

---

## Estructura de código y declaraciones lógicas

---

En el documento anterior, cubrimos las definiciones fundamentales de JavaScript y dónde lo escribimos. Ahora que hemos discutido los conceptos básicos, comencemos a aprender cómo escribimos el código JavaScript. En este documento, cubriremos los fundamentos, que incluyen estructura de código, declaraciones lógicas y variables. Dominar estos conceptos es un requisito para escribir un código de JavaScript útil. Cuando sea relevante, también nos sumergiremos más en cómo funcionan estos conceptos básicos, para que tengas una comprensión más profunda del código que estás escribiendo.

### Empezando

A medida que avanzamos en este documento, será bueno tener un espacio de trabajo donde puedas escribir y probar tu JavaScript. Para estos fines, has creado una carpeta llamada "ProyectosNode" en tu carpeta de documentos. Dentro de eso, has creado dos archivos: index.html y index.js.

Dado que nuestro enfoque será escribir JavaScript, tu archivo HTML puede ser relativamente simple:

```
<!DOCTYPE html>
<html>
<head>
  <title>My First JavaScript</title>
</head>
<body>
  <p>Hello World</p>
  <script src=""index.js""></script>
</body>
</html>
```

Cualquier código JavaScript que desees probar se puede poner en index.js. Por ahora, solo hemos puesto un método simple console.log:

```
console.log("Hello World!")
```

El método `console.log` es realmente útil. Se usa ampliamente para la depuración, y lo usaremos a lo largo de este curso. Cuando se ejecuta, registra un mensaje en la consola de tu navegador, para que puedas ver la salida de tu código fácilmente. Puedes encontrar la consola haciendo clic derecho en cualquier lugar de tu página web y seleccionando "Inspeccionar". Los registros de la consola se pueden ver en la pestaña "Consola", como se muestra en la figura 1.

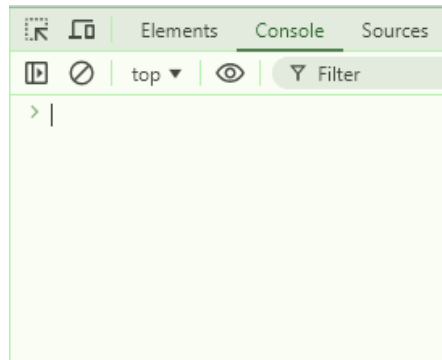


Figura 2-1. Haz clic derecho en el navegador web y selecciona "Inspeccionar" en Google Chrome (u otros navegadores) te permite acceder al registro de la consola. Si haces esto con el archivo `index.html` que definimos antes, verás "¡Hola mundo!" escrito aquí. La consola es una herramienta poderosa utilizada para la depuración.

## Convenciones de código comunes

Antes de comenzar a escribir código real, primero consideremos algunas convenciones de código básico. Como es el caso en la mayoría de la programación, los desarrolladores de JavaScript intentan seguir algunas convenciones comunes al escribir código. Esta sección es en gran medida obstinada, pero te proporciona pautas sobre cómo debes escribir JavaScript. También te hará saber cómo haremos para escribir código a lo largo del curso.

### Semicolones

Para la legibilidad, JavaScript a veces se escribe con un punto y coma al final de cada línea. Por ejemplo:

```
console.log ("¡Hola mundo!");  
console.log ("¡Adiós Mundo!");
```

Sin embargo, esto no es necesario, y también es igual de común ver el mismo código escrito sin semicolones:

```
console.log ("¡Hola mundo!")  
console.log ("¡Adiós Mundo!")
```

Podemos hacer esto porque JavaScript intuitivamente “descubrirá” a dónde deberían ir las semicolones. Incluso funciona si haces una ruptura de línea accidental donde una expresión parece incompleta. Por ejemplo, si una línea termina con un símbolo +, JavaScript asumirá que la expresión debe continuar en la siguiente línea, lo que significa que el código seguirá funcionando como se esperaba:

```
console.log (5 +  
6)
```

Para el código en este curso, omitiremos el punto y coma a menos que sea realmente necesario.

## Espaciado

Una de las convenciones de código más luchadas por los códigos es si se debes usar pestañas o espacios para sangrar. Si bien esencialmente no hay una respuesta correcta o incorrecta a esto, es importante ser consistente. Si usas pestañas, siempre usa pestañas para muescas y de la misma manera para espacios.

A diferencia de Python, las identaciones no juegan un papel funcional en JavaScript, pero sí sirven para hacer que tu código sea más legible cuando otros lo miran. Si bien está bien usar pestañas, los espacios te causarán menos dolor de cabeza. Esto se debe a que se pueden configurar diferentes ecosistemas y sistemas operativos para manejar las pestañas de manera diferente, mientras que los espacios tienen un tamaño constante en todos los sistemas.

En documento y quizás más adelante, sangraremos con cuatro espacios. Aquí hay un ejemplo de cómo se verá eso:

```
let myVariable = 5  
if(myVariable === 5) {  
    console.log("The variable is 5!")  
}
```

**Nota:** Si usas la tecla Tab en lugar de los espacios en el VS Code para sangrar tu código, puedes configurar VS Code para convertirlos automáticamente en espacios si lo deseas. La opción de configuración se encuentra abriendo cualquier archivo y seleccionando la opción “Espacios/Tamaño de pestaña” en la esquina inferior derecha (Spaces/Tab Size).

## Nombramiento de la variable y función

Cuando se trata del nombrado de funciones y variables, encontrará tres paradigmas diferentes: caso de camello, caso pascal y subrayado:

- El caso de camello se refiere a nombrar una variable donde cada palabra después de la primera tiene una letra mayúscula, por ejemplo, `thisIsCamelCase`.
- El caso pascal es el mismo, excepto que la primera letra también está capitalizada. Por ejemplo, `ThisIsPascalCase`.
- El subrayado se refiere a la separación de palabras con `_` (underscores). Por ejemplo, `these_are_underscored`.

Al nombrar variables y funciones, todas estas están bien de usar, pero nuevamente, es importante ser consistente. Si decides usar el caso del camello, asegúrate de usarlo en todas partes. Para los fines de esto, intentemos utilizar el caso camello.

## Variables JavaScript

Al comenzar a escribir JavaScript, lo primero que tendrás que aprender son las variables. Las variables son una forma de asignar un nombre fijo a un valor de datos. Hay tres formas de crear una variable en JavaScript, utilizando tres palabras clave diferentes:

- `var`
- `let`
- `const`

Todas las variables de JavaScript son sensibles a los casos (case-sensitive), por lo que `myVariable` es diferente de `MYVARIABLE`. En la nomenclatura de JavaScript, generalmente decimos que “declaramos” variables.

## Configuración de variables con `let`

La mayoría de las variables en JavaScript se establecen con `let`. Un conjunto de variable con `let` comienza con la palabra clave `let`, seguido del nombre de la variable y un signo igual, y luego el valor de su variable. En el siguiente ejemplo, creamos una variable llamada `myVariable`, y usando `console.log` para que aparezca en la consola del navegador:

```
let myVariable = 5  
console.log (myVariable)
```

**Nota:** Puede parecer que estamos poniendo datos en la variable, pero una mejor manera de pensarlo es que estamos haciendo datos y luego señalando la palabra clave “myVariable” en los datos que acabamos de hacer.

No es posible asignar una variable con let dos veces. Si piensas que tu variable apunta a algunos datos, es fácil ver por qué: la misma variable no puede apuntar dos datos diferentes. Por ejemplo, el siguiente código producirá el error, que se muestra en la figura 2.

```
let myVariable = 5  
let myVariable = 10  
console.log(myVariable)
```

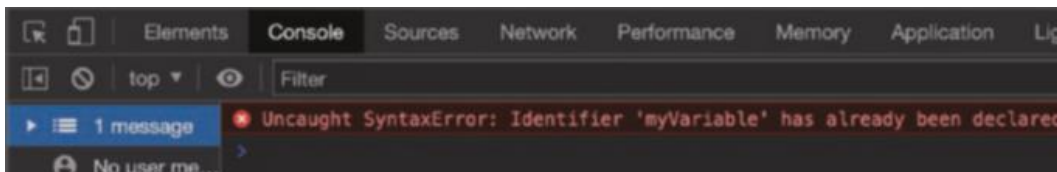


Figura 2-2: Las variables definidas con let no se pueden definir varias veces. Si lo intentas, se producirá un error como el anterior.

Por lo tanto, las variables definidas con let no se pueden redefinir nuevamente. Sin embargo, pueden reasignarse a una nueva pieza de datos mencionándolos nuevamente sin la palabra clave 'let'. Puedes ver eso en el siguiente ejemplo, con los registros de la consola "10", ya que cambiamos el valor de myVariable:

```
let myVariable = 5  
myVariable = 10  
console.log(myVariable)
```

Puede parecer que los datos han cambiado (o “mutado”), pero en realidad acabamos de hacer nuevos datos en algún lugar y señalamos nuestra variable a eso. Los datos "5" todavía existen en alguna parte. Simplemente ya no tienen una variable apuntando.

Cuando ya no se hace referencia a un dato en nuestro código, JavaScript puede eliminarlo de la memoria usando algo llamado “recolección de basura”. Esto permite que JavaScript libere la memoria cuando los datos ya no se usan.

### **Bloquear el alcance con variables**

En el ejemplo anterior vimos que no es posible definir la misma variable, pero hay excepciones a esto. Esto se debe a que las variables definidas con let están en alcance de

bloque. El alcance limita la funcionalidad, los nombres variables y los valores a ciertas secciones de nuestro código.

Los ámbitos de bloques se crean con llaves `{ }`. En la mayoría de los escenarios, creamos ámbitos de bloque definiendo nuevas funciones o declaraciones lógicas.

Dado que los `{ }` independientes también crean un nuevo alcance de bloque, una variable se puede configurar dos veces simplemente definiéndolo nuevamente dentro de `{ }`:

```
let myVariable = 5
{
  let myVariable = 10
  console.log(myVariable)
}
console.log(myVariable)
```

Aunque creamos la misma variable dos veces, no se lanza ningún error, y eso se debe a que uno está asignado a un nuevo alcance de bloque. La primera variable se asigna al alcance “global” de nuestro programa, mientras que la segunda se asigna solo al alcance del bloque `{ }`. El resultado de esto se puede ver en la figura 3.

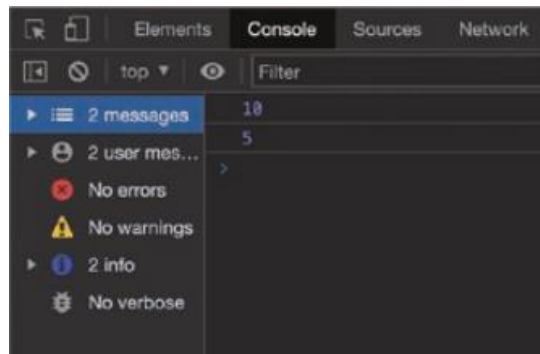


Figura 3: Mientras usas `let`, nuestra consola produce dos líneas diferentes, 5 y 10. Esto se debe a que `let` se asigna a su alcance actual, por lo que establecer `myVariable` en 10 en un alcance separado no afecta la variable original. Si usamos `var`, ambas líneas dirían 10 ya que el alcance se ignora.

### Configuración de variables con `var`

La mayoría de las variables se establecen con `let`, pero también puedes ver la palabra clave `var` que se usa para establecer variables a veces. Usar `var` es la forma original de establecer variables en JavaScript. Es valioso saber que esto existe, pero generalmente no se recomienda que lo uses en lugar de `let`.

Establecer una variable con `var` se parece mucho a lo que hicimos con `let`. Por ejemplo, aquí hay una variable llamada `myVariable`, con un valor de 5:

```
var myVariable = 5
```

La razón por la que usamos `let` en lugar de `var` es porque `var` tiene algunas peculiaridades que no lo hacen. Por ejemplo, puedes definir una variable dos veces con `var`, y no se lanzarán errores:

```
var myVariable = 5  
var myVariable = 10  
console.log(myVariable)
```

Las variables definidas con `var` tampoco están en el alcance del bloque, lo que significa que tu código puede producir algunos resultados impares al redefinir variables en ámbitos de bloque con `var`.

Tal vez te preguntes: “¿Por qué JavaScript tiene dos formas de definir variables cuando `let` es una versión más controlada de `var`?”. La respuesta a eso es bastante simple, y es porque `var` solía ser la única forma de definir variables en JavaScript, y mucho código heredado lo usa. Más adelante, JavaScript creó una mejor manera de definir variables usando `let`, pero `var` no se pudo eliminar, ya que rompería muchas bases de código más antiguas.

### **Establecer variables con `const`**

El tipo de variable final que cubriremos es `const`. Cuando definimos variables con `let`, hablamos sobre cómo se puede reasignar una variable a otro valor:


```
let myVariable = 5  
myVariable = 10  
console.log(myVariable)
```

Recuerda, en realidad no cambiamos ni “mutamos” los datos aquí - solo creamos nuevos datos y “volvemos a señalar” nuestra variable a esa nueva fuente de datos.

Si bien esto funciona para `let`, no funcionará para `const`. Las variables definidas con `const` son constantes y no se pueden reasignar:

```
const myConst = 5  
console.log(myConst)
```

Si intentas reasignar el valor de una variable `const`, obtendrá un error.

```
 const myConst = 5  
myConst = 10  
console.log(myConst)
```

Usar `const` para definir variables es mejor cuando puedes usarlo. Para entender por qué, puedes pensar en el ejemplo que discutimos anteriormente, donde usamos `let` para reasignar una variable de "5" a "10". Hablamos sobre cómo el valor "5" todavía existe en la memoria a menos que se recolecte la basura. Dado que las variables `const` no se pueden cambiar, la recolección de basura nunca tiene que funcionar. Eso significa que se requiere menos limpieza, lo que resulta en una utilización de memoria más eficiente.

### **Mutación de variables `const`**

Las variables no se pueden reasignar con `const`, pero pueden ser mutadas.

Esto se debe a que la reasignación y la mutación no son lo mismo. Cuando reasignamos una variable, no hay cambios en los datos, pero es posible cambiar los datos subyacentes en JavaScript utilizando la mutación. En JavaScript, no es posible la mutación de valores primitivos como números, cadenas y booleanos, pero los objetos pueden ser mutados.

Cubriremos cómo funcionan los arreglos y los objetos con más profundidad en el próximo documento. Por ahora, todo lo que necesitas entender es que los arreglos y los objetos son contenedores de datos. Un arreglo se puede definir como se muestra en el siguiente ejemplo:

```
const myArray = [ "some", "set", "of", "content" ]  
console.log(myArray)
```

Los arreglos pueden contener muchos datos, y podemos llevar nuevos datos a un arreglo utilizando un método especial llamado `push`:

```
const myArray = [ "some", "set", "of", "content" ]  
myArray.push("new data!")  
console.log(myArray)
```

Al usar `push`, podemos mutar nuestro arreglo, lo que significa que los datos subyacentes cambian y los datos continúan siendo referenciados y almacenados en el mismo lugar. En otras palabras, no creamos nuevos datos y apuntamos nuestra variable en otro lugar, sino que mutamos los datos originales.



Esto es confuso para los principiantes ya que el arreglo fue apuntado por una variable `const`, por lo que se supone que, dado que la variable `const` es una constante, los datos internos siempre deben permanecer constantes. Este no es el caso en JavaScript. Entonces, en resumen, mientras la reasignación debe permanecer constante en una variable `const`, la mutación de datos está bien.

### Definición de variables sin valores

También es posible definir variables que apunten a ninguna parte. Si intentas usar una variable en la consola `log` sin valor asignado, quedará indefinida como resultado:

```
let myVariable  
console.log(myVariable)
```

Esto a veces se hace cuando una variable no tiene un valor cuando la declaras, pero se le puede asignar un valor más adelante en el código. Cuando muchas variables deben declararse sin valores, pueden separarse por comas. En el siguiente ejemplo, definimos tres variables con la palabra clave `let`:

```
let myVariable, myOtherVariable, myFinalVariable
```

Aunque la notación de coma como esta se usa más comúnmente con variables que no tienen valor, también puedes declarar variables con valores de esta manera:

```
let myVariable = 5, myOtherVariable = 4, myFinalVariable = 3
```

### Operadores de asignación

Ahora que hemos cubierto los conceptos básicos de establecer variables, veamos a los operadores de asignación. Estos nos permiten modificar una variable existente, cambiando su valor. Por ejemplo, considera esta variable:

```
let x = 5
```

Supongamos que queríamos multiplicar `x` por 5. Una forma es reasignar `x` a `5 * 5`, pero una mejor manera es usar una operación de asignación. La razón por la que es mejor es porque `x` puede no ser siempre 5, por lo que esto es particularmente útil si queremos cambiar el valor de las variables en función de las condiciones (que cubriremos con más detalle a continuación).

Para multiplicar `x` por 5, usamos el operador de asignación `*=`:

```
let x = 5
x *= 5
console.log(x) // Console logs 25 (5 multiplied by 5 = 25)
```

Hay muchos otros operadores de asignación. Se muestran en el siguiente ejemplo:

```
let x = 5
x *= 5
console.log(x) // Console logs 25 (5 multiplied by 5 = 25)
```

```
x += 5
console.log(x) // Console logs 30 (25 plus 5 = 30)
```

```
x /= 5
console.log(x) // Console logs 6 (30 divided by 5 = 6)
```

```
x -= 1
console.log(x) // Console logs 5 (6 minus 1 = 5)
```

```
x %= 4
console.log(x)
/*
  Console logs 1 (if you divide 5 by 4, the remainder is 1.
  % is the remainder operator
  */
```

## Concatenación variable

Cuando tenemos variables que consisten en al menos una cadena, el uso del operador + hace que las cadenas se concatenen. Para comprender esto, echa un vistazo al siguiente ejemplo, donde concatenamos dos cadenas en una nueva variable:

```
let myVariable = "hello"
let myOtherVariable = "world"
let combine = myVariable + myOtherVariable // "helloworld"
```

Si también necesitamos un espacio, podemos agregarlo con otro +:

```
let myVariable = "hello"
let myOtherVariable = "world"
// "hello world"
let combine = myVariable + " " + myOtherVariable
```

Solo ten cuidado, ya que si intentas usar a + con números, los agregará en su lugar!

```
let myVariable = 5
let myOtherVariable = 5
let combine = myVariable + myOtherVariable // 10
```

Esto nos lleva a una nueva peculiaridad causada por el tipeado dinámico de JavaScript. Si un número está en comillas, se supone que es una cadena. Agregar un número y una cadena da como resultado una nueva cadena concatenada en lugar de un cálculo:

```
let myVariable = "5"
let myOtherVariable = 5
let combine = myVariable + myOtherVariable // "55"
```

Otra forma de combinar variables de cadena es con un método especial llamado concat. Este es un método que existe en todas las cadenas, que se sumará al final de una cadena cualquier cantidad de cosas nuevas que separamos con las comas:

```
let myVariable = "hello"
myVariable.concat(" ", "world", "!") // hello world!
```

Los diferentes tipos de datos tienen diferentes métodos incorporados, que veremos con mucho más detalle más adelante.

## Literales de plantilla

Otra forma final de concatenarse de manera más elegante es a través de un tipo de funcionalidad llamada literales de plantilla. Las literales de la plantilla siguen siendo cadenas, pero usan el retroceso "`" para transformar cualquier contenido en una plantilla literal.

Las literales de plantilla tienen el beneficio adicional de permitir descansos de línea, algo que los números y las comillas no lo hacen. También permiten sustitución. Aquí hay un ejemplo de una plantilla particularmente desordenada literal con descansos de línea en todo:

```
let myVariable = `hello world
!!
how are you?`
```

Las literales de plantilla como la anterior se tomarán con saltos de línea y espacio en blanco incluidos, lo que significa que evitará la pérdida de este contenido al usarlos. También permiten sustitución. Agregar una variable en \$ { } lo sustituirá en una plantilla literal:

```
let someWord = "world"  
let myVariable = `hello ${someWord}!` // hello world!
```

## Comentarios de JavaScript

Es posible que ya hayas notado que en el código anterior, usamos doble slash para dejar algunos comentarios sobre lo que hace el código. A medida que nuestro código se vuelve más complicado, es útil dejar mensajes para ti u otros desarrolladores sobre lo que está sucediendo. Para este propósito, se utilizan comentarios.

Un comentario en JavaScript puede tomar una de las dos formas. El primero se ve así:

```
// ¡Soy un comentario!
```

Y el segundo se ve así, donde el comentario está encerrado en `/*` y `*/`:

```
/* ¡Soy un comentario! */
```

Ambos hacen lo mismo, pero el segundo permite comentarios multi línea.

Los comentarios no tienen relación con la funcionalidad de tu código y en su lugar pueden proporcionar información útil sobre lo que está sucediendo. Por ejemplo, podríamos comentar nuestro código anterior así:

```
// This code will console log myConst  
const myConst = 5  
console.log(myConst)
```

## Declaraciones lógicas

Ahora que hemos cubierto variables, probablemente te preguntarás cómo podemos usarlas. Las declaraciones lógicas son una de las formas en que podemos comenzar a aprovechar nuestras variables.

Las declaraciones lógicas se usan cuando queremos verificar si algo es verdadero o falso y luego ejecutar una acción particular basada en ese resultado. Si se encuentra que una declaración lógica es verdadera, todo dentro de su “alcance de bloque” se ejecuta.

Del mismo modo, si una declaración lógica es falsa, entonces el alcance del bloque nunca se ejecutará en absoluto. En otras palabras, las declaraciones lógicas nos permiten generar condicionalidad en nuestros programas.

## Declaraciones if...else

if ... else son quizás una de las declaraciones lógicas más utilizadas. Todo lo que hacen es verificar si una declaración es verdadera. Si es así, ejecutan algún código, de lo contrario, ejecutan algún otro código. En el siguiente ejemplo, verificamos si myVariable está asignada en 5. Si es así, mostramos "¡la variable es 5!" en la consola.

Si no es así, entonces mostramos texto alternativo.

Puedes ver este código que se ejecuta en la figura 4, donde se ejecuta directamente desde la consola en el navegador web. También puedes probar esto actualizando tu archivo index.js usado anteriormente:

```
// We have set myVariable to 5
let myVariable = 5
if(myVariable === 5) {
  console.log("The variable is 5!")
}
else {
  console.log("The variable is not 5.")
}
```

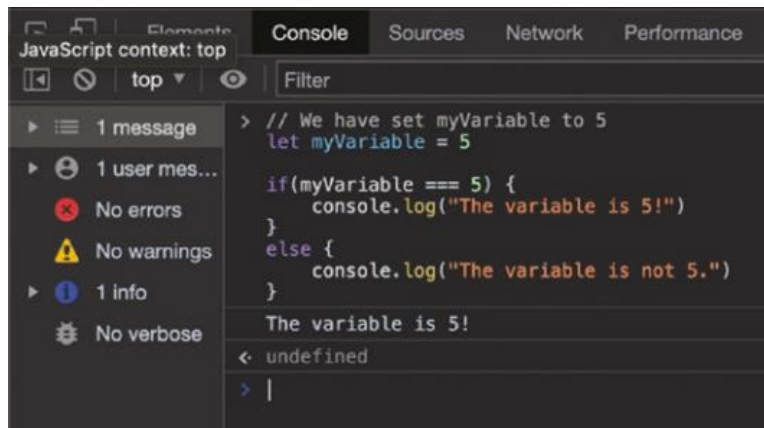


Figura 2-4: Dado que myVariable es 5, la declaración myVariable === 5 siempre es verdadera. Entonces, el bloque de código dentro de la primera declaración if se ejecuta, y la declaración de else nunca se dispara.

Podemos agregar más condiciones a nuestro código usando else if después de una instrucción if. En el siguiente ejemplo, verificamos una tercera condición, donde myVariable podría ser 6:

```
// We have set myVariable to 5
let myVariable = 5
if(myVariable === 5) {
```

```
    console.log("The variable is 5!")
  }
  else if(myVariable === 6) {
    myVariable = 7
    console.log("The variable is 6, but I set it to 7!")
  }
  else {
    console.log("The variable is not 5.")
  }
}
```

Como puedes ver en la condición else if, cualquier código se puede escribir en el alcance de bloque creado por una declaración lógica. En el otro if, reasignamos myVariable para tener un valor de 7.

Si la salida de una instrucción if es una línea de código, puedes omitir los { } y escribirlo todo en una línea de código como se muestra en el siguiente ejemplo. Esto también se aplica a else if y else:

```
// We have set myVariable to 5
let myVariable = 5
if(myVariable === 5) console.log("The variable is 5!")
```

Nuestros ejemplos hasta ahora han tratado una igualdad estricta, lo que significa que estamos utilizando el signo triple igual. ¡Sin embargo, hay más formas de verificar los valores variables! Los cubriremos a todos en este documento.

## Declaraciones switch

Otra forma común de crear condicionalidad en tu JavaScript es a través de declaraciones switch. Toman una sola variable y verifican si equivale a ciertos valores utilizando la palabra clave case para definir el valor que podría ser igual y un break para declarar el final de esa cláusula. Por ejemplo:

```
// Let's set x to 5
let x = 5
switch(x) {
  case 5:
    console.log("hello")
    break
  case 6:
    console.log("goodbye")
    break
}
```

En el ejemplo anterior, tenemos dos cláusulas: una en la que x es igual a 5, en cuyo caso consola log imprime "hello" y otro donde es igual a 6, donde consola log imprime "goodbye". Es importante tener un break después de cada cláusula de case, ya que si no lo haces, todos los casos se ejecutarán después del que es correcto! Por ejemplo, el siguiente código console log imprime tanto "hello" como "goodbye", lo cual es bastante confuso:

```
// Let's set x to 5
let x = 5
switch(x) {
  case 4:
    console.log("4!")
  case 5:
    console.log("hello")
  case 6:
    console.log("goodbye")
}
```

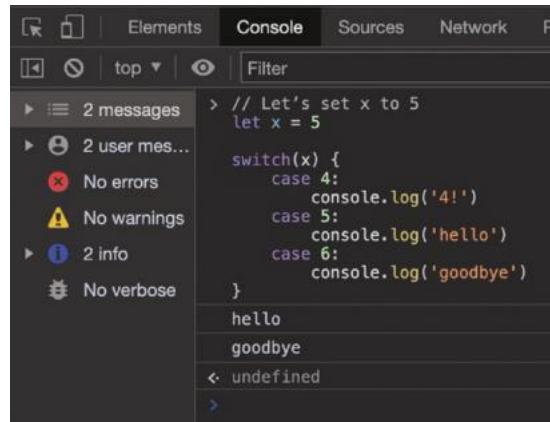


Figura 5: Dado que no usamos la palabra clave "break" en el código anterior y x es igual a 5, tanto "Hello" como "goodbye" están impresas en la consola. La palabra clave "break" es importante para declarar el final de una cláusula en una declaración de conmutación.

Sin embargo, este mecanismo de "caída" en el que dejamos de lado una declaración break a veces se puede usar para tu ventaja. En el siguiente ejemplo, tanto las manzanas como las fresas son rojas, por lo que registrar lo mismo para ambos no es realmente un problema:

```
let x = "Apples"

switch(x) {
  case "Apples":
  case "Strawberries":
    console.log("Apples and Strawberries can be red.")
    break
  case "Bananas":
```

```
    console.log("Bananas are yellow.")
  }
```

### Predeterminado una cláusula con declaraciones switch

Las declaraciones switch utilizan la palabra clave predeterminada para declarar lo que debe suceder si ninguno de los casos coincide. En el siguiente ejemplo, la consola mostrará "goodbye" ya que ninguna cláusula es correcta:

```
// Let's set x to 5
let x = 5
switch(x) {
  case 4: {
    console.log("hello")
    break
  }
  default: {
    console.log("goodbye")
    break
  }
}
```

### Igualdad con declaraciones switch

Al igual que nuestro ejemplo anterior con if...else, las declaraciones switch requieren igualdad “estricta”. Eso significa que debajo del capó, están usando el signo triple igual (===). La igualdad estricta significa que tanto el valor como el tipo de datos deben coincidir para que una cláusula sea verdadera. Esto a veces puede ser confuso en JavaScript ya que los tipos se infieren. En el siguiente ejemplo, tanto 5 como "5" tienen el mismo valor pero diferentes tipos. Como tal, solo se muestra "goodbye" en la consola:

```
// Let's set x to 5
let x = 5
switch(x) {
  case "5": {
    console.log("hello")
    break
  }
  case 5: {
    console.log("goodbye")
    break
  }
}
```



## Alcance de bloquea con declaraciones lógicas

Anteriormente, discutimos cómo los ámbitos de bloque afectan las variables `let` y `const`. Las declaraciones estándar `if...else` tienen por defecto ámbitos de bloque; como usan `{ }`, las declaraciones `switch` no, por sí mismas, no crean ámbitos de bloque. Podemos agregar el alcance del bloque a nuestras declaraciones `switch` encerrando cada cláusula con `{ }` si es necesario.

El resultado del siguiente código es un error ya que reasignamos `x` varias veces: `SyntaxError: Identifier 'x' has already been declared`:

```
// Let's set x to 5
let x = 5
switch(x) {
  case 5:
    let x = 6
    console.log("hello")
    break
  case 6:
    let x = 5
    console.log("goodbye")
    break
}
```

Para evitar este error, se puede crear un alcance de bloque agregando `{ }` a nuestra instrucción `switch`:

```
// Let's set x to 5
let x = 5
switch(x) {
  case 5: {
    let x = 6
    console.log("hello")
    break
  }
  case 6: {
    let x = 5
    console.log("goodbye")
    break
  }
}
```

## Operador condicional en variables

Hasta ahora hemos cubierto declaraciones que transmiten condicionalidad, pero la condicionalidad también se puede extender a declaraciones de variables.

Esto se realiza a través de un operador especial llamado operador condicional.

Puedes probar cualquier declaración dentro de una variable siguiéndola con un signo de interrogación. Esto es seguido por dos valores separados por un colon. El primer valor después del signo de interrogación será lo que muestra si la declaración es verdadera, y la segunda se devolverá si la declaración es falsa.

En el siguiente ejemplo, y se convierte en "Big number" ya que x es realmente más grande que 3:

```
let x = 5
// Returns "Big Number"
let y = x > 3 ? "Big Number" : "Small Number"
```

## Operadores de comparación de instrucciones lógicas

En nuestros ejemplos anteriores, hemos cubierto declaraciones lógicas básicas utilizando una igualdad estricta (===). Por ejemplo, declaraciones if...else:

```
if (myVariable === 5)
```

y declaraciones de cambio, que siempre usan una igualdad estricta.:

```
switch (variable)
```

Eso significa que tanto el valor como el tipo de datos deben coincidir para que la declaración sea verdadera. Por ejemplo, '5' no es el mismo tipo que 5, aunque los valores son los mismos, ya que los tipos son diferentes:

```
let myVariable = 5
if(myVariable === 5) // True!
if(myVariable === "5") // False!
```

Además de la igualdad estricta, hay muchas otras formas en que podemos comparar datos en declaraciones lógicas. En el siguiente ejemplo, utilizamos el operador más que el operador

para verificar si myVariable es más que valores específicos. El resultado de este código puede verse en la figura 6.

```
let myVariable = 5
if(myVariable > 4) console.log("Hello World") // True!
if(myVariable > 6) console.log("Hello World 2!") // False!
```



Figura 6: Se pueden usar otros operadores para probar declaraciones lógicas. En el ejemplo anterior, usamos el operador más que el operador para verificar si MyVariable cumple con ciertas condiciones.

También podemos verificar la igualdad “regular”, lo que ignora el tipo, usando el operador doble igual (==):

```
let myVariable = 5
if(myVariable == '5') // True!
```

Los símbolos que usamos para comparar datos en declaraciones lógicas generalmente se conocen como “operadores”. Todos los operadores a los que tenemos acceso se resumen en la tabla 1.

### Declaración lógica de operadores lógicos

A veces, tus declaraciones lógicas pueden requerir más que una declaración sea verdadera. Por ejemplo, verificar si algo es más que algo y menos que otra cosa. Existen tres operadores lógicos adicionales para combinar declaraciones, que se pueden encontrar en la tabla 2.

En el siguiente ejemplo, estoy verificando si una variable x es más de 5, pero también menos de o igual a 10 usando && y operador:

```
let x = 6
if(x > 5 && x <= 10) {
  console.log('Hello World!')
}
```

Tabla 1: Operadores de JavaScript para comparaciones lógicas.

Operadores	Definición	Ejemplo
===	Igualdad estricta. Tanto el tipo como el valor de los datos deben ser los mismos.	5 === 5 // True 5 === '5' // False
==	Igualdad regular. Solo el valor debe ser el mismo.	5 == 5 // True 5 == '5' // True
>	Más que. El valor a la izquierda debe ser más que la derecha.	6 > 5 // True 5 > 5 // False 4 > 5 // False
>=	Mayor o igual a. El valor a la izquierda debe ser más o igual al de la derecha.	6 >= 5 // True 5 >= 5 // True 4 >= 5 // False
<	Menor que. El valor a la izquierda debe ser menor que el de la derecha.	6 < 5 // False 5 < 5 // False 4 < 5 // True
<=	Menor que o igual a. El valor de la izquierda debe ser menor o igual al de la derecha.	6 <= 5 // False 5 <= 5 // True 4 <= 5 // True
!==	Estricto no igual a. Los dos valores no deben coincidir con el valor y el tipo.	5 !== 5 // False 5 !== '5' // True 5 !== 4 // True
!=	Regular no igual a. Los dos valores no deben coincidir con el valor.	5 != 5 // False 5 != '5' // False 5 != 4 // True

Tabla 2: Operadores lógicos de JavaScript.

Operadores	Definición
&&	AND - algo AND algo entonces es cierto (true).
	OR, algo OR algo entonces es cierto.
!	NOT - algo NOT es cierto.

Mientras tanto, en este ejemplo, verificamos si x es más de 5 OR más de 3, usando el operador || operador OR:

```
let x = 6
if(x > 5 || x > 3) {
  console.log('Hello World!')
}
```

También podemos usar el operador no para permitarnos invertir un resultado. Por ejemplo, en lo siguiente verificamos si X no es más de 5 (invirtiendo x > 5):

```
let x = 6
if(!(x > 5)) {
```

```
    console.log('Hello World!')  
  }
```

## Resumen

En este segundo documento, hemos cubierto algunos de los conceptos fundamentales más importantes de JavaScript. Esto ha incluido algunas de las mejores prácticas para escribir tu código. También hemos analizado cómo puedes usar variables para señalar los datos y la diferencia entre mutar datos en lugar de reasignarlos.

Hemos aprendido cómo actualizar esas variables a través de operadores de asignación y cómo se pueden usar en una variedad de declaraciones lógicas. También hemos discutido los comentarios y cómo agregarlos a tu código. Todos estos conceptos son realmente importantes y requeridos para los siguientes temas. Como tal, tener una buena comprensión de esto es fundamental para escribir JavaScript más complicado.