# CallBacks

function ask(question, yes, no) {

if(confirm(question)) yes;

else no;

}

function yes() {

console.log("GRacias por aceptar");

}

function no() {

console.log("Otra vez será");

}

ask("¿Quieres jugar conmigo?", yes(), no());

GRacias por aceptar

Otra vez será

undefined

Tu código tiene un error de sintaxis y lógica al pasar las funciones yes y no como argumentos a ask(). Vamos a analizarlo:

**Errores en tu código:**

1. **Llamado incorrecto a las funciones yes y no en ask()**
2. ask("¿Quieres jugar conmigo?", yes(), no());
   * Estás llamando (yes(), no()) a las funciones en lugar de pasarlas como referencias.
   * Esto significa que yes() y no() se ejecutarán inmediatamente al evaluar ask(...), antes de que confirm(question) se ejecute.
3. **Problema en ask()**
4. function ask(question, yes, no) {
5. if (confirm(question)) yes;
6. else no;
7. }
   * Aunque yes y no se pasen correctamente, el código no los ejecuta porque yes y no son funciones, pero en ask() no se están llamando con ().

**Corrección del código:**

Corrigiendo los problemas, el código debería verse así:

function ask(question, yes, no) {

if (confirm(question)) yes(); // Llamamos a la función yes

else no(); // Llamamos a la función no

}

function yes() {

console.log("Gracias por aceptar");

}

function no() {

console.log("Otra vez será");

}

ask("¿Quieres jugar conmigo?", yes, no); // Pasamos las funciones sin los paréntesis

**Explicación de la corrección:**

1. **Pasar funciones como referencias:**
   * ask("¿Quieres jugar conmigo?", yes, no); → Pasa yes y no sin ejecutarlas inmediatamente.
2. **Llamar a las funciones dentro de ask():**
   * yes(); y no(); dentro de ask() aseguran que las funciones se ejecuten en el momento correcto.

**Ejemplo de ejecución:**

Si el usuario **acepta** en el cuadro de diálogo confirm, verá en la consola:

Gracias por aceptar

Si el usuario **cancela**, verá:

Otra vez será

Esta corrección asegura que el código funcione como se espera. 🚀

Podemos hacr lo anterior con function expresions

function ask(question,

yes = function(){console.log('yes');},

no = function(){console.log('no');})

{

if(confirm(question)) yes();

else no();

}

ask('Quieres comer');

# Promesas

new Promise ((resolve, reject) =>{

setTimeout(() => resolve('result'), 3000);

})

.finally(()=> alert('Promise ready'))

.then(result => alert(result));

Este código en JavaScript crea y maneja una **promesa** (Promise) con los métodos .finally() y .then(). Vamos a analizarlo paso a paso:

**1️⃣ Creación de la promesa:**

new Promise((resolve, reject) => {

setTimeout(() => resolve('result'), 3000);

})

* Se crea una nueva **promesa** con dos parámetros: resolve y reject.
* Dentro de la promesa, se usa setTimeout para simular una operación asíncrona que dura 3 segundos (3000 ms).
* Una vez pasados esos 3 segundos, la promesa se **resuelve** (resolve('result')), enviando el valor 'result'.

**2️⃣ Uso de .finally()**

.finally(() => alert('Promise ready'))

* .finally() se ejecuta **siempre**, independientemente de si la promesa se resuelve (resolve) o se rechaza (reject).
* En este caso, cuando pasan los 3 segundos, se mostrará una alerta con 'Promise ready'.
* **Importante**: .finally() no modifica el resultado de la promesa, solo se ejecuta código adicional.

**3️⃣ Uso de .then()**

.then(result => alert(result));

* .then() se ejecuta **después de que la promesa se haya resuelto**.
* Recibe el valor resuelto de la promesa ('result') y lo muestra en una alerta (alert(result)).

**🕒 Orden de ejecución:**

1. Se crea la promesa y se ejecuta setTimeout (espera 3 segundos).
2. **Después de 3 segundos**, la promesa se **resuelve** (resolve('result')).
3. **.finally() se ejecuta primero** → Muestra la alerta 'Promise ready'.
4. Luego, .then() recibe el resultado 'result' y muestra la alerta con 'result'.

**🔥 Salida esperada en la pantalla:**

Después de 3 segundos, se mostrarán dos alertas en este orden:

1️⃣ Alert: "Promise ready"

2️⃣ Alert: "result"

**📌 Resumen:**

* La promesa se ejecuta y espera 3 segundos.
* .finally() se ejecuta sin importar el resultado.
* .then() recibe y muestra el valor resuelto.

Si en lugar de resolve, hubiéramos usado reject, **.finally() se ejecutaría igual**, pero .then() no se activaría (a menos que haya un .catch()).

El código tiene un **error**, y analizarlo nos ayudará a entender mejor cómo funcionan las promesas en JavaScript.

**🔍 Análisis paso a paso**

new Promise((resolve, reject) => {

let m = document.firstChild;

document.onload(() => resolve(m)); // ❌ ERROR aquí

})

.finally(() => alert('Promise ready'))

.then(result => console.log(m)); // ❌ ERROR aquí también

**🛑 Errores en el código**

1. **Uso incorrecto de document.onload**
2. document.onload(() => resolve(m));
   * **Error:** document.onload **no es una función**, es una propiedad.
   * **Corrección:** Debe asignarse como una función, así:
   * document.onload = () => resolve(m);
   * Sin embargo, document.onload **ya se ejecutó cuando este código corre**, lo que significa que resolve(m) **nunca se ejecutará** porque la promesa depende de un evento que no ocurrirá nuevamente.
3. **Uso incorrecto de console.log(m)**
4. .then(result => console.log(m));
   * **Error:** m está **fuera de alcance** (m fue declarado dentro de la función new Promise).
   * **Corrección:** Se debe usar el parámetro result de .then(), así:
   * .then(result => console.log(result));

**✅ Corrección del código**

Si queremos resolver la promesa cuando el documento haya terminado de cargar, podemos hacer esto:

new Promise((resolve, reject) => {

let m = document.firstChild;

window.onload = () => resolve(m);

})

.finally(() => alert('Promise ready'))

.then(result => console.log(result));

**🛠 Explicación de la corrección**

1. **Se usa window.onload en lugar de document.onload**, ya que es la forma correcta de detectar cuando el documento ha cargado completamente.
2. **Se usa el parámetro result dentro de .then()**, en lugar de la variable m, para evitar problemas de alcance.

**🔥 Flujo de ejecución**

1. Se crea la promesa.
2. Se almacena document.firstChild en m.
3. Se espera a que la ventana termine de cargar (window.onload).
4. Cuando la página se carga:
   * La promesa se resuelve con el valor m.
   * **.finally()** se ejecuta y muestra alert('Promise ready').
   * **.then()** recibe m y lo muestra en la consola.

**📌 Salida esperada en la consola**

Después de que la página termine de cargar:

1️⃣ Alert: "Promise ready"

2️⃣ (En consola) <html>...</html> // O el primer nodo del documento (puede variar)

**🚀 Conclusión**

* **No se puede llamar document.onload() como si fuera una función**.
* **window.onload es la forma correcta** de ejecutar código cuando la página carga.
* **Se debe usar result dentro de .then()** en lugar de intentar acceder a variables de fuera de su alcance.

Con esta corrección, el código funcionará correctamente. ✅

# Manipulación del document

El código tiene un **error lógico** que hace que el bucle while **nunca se ejecute**. Vamos a analizarlo paso a paso.

**🔍 Análisis del código**

let elem = document.body;

while (elem == elem.parentElement) {

console.log(elem);

}

**Explicación línea por línea**

1. **Se obtiene el elemento <body> del documento**:
2. let elem = document.body;
   * document.body devuelve el elemento <body> del documento.
3. **Se evalúa la condición del while**:
4. while (elem == elem.parentElement)
   * elem.parentElement devuelve el **elemento padre de elem**.
   * Como elem es document.body, su parentElement es <html>.
   * **Comparación (==)**:
   * document.body == document.documentElement // false
     + <body> **no es igual a** <html>, por lo que la condición **es falsa** desde el inicio.
5. **El bucle while no se ejecuta**:
   * Dado que document.body != document.documentElement, el while nunca entra en su bloque de código.
   * **El console.log(elem) jamás se ejecuta.**

**🔥 Conclusión**

* **El bucle while nunca se ejecutará** porque document.body == document.body.parentElement **siempre es falso**.
* **Si el objetivo era recorrer los ancestros del elemento body hasta la raíz del documento (<html>), entonces while debe usar != en lugar de ==.**

**✅ Corrección para recorrer los ancestros del body**

Si queremos recorrer los elementos padres hasta document.documentElement, podemos hacer esto:

let elem = document.body;

while (elem.parentElement != null) {

console.log(elem);

elem = elem.parentElement;

}

**🔹 ¿Cómo funciona esta corrección?**

* Se inicia con elem = document.body.
* El while se ejecuta **mientras elem.parentElement no sea null**.
* En cada iteración:
  1. Se imprime elem en la consola.
  2. Se actualiza elem = elem.parentElement (subiendo un nivel en el DOM).
* Cuando elem.parentElement sea null, el bucle se detiene.

**🔥 Salida esperada en consola**

<body>...</body>

<html>...</html>

Se imprimen el <body> y luego el <html>, ya que este es el **último elemento padre** antes de null.

**📌 Resumen**

1. **Error original**: while (elem == elem.parentElement) siempre es false, así que el bucle nunca se ejecuta.
2. **Solución**: Usar while (elem.parentElement != null) para recorrer los elementos padres.
3. **Salida esperada**: Se imprimen <body> y <html> antes de detenerse.

Así, el código logra su propósito correctamente. 🚀✅

Si quieres recorrer **todos los elementos dentro del <body>** (de afuera hacia adentro), puedes hacerlo con **un recorrido en profundidad (Depth-First Search, DFS)**. Aquí hay varias formas de hacerlo:

**✅ Opción 1: Recorrido con un bucle while y firstElementChild**

Esta versión usa firstElementChild y nextElementSibling para recorrer todos los elementos en orden.

let elem = document.body;

let stack = [elem]; // Usamos un stack para simular un recorrido en profundidad

while (stack.length > 0) {

let current = stack.pop(); // Tomamos el último elemento del stack

console.log(current); // Mostramos el elemento actual

// Agregamos los hijos al stack en orden inverso para que se procesen en el orden correcto

let child = current.lastElementChild;

while (child) {

stack.push(child);

child = child.previousElementSibling;

}

}

**🔥 Explicación**

1. Se inicia con el <body>.
2. Se usa un **stack** (stack[]) para manejar el recorrido.
3. En cada iteración:
   * Se saca el último elemento del stack y se imprime.
   * Se añaden sus hijos al stack, **de derecha a izquierda**, para que se impriman en el orden correcto.
4. El bucle termina cuando ya no quedan elementos en el stack.

🔹 **Salida esperada en la consola (depende del HTML de la página)**:

<body>

<div>

<p>Texto</p>

</div>

<span>Otro texto</span>

</body>

Salida en consola:

<body>

<div>

<p>

#text

</p>

</div>

<span>

#text

</span>

</body>

**✅ Opción 2: Recorrido con Recursión (DFS)**

Si prefieres un enfoque recursivo, puedes hacerlo así:

function traverse(elem) {

console.log(elem); // Mostrar el elemento actual

let child = elem.firstElementChild;

while (child) {

traverse(child); // Llamada recursiva para el siguiente hijo

child = child.nextElementSibling;

}

}

traverse(document.body);

**🔥 Explicación**

1. Se imprime el elemento actual.
2. Se recorre cada hijo con firstElementChild y nextElementSibling.
3. La función se llama a sí misma para recorrer todos los elementos internos.

🔹 **Salida esperada** → Igual que la versión anterior.

**📌 Comparación de Métodos**

| **Método** | **Pros** | **Contras** |
| --- | --- | --- |
| **Bucle while con stack (iterativo)** | ✅ Más eficiente en memoria (evita sobrecarga de la pila de llamadas).✅ Evita problemas de profundidad en el DOM. | ❌ Más código (gestión manual del stack). |
| **Recursión (DFS)** | ✅ Código más limpio y fácil de entender. | ❌ Puede consumir más memoria si el DOM es muy profundo (stack overflow). |

**🚀 Conclusión**

* Si el DOM es **pequeño o mediano**, usa **recursión** (traverse(elem)).
* Si el DOM es **muy grande**, usa el **bucle con stack** para evitar problemas de memoria.

Ambas soluciones recorren todos los elementos **del <body> hacia adentro** correctamente. 🔥🚀