido web. Este objeto dispone de una serie de métodos muy utilizados.

```
// DOM: document

document.getElementById(id): // Selecciona un elemento por su atributo id.

document.querySelector(selector): // Selecciona el primer elemento que
coincida con el selector CSS proporcionado.

document.querySelectorAll(selector): // Selecciona todos los elementos que
coincidan con el selector CSS proporcionado y devuelve una NodeList.

document.createElement(tagName): // Crea un nuevo elemento con la etiqueta
especificada.

document.createTextNode(data): // Crea un nuevo nodo de texto con el contenido
especificado.
```

element

Representa un objeto que corresponde a un elemento seleccionado del DOM. Algunos de los métodos y propiedades para manipular este objeto son:

```
// DOM: element

const element = document.querySelector("p.title") // Obtiene un elemento <p> del DOM con la clase title

element.innerHTML: // Permite obtener o establecer el contenido HTML del elemento.

element.outerHTML: // Permite obtener o establecer el contenido HTML del elemento, incluyendo el elemento mismo.

element.textContent: // Permite obtener o establecer el contenido de texto del elemento.

element.setAttribute(name, value): // Establece un atributo y su valor al elemento.

element.getAttribute(name): // Obtiene el valor de un atributo del elemento.

element.classList: // Proporciona métodos para añadir, eliminar y comprobar clases CSS en un elemento.

element.appendChild(childElement): // Añade un elemento hijo al elemento.

element.removeChild(childElement): // Elimina un elemento hijo del elemento.

element.addEventListener(type, listener): // Añade un oyente de eventos al objeto.

element.removeEventListener(type, listener): // Elimina un oyente de eventos del objeto.
```

window

Representa la ventana que contiene el DOM. Ofrece numerosas funciones y propiedades para interactuar con el entorno del navegador.

```
// DOM: window

window.alert(message): // Muestra un cuadro de diálogo con un mensaje.

window.prompt(message): // Muestra un cuadro de diálogo que pide al usuario una entrada.

window.innerWidth: // Proporcionan el ancho interno de la ventana del navegador.

window.innerHeight: // Proporcionan la altura interna de la ventana del navegador.
```

Ssi bien en este fastbook se explica JavaScript lo más independiente de entorno posible, muchos de los ejemplos están orientados hacia la parte frontend, ya que de esa manera es más fácil representarlos visualmente.

Eventos



Qualentum Lab

En la vida real, estamos acostumbrados a responder a eventos: si suena el teléfono, lo contestamos; si llueve, abrimos un paraguas. De manera similar, en la web, los eventos son acciones o sucesos que ocurren en el navegador, a las que podemos responder usando JavaScript.

Por ejemplo:

- Un clic del *mouse*.
- Presionar una tecla del teclado.
- Pasar el *mouse* sobre un elemento.
- Enviar un formulario.
- Cambiar el tamaño de la ventana del navegador.

Así como en los navegadores interactuamos con páginas web, en **Node.js** (un entorno de ejecución de JavaScript fuera del navegador) los eventos son a menudo acciones relacionadas con operaciones de sistema como, por ejemplo:

- Leer o escribir un archivo, detectar cambios en un archivo.
- Recibir una solicitud HTTP, establecer una conexión TCP, recibir un paquete de datos.
- Eventos del sistema operativo, procesos que envían señales, eventos relacionados con el ciclo de vida de la aplicación.
- Obtener datos de una base de datos, procesar flujos de datos.

JavaScript nos permite *escuchar* estos eventos y *reaccionar* a ellos, por ejemplo, mostrando un mensaje cuando un usuario hace clic en un botón.



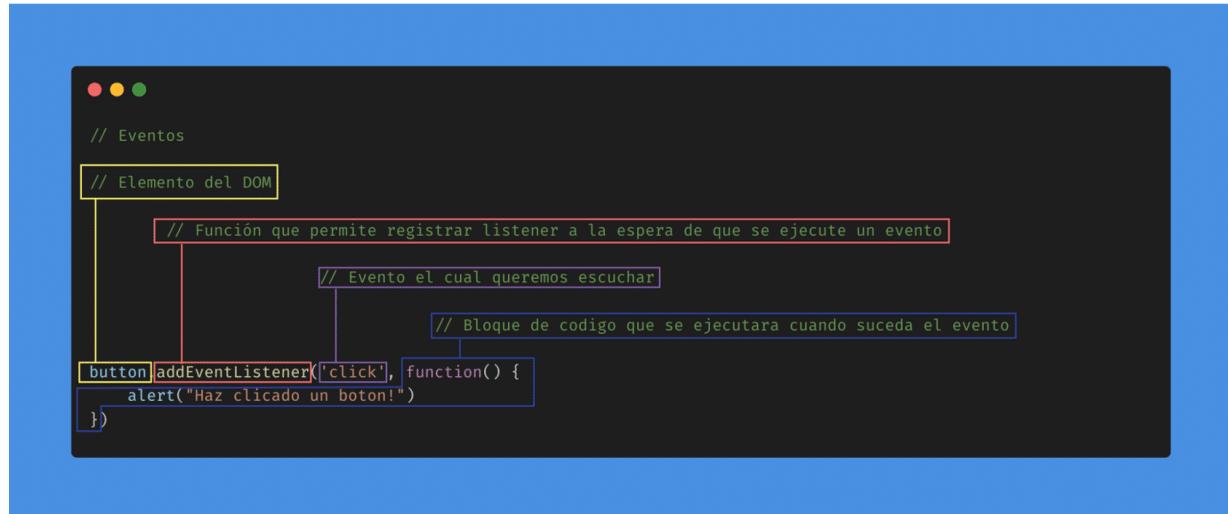
```
// Eventos

// Supongamos que existe un elemento <button> en nuestro html

const button = document.querySelector( "button" ) // Usamos el objeto document para obtener <button> del DOM

button.addEventListener('click', function() { // "Escuchamos" a que ocurra el evento click del botón
    alert("Haz clicado un botón!")
})
```

Ahora analicemos con más detalle el código mostrado anteriormente.

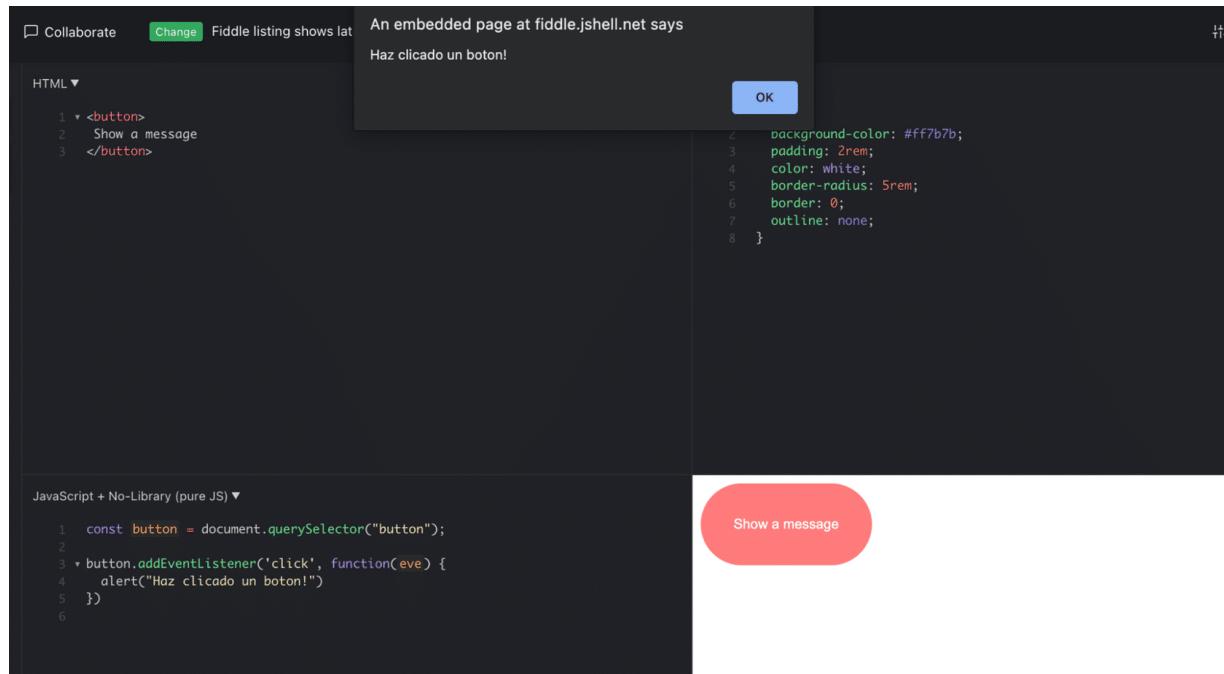


```
// Eventos

// Elemento del DOM
// Función que permite registrar listener a la espera de que se ejecute un evento
// Evento el cual queremos escuchar
// Bloque de código que se ejecutara cuando suceda el evento

button.addEventListener([click], [function() {
    alert("Haz clicado un botón!")
}])
```

En primer lugar, obtenemos el elemento `button` del DOM a través de la función `querySelector` del objeto `document`. Una vez tenemos el elemento, registramos un **listener** al evento clic del botón, pasando la funcionalidad que queremos ejecutar cuando ocurra dicho evento. Cabe destacar que, para registrar los eventos, estos se pasan como primer argumento de la función `addEventListener` como una cadena de texto.



Recuerda: para profundizar más en los tipos de eventos que poseen diferentes elementos HTML, sigue este [enlace](#).

A la hora de trabajar con eventos debemos tener en cuenta un concepto que toma JavaScript para el manejo eventos del DOM, el **bubbling**.

¿Qué es el bubbling?

Cuando ocurre un evento en un elemento, ese evento no afecta solo a ese elemento, sino también a todos sus ancestros en la estructura del DOM. Esto significa que, si tenemos un evento clic en un botón que está dentro de un 'div', y ambos (el botón y el div) tienen manejadores de eventos para el evento 'clic', los dos manejadores se ejecutarán.

La fase de *bubbling* describe este comportamiento: un evento comienza desde el elemento que desencadenó el evento y *burbujea* hacia arriba, pasando por todos los ancestros del elemento, y ejecutando los manejadores de eventos correspondientes en el orden que encuentra.

Para entender mejor este comportamiento, lo veremos con un ejemplo práctico.

Comienzo de la fase de bubbling

```
!— Bubbling

// Supongamos que tenemos los siguientes elementos html

→
<div id="container">
    <button id="button">Haz clic en mí</button>
</div>
```

Añadimos 2 *listeners*, uno al contenedor y otro al botón hijo;

```
// Bubbling

// Usamos la función getElementById para obtener el contenedor
document.getElementById('container').addEventListener('click', () => {
    console.log('Div contenedor fue clicado!');
});

// También podemos usar la función querySelector añadiendo # para indicarle
// que buscamos un id
document.querySelector('#button').addEventListener('click', () => {
    console.log('Botón hijo fue clicado!');
});
```

Si haces clic en el botón 'button', verás dos mensajes: primero, 'Botón hijo fue clicado!' y luego 'Div contenedor fue clicado!'. Esto es debido al *bubbling* que primero maneja el evento en el botón y luego burbujea hacia arriba y se maneja en el 'div contenedor'.

JavaScript + No-Library (pure JS) ▾ Tidy

Haz click en mi

```
4 // Usamos la función getElementById para obtener el contenedor
5 document.getElementById('container').addEventListener('click', () => {
6     console.log('Div contenedor fue clicado!');
7 });
8 // También podemos usar la función querySelector añadiendo # para indicarle que buscamos un id
9 document.querySelector('#button').addEventListener('click', () => {
10    console.log('Botón hijo fue clicado!');
11 });
12
```

Console (beta) ① 2 ② 0 ③ 0 ④ 0 ⑤ 0 Clear console Minimize

"Botón hijo fue clicado!"
"Div contenedor fue clicado!"

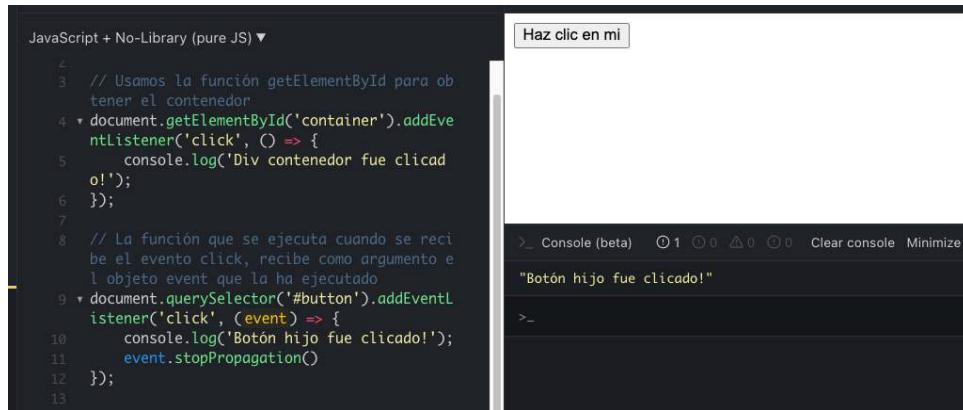
Si, por alguna razón, no quieras que un evento se propague hacia arriba, puedes utilizar el método **stopPropagation** del objeto del evento.



```
// Bubbling

// La función que se ejecuta cuando se recibe el evento click, recibe
// como argumento el objeto event que la ha ejecutado
document.querySelector('#button').addEventListener('click', (event) => {
    console.log('Botón hijo fue clicado!');
    event.stopPropagation()
});
```

Con **stopPropagation**, si haces clic en el botón, solo verás el mensaje 'Botón hijo fue clicado!' y el *bubbling* se detendrá allí, evitando que el evento llegue al 'div contenedor'.



JavaScript + No-Library (pure JS) ▾

```
4 // Usamos la función getElementById para obtener el contenedor
5 document.getElementById('container').addEventListener('click', (event) => {
6     console.log('Div contenedor fue clicado!');
7 });
8 // La función que se ejecuta cuando se recibe el evento click, recibe como argumento el objeto event que la ha ejecutado
9 document.querySelector('#button').addEventListener('click', (event) => {
10     console.log('Botón hijo fue clicado!');
11     event.stopPropagation()
12 });
13
```

Haz clic en mí

>_ Console (beta) ⏹ 1 ⏹ 0 ⏹ 0 ⏹ 0 ⏹ 0 Clear console Minimize

"Botón hijo fue clicado!"

>_



Puedes profundizar más en detalle sobre el *bubbling* y eventos en la web oficial de [Mozilla](#).

Qué hemos aprendido



En esta exploración de los conceptos fundamentales de JavaScript, hemos adquirido una sólida comprensión de varios aspectos clave de la programación en este lenguaje, desde su creación como un lenguaje de *scripting* en el navegador hasta su ubicuidad en el desarrollo web y más allá.

- Hemos explorado cómo declarar **variables**, entender su alcance y cómo almacenar diferentes tipos de **datos** como números, cadenas, booleanos, cómo utilizar **operadores** para realizar cálculos y concatenar cadenas.
- Exploramos las **estructuras condicionales**, como *if-else* y *switch*, junto con los bucles *while*, *for* y *do-while*, que permiten controlar el flujo y la repetición en nuestros programas.
- También hemos abordado cómo declarar, llamar y utilizar **funciones** para encapsular bloques de código reutilizables y cómo los parámetros y retornos de valores agregan versatilidad.
- Por otra parte, ya comprendemos cómo los **objetos** agrupan propiedades y métodos para representar entidades y cómo utilizarlos para organizar y reutilizar la funcionalidad.
- Y para finalizar hemos explorado la creación y manipulación de **arreglos** para almacenar y trabajar con conjuntos de datos, qué es el **DOM**, cómo manipularlo y cómo registrarse a **eventos** para añadir funcionalidad a nuestras webs.

Estos conceptos forman los cimientos de la programación en JavaScript y son esenciales para crear aplicaciones interactivas y funcionales en la web y más allá.

La clave del buen desarrollador es que continúe explorando y practicando y así podrá construir aplicaciones más complejas y resolver desafíos cada vez mayores. En otras palabras: nunca pares de investigar y practicar por tu cuenta.

¡Enhorabuena! Fastbook superado



Qualentum.com