8. ESTRUCTURAS DE CONTROL DE FLUJO DE EJECUCION

Introducción a las estructuras de control

JavaScript tiene un conjunto de **estructuras de control / estructuras de flujo de control** similar al de otros lenguajes de la familia C.

- **secuencia:** el carácter de punto y coma (;) se usa para separar declaraciones en código JavaScript
  ```
  sentencia;
  sentencia;
  sentencia;
  ...
  ```

- **bloque**

- **sentencias condicionales**
  - `if`, `switch`
  - gestión de errores (***exception handling***): `try`...`catch`...`finally`

- sentencias de repetición: `for` ..., `for` ... `of`, `for` ... `in`, `while`, `do` ... `while`, `break`, `continue`

.
Una **expresión** de JavaScript también es una declaración: a diferencia de algunos lenguajes como Rust, las **estructuras de flujo de control** son **declaraciones** en JavaScript, lo que significa que no puedes asignarlas a una variable (aunque sí crear una función), como

```
const a = if (x) { 1 } else { 2 }
```

Declaración de bloque

La sentencia más básica es una ***sentencia de bloque,*** que se utiliza para agrupar sentencias.

- El **bloque** está delimitado por un par de corchetes:

```
{
  declaración1;
  declaración2;
  // …
  declaraciónN;
}
```

Ejemplo

Las declaraciones de bloque se usan comúnmente con **declaraciones de flujo de control** (`if`, `for`, `while`).

```
while (x < 10) {
  x++;
}
```

Aquí, `{x++; }` es la sentencia del bloque.

Nota: Las variables declaradas `var` no tienen un **alcance de bloque** (***block scope***), sin que tienen **alcance de función** o script que las contiene, y los efectos de establecerlas persisten más allá del bloque en sí. Por ejemplo:

```
var x = 1;
{
   var x = 2;
}
console.log(x); // 2
```

- esto da como resultado `2` porque la declaración `var x` dentro del bloque está en el mismo alcance que la declaración `var x = 1` antes del bloque. *(En C o Java, el código equivalente tendría la salida `1`).*
- este efecto de alcance se puede mitigar usando `let` o `const`

Declaraciones condicionales

Una **declaración condicional**:

- es un bloque de sentencias que se ejecutan si una **condición** especificada evalúa a `true`.
- JavaScript admite dos declaraciones condicionales: `if...else` y `switch`.

Sentencia `if...else`

Usa el la declaración `if` para ejecutar una declaración si una **condición lógica** evalúa a `true`.

- la cláusula `else` opcional para ejecutar una declaración si la condición evalúa a `false`.

Una declaración `if` se ve así:

```
if (condición) {
   declaración1;
} else {
   declaración2;
}
```

- aquí, la ***condición*** puede ser cualquier **expresión** que se evalúe como `true` o `false`.
- si la **condición** se evalúa como `true`, se ejecuta la ***declaración1***.
- de lo contrario, se ejecuta la ***declaración2***.
- ***declaración1*** y ***declaración2*** pueden ser cualquier declaración, incluidas otras declaraciones `if` **anidadas**.
- se ejecutará la primera condición lógica que se evalúe como `true`.
- para ejecutar varias sentencias, agrúpelas dentro de una sentencia de bloque (`{ /*` … `*/ }`).

Combinación (es también un anidamiento, pero por la parte del `else`)

Puedes **combinar** las declaraciones `if...else` para probar múltiples condiciones en secuencia, de la siguiente manera.

```
if (condición1) {
  sentencia1;
} else if (condición2) {
  sentencia2;
} else if (condición3) {
  sentencia3;
} else {
  sentencia4;
}
```

También se puede escribir así, pero es menos recomendable pues aparece el código muy sangrado dificultando si legibilidad:

```
if (condición1) {
  sentencia1;
} else
    if (condición2) {
      sentencia2;
    } else
      if (condición3) {
       sentencia3;
      } else {
       sentencia4;
      }
```

La semántica de la anterior expresión es: para que se ejecute:

- la *sentencia1* : tendrá que cumplirse la *condición1* y no se ejecutan más sentencias
- la *sentencia2* : tendrán que cumplirse al mismo tiempo *condición1* ***es falsa,*** *condición2* ***es verdadera*** y no se ejecutan más sentencias
- la *sentencia3* : tendrán que cumplirse al mismo tiempo *condición1* ***es falsa,*** *condición2* ***es falsa*** , *condición3* ***es verdadera*** y no se ejecutan más sentencias
- la *sentencia4* : todas las condiciones deben ser falsas para que se ejecute.

Anidamiento (***nesting***)

Las declaraciones `if` también se pueden anidar: en el ejemplo siguiente, para que se ejecute:

- la *sentencia1* : tendrán que cumplirse al mismo tiempo la *condición1*, *condición2* y *condición3*:
- la *sentencia2* : tendrán que cumplirse al mismo tiempo la *condición1*, *condición2* y *ser falsa* *condición3*:
- la *sentencia3* : tendrán que cumplirse al mismo tiempo la *condición1*, *condición2*
- la *sentencia4* : tendrán que cumplirse al mismo tiempo la *condición1* *y falsa* *condición2* .

Aunque algunas llaves no son imprescindibles, pues hay una única sentencia dentro de los `if`, por **legibilidad** y evitar futuros errores al modificar/mantener el código, es mejor ponerlas:

```
if (condición1) {
    if (condición2) {
        if (condición3) {
            sentencia1;
        } else {
            sentencia2;
        }
        sentencia3;
    } else {
        sentencia4;
    }
}
```

Buenas prácticas

En general, es una buena práctica usar siempre **sentencias de bloque**, *especialmente* cuando se anidan sentencias `if` :

```
if (condición) {
    // Declaraciones para cuando la condición es true
    // …
} else {
    // Declaraciones para cuando la condición es falsa
    // …
}
```

En general, es una buena práctica no tener un `if...else` con una **asignación** como `x = y` como condición:

```
if (x = y) {
    /* declaraciones aquí */
}
```

valores falsy ("falsetes")

Los siguientes **valores evalúan a `false`** (también conocido como **valores false**):

- todos los demás valores, incluidos todos los objetos, se evalúan como `true` cuando se pasan a una declaración condicional.

Valor	Descripción
`false`	La palabra clave `false` .
`0`	El Número cero (entonces, también 0.0 , etc., y 0x0).
`-0`	El Número menos cero (entonces, también -0.0 , etc., y -0x0).
`0n`	El BigInt cero (entonces, también 0x0n). Tenga en cuenta que no hay un cero negativo de BigInt : la negación de 0n es 0n .

`""`
`''`
`` ``
	Valor de cadena vacía.

`null`	null — la ausencia de referencia a objeto.
`undefined`	`undefined` — el valor primitivo.
`NaN`	NaN: no es un número.
`document.all`	Los objetos son falses si y solo si tienen la ranura interna `[[IsHTMLDDA]]`. Dicha ranura solo existe en `document.all` y no se puede configurar mediante JavaScript.

Nota: No confunda los **valores booleanos primitivos** `true` y `false` con los valores **true** y **false** de un valor **objeto** `Boolean`

- por ejemplo:

```
const b = new Boolean(false);
if (b) {
   // esta condición se evalúa como true
}
   if (b == true) {
      // esta condición se evalúa como falsa (debido a que objeto Boolean(false)
      //  es coercionado/promovido automáticamente a primitivo false con el operador ==
}
```

Ejemplo

En el siguiente ejemplo, la función `comprobarDatos` devuelve

- `true` si el número de caracteres en un objeto de `texto` es tres.
- de lo contrario, muestra una alerta y devuelve `false`.

```
<form name="formulario1">
   <input type="text" name="trescaracteres" onchange="comprobarDatos();" />
</form>
```

```
function comprobarDatos() {
   if (document.formulario1.trescaracteres.value.length === 3) {
     return true;
   } else {
     alert(`Ingrese exactamente tres caracteres.
          ${document.formulario1.trescaracteres.value.length} no es válido.`);
     return false;
   }
}
```

Sentencia `switch`

La **sentencia** `switch`:

- permite que un programa **evalúe una expresión** e intente hacer coincidir el valor de la expresión con una **etiqueta** `case`: si se encuentra una coincidencia, el programa ejecuta la instrucción asociada.
- una declaración de `switch` se ve así:

```
switch (expresión) {
  case etiqueta1:
    sentencias1;
    break;
  case etiqueta2:
    sentencias2;
    break;
      // …
  default:
    sentenciasPorDefecto;
}
```

JavaScript evalúa la *expresión* y luego procede de la siguiente manera:

- el programa primero busca una cláusula de `caso` con una **etiqueta** que coincida con el valor de *expresión* y luego **transfiere el control** a esa **cláusula**, ejecutando las sentencias asociadas.
- si no se encuentra ninguna etiqueta coincidente, el programa busca la **cláusula** `default`, que es opcional :
  - si se encuentra una cláusula `default`, el programa **transfiere el control** a esa cláusula, ejecutando las sentencias asociadas.
  - si no se encuentra ninguna cláusula `default`, el programa reanuda la ejecución en la sentencia que sigue al final de `switch`.
  - por convención, la cláusula `default` se escribe como la última cláusula, pero no es necesario que sea así.

Ejemplo:

```
const accion = "comer";

switch (accion) {
  case "dibujar":
    console.log("dibujando ....");
    break;
  case "comer":
    console.log("comiendo ....");
    break;
  default:
   console.log("descansando ....");
}
```

Si no ponemos la sentencia **break** al final de una cláusula **case**, se continuará ejecutando las instrucciones de la siguiente cláusula **case**.

- en cada cláusula **case** se puede utilizar como **etiqueta** cualquier expresión, y no sólo una cadena o número, y serán evaluadas una a una en tiempo de ejecución.
- cada comparación entre la *expresión* del **switch** y la etiqueta de una cláusula **case**, se realiza utilizando el operador de igualdad estricta **===**.

Ejemplo

En el siguiente ejemplo

- si **fruitType se** evalúa como **'Bananas'**, el programa hace coincidir el valor con el caso **'Bananas'** y ejecuta la instrucción asociada.
- cuando se encuentra una **interrupción, el programa sale del interruptor** y **continúa la ejecución** desde la instrucción que sigue al **interruptor.**
- si se omitiera **break,** también se ejecutaría la declaración para **el caso 'Cerezas'.**

```
switch (tipoFruta) {
  case 'Naranjas':
    console.log('Las naranjas cuestan 0.59€ el kilo.');
    break;
  case 'manzanas':
    console.log('Las manzanas cuestan 0.32€ el kilo.');
    break;
  case 'Plátanos':
    console.log('Las bananas cuestan 0.48€ el kilo.');
    break;
  case 'Cerezas':
    console.log('Las cerezas cuestan 3.00€ el kilo.');
    break;
  case 'Mangos':
    console.log('Los mangos cuestan 0.56€ el kilo.');
    break;
  case 'Papayas':
    console.log('Los mangos y las papayas cuestan 2.79€ el kilo.');
    break;
  default:
    console.log(`Lo sentimos, no tenemos ${tipoFruta}.`);
}
```

Declaraciones de manejo de excepciones

Puede **lanzar excepciones** (***throw exceptions***) usando la **declaración** `throw` y **manejarlas** (***handle***) usando las **declaraciones** `try...catch.`

Tipos de excepción

Casi cualquier objeto se **puede lanzar** en JavaScript.

- sin embargo, no todos los objetos lanzados son iguales.
- si bien es común arrojar números o cadenas como errores, con frecuencia es más efectivo usar uno de los **tipos de excepción** creados específicamente para este propósito:
    - Excepciones ECMAScript
    - `DOMException` y `DOMError`

Sentencia `throw`

Utilice la sentencia `throw` para **lanzar una excepción.**

- una declaración `throw` especifica el valor que se lanzará:

```
throw expresión ;
```

- puede lanzar cualquier expresión, no solo expresiones de un tipo específico.
- el siguiente código arroja varias excepciones de diferentes tipos:

```
throw 'Error2';                   // tipo de cadena
throw 42;                         // tipo de numero
throw true;                       // tipo booleano
throw {toString() { return "¡Soy un objeto!"; } };
```

Sentencia `try...catch`

La **sentencia** `try... catch`

- marca un **bloque de declaraciones** para vigilar si se produce una exepción en alguna sentencia del bloque y especifica una o más respuestas en caso de que se produzca una excepción.
- si se lanza una excepción, la sentencia **`try...catch`** la detecta.
- consiste en
    - un **bloque de** `try,` que contiene una o más declaraciones,
    - un **bloque** `catch,` que contiene declaraciones que especifican qué hacer si se lanza una excepción en el bloque **`try.`**
- en otras palabras, se parte que se busca que en el bloque `try` no se produzcan excepciones/errores en tiempo de ejecución, pero si no es así, se desea que el control pase al bloque `catch.`
    - si alguna declaración dentro del bloque **`try (`** o en una función llamada desde dentro del bloque **`try`**) **lanza una excepción,** el **control de flujo** cambia *inmediatamente al* bloque **`catch.`**
    - si no se lanza ninguna excepción en el bloque **`try`**, se omite el bloque **`catch.`**
    - siempre, el bloque **`finally`** se ejecuta después de que se ejecuten los bloques **`try`** y **`catch`**, pero antes de las declaraciones que siguen a toda la declaración **`try...catch.`**

El siguiente ejemplo usa una sentencia **`try...catch.`**

- el ejemplo llama a una función que recupera el nombre de un mes de un array en función del valor pasado a la función.
- si el valor no corresponde a un número de mes (**`1 – 12`**), se lanza una excepción con el valor **`'NombreDeMesNoValido'`** y las declaraciones en el bloque **`catch`** establecen la variable **`nombreMes`** en '**`Desconocido`**'.

```
function obtenerNombreMes(mes) {
  mes--;    // Ajustar número mes para el índice del array (para que 0 = enero, 11 = diciembre)

  const meses = [
    'Ene', 'Feb', 'Mar', 'Abr', 'May', 'Jun',
    'Jul', 'Ago', 'Sep', 'Oct', 'Nov', 'Dic',
  ];

  if (meses[mes]) {
    return meses[mes];
  } else {
```

```
    throw new Error('NombreDeMesNoValido');
  }
}


const miMes = 34;

try {
  nombreMes = obtenerNombreMes(miMes);
} catch (e) {
  nombreMes = 'Desconocido';
  console.log(e); // pasamos la información de la excepción a una función diseñada por nosotros
}
```

Observaciones sobre bloque `catch`

Puedes usar un bloque **`catch para`** **manejar** todas las **excepciones** que pueden generarse en el bloque **`try`**.

```
catch (catchID) {
  sentencias
}
```

- el bloque **`catch`** especifica un **identificador** (*`catchID`* en la sintaxis anterior) que contiene el valor especificado por la instrucción **`throw`**.
- puede usar este identificador para obtener información sobre la excepción que se lanzó.
- JavaScript crea en tiempo de ejecución de este identificador cuando se ingresa el bloque **`catch`**.
- el identificador dura solo la duración del bloque **`catch`**: una vez que el bloque **`catch`** termina de ejecutarse, el identificador ya no existe.

Por ejemplo, el código siguiente genera una excepción.

- cuando ocurre la excepción, el **control se transfiere** al bloque **`catch`**.

```
try {
  throw 'miExcepcion';          // genera una excepción
} catch (err) {
   // declaraciones para manejar cualquier excepción
  registrarError(err);          // pasar el objeto de excepción al controlador de errores
}
```

Nota: al registrar errores en la consola dentro de un bloque **`catch`**, se recomienda usar **`console.error()`** en lugar de **`console.log()`** para la depuración. Da formato al mensaje como un error y lo agrega a la lista de mensajes de error generados por la página.

En general:

- no puedes decir el **tipo de error** que acaba detectándose, porque cualquier cosa puede aparecer en un bloque **`try`**.
- sin embargo, generalmente puede asumir que es una instancia de **`Error`**, como en un ejemplo anterior.
- hay algunas **subclases** de **`Error`**, como **`TypeError`** y **`RangeError`**, que puedes usar para proporcionar semántica adicional sobre el error.

No hay una captura condicional en JavaScript: si solo deseas manejar un tipo de error, debes capturar todo, identificar el tipo de error usando **`instanceof`** y luego volver a **lanzar** los otros casos.

```
try {
   construirMiSitio("./sitioweb");
} catch ( e ) {
   if( e instanceof RangeError ) {
      console.error("Parece que un parámetro está fuera de rango:", e);
      console.log("Reintentando...");
      construirMiSitio("./sitioweb");
   }
} finally {
    // No sé cómo manejar otros tipos de errores; tíralos así
    // algo más en la pila de llamadas puede capturarlo y manejarlo
    throw e;
}
```

Si un error no es **procesado** por ningún `try... catch` en la **pila de llamadas,** el script se fallará.

Observaciones sobre el bloque `finally`

El **bloque** `finally` contiene declaraciones que se ejecutarán *simpre y después de que se ejecuten* los bloques **`try`** y **`catch`.**

- además, el bloque **`finally`** se ejecuta antes que el código que sigue a la instrucción **`try`**…**`catch`**…**`finally`.**
- también es importante tener en cuenta que el bloque **`finally`** se ejecutará independientemente de que se produzca una excepción *o no.*
- sin embargo, si se lanza una excepción, las declaraciones en el bloque **`finally`** se ejecutan incluso si ningún bloque **`catch`** maneja la excepción que se lanzó.
- puedes usar el bloque **`finally`** para conseguir que el script falle de forma organizada cuando ocurra una excepción.
  - por ejemplo, es posible que debas liberar un recurso que tu script ha bloqueado (fichero, conexión de red, base de datos, ...)

El siguiente ejemplo abre un archivo y luego ejecuta declaraciones que usan el archivo.

- **JavaScript del lado del servidor** te permite acceder a los archivos
- si se lanza una excepción mientras el archivo está abierto, el bloque **`finally`** cierra el archivo antes de que falle el script.
- usar **`finally`** aquí *asegura* que el archivo nunca se quede abierto, incluso si ocurre un error.

```
abrirMiArchivo();

try {
   escribirMiArchivo(losDatos);            // Esto puede arrojar un error
} catch (e) {
   manejarError(e);                        // Si ocurrió un error, manejarlo
} finally {
   cerrarMiArchivo();                       // Siempre cerrar el recurso
}
```

Si el bloque **`finally`** **devuelve un valor**, este valor se convierte en el **valor de retorno** de toda la sentencia **`try`**…**`catch`**…**`finally`,** independientemente de las declaraciones de retorno en los bloques **`try`** y **`catch`** :

```
function f() {
  try {
    console.log(0);
    throw 'PRUEBA';
  } catch (e) {
    console.log(1);
    return true;                           // esta declaración de devolución queda anulada
                                           // hasta que el bloque finally se complete
    console.log(2);                        // Esta sentencia es inalcanzable
  } finally {
    console.log(3);
    return false;                          // anula el return del catch
    console.log(4);                        // Esta sentencia es inalcanzable
  }
  // ahora ejecuta "return false"
  console.log(5);                          // Esta sentencia es inalcanzable
}

console.log( f() );                        // 0, 1, 3, false
```

La sobrescritura de los valores `return` por el bloque `finally` también se aplica a las excepciones lanzadas o **re-lanzadas** dentro del bloque **`catch`** :

```
function f() {
  try {
    throw 'PRUEBA';
  } catch (e) {
    console.log('capturada la "PRUEBA" interna');
    throw e;                               // esta declaración de lanzar queda anulada
                                           // hasta que el bloque finally se complete
  } finally {
    return false;                          // anula el throw del catch
```

```
  }
  // ahora ejecuta "return false"
}


try {
  console.log( f() );
} catch (e) {
  // ¡esto nunca se alcanza!
  // mientras f() se ejecuta, el bloque `finally` devuelve false,
  // que sobrescribe el `throw` dentro del `catch` anterior
  console.log('capturada la "PRUEBA" externa');
}

// Registra:
//  capturada la "PRUEBA" interna
// false
```

Anidación de sentencias `try...catch`

Puedes **anidar** una o más sentencias **`try...catch`**.

- si un bloque **`try`** interno *no* tiene un bloque **`catch`** correspondiente :
  - debe contener un bloque **`finally`**, *y*
  - se comprueba la instrucción **`try...catch`** que lo encierra en busca de una coincidencia.

Uso de objetos objetos `Error`

Según el **tipo de error,** es posible que pueda usar las propiedades **`name`** y **`message`** para obtener un mensaje más refinado.

- la propiedad de **`name`** proporciona la clase general de **`Error`** (como **`DOMException`** o **`Error`**),
- **`message`** generalmente proporciona un mensaje más breve que el que se obtendría al convertir el objeto de error en una cadena.

Si lanzas tus propias excepciones, para aprovechar estas propiedades (por ejemplo, si tu bloque **`catch`** no discrimina entre sus propias excepciones y las **excepciones del sistema**), puede usar el constructor **`Error()`**:

- por ejemplo:

```
function hacerAlgoQueCauseExcepcion() {
  if (miCodigoCometeError()) {
    throw new Error('Mi mensaje de error');
  } else {
    hacerAlgoQueCauseUnErrorJS();
  }
}


try {
  hacerAlgoQueCauseExcepcion();
} catch (e) {
  // Ahora, es mejor usar `console.error()`
  console.error(e.name);        // 'Error'
  console.error(e.message);     // 'Mi mensaje de error', o un mensaje de error de JavaScript
}
```

Bucles e iteraciones

Los **bucles** (***loops***) ofrecen una forma rápida y fácil de hacer algo repetidamente.

- puedes pensar en un bucle como una versión computerizada del juego donde le dice a alguien que dé *X* pasos en una dirección, luego *Y* pasos en otra.
- por ejemplo, la idea "Ve cinco pasos hacia el este" podría expresarse de esta manera como un bucle:

```
for (let paso = 0; paso < 5; paso++) {
   // Se ejecuta 5 veces, con valores del paso 0 al 4.
   console.log('Caminando hacia el este un paso');
}
```

- hay muchos tipos diferentes de bucles, pero básicamente todos hacen lo mismo: repiten una **acción** varias veces.
- ten en cuenta que es posible que ese número sea cero
- los diversos mecanismos de bucle ofrecen diferentes formas de determinar los puntos de inicio y fin del bucle.
- hay varias situaciones que se resuelven más fácilmente con un tipo de bucle que con los demás.

Las sentencias para bucles proporcionadas en JavaScript son:

- **`for`**
- **`for ...in`**
- **`for ...of`**
- **`while`**
- **`do...whiile`**
- ***`etiqueta`***
- **`break`**
- **`continue`**

Sentencia `for`

Un **bucle `for`** se repite hasta que una **condición** específica **se evalúa** como falsa.

- **`for`** de JavaScript es similar al bucle for de Java y **`C.`**
- una instrucción **`for`** tiene el siguiente aspecto: (***`[ soy opcional ]`*** cuando se describe una sintaxis los corchetes significan "puede aparecer opcionalmente"; son metacaracteres y no elementos de JavaScript)

```
for ([ expresiónInicial ]; [ expresiónCondición ]; [ expresiónIncremento ])
  sentencia
```

Cuando se ejecuta un bucle **`for`**, ocurre lo siguiente:

1. La **expresión de inicialización** *`expresiónInicial`*, si existe, se ejecuta.
   - esta expresión normalmente inicializa uno o más **contadores de bucle,** pero la sintaxis permite una expresión de cualquier grado de complejidad.
   - esta expresión también puede **declarar variables.**
2. Se evalúa la expresión *`expresiónCondición`*.
   - si el valor de *`expresiónCondición`* es **`true`**
     - se ejecutan las sentencias de bucle.
     - de lo contrario, el ciclo **`for`** termina.
   - si la expresión *`expresiónCondición`* se omite por completo, se supone que la condición es `true`.
3. La *`sentencia se`* ejecuta: para ejecutar varias declaraciones, use una **declaración de bloque** (**`{ }`**) para agrupar esas declaraciones.
4. Si está presente, la **expresión de actualización** *`expresiónIncremento`* se ejecuta.
5. El control vuelve al Paso 2.

Puesto como si fuera un bucle while, queraría así:

```
expresiónInicial
while (expresiónCondición) {
   sentencia
   expresiónIncremento
}
```

Ejemplo

En el siguiente ejemplo, la función contiene una instrucción **`for`** que cuenta el número de opciones seleccionadas en una lista de desplazamiento (un elemento **`<select>`** que permite múltiples selecciones).

HTML

```
<form name="formularioSeleccion">
  <label for="tiposDeMusica">Elija algunos tipos de música, luego haga
     clic en el botón a continuación:</label>
  <select id="tiposDeMusica" name="tiposDeMusica" multiple>
    <option selected>R&B</option>
    <option>Jazz</option>
    <option>Blues</option>
    <option>New Age</option>
    <option>Clasica</option>
    <option>Opera</option>
  </select>
  <button id="boton" type="button">¿Cuántos están seleccionados?</button>
</form>
```

JavaScript

Aquí,

- la sentencia **`for`** declara la variable **`i`** y la inicializa a **`0`**.
- comprueba que **`i`** es menor que el número de opciones en el elemento **`<select>`**,
- realiza la instrucción **`if siguiente`** e incrementa **`i`** en 1 después de cada paso por el ciclo.

```
function cuantosHay(objetoSelect) {
   let numeroSeleccionados = 0;
   for (let i = 0; i < objetoSelect.options.length; i++) {
      if (objetoSelect.options[i].selected) {
         numeroSeleccionados++;
      }
   }
   return numeroSeleccionados;
}

const boton = documento.getElementById('boton');

boton.addEventListener('click', () => {
   const tiposDeMusica = document.formularioSeleccion.tiposDeMusica ;
   console.log(`Ha seleccionado la(s) opción(es) ${cuantosHay(tiposDeMusica)} .`);
});
```

Sentencia `for...in`

La **declaración `for...in` itera** una variable especificada sobre todas las **propiedades enumerables** de un objeto.

- para cada propiedad distinta, JavaScript ejecuta las sentencias especificadas.
- una instrucción **`for...in`** tiene el siguiente aspecto:

```
for ( variable in objeto )
  sentencia
```

Ejemplo

La siguiente función

- toma como **argumento** un objeto y el **nombre del objeto.**
- luego itera sobre todas las propiedades del objeto y devuelve una cadena que enumera los nombres de las propiedades y sus valores.

```
function volcarPropiedades(objeto, nombreObjeto) {
  let resultado = '';
  for (const i in objeto) {
    result += `${nombreObjeto}.${i} = ${objeto[i]}<br>`;
  }
  resultado += '<hr>';
  return resultado;
}
```

Para un objeto **`coche`** con propiedades **`marca`** y **`modelo`**, el **`resultado`** sería:

```
coche.marca = Ford
coche.modelo = Mustang
```

Arrays

Aunque puede ser tentador usar esto como una forma de iterar sobre los elementos de `Array`,

- la instrucción **`for...in`** devolverá el nombre de sus **propiedades definidas por el usuario** además de los **índices numéricos.**
- por lo tanto, es mejor usar un bucle **`for`** tradicional con un **índice numérico** al iterar sobre arrays, porque la instrucción **`for...in`** itera sobre **propiedades definidas por el usuario** además de los elementos del array.

Sentencia `for...of`

La **declaración** `for...of` crea un bucle iterando sobre **objetos iterables** (incluidos `Array`, `Map`, `Set`, el objeto `arguments`, etc.), invocando un **gancho de iteración personalizado** con declaraciones que se ejecutarán para el valor de cada propiedad distinta.

```
for ( variable of objeto )
  sentencia
```

El siguiente ejemplo muestra la diferencia entre un bucle `for...of` y un bucle `for...in`.

- mientras que `for...in` itera sobre los **nombres** de las propiedades, `for...of` itera sobre los **valores** de las propiedades:

```
const listado = [3, 5, 7];
listado.foo = 'Hola';

for (const i in arr) {
  console.log(i);
}
// "0" "1" "2" "foo"

for (const i of arr) {
  console.log(i);
}
// Registra: 3 5 7
```

Las sentencias `for...of` y `for...in` también se pueden usar con la **desestructuración.**

- por ejemplo, puede recorrer simultáneamente las **claves** y los **valores** de un objeto usando `Object.entries()`.

```
const obj = { foo: 1, bar: 2 };

for (const [clave, valor] of Object.entries(obj)) {    // operador desestructurante
   consola.log(clave, valor);
}
// "foo" 1
// "bar" 2
```

Sentencia `do...while`

La **instrucción** `do...while` se repite hasta que una condición específica se evalúa como `false`.

- una instrucción `do...while` se ve de la siguiente manera:

```
do
  sentencia
while ( condición );
```

- *`sentencia`* siempre se ejecuta una vez antes de comprobar la **condición** *`condición`*.
- para ejecutar varias declaraciones, use una **declaración de bloque** (`{ }`) para agrupar esas declaraciones.
- si la *`condición`* es `true`, la *`sentencia`* se ejecuta de nuevo.
- al final de cada ejecución, se comprueba la *`condición`*.
  - cuando la condición es `false`, la ejecución se detiene y el **control** pasa a la sentencia que sigue a `do...while`.

Ejemplo

En el siguiente ejemplo, el ciclo/bucle `do` **itera** al menos una vez y se repite hasta que `i` ya no es menor que `5`.

```
let i = 0;
do {
  i += 1;
  console.log(i);
} while (i < 5);
```

Sentencia `while`

Una **declaración** `while`

- ejecuta sus *`sentencias`* siempre que una condición *`condición`* especificada se evalúe como `true`.
- una instrucción `while` tiene el siguiente aspecto:

```
while ( condición )
  sentencia
```

- si la *`condición`* se vuelve `false`, la *`sentencia`* dentro del bucle deja de ejecutarse y el control pasa a la instrucción que sigue al bucle.
- la prueba de condición ocurre *antes* se que se ejecute *`sentencia`*.
  - si la *`condición`* devuelve `true`, la *`sentencia`* se ejecuta y la *`condición`* se prueba nuevamente.
  - si la *`condición`* devuelve `false`, la ejecución se detiene y el control pasa a la instrucción que sigue a `while`.
- para ejecutar varias declaraciones, use una **declaración de bloque** (`{ }`) para agrupar esas declaraciones.

Ejemplo 1

El siguiente bucle `while` itera siempre que `n` sea menor que `3` :

```
let n = 0;
let x = 0;
while (n < 3) {
  n++;
  x += n;
}
```

- con cada **iteración,** el ciclo incrementa `n` y agrega ese valor a `x`.
- por lo tanto, `x` y `n` toman los siguientes valores:
  - después de la primera pasada: `n = 1` y `x = 1`
  - después de la segunda pasada: `n = 2` y `x = 3`
  - después del tercer pase: `n = 3` y `x = 6`
- después de completar el tercer paso, la condición `n < 3` ya no es `true`, por lo que el ciclo termina.

Ejemplo 2

Evite **bucles infinitos.**

- asegúrese de que la condición en un ciclo finalmente se vuelva `false` : de lo contrario, ¡el ciclo nunca terminará!
- las declaraciones en el siguiente ciclo `while` se ejecutan para siempre porque la condición nunca se vuelve `false`:

```
// ¡Los bucles infinitos son malos (....casi siempre)!
while (true) {
   console.log('¡Hola, mundo!');
}
```

`etiqueta`

Una ***`etiqueta`*** proporciona una sentencia que consiste en **identificador** que permite que el flujo de control sea dirigido a ella en cualquier otra parte de su programa.

- por ejemplo, puede usar una etiqueta para identificar un **ciclo** y luego usar la declaración de `break` o `continue` para indicar si un programa debe interrumpir el ciclo o continuar su ejecución.
- la sintaxis de la instrucción etiquetada tiene el siguiente aspecto:

```
etiqueta:
  sentencia
```

- el valor de la ***`etiqueta`*** puede ser cualquier **identificador de JavaScript** que no sea una palabra reservada.
- la ***`sentencia`*** que identificas con una etiqueta puede ser cualquier sentencia.

Ejemplo

En este ejemplo, la etiqueta **`bucleMarcar`** identifica un ciclo **`while`**.

```
bucleMarcar:
while (hayMarca) {
   marcarAlgo();

}
```

Sentencia `break`

Utilice la **sentencia** **`break`** para **terminar un bucle,** una cláusula **`case`** de **`switch`** o para llegar a una **sentencia etiquetada.**

- cuando usa **`break`** sin una etiqueta, termina el **bucle más interno** **`while`**, **`do-while`**, **`for`** o **`switch`** inmediatamente y transfiere el control a la siguiente sentencia.
- cuando usas **`break`** con una etiqueta, finaliza la instrucción etiquetada especificada.
- la sintaxis de la instrucción **`break`** se ve así:

```
break;
break etiqueta;
```

1. La primera forma de la sintaxis finalizaría el **bucle** o **`switch`** más interno.
2. La segunda forma de la sintaxis finaliza la instrucción con la etiqueta justo antes de dicha instrucción.

Ejemplo 1

El siguiente ejemplo itera a través de los elementos de un array hasta que encuentra el **índice** de un elemento cuyo valor es **`elValor`**:

```
for (let i = 0; i < a.length; i++) {
  if (a[i] === elValor) {
    break;
  }
}
```

Ejemplo 2: romper con una etiqueta

```
let x = 0;
let z = 0;
etiquetaExterna:
  while (true) {
     console.log('Bucle externo: ', x);
     x += 1;
     z = 1;
     while (true) {
        console.log('Bucle interno: ', z);
        z += 1;
        if (z === 10 && x === 10) {
           break etiquetaExterna;
        } else if (z === 10) {
          break;
        }
   }
}
```

Sentencia `continue`

La **sentencia** **`continue`** se puede usar para **reiniciar** una sentencia **`while`**, **`do-while`**, **`for`** o **`label`**.

- cuando usas **`continue`** sin etiqueta,
  - termina la **iteración actual** del bucle **más interno** de una sentencia **`while`**, **`do-while`** o **`for`**
  - y **continúa la ejecución del bucle** con la **siguiente iteración.**
- a diferencia de la instrucción **`break`**, **`continue`** no finaliza la ejecución del bucle por completo.

    - en un bucle **`while`**, vuelve a la condición.
    - en un bucle **`for`**, salta a la **`expresión-incremento`**.
- cuando usa **`continue`** con una etiqueta, el control de flujo se transfiere a continuar en el bucle etiquetado con esa etiqueta.
- la sintaxis de la sentencia **`continue`** es similar a la siguiente:

```
continue;
continue etiqueta;
```

Ejemplo 1

El siguiente ejemplo muestra un ciclo **`while`** con una declaración **`continue`** que se ejecuta cuando el valor de **`i`** es **`3`**. Así, **`n`** toma los valores **`1`**, **`3`**, **`7`** y **`12`**.

```
let i = 0;
let n = 0;
while (i < 5) {
  i++;
  if (i === 3) {
    continue;
  }
  n += i;
  console.log(n);
}
//1,3,7,12
```

Si comentas el **`continue;`** el bucle se ejecutaría hasta el final y verías **`1,3,6,10,15`**.

Ejemplo 2

Una sentencia etiquetada como **`comprobariyj`** contiene una sentencia etiquetada como **`comprobarj`**.

- si se encuentra **`continue`**, el programa finaliza la iteración actual de **`comprobarj`** y comienza la siguiente iteración.
- cada vez que se encuentra **`continue`**, **`comprobarj se`** reitera hasta que su condición devuelve **`false`**.
- cuando se devuelve **`false`**, el resto de la sentencia **`comprobariyj`** se completa y **`comprobariyj`** se repite hasta que su condición devuelve **`false`**.
- cuando se devuelve **`false`**, el programa continúa en la sentencia que sigue a **`comprobariyj`**.
- si **`continue`** tuviera la etiqueta **`comprobariyj`**, el programa continuaría en la parte superior de la instrucción **`comprobariyj`**.

```
let i = 0;
let j = 10;
comprobariyj:
  while (i < 4) {
     console.log(i);
     i += 1;
  comprobarj:
     while (j > 4) {
       console.log(j);
       j -= 1;
       if ((j % 2) === 0) {
         continue comprobarj;
       }
       console.log(j, ' es impar.');
     }
  console.log('i = ', i);
  console.log('j = ', j);
}


// 0
// 10
//  9 ' es impar.'
// 9
// 8
//  7 ' es impar.'
// 7
// 6
//  5 ' es impar.'
// 5
//  i =  1
//  j =  4
```

```
// 1
//  i =  2
//  j =  4
// 2
//  i =  3
//  j =  4
// 3
//  i =  4
//  j =  4
```