1. Gramática, comentarios, variables, ámbito/alcance, *hoisting*

Lo esencial

JavaScript

- distingue entre mayúsculas y **minúsculas** y utiliza el conjunto de **caracteres Unicode** .
  - Por ejemplo, la palabra Früh (que significa "temprano" en alemán) podría usarse como nombre de variable.

```
const Früh = "foobar";
```

- Las instrucciones se llaman **instrucciones** y están separadas por punto y **coma** (;).
  - un punto y coma no es necesario después de una declaración si está escrito en su propia línea.
  - pero si se desea más de una declaración en una línea, entonces *deben* estar separadas por punto y coma.

    Nota: ECMAScript también tiene reglas para la inserción automática de punto y coma (ASI) al final de las declaraciones. (Para obtener más información, consulte la referencia detallada sobre la gramática léxica de JavaScript).
  - Sin embargo, se considera una buena práctica escribir siempre un punto y coma después de una instrucción, incluso cuando no sea estrictamente necesario.
    - Esta práctica reduce las posibilidades de que se introduzcan errores en el código.
- El texto fuente del script JavaScript se escanea de izquierda a derecha y se convierte en una secuencia de elementos de entrada que son ***tokens*** , ***caracteres de control*** , ***terminadores de línea*** , ***comentarios*** o espacios en **blanco** . (Los espacios, las tabulaciones y los caracteres de nueva línea se consideran espacios en blanco).

JavaScript es un **lenguaje dinámico** y **multiparadigma**

- con
  - **tipos** y **operadores**,
  - **objetos incorporados** (built-in) estándar,
  - y **métodos**.
- su **sintaxis** se basa en los lenguajes Java y C: muchas estructuras de esos lenguajes también se aplican a JavaScript.
- admite la **programación orientada a objetos** con **prototipos de objetos** y **clases de objetos** .
- admite **la programación funcional** ya que las funciones son **funciones de primera clase** (funciones que pueden crearse fácilmente a través de expresiones y transmitirse como cualquier otro objeto).
  *Se dice que un lenguaje de programación tiene **funciones de primera clase** cuando las funciones en ese lenguaje se tratan como cualquier otra variable. Por ejemplo, en dicho lenguaje, una función puede pasarse como argumento a otras funciones, puede ser devuelta por otra función y puede asignarse como valor a una variable.*

Comentarios

La sintaxis de los **comentarios** es la misma que en C++ y en muchos otros lenguajes:

```
// un comentario de una línea

/* esto es más largo,
 * comentario de varias líneas
*/
```

No se pueden anidar comentarios unos dentro de otros:

- esto sucede a menudo cuando accidentalmente incluye una secuencia `*/` en su comentario, lo que terminará el comentario.

```
/* Sin embargo, no puede /* anidar comentarios */ SyntaxError */
```

En este caso, debes romper el patrón `*/` .

- por ejemplo, al insertar una **barra invertida** :

```
/* Puede /* anidar comentarios *\/ escapando de barras */
```

Los comentarios se comportan como espacios en blanco y se descartan durante la ejecución del script.

Nota: si JavaScript se ejecuta en un shell:

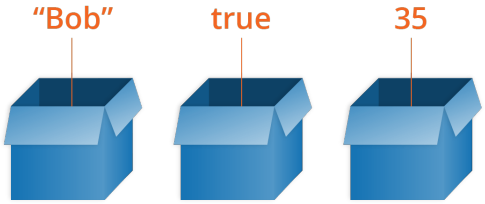
También puedes ver un tercer tipo de sintaxis de comentarios al comienzo de algunos archivos JavaScript, que se parece a esto: `#!/usr/bin/env node` .

Se llama sintaxis de **comentario hashbang** , y es un comentario especial que se usa para especificar la ruta a un motor de JavaScript
en particular que debe ejecutar el script. Consulte los comentarios de Hashbang para obtener más detalles.

Variables

Tenemos que:

- una **variable** es un contenedor de un valor, como un número que podríamos usar en una suma o una cadena que podríamos usar como parte de una sentencia.
- un **identificador de variable** es el nombre del contenedor
- decimos que las variables contienen **valores** .
  - esta es una distinción importante que hacer.
  - no son los valores mismos; son contenedores de valores.
  - puedes pensar en ellos como pequeñas cajas de cartón en las que puedes guardar cosas.



Utilizas nombres de **variables** como nombres simbólicos para **valores** en tu aplicación.

- los nombres de las variables, llamados **identificadores**, se ajustan a ciertas reglas.
- un identificador generalmente comienza con una letra, un guión bajo (**_**) o un signo de dólar (**$**).
  - los caracteres subsiguientes también pueden ser dígitos (**0** – **9**).
  - debido a que JavaScript distingue entre mayúsculas y minúsculas, las letras incluyen los caracteres de la **A** a la **Z** (mayúsculas) así como **de la a** a la **z** (minúsculas).
- puedes usar la mayor parte de los caracteres **ISO 8859-1** o caracteres **Unicode** como `å` y `ü` en los identificadores.
- también puedes usar las **secuencias de escape Unicode** como caracteres en los identificadores.
- no uses guiones bajos al comienzo de los nombres de variables: esto se usa en ciertas construcciones de JavaScript para significar cosas específicas, por lo que puede resultar confuso.
- no utilices dígitos al principio de las variables: esto no está permitido y provoca un error.
- una convención segura a seguir es la llamada " **lower camel case**", en la que se unen varias palabras, usando minúsculas para la primera palabra completa y luego mayúsculas para las palabras siguientes.
- hacer que los nombres de las variables sean intuitivos, para que describan los datos que contienen.
- no uses solo letras/números individuales, o frases grandes y largas.
- las variables distinguen entre **mayúsculas y minúsculas**: entonces `miEdad` es una variable diferente de `miEdad` .
- también debes evitar el uso de **palabras reservadas de JavaScript** como nombres de variables
  - con esto, nos referimos a las palabras que componen la sintaxis real de JavaScript.
  - por lo tanto, no puedes usar palabras como `var` , `function` , `let` y `for` como nombres de variables: los navegadores los reconocen como elementos de código diferentes, por lo que obtendrás errores.
- una cosa especial acerca de las variables es que pueden contener casi cualquier cosa:
  - no solo cadenas y números.
  - también puede contener datos complejos como cualquier tipo de objeto y función

Ejemplos de identificadores buenos:	Ejemplos de identificadores malos:

`anos`
`miEdad`
`inicial`
`colorInicial`
`valorSalidaFinal`
`audio1`
`audio2`
`exitos_Lucas`
`temp99`
`$credito`
`_nombre`
	`1`
	`a`
	`_12`
	`miedad`
	`MIEDAD`
	`var`
	`document`
	`skjfndskjfnbdskjfb`
	`esteesunnombredevariablemuymalo`

EJEMPLO

Ejemplo de variables

Veamos un ejemplo sencillo:

```
<button id="boton_A">Presioname</button>
<h3 id="encabezado_A"></h3>

const botonA      = document.querySelector('#boton_A');
const encabezadoA = document.querySelector('#encabezado_A');

boton_A.onclick = () => {
  const nombre = prompt('¿Cuál es tu nombre?');
  alert(`Hola ${nombre}, ¡encantado de verte!`);
  encabezadoA.textContent = `Bienvenido ${nombre}`;
}
```

En este ejemplo, al presionar el botón se ejecuta un código.

- la primera línea abre un cuadro en la pantalla que le pide al lector que ingrese su nombre y luego almacena el valor en una variable.
- la segunda línea muestra un mensaje de bienvenida que incluye su nombre, tomado del valor de la variable
- y la tercera línea muestra ese nombre en la página.

..... y sin variable

Para entender por qué esto es tan útil, pensemos cómo escribiríamos este ejemplo sin usar una variable. Terminaría luciendo algo como esto:

```
<button id="boton_B">Presioname</button>
<h3 id="encabezado_B"></h3>

const botonB = document.querySelector('#boton_B');
const encabezadoB = document.querySelector('#encabezado_B');

boton_B.onclick = () => {
alert(`Hola ${ prompt('¿Cuál es tu nombre?') }, ¡me alegro de verte!`);
encabezado_B.textContent = `Bienvenido ${prompt('¿Cuál es tu nombre?')}`;
}
```

- es posible que no comprendas completamente la sintaxis que estamos usando todavía, pero deberías poder hacerte una idea. Si no tuviéramos variables disponibles, ¡tendríamos que pedirle al lector su nombre cada vez que necesitáramos usarlo!
- las variables simplemente tienen sentido y, a medida que aprendas más sobre JavaScript, comenzarán a convertirse en lo normal.

Declarar una variable

Para **usar una variable**, primero tienes que crearla: más exactamente, llamamos a esto **declarar la variable** .

Puedes **declarar una variable** de dos maneras:

- **ámbito global**
  - con la palabra clave **`var`**.
  - por ejemplo, **`var x = 42`** .
  - esta sintaxis se puede utilizar para declarar tanto variables **locales** como **variables globales ,** según el ***contexto de ejecución*** .
- **ámbito de bloque**: **variable local**
  - con la palabra clave **`const`** (**solo lectura**) o **`let`** .
  - por ejemplo, **`let y = 13;`**
  - las **reglas de alcance para las constantes** son las mismas que para las **variables de alcance de bloque** **`let`** .

Ex:

```
let miNombre;
let miEdad;
```

- aquí estamos creando dos variables llamadas **`miNombre`** y **`miEdad`**.
- intenta escribir estas líneas en la consola de tu navegador web.
- después de eso, intenta crear una variable (o dos) con sus propias opciones de nombre.

Nota: en JavaScript, todas las instrucciones del código deben terminar con un punto y coma (`;`) — su código puede funcionar correctamente para líneas individuales, pero probablemente no funcionará cuando esté escribiendo varias líneas de código juntas.

Puedes probar si estos valores ahora existen en el **entorno de ejecución** escribiendo solo el nombre de la variable, por ejemplo

```
miNombre;
miEdad;
```

- actualmente no tienen valor;
- son **contenedores vacíos** .
- cuando ingresa los nombres de las variables, debe obtener un valor de retorno `undefined` .
- si no existen, obtendrás un **mensaje de error**; intente escribir

```
perroBonito;
```

Nota: no confundas una variable que existe pero que no tiene un **valor definido** con una variable que no existe en absoluto

- son cosas muy diferentes.
- en la analogía de las cajas que viste arriba, no existir significaría que no hay una caja (variable) para que entre un valor.
- ningún valor definido significaría que hay una caja, pero no tiene ningún valor dentro de ella.

Puedes declarar variables para desempaquetar valores utilizando la sintaxis de **asignación de desestructuración** .

- por ejemplo,

```
const foo = { nombre: "Juan", bar: "López", direccion: "Avenida Camelias 175" };

const {bar, nombre} = foo;
```

Esta última sentencia equivale a:

```
const bar = "López";
const nombre = "Juan";
```

let

Tenemos que:

- te permite declarar **variables a nivel de bloque** .
- la variable declarada está disponible desde el ***bloque*** en el que está encerrada.

```
let a;
let nombre = "Simón";

// miVariableLet *no* es visible aquí

for (let miVariableLet = 0; miVariableLet < 5; miVariableLet++) {
   // miVariableLet solo es visible aquí
}

// miVariableLet *no* es visible aquí
```

const

Tenemos que:

- te permite declarar variables cuyos **valores** se supone que no van a cambiar (el ***runtime*** se encarga de verificar que eso sea así)
- la variable está disponible solo desde el ***bloque*** en el que **se declara**.

```
const Pi = 3,14; // Declarar la variable Pi
console.log(Pi); // 3.14
```

Una variable declarada con `const` no se puede **reasignar** .

```
const Pi = 3,14;
Pi = 1;          // arrojará un error porque no puede cambiar una variable constante.
```

Las **declaraciones** `const` solo previenen las ***reasignaciones***: no previenen las ***mutaciones del valor*** de la **variable**, si es un objeto.

- puede actualizar, agregar o eliminar propiedades de un objeto declarado usando **`const`**, porque aunque el contenido del objeto haya cambiado, la constante aún apunta al mismo objeto:

```
const ave = { especie : 'Cernicalo'};
console.log(ave.especie);              // "Cernicalo"

ave.especie = 'Caracara Estriada';
console.log(ave.especie);             // "Caracara Estriada"
```

No puedes declarar una constante con el mismo nombre que una función o variable en el mismo **ámbito**; por ejemplo:

```
// ESTO CAUSARÁ UN ERROR
function f() {};
const f = 5;

// ESTO TAMBIÉN CAUSARÁ UN ERROR
function f() {
  const g = 5;
  var g;
//declaraciones
}
```

var

- las **declaraciones con `var`** pueden tener comportamientos sorprendentes (por ejemplo, no tienen **un alcance de bloque**),
- y se desaconsejan en el código JavaScript moderno.

Probablemente también verá una forma diferente de declarar variables, usando la palabra clave **`var`** :

```
var miNombre;
var miEdad;
```

- cuando se creó JavaScript por primera vez, esta era la única forma de declarar variables.
- el diseño de **`var`** es confuso y propenso a errores.
- así que **`let`** se creó en las versiones modernas de JavaScript, una nueva palabra clave para crear variables que funciona de manera algo diferente a **`var`** , solucionando sus problemas en el proceso.

Problemas con `var` :

1) Para empezar, si escribes un programa JavaScript de varias líneas que **declara** e **inicializa** una variable, puedes declarar una variable con **`var`** después de inicializarla y seguirá funcionando. Por ejemplo:

```
miNombre = 'Chris';

function nombreRegistro() {
   console.log(miNombre);
}

nombreRegistro();

var miNombre ;
```

- esto funciona debido a la **elevación** (***hoisting***)
- **elevar** no funciona con **`let`** .
  - si cambiamos **`var`** por **`let`** en el ejemplo anterior, fallaría con un error.
  - esto es algo bueno: declarar una variable después de inicializarla da como resultado un código confuso y más difícil de entender.

2) En segundo lugar, cuando usas **`var`**, puedes declarar la misma variable tantas veces como quiera, pero con **`let`** no puede. Lo siguiente funcionaría:

```
var miNombre = 'Chris';
var miNombre = 'Bob';
```

- pero lo siguiente **arrojaría un error** en la segunda línea:

```
let miNombre = 'Chris';
let miNombre = 'Bob';
```

- tendrías que hacer esto en su lugar:

```
let miNombre = 'Chris';
miNombre = 'Bob' ;
```

- nuevamente, esta es una decisión de lenguaje sensata: no hay razón para volver a **declarar variables**, solo hace que las cosas sean más confusas.

Por estas razones y más, te recomendamos que uses `let` en tu código, en lugar de `var` .

- ya no hay ninguna razón para usar `var`, ya que `let` se admite desde Internet Explorer 11.

CONSEJO PRÁCTICO CONSOLA DE CHROME:

```
> let miNombre = 'Chris';
let miNombre = 'Bob';
// Como una entrada: SyntaxError : el identificador 'miNombre' ya ha sido declarado

> let miNombre = 'Chris';
> let miNombre = 'Bob';
// Como dos entradas: ambas tienen éxito
```

CONSEJOS: Cuando usar `const` y cuando usar `let`

1) ESCAPAR DE var:
Las variables siempre deben declararse antes de que se utilicen.

- JavaScript solía permitir la asignación a **variables no declaradas**, lo que crea una **variable global no declarada** .
- este es un error en **modo estricto** y debe evitarse por completo.

2) Usar `const` siempre por defecto y, sólo si es necesario cambiar el valor, usar `let`
Si no puedes hacer tanto con `const` como con `let`, ¿por qué preferirías usarlo en lugar de `let` ?

- de hecho `const` es muy útil.
- si usas `const` para nombrar un valor,
  - le dice a cualquiera que mire tu código que este identificador está pensado para que nunca se le asigne un valor diferente.
  - cada vez que otro programador vea este identificador, sabrán a qué se refiere.

En este curso, adoptamos el siguiente principio sobre cuándo usar `let` y cuándo usar `const` :

- usar `const` cuando puedas, y usa `let` cuando tengas que hacerlo: esto significa que si puedes inicializar una variable cuando la declaras, y no necesita reasignarla más tarde o cambiar su valor, conviértela en una constante.

Inicializar una variable

Una vez que hayas declarado una variable, puedes **inicializarla** con un valor.

- lo haces escribiendo el nombre de la variable, seguido de un signo igual (`=`), seguido del valor que desea darle. Por ejemplo:

```
let miNombre;
let miEdad;

miNombre = 'Chris';
miEdad = 37;
```

- intenta volver a la consola ahora del navegador y escribe estas líneas.
- deberías ver el valor que has asignado a la variable devuelto en la consola para confirmarlo, en cada caso.
- de nuevo, puedes devolver los valores de sus variables escribiendo su nombre en la consola; inténtela de nuevo:

```
miNombre
miEdad;
```

Declaración e inicialización

Puedes **declarar e inicializar una variable** al mismo tiempo, así:

```
let miPerro = 'Rover';
```

Esto es probablemente lo que harás la mayor parte del tiempo, ya que es más rápido que hacer las dos acciones en dos líneas separadas.

En una declaración como `let miPerro = 'Rover';`

- la parte `let miPerro` se llama ***declaración*** ,
  - la **declaración** permite acceder a la variable más adelante en el código sin arrojar un `ReferenceError` ,
- la parte `= 'Rover'` se llama ***inicializador*** .
  - mientras que el **inicializador** asigna un valor a la variable.

En las declaraciones `var` y `let`, el inicializador es opcional.

- si una variable se declara sin un inicializador, se le asigna el valor `undefined` .

Las declaraciones `const` siempre necesitan un **inicializador**, porque prohíben cualquier tipo de asignación después de la declaración, y es probable que inicializarlas implícitamente con `undefined` sea un error del programador.

Actualizar una variable

Una vez que una variable se ha inicializado con un valor, puede cambiar (o actualizar) ese valor dándole un valor diferente.

```
let miPerro = 'Rover';
miPerro = 'Tipi';
miPerro = 'Romeo';
```

Alcance variable (*variable scope*)

Una variable puede tener uno de los **alcances/ámbitos** (***scopes***) siguientes:

- **Alcance global** :
  - el ámbito predeterminado para todo el código que se ejecuta en modo script.
  - cuando declara una variable fuera de cualquier función, se llama **variable *global*** , porque está disponible para cualquier otro código en el documento actual.
- **Alcance de función** :
  - el ámbito creado con una **función** (declarada con palabra clave `function`)
  - cuando declara una variable dentro de una función, se llama **variable *local***, porque solo está disponible dentro de esa función.
- **Alcance de bloque**: el ámbito creado con un par de llaves (un **bloque de sentencias**): las variables declaradas con `let` o `const` pueden pertenecer a un bloque.
- **Alcance de módulo**: el ámbito del código que se ejecuta en **modo módulo** .

Variables globales

Las variables globales son, de hecho, propiedades del ***objeto global (`globalThis`)***.

- en las páginas web, el objeto global es `window`, por lo que puede establecer y acceder a las variables globales usando la sintaxis `window.variable`
- en todos los entornos de JavaScript (front-end, back-end, consola), puede utilizar la variable `globalThis` (que en sí misma es una variable global) para acceder a las variables globales.

En consecuencia, puedes acceder a las variables globales declaradas en una ventana o marco desde otra ventana o marco especificando el nombre de la `window` o `frame`.

- por ejemplo, si en un documento se declara una variable denominada `numeroDeTelefono`, puede hacer referencia a esta variable desde un iframe como `parent.numeroDeTelefono` .

Ámbito local y de bloque

EJEMPLOS:

1) El alcance de `let` y `const` también se limita al **bloque de declaración** en la que se declaran.

```
if (Math.random() > 0.5) {
   const y = 5;
}
console.log(y); // ReferenceError : y no está definido
```

2) Sin embargo, las variables creadas con **`var`** no tienen alcance de bloque, sino *alcance global* en el que reside el bloque.

- por ejemplo, el siguiente código registrará **`5`**, porque el alcance de **`x`** es el **contexto global** (o el contexto de la función si el código es parte de una función).
- el alcance de **`x`** no se limita al bloque de declaración **`if`** inmediato.

```
if (true) {
   var x = 5;
}
console.log(x); // x es 5
```

Si **declaras una variable** sin asignarle ningún valor, su valor es **`undefined`** .

- no puedes declarar una variable **`const`** sin un **inicializador**, porque de todos modos no puede cambiarla más tarde.
- las variables declaradas **`let`** y **`const`** aún ocupan todo el **ámbito** en el que están definidas y se encuentran en una región conocida como **zona muerta temporal** antes de la línea real de declaración.
  - esto tiene algunas interacciones interesantes con la **sombreado de variable** (***variable shadowing***), que no ocurre en otros lenguajes de programación.

```
function foo(x, condicion) {
   if (condicion) {
     console.log(x);
     const x = 2;
     console.log(x);
   }
}

foo(1, true);
```

- en la mayoría de los otros idiomas, esto registraría "1" y "2", porque antes de la línea **`const x = 2`** , x aún debería referirse al parámetro x en el **alcance superior**.
- en JavaScript, debido a que cada declaración ocupa todo el **alcance** , esto **arrojaría un error** (**throw** an error) en el primer **`console.log`** : " **`No se puede acceder a 'x' antes de la inicialización`** ".

Zona muerta temporal (TDZ)

Una variable **`let`** o **`const`**

- se dice que está en una " **zona muerta temporal** " (**TDZ**) desde el inicio del bloque hasta **que la ejecución del código** llega a la línea donde se declara e inicializa la variable .
- mientras que dentro de la TDZ, la variable no se ha inicializado con un valor, y cualquier intento de acceder a ella resultará en un **`ReferenceError`** .
- la variable se inicializa con un valor cuando la ejecución llega a la línea de código donde se declaró.
- si no se especificó ningún valor inicial con la declaración de la variable, se inicializará con un valor de **`undefined`** .
- esto difiere de las variables **`var`**, que devolverán un valor **`undefined`** si se accede a ellas antes de que se declaren.

El siguiente código demuestra el resultado diferente cuando se accede a **`let`** y **`var`** en el código antes de la línea en la que se declaran.

```
{ // TDZ comienza al principio del alcance
  console.log(barra); // undefined
  console.log(foo);   // ReferenceError
  var barra = 1;
  let foo = 2;        // Fin de TDZ (para foo)
}
```

El término "temporal" se usa porque la zona depende del **orden de ejecución** (=tiempo = momento) en lugar del **orden en que se escribe el código** (posición).

- por ejemplo, el siguiente código funciona porque, aunque la función que usa la variable **`let`** aparece antes de que se declare la variable, la función se llama fuera de la TDZ.

```
{
  // TDZ comienza al principio del alcance
  const miFuncion = () => console.log(letVariable); // CORRECTO!

  // en esta zona, dentro del acceso TDZ, letVariable arroja `ReferenceError`

  let letVariable = 3; // Fin de TDZ (para letVariable)
  miFuncion(); // ¡Llamado fuera de TDZ!
}
```

Elevación de variables y elevación de funciones funcional

Variables

Las variables declaradas con **`var`** son **elevadas** (***hoisted***) (elevadas en el texto del código JavaScript a la parte de arriba de todo):

- lo que significa que puede hacer referencia a la variable en cualquier lugar de su **alcance**, incluso si aún no se ha alcanzado su declaración.
- puedes imaginarte la declaración **`var`** como "elevadas" a la parte superior de su función o alcance global, aunque aparezca más abajo
- sin embargo, si accede a una variable antes de que se declare, el valor siempre es **`undefined`**, porque solo se eleva su *declaración, pero no su* ***inicialización*** .

	Equivale a:

Ejemplo 1:
 	
`console.log(x === undefined); // true`
`var x = 3;`
	`var x;  // declaración elevada,`
	`// pero no su inicialización`
	`console.log(x === undefined); // verdadero`
	`x = 3;`

Ejemplo 2:
 	
`y = 3;`
`console.log(y === undefined); // false`
`var y;`
	`var y; // declaración elevada`
	`y = 3;`
	`console.log(y === undefined);  // false`

Ejemplo 3: (función anónima)
 	
`(function() {`
`  console.log(x);  // undefined`
`  var x = 'valor local';`
`})();`
	`(function() {`
	`  var x;  // declaración elevada`
	`  console.log(x);  // undefined`
	`  x = 'valor local';`
	`})();`

Debido a la elevación (hoisting),

- todas las declaraciones **`var`** en una función deben colocarse lo más cerca posible de la parte superior de la función.
- esta mejor práctica aumenta la claridad del código.

Funciones

Elavación de funciones: a diferencia de las declaraciones **`var`**, que solo elevan la declaración pero no su valor , **las declaraciones de funciones** se elevan por completo: puede llamar a la función de manera segura en cualquier lugar de su alcance.