

JavaScript es un lenguaje de programación con el **ámbito global** como ámbito, visibilidad o *scope* predeterminado y en el que todo se pasa por referencia también de forma predeterminada. Esto significa que una variable declarada fuera de una función es una **variable global** y es pasada por referencia a scopes descendientes o herederos:

```
i = "global";
function foo() {
  i = "local";
  console.log(i); // local
}
foo();
console.log(i); // local
```

En el snippet anterior se puede ver como la variable `i` es una variable del ámbito global y **es sobreescrita dentro de la función** por ser pasada por referencia; la referencia al identificador `i` dentro y fuera de la función es la misma referencia. Sería equivalente a declarar la variable con el keyword `var` :

```
var i = "global";
function foo() {
  i = "local";
  console.log(i); // local
}
foo();
console.log(i); // local
```

Sin embargo, si declaramos la variable con `var` también dentro de la función, se crea una **variable local** cuya visibilidad se reduce a la propia función donde ha sido declarada:

```
var i = "global";
function foo() {
  var i = "local"; // Otra variable local solo para esta función
  console.log(i); // local
}
foo();
console.log(i); // global
```

Ahora hay dos variables con el mismo identificador pero en ámbitos diferentes y con referencias en memoria diferentes. La variable `i` dentro de la función tiene un *scope* reducido al ámbito de la función y no sobreescibe la variable global.

En resumen, **la declaración con `var` define una variable en el ámbito local actual** (lease función) y se hereda a scopes descendientes por referencia. Si la variable es declarada fuera de una función, la variable será una variable global.

Un último apunte sobre el ámbito de `var`, es que una variable declarada exclusivamente dentro de una función, no existe fuera de la misma, pues sería una variable local para esa función:

```
function foo() {  
    var i = "local";  
}  
console.log(i); // ReferenceError: i is not defined
```

Hoisting

Además del ámbito de aplicación visto antes, una variable declarada con `var` es sometida a *hoisting* («izamiento» o «levantamiento»): la declaración de la variable es «levantada» hasta el inicio del ámbito de aplicación pero la asignación al valor permanece en el sitio donde se realice.

Si intentamos acceder a su valor antes de que se asigne el valor, obtendremos un valor indefinido (`undefined`),:

```
console.log(i); // undefined  
var i = 1;
```

Este comportamiento se puede entender como «la variable `i` ha sido declarada en el programa pero en el momento de intentar acceder a ella aún no tenía un valor asignado». La interpretación sería similar al siguiente código:

```
var i; // Variable declarada pero valor no definido  
console.log(i); // undefined  
i = 1;  
console.log(i); // 1
```

Sin embargo, **si la variable no es declarada en absoluto obtendremos un `ReferenceError`**, que no es lo mismo que obtener un valor indefinido:

```
console.log(x); // ReferenceError: x is not defined
var i = 1;
```

Debido a este comportamiento, se suele recomendar mover todas las declaraciones de variables al inicio del scope aunque no se asigne valor alguno, de esta forma se evitan estos posibles errores de ejecución.

El hoisting no es posible en variables declaradas con `let` o `const`; estas variables siempre darán un `ReferenceError` si se intenta acceder a ellas antes de que sean declaradas:

```
console.log(x); // ReferenceError: x is not defined
let x = 1;
```

Declaración de variables con `let` y `const`

`let` y `const` son dos formas de declarar variables en JavaScript introducidas en ES6 que **reducen el ámbito de la variable a bloques** (con `var` el ámbito era la función actual) y **no admiten hoisting**. Además, **las variables declaradas con `const` no pueden ser reasignadas** (aunque no significa que su valor sea inmutable, como veremos a continuación).

`let`

Un bloque en JavaScript se puede entender como «lo que queda entre dos corchetes», ya sean definiciones de funciones o bloques `if`, `while`, `for` y loops similares. Si una variable es declarada con `let` en el ámbito global o en el de una función, la variable pertenecerá al ámbito global o al ámbito de la función respectivamente, de forma similar a como ocurría con `var`.

Por ejemplo, en el siguiente snippet la variable `i` es una variable global y la variable `j` es una variable local:

```

let i = 0;
function foo() {
  i = 1;
  let j = 2;
  if(true) {
    console.log(i); // 1
    console.log(j); // 2
  }
}
foo();

```

Pero si declaramos una variable con `let` dentro un bloque que a su vez está dentro de una función, la variable pertenece solo a ese bloque:

```

function foo() {
  let i = 0;
  if(true) {
    let i = 1; // Sería otra variable i solo para el bloque if
    console.log(i); // 1
  }
  console.log(i); // 0
}
foo();

```

Fuera del bloque donde se declara con `let`, la variable no está definida:

```

function foo() {
  if(true) {
    let i = 1;
  }
  console.log(i); // ReferenceError: i is not defined
}
foo();

```

Debido a este comportamiento, muchos desarrolladores se inclinan hacia `let` como la forma predeterminada de declarar variables en JavaScript y abandonar `var` (1 (<https://medium.com/craft-academy/javascript-variables-should-you-use-let-var-or-const-394f7645c88f>), 2 (<https://medium.com/javascript-scene/javascript-es6-var-let-or-const-ba58b8dcde75>), 3 (<https://hackernoon.com/js-var-let-or-const-67e51dbb716f>)), pues el scope más específico previene la sobreescritura de variables de forma accidental al declarar variables sin ensuciar el scope superior.

const

El ámbito o scope para una variable declarada con `const` es, al igual que con `let`, el bloque, pero si la declaración con `let` previene la sobreescritura de variables, `const` **directamente prohíbe la reasignación de valores** (`const` viene de *constant*).

Con `let` una variable puede ser reasignada:

```
function foo() {  
  let i = 0;  
  if(true) {  
    i = 1;  
  }  
  console.log(i); // 1  
}  
foo();
```

Con `const` no es posible; **si se intenta reasignar una variable constante se obtendrá un error tipo** `TypeError`:

```
const i = 0;  
i = 1; // TypeError: Assignment to constant variable
```

Pero que no se puedan reasignar no significa que sean inmutables. Si el valor de una variable constante es «algo» mutable, como **un array o un objeto**, se pueden cambiar los valores de sus elementos.

Por ejemplo, una variable constante se puede asignar a un objeto con determinadas propiedades. Aunque la variable no se pueda asignar a un nuevo valor, si se puede cambiar el valor de sus propiedades. Esto sería posible:

```
const user = { name: 'Juan' };  
user.name = 'Manolo';  
console.log(user.name); // Manolo
```

Pero esto no sería posible:

```
const user = 'Juan';  
user = 'Manolo'; // TypeError: Assignment to constant variable
```

Es decir, las variables declaradas con `const` son constantes (no reasignables) pero pueden ser mutables. Si es un objeto, incluso se podrían crear nuevas propiedades:

```
const user = { name: 'Juan' };
user.surname = 'Padial';
console.log(user); // {name: 'Juan', surname: 'Padial'}
```

Resumen

1. `var` declara una variable de **scope global o local para la función** sin importar el ámbito de bloque. Permite **hoisting**.
2. `let` declara una variable de **scope global, local para la función o de bloque**. Es **reassignable** y no permite hoisting.
3. `const` declara una variable de scope global, local para la función o de bloque. **No es reassignable, pero es mutable**. No permite hoisting.

En general, `let` sería todo lo que se necesita dentro de un bloque, función o ámbito global. `const` sería para variables que no van a sufrir una reasignación. `var` se puede relegar para cuando necesitemos hacer hoisting, vamos, casi nunca.