Diferencias entre var, let y const ------- ámbitos/scopes

var define/declaramos variables en dos ámbitos diferentes, el primero el **ámbito global**. Si la declaramos dentro de la función sólo será accesible en el **ámbito a nivel de función** (*esto hasta el ES5*) pero **admite hoisting** (**Debido a un concepto llamado *hoisting*, no importa el lugar en que se declaró una variable, ésta siempre se moverá hasta el principio de una función//bloque por lo que podemos acceder a una variable aunque aún no sea declarada sin generar un error (aunque su valor es `undefined`))**

En ES6 aparecen:

let funciona como variable de ámbito bloque (entendiendo por bloque lo que está dentro de { }) y **no admite Hoisting**
Si la definimos en el bloque global, tendrá ámbito global.

const el ámbito de const es el bloque (como let) pero directamente **prohíbe la reasignación de valores en variables simples**, aunque si son **objects** podemos *añadir/borrar/cambiar el contenido* de las propiedades y si son **arrays** podemos añadir/eliminar elementos del array.

¿Cuál usar? ¿Const o Let? Depende de la situación, personalmente recomiendo usar **`let`** cuando sepamos que el elemento puede sufrir mutación y/o cambios en un futuro. Y **`const`** cuando sepamos que nuestro elemento no va a sufrir ninguna alteración.

Hoisting

Con var si existe el hoisting en variables declaradas y sin valor asignado dará *undefined* no *ReferenceError* como sucede con let y const

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;
```

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 **reasignable** y no permite hoisting.
3. `const` declara una variable de scope global, local para la función o de bloque. **No es reasignable, 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.

Otro autor:

Conclusión

La palabra `let` fue introducida en ES6 para declarar variables definidas solo en un bloque de código, y aunque es de mucha utilidad, también es importante mecionar que `let` no es un reemplazo de `var`, no debemos empezar a reemplazar en todos nuestros archivos `var` por `let`, más bien debemos utilizarlas en conjunto para aprovecharlas al máximo.

https://medium.com/@germancutraro/var-let-y-const-js-es6-ed9222cad422

En ES6 ya tenemos varias formas de declarar nuestras variables. En ES5 lo hacíamos con la palabra **`var`** ***como en el siguiente ejemplo:***

```
var numero = 3;
```

El problema de **`var`** es el *scope* que genera, es decir el ámbito que crea para la variable en cuestión.

Podemos ver el problema en el siguiente ejemplo:

```
for (var i = 1; i <= 5; i++) {
  console.log(i);
}
```

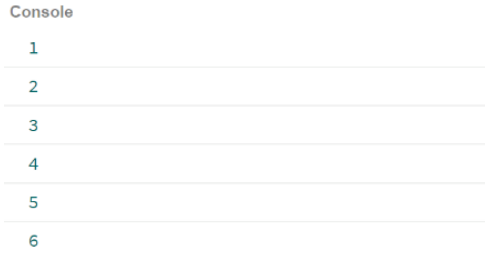
Tenemos un bucle ***for*** que imprime del uno al cinco.

Ahora bien, si insertamos un ***console.log*** fuera del bucle vemos como sigue accediendo al valor de la variable, de esta forma, imprime el siguiente valor (6).

```
for (var i = 1; i <= 5; i++) {
  console.log(i);
}

console.log(i);
```

Se puede acceder al valor de la variable fuera del bucle



Si la variable fuese de bloque no podría imprimir el número 6

Por lo tanto, se recomienda dejar de usar **`var`** y de esta forma, reemplazarla por **`let`** y **`const`.**

Let

`let` funciona como variable de bloque. Ahora bien, ¿Qué es un bloque?

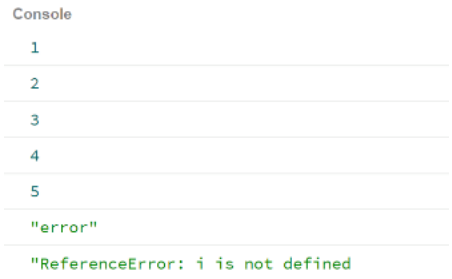
Un bloque es básicamente una estructura de código creada por llaves.

Sintaxis

Veamos el mismo ejemplo pero usando **`let`:**

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

console.log(i);
```



Según la consola el mensaje de error será más o menos parecido

Como podemos observar si utilizamos **`let` la variable solo existirá dentro del bucle** por lo tanto no podremos acceder a ella fuera del bloque. Este ejemplo funcionaría para un condicional o para cualquier otro caso de bloque.

Const

Const tiene un funcionamiento parecido a **`let`,** con la diferencia que el **valor de una constante no puede cambiarse a través de la reasignación, y no se puede redeclarar.**

Sintaxis

```
const numero = 3;
```

Como se mencionó, no se puede reasignar una variable constante, por lo tanto, si quisiéramos hacer:

```
const numero = 3;
numero = 10;
console.log(numero);
```

Nos daría un error:

```
Console
  "error"
  "TypeError: Assignment to constant variable.
```

Sin embargo, no hay que confundirse con el tema de constantes y cuando usamos *arrays* y *objects*, para entenderlo veamos un ejemplo:

```
const lenguajes = ['JavaScript', 'Python'];
lenguajes.push('Java');
console.log(lenguajes);
```

Como podemos observar tenemos un array de lenguajes, el cual luego le agregamos un nuevo elemento. Es decir, aunque nuestro array sea definido como constante, le podemos agregar y eliminar elementos.

Resultado:

```
Console
  ["JavaScript", "Python", "Java"]
```

Lo mismo para un objeto:

```
const persona = {
  nombre: 'John Doe',
};
persona.edad = 33;
console.log(persona);
```

En este caso le agregamos un atributo al objeto persona, como ves, sin errores:

```
▶ {nombre: "John Doe", edad: 33}
```

¿Cuál usar? ¿Const o Let?

Depende de la situación, personalmente recomiendo usar **`let`** cuando sepamos que el elemento puede sufrir mutación y/o cambios en un futuro. Y **`const`** cuando sepamos que nuestro elemento no va a sufrir ninguna alteración.

https://cybmeta.com/var-let-y-const-en-javascript

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 sobreescribe la variable global.

En resumen, **la declaración con `var` define una variable en el ámbito local actual** (léase 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 llaves", 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 delcara 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, 2, 3), 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, la 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

4. `var` declara una variable de **scope global o local para la función** sin importar el ámbito de bloque. Permite **hoisting**.
5. `let` declara una variable de **scope global, local para la función o de bloque**. Es **reasignable** y no permite hoisting.
6. `const` declara una variable de scope global, local para la función o de bloque. **No es reasignable, 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 sufrir una reasignación. `var` se puede relegar para cuando necesitemos hacer hoisting, vamos, casi nunca.

Otro: https://yeisondaza.com/var-let-y-const-en-javascript