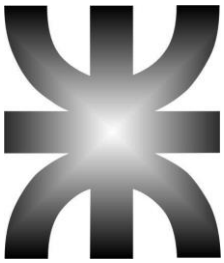

# VAR, LET Y CONST

## UNIDAD #5 - JS

*TUP - Laboratorio de Computación II*

*MATERIAL COMPLEMENTARIO*

## Diferencia entre var, let y const

En la mayoría de los lenguajes de programación, el alcance de las variables locales está vinculado al bloque donde se declaran. Como tal, "mueren" al final de la instrucción en la que se realizan. ¿Esto también se aplica al lenguaje JavaScript? Vamos a revisar:

```
var exibeMensaje = function() {
    var mensajeFueraDelIf = 'Caelum';
    if(true) {
        var mensajeDentroDelIf = 'Alura';
        console.log(mensajeDentroDelIf)// Alura ;
    }
    console.log(mensajeFueraDelIf); // Caelum

    console.log(mensajeDentroDelIf); // Alura
}
```

Estamos declarando dos variables en diferentes bloques de código, ¿cuál será el resultado? Vamos a probar:

```
exibeMensaje(); // Imprime 'Alura', 'Caelum' y 'Alura'
```

Si se declaró `mensajeDentroDelIf` dentro del `if`, ¿por qué todavía tenemos acceso a él fuera del bloque de esta declaración?

## Usar antes de la declaración

A continuación, se muestra otro ejemplo de código JavaScript:

```
var exibeMensaje = function () {
    mensaje = 'Alura';
    console.log (mensaje);
```

```
    var mensaje;
}
```

Tenga en cuenta que estamos declarando la variable del mensaje solo después de asignar un valor y mostrarlo en el registro, ¿funciona? ¡Vamos a probar!

```
exibeMensaje(); // Imprime 'Alura'
```

¡Funciona! ¿Cómo es posible usar la variable mensaje incluso antes de declararla? ¿El alcance está garantizado solo dentro del lugar donde se creó la variable?

## Hoisting

En JavaScript, cada variable es **"elevada"** ***(hoisting)*** a la parte superior de su contexto de ejecución. Este mecanismo mueve las variables a la parte superior de su alcance antes de que se ejecute el código.
En nuestro ejemplo anterior, como la variable `mensaje` está dentro de una *función*, su declaración se eleva (*alzando*) a la parte superior de su contexto, es decir, a la parte superior de la *función*.
Es por esta misma razón que "es posible usar una variable antes de que se haya declarado": en tiempo de ejecución, la variable se elevará (*hoisting*) y todo funcionará correctamente.

## var

Considerando el concepto de *hoisting*, hagamos una pequeña prueba usando una variable declarada con var incluso antes de que se haya declarado:

```
void function() {
    console.log(mensaje);
} ();
var mensaje;
```

En el caso de la palabra clave `var`, además de la variable que se iza(*hoisting*), se inicializa automáticamente con el valor indefinido (si no se asigna ningún otro valor).
De acuerdo, pero ¿cuál es el impacto que tenemos cuando hacemos este tipo de uso? Imagine que nuestro código contiene muchas líneas y que su complejidad no es tan trivial de entender.

A veces, queremos declarar variables que se usarán solo dentro de una pequeña porción de nuestro código. Tener que lidiar con el alcance de la función de las variables declaradas con `var` (alcance integral) puede confundir las mentes hasta de los programadores más experimentados.

## “var” no tiene alcance (visibilidad) de bloque.

Las variables declaradas con `var` pueden: tener a la función como entorno de visibilidad, o bien ser globales. Su visibilidad atraviesa los bloques.

Por ejemplo:

```
if (true) {
    var test = true; // uso de "var" en lugar de "let"
}

alert(test); // true, la variable vive después del if
```

Como `var` ignora los bloques de código, tenemos una variable global `test`.

Si usáramos `let test` en vez de `var test`, la variable sería visible solamente dentro del `if`:

```
if (true) {
    let test = true; // uso de "let"
}
alert(test); // ReferenceError: test no está definido
```

Lo mismo para los bucles: `var` no puede ser local en los bloques ni en los bucles:

```
for (var i = 0; i < 10; i++) {
    var one = 1;
    // ...
}

alert(i); // 10, "i" es visible después del bucle, es una variable global
alert(one); // 1, "one" es visible después del bucle, es una variable global
```

Si un bloque de código está dentro de una función, `var` se vuelve una variable a nivel de función:

```
function sayHi() {
    if (true) {
        var phrase = "Hello";
    }
```

```
    alert(phrase); // funciona
}

sayHi();
alert(phrase); // ReferenceError: phrase no está definida
```

Como podemos ver, `var` atraviesa `if`, `for` u otros bloques. Esto es porque mucho tiempo atrás los bloques en JavaScript no tenían ambientes léxicos. Y `var` es un remanente de aquello.

## "var" tolera redeclaraciones

Declarar la misma variable con `let` dos veces en el mismo entorno es un error:

```
let user;
let user; // SyntaxError: 'user' ya fue declarado
Con var podemos redeclarar una variable muchas veces.Si usamos var con una
variable ya declarada, simplemente se ignora:
    var user = "Pete";

var user = "John"; // este "var" no hace nada (ya estaba declarado)
// ...no dispara ningún error

alert(user); // John
```

## Las variables "var" pueden ser declaradas debajo del lugar en donde se usan

Las declaraciones `var` son procesadas cuando se inicia la función (o se inicia el script para las globales).

En otras palabras, las variables `var` son definidas desde el inicio de la función, no importa dónde esté tal definición (asumiendo que la definición no está en una función anidada).

Entonces el código:

```
function sayHi() {
    phrase = "Hello";

    alert(phrase);

    var phrase;
```

```
}
sayHi();
```

...es técnicamente lo mismo que esto (se movió `var phrase` hacia arriba):

```
function sayHi() {
    var phrase;

    phrase = "Hello";

    alert(phrase);
}
sayHi();
```

...O incluso esto (recuerda, los códigos de bloque son ignorados):

```
function sayHi() {
    var phrase;

    phrase = "Hello";

    alert(phrase);
}
sayHi();
```

Este comportamiento también se llama “hoisting” (elevamiento), porque todos los `var` son “hoisted” (elevados) hacia el tope de la función.

Entonces, en el ejemplo anterior, la rama `if (false)` nunca se ejecuta, pero eso no tiene importancia. El `var` dentro es procesado al iniciar la función, entonces al momento de `(*)` la variable existe.

**Las declaraciones son “hoisted” (elevadas), pero las asignaciones no lo son.**

```
Es mejor demostrarlo con un ejemplo:
function sayHi() {
  alert(phrase);

  var phrase = "Hello";
}

sayHi();
```

La línea `var phrase = "Hello"` tiene dentro dos acciones:

1. La declaración `var`
2. La asignación `=`.

La declaración es procesada al inicio de la ejecución de la función (“hoisted”), pero la asignación siempre se hace en el lugar donde aparece. Entonces lo que en esencia hace el código es:

```
function sayHi() {
    var phrase; // la declaración se hace en el inicio...

    alert(phrase); // undefined

    phrase = "Hello"; // ...asignación - cuando la ejecución la alcanza.
  }

  sayHi();
```

Como todas las declaraciones `var` son procesadas al inicio de la función, podemos referenciarlas en cualquier lugar. Pero las variables serán indefinidas hasta que alcancen su asignación.

En ambos ejemplos de arriba `alert` se ejecuta sin un error, porque la variable `phrase` existe. Pero su valor no fue asignado aún, entonces muestra `undefined`.

Conociendo las "complicaciones" que pueden causar las variables declaradas con `var`, ¿qué podemos hacer para evitarlas?

## let

Fue pensando en traer el alcance de bloque (tan conocido en otros lenguajes) que ECMAScript 6 tenía la intención de proporcionar la misma flexibilidad (y uniformidad) para el lenguaje.
Usando la palabra clave `let`, podemos declarar variables con alcance de bloque. Vamos a ver:

```
var exibeMensaje = function () {
    if (true)) {
        var scopeFunction = 'Caelum';
        let scopeBlock = 'Alura';
```

```
        console.log(scopeBlock); // Alura
    }
    console.log(scopeFunction); // Caelum
    console.log(scopeBlock);
}
```

¿Cuál será la salida del código anterior?

```
exibeMensaje(); // Imprime 'Alura', 'Caelum' y da un error
```

Tenga en cuenta que cuando intentamos acceder a una variable que ha sido declarada usando la palabra clave `let` dejada fuera de su alcance, se presentó el error *Uncaught ReferenceError: scopeBlock no está definido*.
Por lo tanto, podemos usar `let` sin problemas, ya que el alcance de bloque está garantizado.

## const

Aunque let garantiza el alcance, todavía existe la posibilidad de declarar una variable con `let` y ella ser *undifined*. Por ejemplo:

```
void function () {
    let mensaje;
    console.log(mensaje); // Imprime indefined
}();
```

Suponiendo que tenemos una variable que queremos garantizar su inicialización con un cierto valor, ¿cómo podemos hacer esto en JavaScript sin causar una inicialización *default* con *undefined*?
Para tener este tipo de comportamiento en una variable en JavaScript, podemos declarar constantes usando la palabra clave `const`. Echemos un vistazo al ejemplo:

```
void function () {
    const mensaje = 'Alura';
    console.log (mensaje); // Alura
    mensaje = 'Caelum';
} ();
```

El código anterior genera un *Uncaught TypeError* : *Assignment to constant variable*, ya que el comportamiento fundamental de una constante es que una vez que se le asigna un valor, no se puede cambiarlo.
Al igual que las variables declaradas con la palabra clave `let`, las constantes también tienen un alcance de bloque.
Además, las constantes deben inicializarse en el momento de la declaración. Aquí hay unos ejemplos:

```
// constante válida
const edad = 18;

// constante inválida: ¿dónde está la inicialización?
const pi;
```

En el código anterior, tenemos el ejemplo de una constante de edad que se declara e inicializa en la misma línea (constante válida) y otro ejemplo en el que el valor no se asigna en la declaración pi (constante no válida) que causa el error *Uncaught SyntaxError*: *Missing initializer in const declaration*.
Es importante usar `const` para declarar nuestras variables, porque de esa manera obtenemos un comportamiento más predecible, ya que el valor que reciben no se puede cambiar.

## Referencias


https://www.aluracursos.com/blog/comprenda-diferencia-entre-var-let-y-const-en-javascript--amp?gclid=CjwKCAjw4c-ZBhAEEiwAZ105RQRWWtMeE0XXFR0n3tO4Y5p65MDSDv-dqna5jyj9vrEcLw29t8fvUhoCIzwQAvD_BwE

https://es.javascript.info/var

https://es.javascript.info/variables