Title: What are the variable scopes in JavaScript?

Answer:

Scope in JavaScript refers to the current context of code, which determines the accessibility of variables to JavaScript. The two types of scope are *local* and *global*:

- Global variables are those declared outside of a block
- Local variables are those declared inside of a block

In the example below, we will create a global variable.

```
// Initialize a global variable
var creature = "wolf";
```

We learned that variables can be reassigned. Using local scope, we can actually create new variables with the same name as a variable in an outer scope without changing or reassigning the original value.

In the example below, we will create a global `species` variable. Within the function is a local variable with the same name. By sending them to the console, we can see how the variable's value is different depending on the scope, and the original value is not changed.

```
// Initialize a global variable
var species = "human";

function transform() {
  // Initialize a local, function-scoped variable
  var species = "werewolf";
  console.log(species);
}

// Log the global and local variable
console.log(species);
transform();
console.log(species);
```

```
Output
human werewolf human
```

In this example, the local variable is *function-scoped*. Variables declared with the `var` keyword are always function-scoped, meaning they recognize functions as having a separate scope. This locally-scoped variable is therefore not accessible from the global scope.

The new keywords `let` and `const`, however, are *block-scoped*. This means that a new, local scope is created from any kind of block, including function blocks, `if` statements, and `for` and `while` loops.

To illustrate the difference between function- and block-scoped variables, we will assign a new variable in an `if` block using `let`.

```
var fullMoon = true;

// Initialize a global variable
let species = "human";

if (fullMoon) {
  // Initialize a block-scoped variable
  let species = "werewolf";
  console.log(`It is a full moon. Lupin is currently a ${species}.`);
}

console.log(`It is not a full moon. Lupin is currently a ${species}.`);
```

```
Output
It is a full moon. Lupin is currently a werewolf. It is not a full moon. Lup
in is currently a human.
```

In this example, the `species` variable has one value globally (`human`), and another value locally (`werewolf`). If we were to use `var` , however, there would be a different result.

```
// Use var to initialize a variable
var species = "human";

if (fullMoon) {
  // Attempt to create a new variable in a block
  var species = "werewolf";
  console.log(`It is a full moon. Lupin is currently a ${species}.`);
}

console.log(`It is not a full moon. Lupin is currently a ${species}.`);
```

```
Output
It is a full moon. Lupin is currently a werewolf. It is not a full moon. Lup
in is currently a werewolf.
```

In the result of this example, both the global variable and the block-scoped variable end up with the same value, `werewolf` . This is because instead of creating a new local variable with `var` , you are reassigning the same variable in the same scope. `var` does not recognize `if` to be part of a different, new scope. It is generally recommended that you declare variables that are block-scoped, as they produce code that is less likely to unintentionally override variable values.

Tags: variables, javascript