JAVASCRIPT VARIABLES

Can be declared using the ***const***, ***var*** or ***let*** keywords.

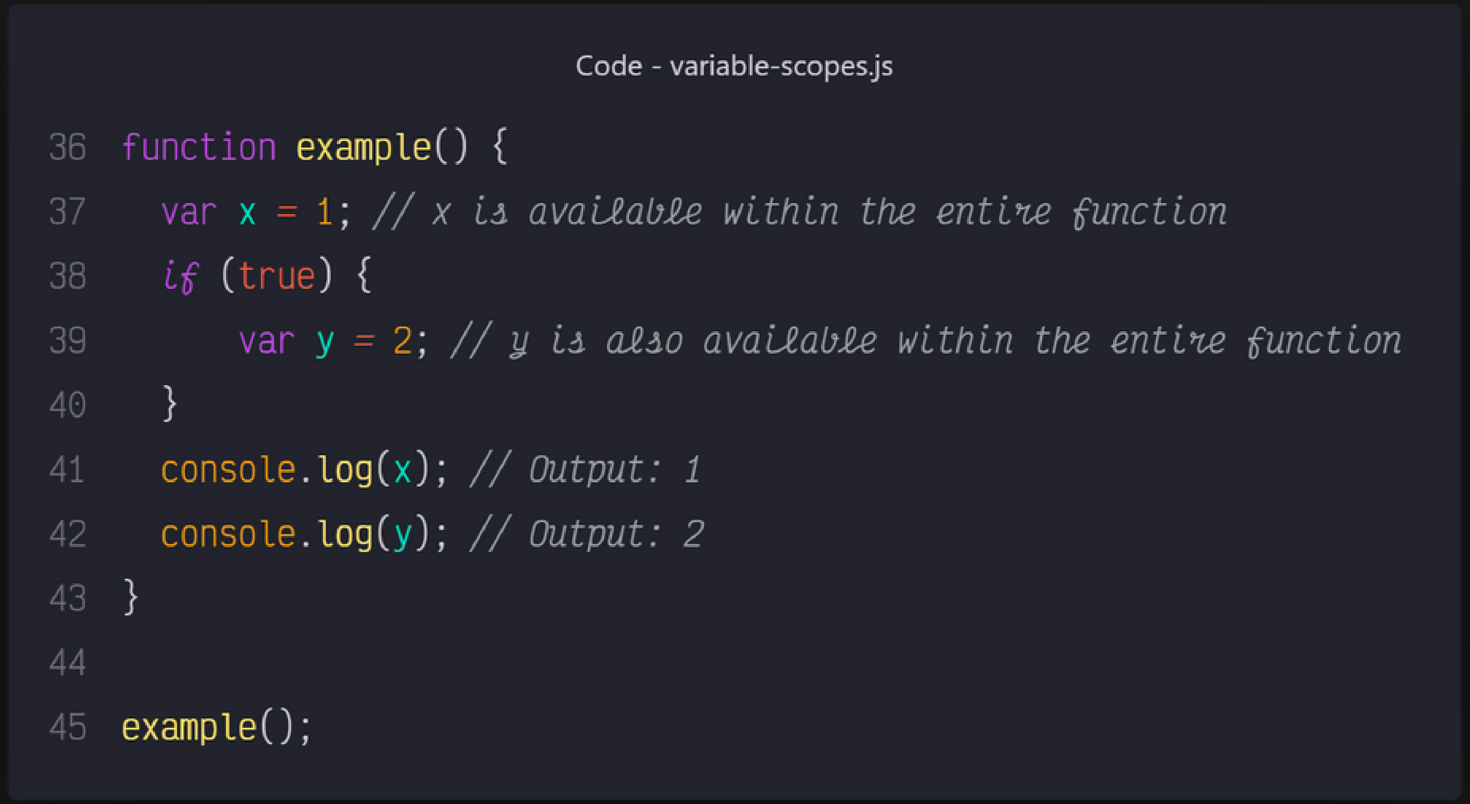
Code - javascript-variables.js

```javascript
var myVariable = "this is a variable declared using 'var'.";

let mySecondVariable = "this is a variable declared using 'let'.";

const myThirdVariable = "this is a variable declared using 'const'.";
```

VAR vs LET

var

var is an older way of declaring variables in JavaScript. When you use *var*, the variable is available throughout the entire function (or global scope if declared outside a function).

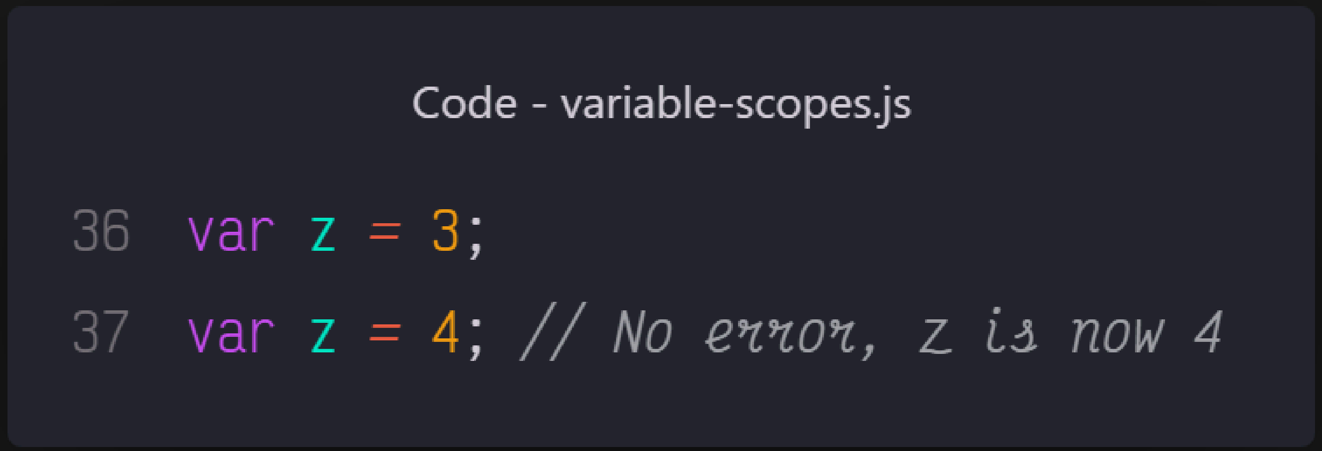


Code - variable-scopes.js

```javascript
function example() {
  var x = 1; // x is available within the entire function
  if (true) {
      var y = 2; // y is also available within the entire function
  }
  console.log(x); // Output: 1
  console.log(y); // Output: 2
}

example();
```

VAR vs LET

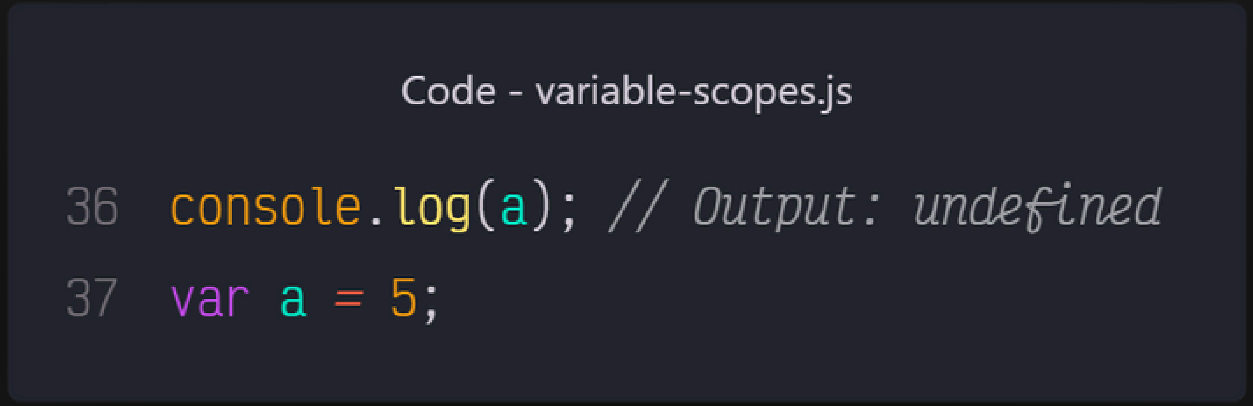
However, *var* has some quirks that can lead to unexpected behavior. For instance, you can declare the same variable twice with *var* without getting an error.



Code - variable-scopes.js

```
var z = 3;
var z = 4; // No error, z is now 4
```

Additionally, variables declared with *var* are hoisted to the top of their scope (function or global). This means that you can use a variable before it's declared.



Code - variable-scopes.js

```
console.log(a); // Output: undefined
var a = 5;
```

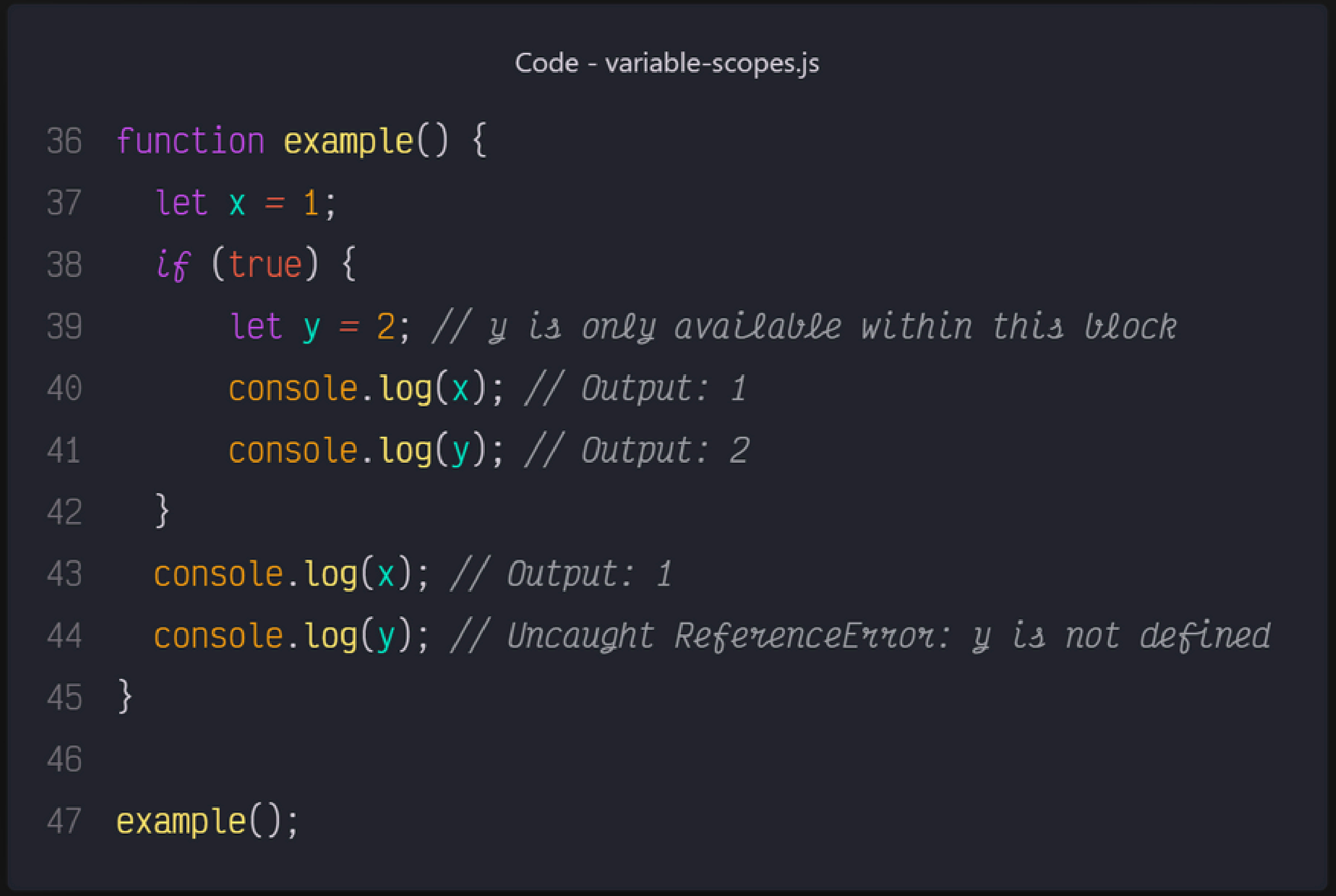
While *hoisting* can be convenient, it can also lead to confusing bugs if you're not aware of how it works.

VAR vs LET

let

Now, let's talk about *let*. *let* was introduced in ES6 (a major update to JavaScript in 2015) to address some of the issues with *var*.

Variables declared with *let* are block-scoped, meaning they are only accessible within the block (denoted by curly braces {}) they are defined in.

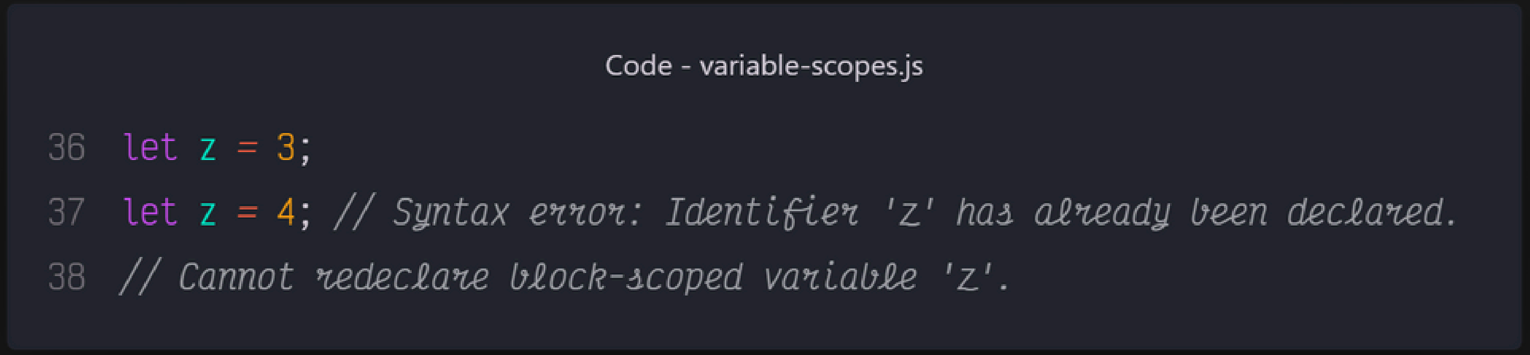


Code - variable-scopes.js

```javascript
function example() {
  let x = 1;
  if (true) {
      let y = 2; // y is only available within this block
      console.log(x); // Output: 1
      console.log(y); // Output: 2
  }
  console.log(x); // Output: 1
  console.log(y); // Uncaught ReferenceError: y is not defined
}

example();
```

VAR vs LET

Unlike *var*, you cannot re-declare a variable with *let* in the same scope

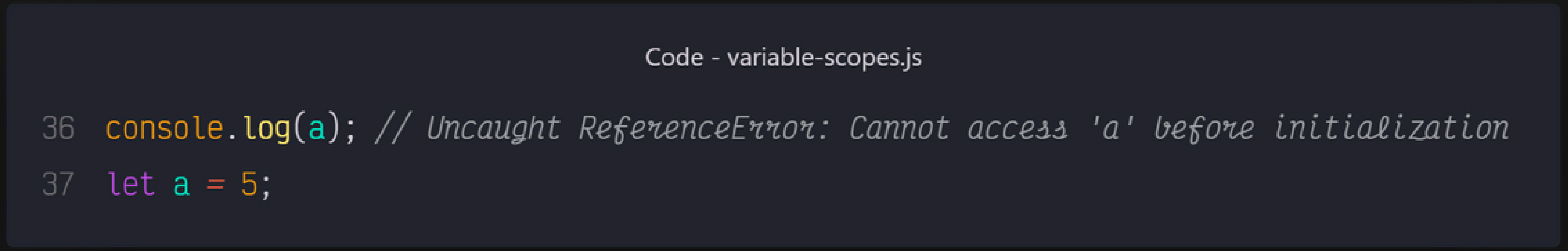


Code - variable-scopes.js

```js
let z = 3;
let z = 4; // Syntax error: Identifier 'z' has already been declared.
// Cannot redeclare block-scoped variable 'z'.
```

Additionally, *let* variables are not hoisted to the top of their scope.

Instead, they are in a "temporal dead zone" until they are declared. This means you cannot access them before they are declared.



Code - variable-scopes.js

```js
console.log(a); // Uncaught ReferenceError: Cannot access 'a' before initialization
let a = 5;
```

VAR vs LET

In general, it's recommended to use let for variable declaration in modern JavaScript, as it provides more predictable behavior and helps prevent common issues that can occur with var.

There are a few scenarios where you might still use var:

- **When you need to support older browsers that don't support *let***: Although most modern browsers support *let*, if you need to support older browsers (like Internet Explorer), you might have to stick with *var*.
- **When you want a variable to be function-scoped**: If you intentionally want a variable to be available throughout the entire function scope (rather than block-scoped), you might use *var*.
- **In certain loop constructs**: In some cases, like for loops, you might use *var* to create a function-scoped variable to work around certain quirks of *let* (more advanced topic).

However, in most modern JavaScript development, ***let*** is the preferred way to declare variables due to its more predictable behavior and better support for block-scoping.

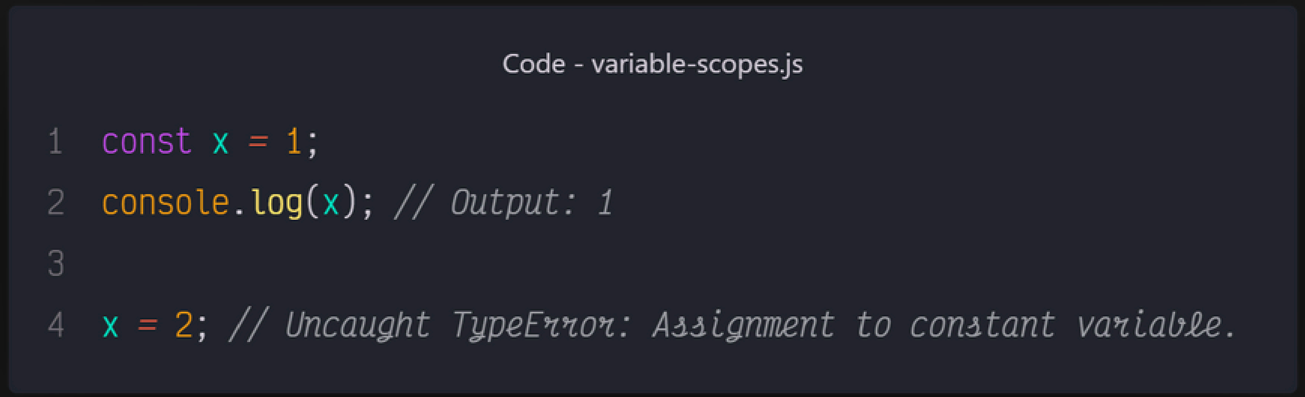
CONST VARIABLES

const

const is the third way to declare variables in modern JavaScript, alongside *var* and *let*.

Like *let*, *const* is block-scoped, meaning it is only accessible within the block (denoted by curly braces {}) it is defined in.

However, the key difference is that *const* variables are immutable, which means their value cannot be reassigned after they are declared.



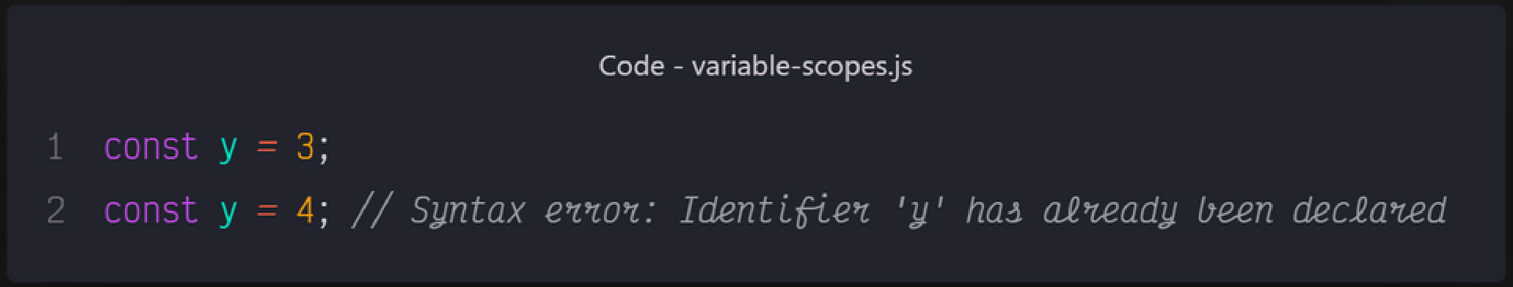
Code - variable-scopes.js

```javascript
const x = 1;
console.log(x); // Output: 1

x = 2; // Uncaught TypeError: Assignment to constant variable.
```

As you can see, we can't reassign the value of x after it has been declared with *const*.

This makes *const* variables useful for declaring constants or values that should not be changed throughout the program.

Like *let*, you cannot re-declare a variable with *const* in the same scope.



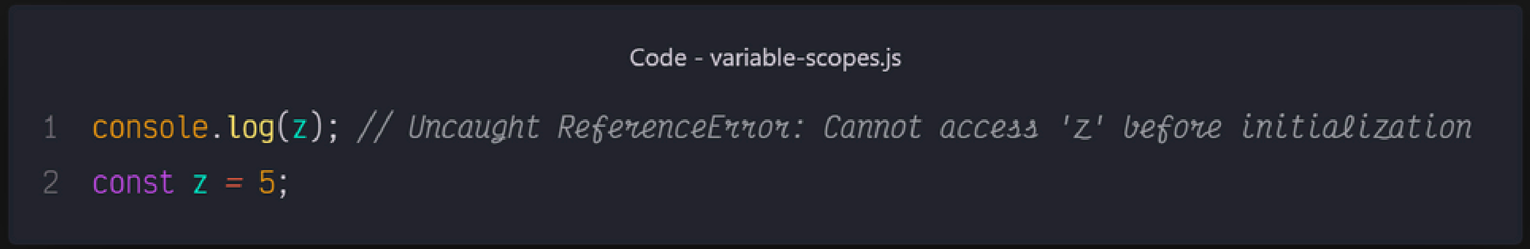
Code - variable-scopes.js

```javascript
const y = 3;
const y = 4; // Syntax error: Identifier 'y' has already been declared
```

CONST VARIABLES

And just like ***let***, ***const*** variables are not hoisted to the top of their scope.

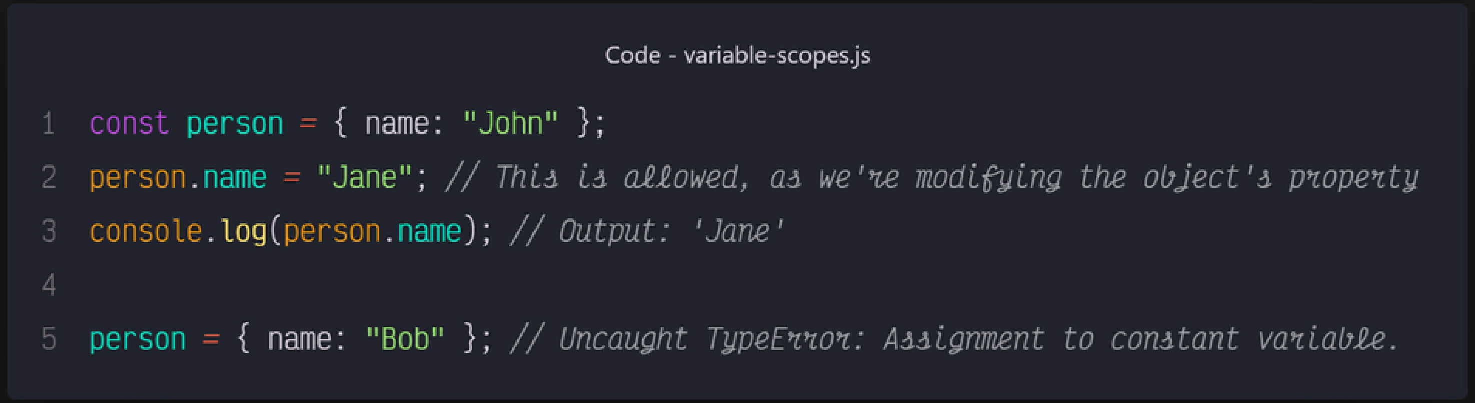
They are also in a "temporal dead zone" until they are declared, meaning you cannot access them before they are declared



Code - variable-scopes.js

```javascript
console.log(z); // Uncaught ReferenceError: Cannot access 'z' before initialization
const z = 5;
```

One important thing to note about ***const*** is that while the variable itself is immutable, if the variable holds a complex data type like an object or an array, the properties or elements of that data type can be modified.



Code - variable-scopes.js

```javascript
const person = { name: "John" };
person.name = "Jane"; // This is allowed, as we're modifying the object's property
console.log(person.name); // Output: 'Jane'

person = { name: "Bob" }; // Uncaught TypeError: Assignment to constant variable.
```

In this example, we can modify the name property of the person object, but we cannot reassign the person variable to a new object.

MORE TIPS

Best practices: It's generally recommended to use *const* as your default way of declaring variables in JavaScript. This is because *const* variables cannot be reassigned, which helps prevent accidental modifications and makes your code more predictable. Only use *let* when you know that you'll need to reassign the value of the variable later on.

Global scope: When you declare a variable (*var*, *let*, or *const*) outside of any function or block, it becomes a global variable. Global variables are properties of the global object (***window*** in web browsers, ***global*** in Node.js). It's generally best to avoid using too many global variables, as they can lead to naming conflicts and make your code harder to maintain.

Variable naming conventions: When naming variables in JavaScript, it's a good practice to use descriptive names that clearly convey what the variable represents. Additionally, variable names are case-sensitive, so **myVar** and **myvar** are treated as two different variables. It's a common convention to use camelCase for variable names (e.g., **myVariable**).