ES6

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

Console

1

2

3

4

5

6

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?

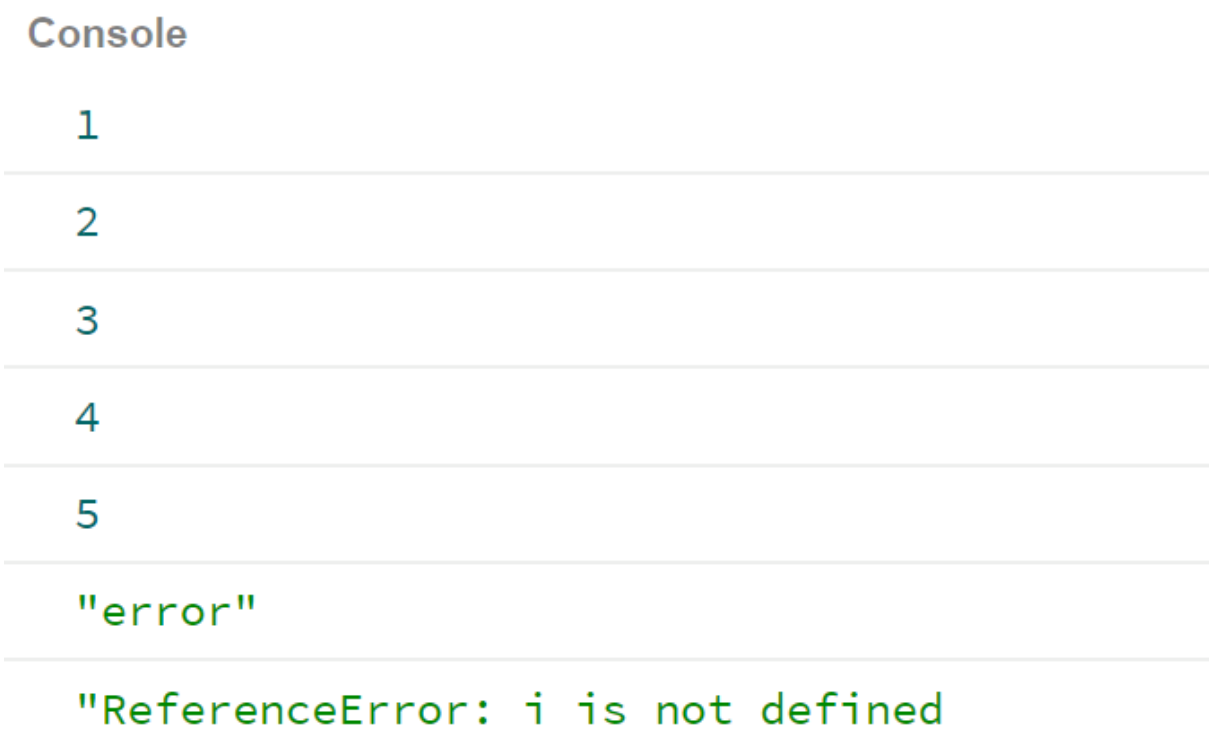
Un bloque es basicamente una estructura de código creada por llaves.

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 mas 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:

Funciones Flecha (Arrow Function)

Comparación de funciones tradicionales con funciones flecha

Observa, paso a paso, la descomposición de una "función tradicional" hasta la "función flecha" más simple:
Nota: Cada paso a lo largo del camino es una "función flecha" válida

```
// Función tradicional
function (a){
  return a + 100;
}

// Desglose de la función flecha
// 1. Elimina la palabra "function" y coloca la flecha entre el argumento y el corchete de apertura.
(a) => {
  return a + 100;
}

// 2. Quita los corchetes del cuerpo y la palabra "return" — el return está implícito.
(a) => a + 100;

// 3. Suprime los paréntesis de los argumentos
a => a + 100;
```

Por ejemplo, si tienes varios argumentos o ningún argumento, deberás volver a introducir paréntesis alrededor de los argumentos:

```
// Función tradicional
function (a, b){
  return a + b + 100;
}

// Función flecha
(a, b) => a + b + 100;

// Función tradicional (sin argumentos)
let a = 4;
let b = 2;
function (){
  return a + b + 100;
}

// Función flecha (sin argumentos)
let a = 4;
let b = 2;
() => a + b + 100;
```

Del mismo modo, si el cuerpo requiere líneas de procesamiento adicionales, deberás volver a introducir los corchetes Más el "return" (las funciones flecha no adivinan mágicamente qué o cuándo quieres "volver"):

```
// Función tradicional
function (a, b){
  let chuck = 42;
  return a + b + chuck;
}

// Función flecha
(a, b) => {
  let chuck = 42;
  return a + b + chuck;
}
```

Y finalmente, en las funciones con nombre tratamos las expresiones de flecha como variables:

```
// Función tradicional
function bob (a){
  return a + 100;
}

// Función flecha
let bob = a => a + 100;
```

Template Literal

Antes de ES6, usaba comillas simples (') o comillas dobles (") para envolver un literal de cadena. Y las cuerdas tienen una funcionalidad muy limitada.

Para permitirle resolver problemas más complejos, los literales de plantilla de ES6 proporcionan la sintaxis que le permite trabajar con cadenas de manera más segura y limpia.

En ES6, crea un literal de plantilla envolviendo su texto en acentos franceses (`) de la siguiente manera:

let simple = `This is a template literal`;

Sustituciones de variables y expresiones

```
let firstName = 'John',
    lastName = 'Doe';

let greeting = `Hi ${firstName}, ${lastName}`;
console.log(greeting); // Hi John, Doe
```

Destructuring

La sintaxis de desestructuración es una expresión de JavaScript que permite desempacar valores de arreglos o propiedades de objetos en distintas variables.

```
let a, b, rest;
[a, b] = [10, 20];

console.log(a);
// expected output: 10

console.log(b);
// expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);
// expected output: Array [30,40,50]
```