



Repaso de JavaScript

Centro de Servicios y Gestión Empresarial
SENA Regional Antioquia



www.sena.edu.co



Funciones Tipo Flecha

Concepto – Funciones Tipo Flecha

Las **funciones flecha** se utilizan principalmente cuando se requiere una sintaxis más concisa y cuando no es necesario un comportamiento propio del contexto de *this*.

Esto las hace ideales para funciones de orden superior como las que se pasan como *callback* en métodos como *map()*, *filter()*, y *reduce()*, o en promesas y eventos.

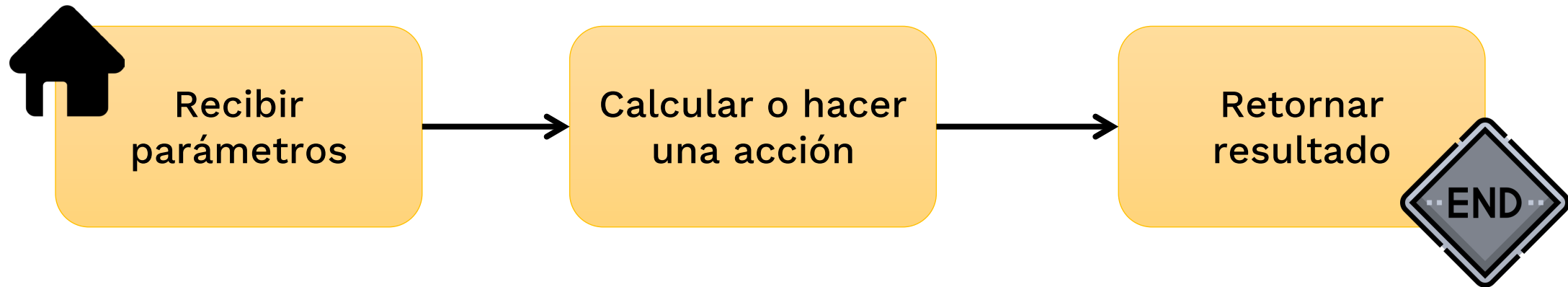
Concepto – Funciones Tipo Flecha

Las funciones *map()*, *filter()* y *reduce()*: son métodos muy útiles para manipular arrays de manera funcional.

- *map()*: **Transforma** cada elemento en el array y devuelve un nuevo array.
- *filter()*: **Filtra** elementos basados en una condición y devuelve un nuevo array con los elementos que cumplen la condición.
- *reduce()*: **Reduce** todos los elementos del array a un solo valor (por ejemplo, una suma o concatenación).

Concepto - Funciones Tipo Flecha

El Flujo de trabajo de una función flecha es:



Sintaxis – Funciones Tipo Flecha

- La sintaxis básica de una función flecha es:

```
(param1, param2, ..., paramN) => {  
    // cuerpo de la función  
}
```

- Si solo hay un parámetro, se pueden omitir los paréntesis, y si el cuerpo de la función contiene solo una expresión, se puede omitir el bloque {} y el return:

```
param => expresión
```

Sintaxis - Funciones Tipo Flecha

Función Tradicional



Función Flecha

```
function sumar(a, b) {  
    return a + b;  
}  
console.log(sumar(5, 3)); // 8
```

```
const sumar = (a, b) => a + b;  
console.log(sumar(5, 3)); // 8
```

Sintaxis – Funciones Tipo Flecha

- Si la función no tiene parámetros, se usa paréntesis vacíos.
- Si solo tienes un parámetro, se puede omitir los paréntesis.

```
const saludar = () => console.log('Hola');  
saludar();  
// Resultado: 'Hola'
```

```
const cuadrado = num => num * num;  
console.log(cuadrado(4));  
// Resultado: 16
```


Ejemplo – Método map()

map(): Este método recorre un array y aplica una función a cada uno de sus elementos, devolviendo un nuevo array con los resultados.

```
array.map(elemento => {  
    // lógica de la función  
});
```

Ejemplo:

```
const numbers = [1, 2, 3, 4, 5];  
const fivetime = numbers.map(num => num * 5);  
console.log(fivetime); // [5, 10, 15, 20, 25]
```

Ejemplo – Método reduce()

reduce(): Este método reduce un array a un solo valor, aplicando una función que acumula el resultado.

```
array.reduce((acumulador, elemento) => {  
    // lógica de la función  
}, valorInicial);
```

Ejemplo:

```
const numbers = [1, 2, 3, 4, 5];  
const sum = numbers.reduce((accumulator, num) =>  
    accumulator + num,  
0);  
console.log(sum); // 15
```

Ejemplo – Método filter()

filter(): Este método recorre un array y devuelve un nuevo array con los elementos que cumplen con una condición específica.

```
array.filter(elemento => {  
    // lógica de la función  
});
```

Ejemplo:

```
const numbers = [1, 2, 3, 4, 5];  
const evenNumbers = numbers.filter(num => num % 2 === 0);  
console.log(evenNumbers); // [2, 4]
```

Ejemplo – Funciones Tipo Flecha

La charcutería SENA, tiene con un array con las ventas diarias de la semana (*miles de pesos*) y requiere saber, cual es el total de las ventas de la semana.

Ventas diarias:

\$200, \$450, \$300, \$600, \$150, \$1.300,
\$800, \$4.500



Ejemplo - Funciones Tipo Flecha

Solución:

```
// Ventas de la empresa diarias
const ventasDiarias = [200, 450, 300, 600, 150,
  1300, 800, 4500];

// Calcular el total de ventas
// Método Reduce: Suma todos los elementos del array
// Parámetros: (Función reductora, valor inicial)
// Función reductora: (sumaTotal, ventaDiaria) => sumaTotal + ventaDiaria
// Valor inicial: 0
// Retorno: Total de ventas
const totalVentas = ventasDiarias.reduce(
  (sumaTotal, ventaDiaria) => sumaTotal + ventaDiaria, 0);

// Imprimir el Total de ventas: $8300
console.log(`Total de ventas: ${totalVentas}`);
```

Ejemplo – Funciones Tipo Flecha

Tu boleta, requiere un script para asignar los turnos de la compra de boletería para el concierto de Medellín al Parque de manera inteligente según la prioridad definida por la empresa (*Alta*).

Usuarios

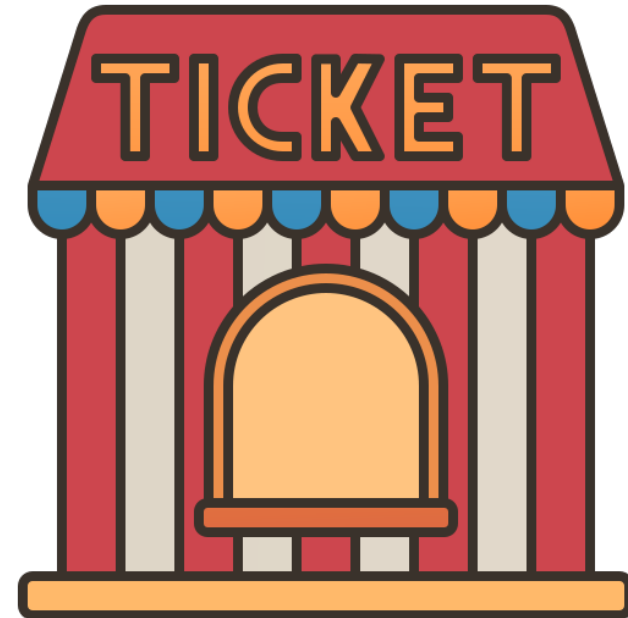
Camilo Diaz → Alta

Andres Sanchez → Media

Julia Zaens → Baja

Luisa Cruz → Media

Andrea Lopez → Alta





Solución:

```
// Usuarios en Fila de espera
const filaEspera = [
  {nombre: 'Camilo Diaz', prioridad: 'Alta'},
  {nombre: 'Andres Sanchez', prioridad: 'Media'},
  {nombre: 'Julia Zaens', prioridad: 'Baja'},
  {nombre: 'Luisa Cruz', prioridad: 'Media'},
  {nombre: 'Andrea Lopez', prioridad: 'Alta'}
];

// Turnos prioritarios
// Filtrar usuarios con prioridad Alta
// Mapear los nombres de los usuarios priorizados
// Parámetros: (Función de filtrado, Función de mapeo)
// Función de filtrado: usuario => usuario.prioridad === 'Alta'
// Función de mapeo: usuario => usuario.nombre
// Retorno: Nombres de los usuarios priorizados
const usuariosPrioritarios = filaEspera.filter(
  usuario => usuario.prioridad === 'Alta')
  .map(usuario => usuario.nombre);

// Imprimir los usuarios con prioridad Alta
console.log(`Los usuarios priorizados son: ${usuariosPrioritarios}`);
```

Funciones Tipo Flecha

Ventajas de Usar Funciones Flecha

- **Sintaxis más concisa:** Permite escribir funciones más rápidas y con menos código.
- **this léxico:** No crean su propio contexto de this, manteniendo el valor del this del contexto donde se declararon. Esto es útil en callbacks, eventos y promesas.
- **Menos verbosidad:** Son ideales para funciones de una sola línea o como callbacks.

Funciones Tipo Callback

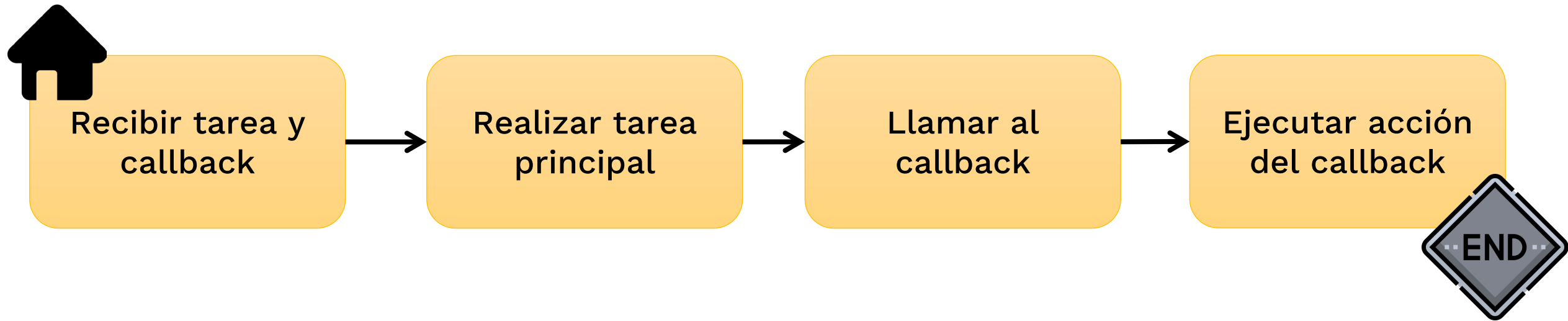
Concepto – Funciones Tipo callback

Un **callback** es una función que se pasa como argumento a otra función y se ejecuta después de que se completa una operación.

Los callbacks son fundamentales para la programación asíncrona, permitiendo ejecutar código después de que una tarea haya terminado.

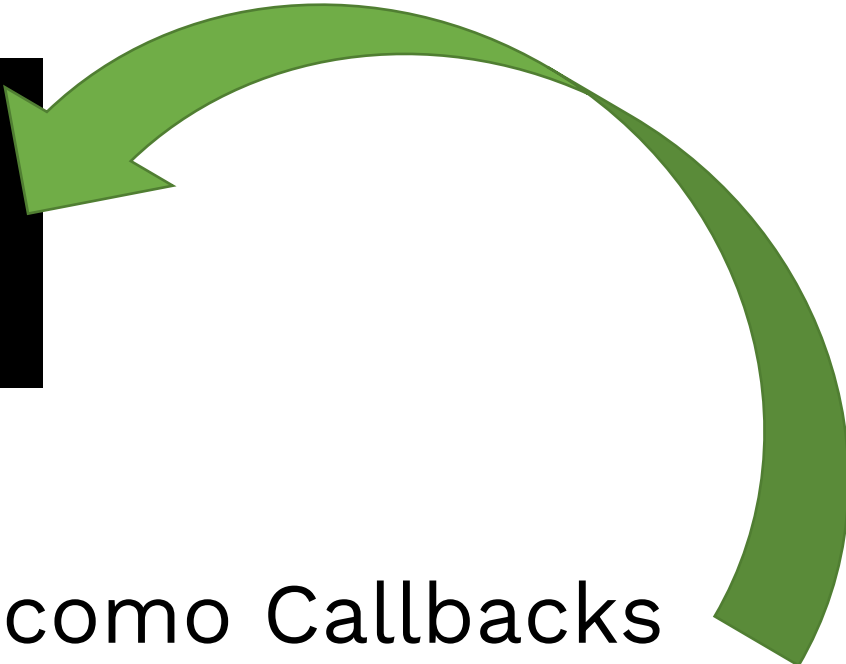
Concepto - Funciones Tipo callback


El Flujo de trabajo de una función tipo callback es:



Sintaxis – Funciones Tipo callback

```
function nombreFuncion(callback) {  
  // Lógica de la función principal  
  callback(); // Llamada al callback  
}
```

A large, thick green curved arrow originates from the right side of the code block and points towards the list of topics on the right.

- 
- A large, thick orange curved arrow originates from the bottom left of the code block and points towards the list of topics on the right.
- Funciones Flecha como Callbacks
 - Callbacks con Parámetros
 - Uso en Funciones Asíncronas
 - Manejo de Errores con Callbacks

Ejemplo - Funciones Tipo callback

```
// Primer paso: Declarar la función callback
function saludar(nombre) {
    // Quinto paso: Imprimir el mensaje
    console.log(`Hola, ${nombre}`);
}

// Segundo paso: Declarar la función que recibe el callback
function procesarEntradaUsuario(callback) {
    const nombre = "Juan";
    // Cuarto paso: Llamar la función callback
    callback(nombre);
}

// Tercer paso: Llamar la función que recibe el callback
procesarEntradaUsuario(saludar);
```

Ejemplo - Funciones Tipo callback

```
// Primer paso: Declarar la función realizarOperacion
function realizarOperacion(a, b, operacion) {
    // Cuarto paso: Llamar a la función operación - callback
    const resultado = operacion(a, b);
    console.log(`El resultado es: ${resultado}`);
}

// Segundo paso: Declarar la función sumar
const sumar = (x, y) => x + y;

// Tercer paso: Llamar a la función realizarOperacion
realizarOperacion(5, 3, sumar);
```

Ejemplo – Funciones Tipo callback

Funciones Flecha como Callbacks

```
// Primer paso: Declarar la función procesarEntradaUsuario
function procesarEntradaUsuario(callback) {
  const nombre = "Ana";
  // Tercer paso: Llamar la función callback
  callback(nombre);
}

// Segundo paso: Llamar la función procesarEntradaUsuario
procesarEntradaUsuario(
  // Cuarto paso: Declarar la función callback - tipo flecha
  (nombre) => {
    // Quinto paso: Imprimir el mensaje
    console.log(`Hola, ${nombre}`);
  }
);
```

Ejemplo - Funciones Tipo callback

Callbacks con Parámetros

```
// Primer paso: Definir la función que recibe el callback
function sumarNumeros(a, b, entregarResultado) {
    const resultado = a + b;
    // Cuarto paso: Llamar la función callback
    entregarResultado(resultado);
}

// Segundo paso: Definir la función callback
function mostrarResultado(res) {
    // Quinto paso: Mostrar el resultado del callback
    console.log(`El resultado es: ${res}`);
}

// Tercer paso: Llamar la función principal y pasarle el callback
sumarNumeros(5, 3, mostrarResultado);
```




```
// Paso 1: Crear la función principal
function iniciarSesion(usuario, password, validarDatos) {
    console.log("Procesando la información...");
    // Paso 4: Simular un tiempo de espera de 2 segundos
    setTimeout(() => {
        // Paso 5: Llamar a la función de validación
        validarDatos(usuario, password);
    }, 2000);
}

// Paso 2: Crear la función de validación
function validarDatos(usuario, password) {
    // Paso 6: Validar las credenciales
    if (usuario === "Alvaro" && password === "1234") {
        console.log("Usuario autenticado correctamente");
    } else {
        console.log("Usuario o contraseña incorrectos");
    }
}

// Paso 3: Llamar a la función principal
iniciarSesion("Alvaro", "1234", validarDatos);
// Intentando iniciar sesión con credenciales incorrectas
iniciarSesion("Alvaro", "wrongpassword", validarDatos);
```



```
// Paso 1: Crear la función principal
function iniciarSesion(usuario, password, validarDatos, manejarError) {
  // Paso 5: Verificar si los valores de inicio de sesión son válidos
  if (!usuario || !password) {
    manejarError("Usuario y contraseña son requeridos.");
    return; // Termina la función si hay un error
  }
  console.log("Procesando la información...");
  // Simulando un tiempo de espera de 2 segundos
  setTimeout(() => {
    // Paso 6: Llamar a la función de validación
    validarDatos(usuario, password);
  }, 2000);
}

// Paso 2: Crear la función de validación
function validarDatos(usuario, password) {
  // Paso 7: Validar las credenciales
  if (usuario === "Alvaro" && password === "1234") {
    console.log("Usuario autenticado correctamente");
  } else {
    // console.log("Usuario o contraseña incorrectos");
    manejarError("Usuario o contraseña incorrectos");
  }
}
```

Manejo de Errores con Callbacks

```
// Paso 3: Crear la función para manejar errores
function manejarError(mensaje) {
    console.log("Error:", mensaje);
}

// Paso 4: Llamar a la función principal con credenciales correctas
iniciarSesion("Alvaro", "1234", validarDatos, manejarError);

// Intentando iniciar sesión con credenciales incorrectas
iniciarSesion("Alvaro", "wrongpassword", validarDatos, manejarError);
// Intentando iniciar sesión sin usuario y contraseña
iniciarSesion("", "", validarDatos, manejarError);
```

Manejo de Errores

El bloque **try...catch** en JavaScript se utiliza para manejar errores de forma controlada. Esto te permite intentar ejecutar un bloque de código (**try**) y, si ocurre un error durante su ejecución, capturar el error en un bloque alternativo (**catch**) para manejarlo sin que el programa falle abruptamente.

```
try {  
    // Código que puede producir un error  
} catch (error) {  
    // Código para manejar el error  
} finally {  
    // (Opcional)  
    // Código que se ejecuta siempre, haya o no errores  
}
```

Manejo de Errores - Ejemplo

```
try {  
  const resultUno = 10 / 0;  
  console.log('Resultado es: ', resultUno);  
  
  let data = JSON.parse('');  
  console.log(data);  
  
} catch (error) {  
  console.log("Ocurrió un error:", error.message);  
} finally {  
  console.log("La aplicacion ha finalizado.");  
}
```

✎ Resultado es: Infinity

✎ Ocurrió un error:
Unexpected end of JSON
input

✎ La aplicacion ha
finalizado.

Manejo de Errores - Ejemplo

También puedes lanzar tus propios errores usando **throw** dentro de un bloque **try** si deseas manejar situaciones específicas.

```
try {  
    const resultUno = 10 / 0;  
    if (resultUno === Infinity) {  
        throw new Error('No se puede dividir por cero.');    }  
} catch (error) {  
    console.log("Ocurrió un error:", error.message);  
} finally {  
    console.log("La aplicacion ha finalizado.");  
}
```

Ocurrió un error: No se puede dividir por cero.

La aplicacion ha finalizado.



Promesas

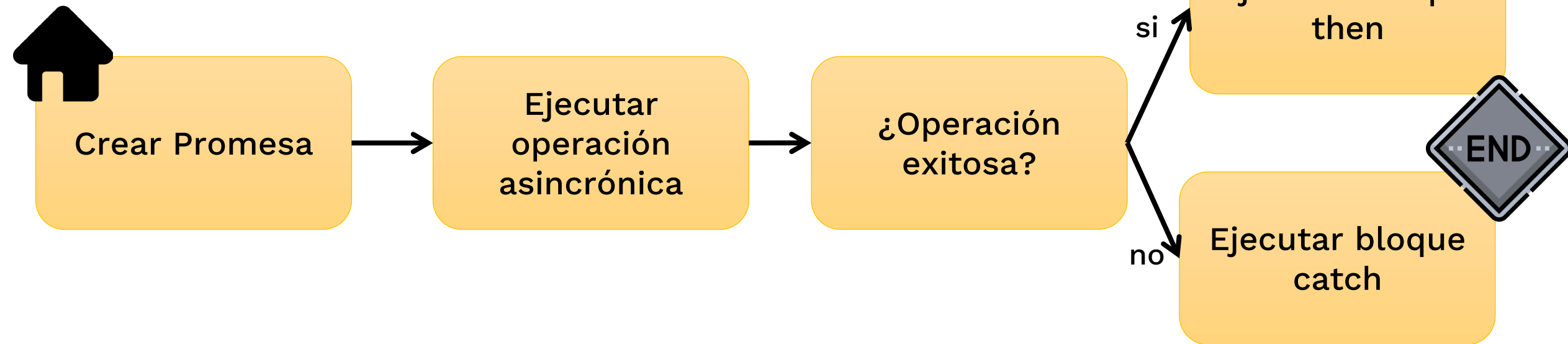
Concepto – Promesas

Las **promesas** en JavaScript son una forma de manejar operaciones asíncronas, ofreciendo una alternativa más estructurada a los *callbacks*. Una promesa representa un valor que puede estar disponible ahora, o en el futuro, o nunca. Es un objeto que puede estar en uno de tres estados:

- 1.Pendiente (Pending):** Estado inicial, la operación aún no se ha completado.
- 2.Cumplida (Fulfilled):** La operación se completó exitosamente, y se obtuvo un resultado.
- 3.Rechazada (Rejected):** La operación falló, y se obtuvo una razón (error).

Concepto - Promesas

El Flujo de trabajo de una Promesa:



Sintaxis – Promesas

Para **crear una promesa**, se utiliza el **constructor Promise**, que toma una función como argumento. Esta función recibe dos parámetros: **resolve** y **reject**. **resolve** se llama cuando la operación se completa con éxito, y **reject** se llama cuando hay un error.

```
const promesa = new Promise((resolve, reject) => {  
  // Simular una operación asincrónica  
  const exito = true; // Simular el resultado de la operación  
  
  if (exito) {  
    resolve("Operación completada exitosamente.");  
  } else {  
    reject("Ocurrió un error en la operación.");  
  }  
});
```

Sintaxis – Promesas

Para **manejar el resultado de una promesa** → **consumirla**, se utilizan los métodos **.then()**, **.catch()**, y **.finally()**:

.then() se utiliza para manejar el caso de éxito.

.catch() se utiliza para manejar el caso de error.

.finally() se ejecuta después de que la promesa se haya completado, independientemente de si fue cumplida o rechazada.

Las promesas ofrecen una forma eficiente y manejable de trabajar con operaciones asíncronas en JavaScript, mejorando la legibilidad del código y evitando complicaciones como la anidación excesiva de callbacks, que puede hacer el código difícil de seguir y mantener.

Sintaxis – Promesas

```
// Consumir la promesa
promesa
  .then((resultado) => {
    console.log(resultado); // Exito
  })
  .catch((error) => {
    console.error(error); // Error
  })
  .finally(() => {
    console.log("Esta línea se ejecuta
siempre.");
  });
```

Ejemplo – Promesas

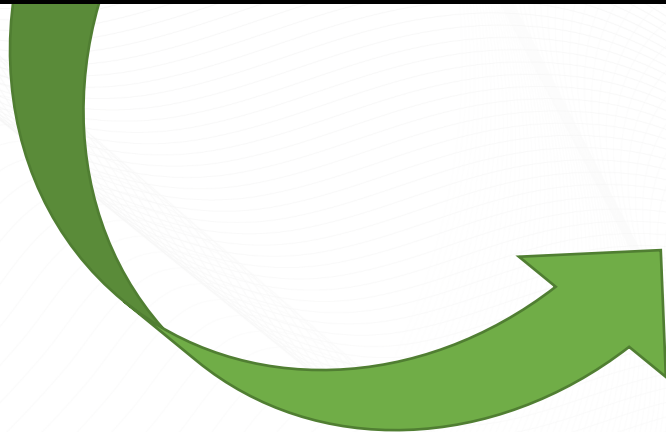


```
const promesa = new Promise((resolve, reject) => {  
  // Simular una operación asincrónica  
  const exito = true; // Simular el resultado de la  
  operación
```

```
  if (exito) {  
    resolve("Operación completada exitosamente.");  
  } else {  
    reject("Ocurrió un error en la operación.");  
  }  
});
```

```
// Consumir la promesa  
promesa
```

```
  .then((resultado) => {  
    console.log(resultado); // Exitoso  
  })  
  .catch((error) => {  
    console.error(error); // Error  
  })  
  .finally(() => {  
    console.log("Esta línea se ejecuta siempre.");  
  });
```



Ventajas – Promesas

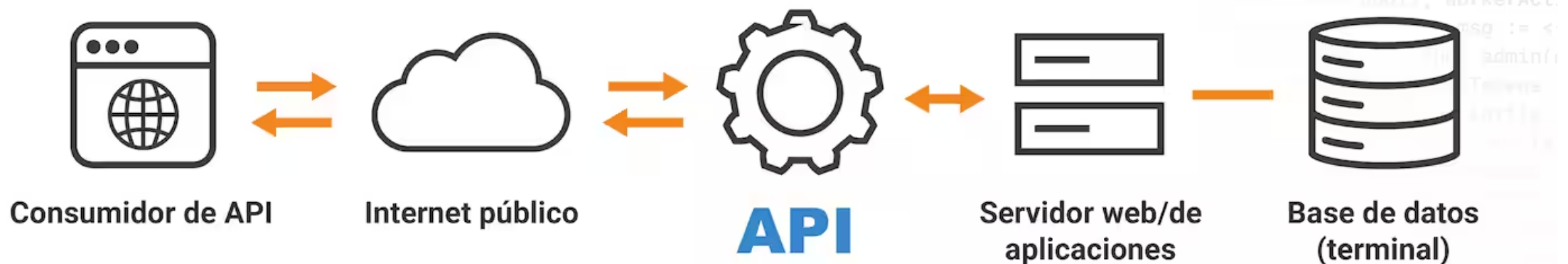
- **Manejo más claro del flujo asíncrono:** Las promesas permiten estructurar operaciones asíncronas de una manera más lineal y fácil de entender, evitando la anidación de callbacks.
- **Encadenamiento de operaciones:** Las promesas pueden encadenarse para ejecutar operaciones secuenciales con `.then()`.
- **Manejo centralizado de errores:** Puedes manejar todos los errores en un solo lugar con `.catch()`.
- **Paralelismo:** Se pueden realizar operaciones paralelas de manera eficiente y gestionar los resultados, con `Promise.all()` o `Promise.race()`



API

API – Concepto

Una **API** (*Interfaz de Programación de Aplicaciones*): es un conjunto de reglas y protocolos que permite la comunicación y el intercambio de datos entre diferentes software o servicios. Actúa como intermediario que permite que dos aplicaciones se "hablen" entre sí de manera estructurada, facilitando así el uso de funciones o datos de una aplicación en otra sin necesidad de conocer su implementación interna.



API – Características

- **Interfaz:** Proporciona un conjunto de métodos y herramientas que los desarrolladores pueden utilizar para interactuar con un servicio o una aplicación. Permitiendo acceder a funcionalidades específicas sin necesidad de comprender completamente el código subyacente.
- **Protocolos de Comunicación:** Las APIs pueden usar diferentes protocolos para la comunicación, siendo los más comunes:
 - **HTTP/HTTPS:** Utilizado en APIs web, donde se envían y reciben solicitudes a través de la web.
 - **REST** (*Representational State Transfer*): Un estilo arquitectónico que utiliza HTTP y es común en APIs modernas. Se basa en recursos y operaciones estándar (GET, POST, PUT, DELETE).
- **Formato de Datos:** Las APIs suelen utilizar formatos de datos estándar para enviar y recibir información, como:
 - **JSON** (*JavaScript Object Notation*): Un formato ligero y fácil de leer y escribir, muy utilizado en APIs web.
 - **XML** (eXtensible Markup Language): Un formato más antiguo y más verboso que JSON, utilizado en algunas APIs.

API – Operaciones estándar

Las operaciones estándar de una API REST son métodos HTTP que se utilizan para realizar diferentes tipos de acciones en los recursos que maneja la API.

- **GET:** Se utiliza para obtener información de un recurso específico o de una colección de recursos.
- **POST:** Se utiliza para crear un nuevo recurso en el servidor.
- **PUT:** Se utiliza para actualizar un recurso existente en el servidor. Puede reemplazar completamente el recurso o actualizar parcialmente.
- **DELETE:** Se utiliza para eliminar un recurso del servidor.

Estas operaciones son fundamentales para trabajar con **APIs RESTful**, permitiendo interactuar de manera efectiva con los datos y recursos que se gestionan en el servidor.

Ejemplo
JS

Ejemplos aplicación – Promesas

Llamadas a APIs o servidores (Axios, Fetch): Las promesas son muy útiles cuando se trabaja con operaciones de red, como solicitar datos a un servidor o una API externa. Estas operaciones no se resuelven inmediatamente y se necesita una forma de manejar el resultado cuando esté disponible.

```
// Realizar una petición a un servidor con Fetch y Promesas
fetch('https://api.example.com/datos')
  // Si la petición es exitosa se ejecuta el método then
  // que recibe la respuesta y la convierte a formato JSON
  .then(response => response.json())
  // Si la respuesta es correcta se imprime en consola
  .then(data => console.log(data))
  // Si ocurre un error se imprime en consola
  .catch(error => console.error('Error en la petición:', error));
```

Ejercicio de aplicación

```
// Nombre del pokemon
const nombrePokemon = 'bulbasaur';
// Realizar una petición a un servidor con Fetch y Promesas
fetch(`https://pokeapi.co/api/v2/pokemon/${nombrePokemon}`)
  // Si la petición es exitosa se ejecuta el método then
  // que recibe la respuesta y la convierte a formato JSON
  .then((response) => {
    if (!response.ok) {
      throw new Error('Pokémon no encontrado');
    }
    return response.json();
  })
  // Si la respuesta es correcta se imprime en consola
  .then((pokemonData) => {
    const abilitiesPokemon = pokemonData.abilities.map(
      (ability) => ability.ability.name).join(', ');
    console.log(`Las habilidades de ${nombrePokemon} son: ${abilitiesPokemon}`);
  })
  // Si ocurre un error se imprime en consola
  .catch(error => console.error('Error en la petición:', error));
```





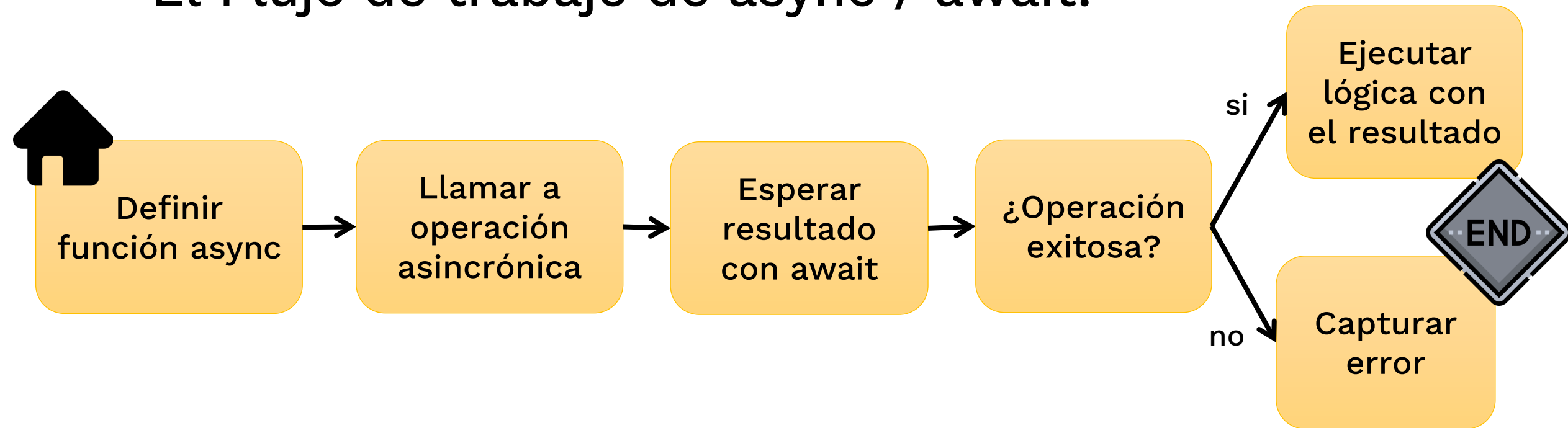
async / await

Concepto – async / await

- **async/await** es una mejora en JavaScript que facilita aún más el trabajo con código asíncrono.
- Permite escribir código asíncrono de manera similar al código síncrono, lo que mejora la legibilidad y facilita el manejo de operaciones asíncronas.

Concepto - async / await

El Flujo de trabajo de async / await:



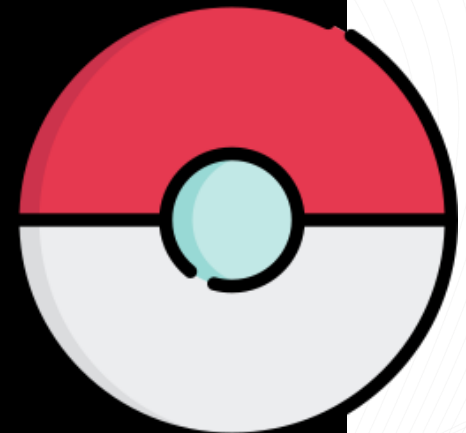
Concepto – `async` / `await`

- **`async`:** Declara una función que devuelve una promesa. Cualquier función marcada con `async` devolverá automáticamente una promesa, ya sea que internamente use promesas explícitas o no.
- **`await`:** Solo puede ser usado dentro de funciones `async`. Pausa la ejecución de la función `async` hasta que la promesa que sigue se resuelve (o rechaza).
- Una vez que la promesa se resuelve, el valor de la promesa se devuelve y la ejecución continúa.

Sintaxis – async / await

```
// Función asíncrona
async function nombreFuncion() {
  try { // Bloque try-catch para manejar errores
    // Lógica de la función
    // Esperar a que la promesa se resuelva
    const resultado = await promesa;
    // Lógica después de que la promesa se resuelve
    console.log(resultado); // Imprimir el resultado
  } catch (error) { // Capturar errores
    // Manejar el error
    console.error(error);
  }
}
```

```
// Nombre del pokemon
const nombredelPokemon = 'bulbasaur';
// Función asíncrona para obtener información de un Pokémon
async function obtenerInfoPokemon(nombre) {
  try {
    // Realizar una petición a un servidor con Fetch y Promesas
    const response = await fetch(`https://pokeapi.co/api/v2/pokemon/${nombre}`);
    if (!response.ok) { // Si la petición falla
      // Lanzar un error
      throw new Error('Pokémon no encontrado');
    }
    // Convertir la respuesta a formato JSON
    const pokemonData = await response.json();
    // Mapear las habilidades del Pokémon y unir las en un string
    const abilitiesPokemon = pokemonData.abilities.map(
      (ability) => ability.ability.name).join(', ');
    // Imprimir las habilidades del Pokémon
    console.log(`Las habilidades de ${nombre} son: ${abilitiesPokemon}`);
  } catch (error) { // Capturar errores
    // Imprimir el error en consola
    console.error('Error en la petición:', error);
  }
}
```



Ventajas – async / await

Cuándo usar async/await en el entorno de desarrollo:

- **Operaciones asíncronas complejas:** Ideal para manejar llamadas a APIs, interacciones con bases de datos o leer archivos de forma asíncrona en un servidor.
- **Mejora la legibilidad:** Usarlo en proyectos de gran escala donde la legibilidad y el manejo de errores son cruciales.
- **Secuencialidad clara:** Cuando necesitas que varias tareas asíncronas se ejecuten en un orden específico (una después de otra), async/await facilita ese flujo.
- **Simplificación de código:** Al refactorizar código con muchas promesas encadenadas o callbacks anidados, async/await ayuda a reducir la complejidad.

A white icon representing a network or object structure, consisting of a central node connected to six peripheral nodes by lines.

Desestructuración de objetos



Concepto – Desestructuración de objetos

La desestructuración de objetos en JavaScript es una característica que permite **extraer propiedades** de un objeto y **asignarlas a variables** de manera concisa. Esto facilita el acceso a los valores de los objetos sin necesidad de escribir varias líneas de código.

Sintaxis

```
const {propiedad1, propiedad2, propiedadN} = objeto;
```

Ejemplo – Desestructuración de objetos

```
// Declarar un objeto con propiedades
const persona = {
  nombre: 'Juan',
  edad: 25,
  ciudad: 'Bogotá'
};

// Desestructurar el objeto persona
const {nombre, edad, ciudad} = persona;

// Imprimir las propiedades desestructuradas
console.log(nombre); // Juan
console.log(edad); // 25
console.log(ciudad); // Bogotá
```



```
const persona = {
  nombre: 'Juan',
  edad: 25,
  ciudad: 'Bogotá'
};
const {nombre, edad} = persona;
console.log(nombre); // Juan
console.log(edad); // 25
```



Ejemplo – Desestructuración de objetos

Desestructuración con nombres de variables personalizados

```
// Declarar un objeto con propiedades
const equipoComputo = {
  marca: 'HP',
  modelo: 'Pavilion',
  procesador: 'Intel Core i5'
};

// Desestructurar el objeto equipoComputo
const {marca: marcaEquipo, modelo: modeloEquipo} = equipoComputo;

// Imprimir las propiedades desestructuradas
console.log(marcaEquipo); // HP
console.log(modeloEquipo); // Pavilion
```



Ejemplo – Desestructuración de objetos

Desestructuración con valores por defecto

```
// Declarar un objeto con propiedades
const aprendiz = {
  nombreApellido: 'Ana Cruz',
};

// Asignar valor por defecto
const { nombreApellido, genero = "F" } = aprendiz;

console.log(nombreApellido); // 'Ana Cruz'
console.log(genero);        // F
```




Desestructuración anidada

```
// Declarar un objeto con propiedades
const fruta = {
  nombreFruta: 'Manzana',
  color: 'Rojo',
  precio: 1.5,
  propiedades: {
    vitaminas: 'A, C',
    origen: 'Colombia'
  }
};
```

```
// Desestructurar el objeto fruta
const {nombreFruta, color, precio, propiedades: {vitaminas, origen}}
= fruta;
```

```
// Imprimir las propiedades desestructuradas
console.log(nombreFruta); // Manzana
console.log(color); // Rojo
console.log(precio); // 1.5
console.log(vitaminas); // A, C
console.log(origen); // Colombia
```

Ejemplo – Desestructuración de objetos

Desestructuración con el resto de las propiedades (...rest)

```
// Declarar un objeto con propiedades
const docente = {
  nombres: 'Luis Dario',
  edad: 32,
  ubicacion: 'Sevilla',
  profesion: 'Ingeniero'
};

// Desestructurar algunas propiedades y capturar el resto
const { nombres, ubicacion, ...resto } = docente;

// Imprimir las propiedades desestructuradas
console.log(nombre); // 'Luis'
console.log(ciudad); // 'Sevilla'
console.log(resto); // { edad: 32, profesion: 'Ingeniero' }
```

Ventajas – Desestructuración de objetos

- Legibilidad: Evita el uso repetitivo de objeto.propiedad y mejora la legibilidad del código.
- Concisión: Reduce la cantidad de código necesario para extraer múltiples propiedades de un objeto.
- Flexibilidad: Permite extraer valores específicos, renombrar propiedades, usar valores predeterminados, y obtener el resto de las propiedades fácilmente.



Proyecto de repaso

Proyecto de repaso

Crear el front-end con (*html, css y bootstrap*), para consumir una API con Javascript. Únicamente implementar el método GET.

Users from JSONPlaceholder

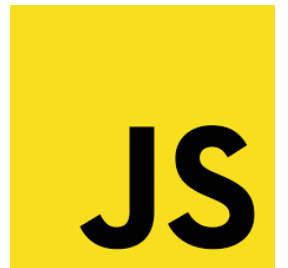
ID	Name	Username	Email	Website
1	Leanne Graham	Bret	Sincere@april.biz	hildegard.org
2	Ervin Howell	Antonette	Shanna@melissa.tv	anastasia.net
3	Clementine Bauch	Samantha	Nathan@yesenia.net	ramiro.info
4	Patricia Lebsack	Karianne	Julianne.OConner@kory.org	kale.biz
5	Chelsey Dietrich	Kamren	Lucio_Hettinger@annie.ca	demarco.info
6	Mrs. Dennis Schulist	Leopoldo_Corkery	Karley_Dach@jasper.info	ola.org
7	Kurtis Weissnat	Elwyn.Skiles	Telly.Hoeger@billy.biz	elvis.io
8	Nicholas Runolfsdottir V	Maxime_Nienow	Sherwood@rosamond.me	jacynthe.com
9	Glenna Reichert	Delphine	Chaim_McDermott@dana.io	conrad.com
10	Clementina DuBuque	Moriah.Stanton	Rey.Padberg@karina.biz	ambrose.net

Front-end



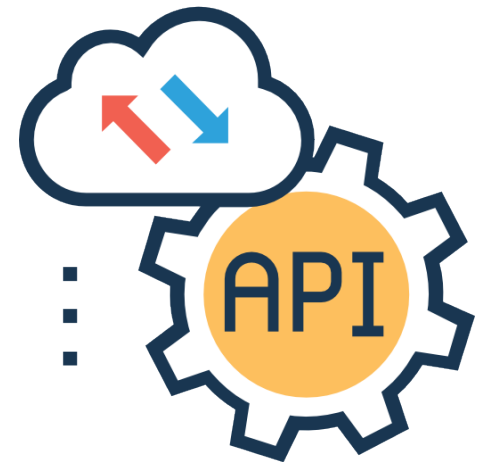
Consumo API

- Función para obtener información de la API
- Función para mostrar la información de la API en el Frontend.



APIs Sugeridas

- PokéAPI → <https://pokeapi.co/api/v2/pokemon/>
- Rick&Morty → <https://rickandmortyapi.com/api/character/>
- ReqRes → <https://reqres.in/api/users>
- Fake Store API → <https://fakestoreapi.com/products>
- The Dog API → <https://api.thedogapi.com/v1/breeds>
- The Cat API → <https://api.thecatapi.com/v1/breeds>
- CoinGecko API → <https://api.coingecko.com/api/v3/coins/list>





GRACIAS

Presentó: Alvaro Pérez Niño
Instructor Técnico

Correo: aperezn@misena.edu.co

<http://centrodeserviciosygestionempresarial.blogspot.com/>

Línea de atención al ciudadano: 01 8000 910270

Línea de atención al empresario: 01 8000 910682



www.sena.edu.co