COURSEWARE

Hoisting

Contents

Overview

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

Tutorial

What is hoisting?

Hoisting circumvents the usual top-to-bottom execution of JavaScript:

- `var` declarations get hoisted to the top of their closest enclosing function scope, their assignment does not.
- `const` and `let` declarations are blessed with a new concept called **Temporal Dead Zones**.

Temporal Dead Zone

Unlike variables declared with `var`, which will start with the value `undefined`, `let` variables are *not* initialised until their definition is evaluated. Accessing the variable before the initialisation results in a `ReferenceError`.

The variable is in the **Temporal Dead Zone (TDZ)** from the start of the block until the initialisation is processed:

```
function doSomething(){
    console.log(bar); // undefined
    console.log(foo); // ReferenceError
    var bar = 1;
    let foo = 2;
}
```

When used inside a block, `let` limits the variable's scope to that block. Note the difference between `var`, whose scope is inside the function where it is declared:

```
var a = 1;
var b = 2;

if (a === 1){
    var a = 11; // the scope is global
    let b = 22; // the scope is inside the if-block

    console.log(a); // 11
    console.log(b); // 22
}

console.log(a); // 11
console.log(b); // 2
```

However, this combination of `var` and `let` declaration below is a `SyntaxError` due to `var` being hoisted to the top of the block. This results in implicit re-declaration of the variable:

```
let x = 1;

{
    var x = 2; // SyntaxError for re-declaration.
}
```

TDZ and `typeof`

Unlike with simply undeclared variables and variables that hold a value of `undefined`, using the `typeof` operator to check for the type of a variable in that variable's temporal dead zone will throw a `ReferenceError`:

```
// Prints out 'undefined'
console.log(typeof undeclaredVariable);

// Results in a 'ReferenceError'
console.log(typeof i);
let i = 10;
```

TDZ combined with lexical scoping

Due to lexical scoping, the identifier `foo` inside the expression `(foo + 60)` evaluates to the `if` block's `foo`, and not the overlying variable `foo` with the value of `33`.

In the same line, the `if` block's `foo` has already been created in the lexical environment, but has not reached (and terminated) its initialisation (which is part of the statement itself).

The `if` block's `foo` is in the temporal dead zone:

```
function test(){
    var foo = 33;
    if(foo){
        let foo = (foo + 60); // ReferenceError
    }
}
test();
```

Examples

Let's have a look at some more examples of hoisting:

No global variable:

Assuming there is no global variable named `notDefined`, the below example will not work and throws a `ReferenceError`:

```
function example(){
    console.log(notDefined);
}
```

Declaration (VAR) after reference:

Creating a variable declaration after you reference the variable will work due to variable hoisting:

```
function example2(){
    console.log(declaredButNotAssigned); // Undefined
    var declaredButNotAssigned = true;
}
```

Note: the assignment value of 'true' is not hoisted.

Declaration (LET) before reference:

The interpreter is hoisting the variable declaration to the top of the scope, which means our example could be rewritten as:

```
function example3(){
    let declaredButNotAssigned;
    console.log(declaredButNotAssigned); // Undefined
    declaredButNotAssigned = true;
}
```

Using CONST and LET

```
function example4(){
    console.log(declaredButNotAssigned); // Throws ReferenceError
    console.log(typeof declaredButNotAssigned); // Throws ReferenceError
    const declaredButNotAssigned = true;
}
```

Function hoisting

Anonymous Functions

Anonymous Functions hoist their variable name but not the function assignment. Let's have a look at what that means:

```
function anonFunction(){
    console.log(anon); // undefined

    anon(); // TypeError anon is not a function

    var anon = function(){
        console.log('anonymous function expression');
    };
}
```

Named Functions

Named function expressions hoist the variable name, not the function name or the function body:

```
function namedFunction(){
    console.log(named); // undefined

    named(); // TypeError named is not a function

    superPower(); // ReferenceError superPower is not defined

    var named = function superPower(){
        console.log("Zoomin' through the sky");
    };
}
```

The same is true when the function name is the same as the variable name:

```
function example(){
    console.log(named); // undefined

    named(); // TypeError named is not a function

    var named = function named(){
        console.log('named');
    }
}
```

Function Declarations

Function declarations hoist their name and the body function:

```
function example(){
    superPower()// => Zoomin' thru the sky.

    function superPower(){
        console.log("Zoomin' thru the sky.");
    }
}
```

In summary, always make sure you are using the appropriate declaration types for your variables.

Exercises

1. What value will be alerted if the following is executed?

```
var foo = 1;
function bar(){
    if(!foo){
        var foo = 10;
    }
    alert(foo);
}
bar();
```

▶ Solution

2. What value will be alerted if the following is execute?

```
var a = 1;
function b(){
    a = 10;
    return;
    function a(){}
}
b();
alert(a);
```

▶ Solution

3. Try it yourself, have a go at experimenting with hoisting and see what you come up with.