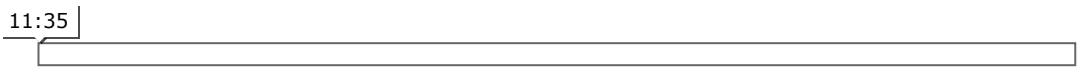
Watch: 👩‍🎨 **Control de la reasignación y mutabilidad con let y const**



Cuando asignamos variables lo hacemos dentro de un scope. Usando `var`, nuestro scope es la función: fuera de ella no podemos acceder a variables que han sido declaradas dentro.

```
function sayHi() {
  var message = "Hi";
  console.log(message);
}

console.log(message); // Uncaught ReferenceError: message is
not defined
```

Si usamos `let` o `const`, el scope pasa a ser un bloque: lo que definimos entre curly braces. Esto quiere decir que además de una función, un `if` o un loop `for` también crean un scope:

```
function sayHi(name) {
  if (name) {
    let message = `Hi ${name}`;
  } else {
    let message = "Hi";
  }
  console.log(message);
}

sayHi() // Uncaught ReferenceError: message is not defined
```

Para que el código anterior funcione, tenemos que declarar la variable fuera del `if`:

```
function sayHi(name) {
  let message;
  if (name) {
    message = `Hi ${name}`;
  } else {
    message = "Hi";
  }
  console.log(message);
}

sayHi() // "Hi"
```

Se suele decir que la diferencia entre `let` y `const` es que `const` es inmutable, pero esto no es del todo correcto. Podemos mutar un objeto asignado a `const`:

```
const user = {
  name: "Núria",
};

user.name = "Javi";
```

Lo que no podemos hacer con `const` es **reasignar** la variable: lo que es inmutable es la asignación, no el valor.

Entonces, ¿cuando usar `let` y cuando usar `const`? Una opción es usar `const` siempre que no vayamos a reasignar esa variable, sea esto casualidad o intencionadamente. Herramientas como ESLint nos permiten forzar esto. Otra corriente aboga por usar `const` solo cuando queremos comunicar que ese valor **no va a cambiar nunca**. Las dos opciones son perfectamente válidas, pero si es conveniente acordar qué convención vamos a seguir para el trabajo en equipo.

Enlaces relacionados

- Artículo de Jamie Builds