Zasięg zmiennych i hoisting

Hoisting (windowanie)

Jako koncept, hoisting sugeruje, że zmienne i deklaracje funkcji są przenoszone, tak aby znalazły się na początku Twojego kodu. Co się jednak naprawdę dzieje to to, że deklaracje zmiennych i funkcji są zapisywane w pamięci podczas fazy kompilacji, ale pozostają dokładnie tam, gdzie zostały wpisane w kodzie. Podstawowe znaczenie windowania polega na tym, że umożliwia ono korzystanie z funkcji przed zadeklarowaniem ich w kodzie.

Oto rzeczy, których możemy nauczyć się z definicji windowania.

- Co zostaje przeniesione, to deklaracje zmiennych i funkcji. Przypisanie zmiennych i inicjalizacja nigdy nie zostaje przeniesiona.
- Deklaracje nie są przenoszone na początek kodu; zamiast tego są zapisywane w pamięci.

W JavaScript wszystkie zmienne zdefiniowane przy użyciu var na początku mają wartość undefined. Jest to spowodowane windowaniem, które zapisuje w pamięci deklaracje zmiennych i inicjuje je wartością undefined. Poniższy przykład dobrze to ilustruje:

```
1  console.log(x); //prints undefined
2  console.log(y); //throws ReferenceError: y is not defined
3  var x = 1;
```

Jednak zmienne zdefiniowane za pomocą słów kluczowych let i const, gdy są windowane, nie są inicjalizowane z undefined. Są one raczej w stanie zwanym Temporal Dead Zone, dopóki ich definicje nie zostaną zewaluowane.

```
1  console.log(x); //throws TDZ ReferenceError: x is not defined
2  let x = 1;
```

Następny fragment kodu pokazuje hoisting zmiennych let i const.

```
1  var x = 10;
2  {
3      console.log(x); //throws TDZ ReferenceError: x is not defined
4      let x = 5;
5  }
```

Zmienna x zdefiniowana w bloku za pomocą słowa kluczowego let jest windowana i ma pierwszeństwo przed zmienną x zdefiniowaną za pomocą var. Jednak nadal znajduje się ona w Temporal Dead Zone, gdy występuje odwołanie do niej z console.log(x), a zatem wyrzuca reference error.

Zasięg

Zmienne zdefiniowane za pomocą słowa kluczowego var mają zasięg, który jest ich bieżącym kontekstem wykonania. Nie mają one zasięgu blokowego, więc można uzyskać do nich dostęp spoza bloku, w którym zostały zdefiniowane. Może się tak stać pod warunkiem, że zmienne te nadal znajdują się w zasięgu kontekstu wykonania. Zmienne let i const mają jednak zasięg blokowy i nie można uzyskać do nich dostępu spoza bloku. Widać to poniżej:

```
(function () {
    {
        var x = 2;
        let y = 3;
        const z = 4;
    }
    if (true) {
        console.log(x) //prints 2
        console.log(y); //throws ReferenceError - out of its scope
        console.log(z); //throws ReferenceError - out of its scope
    }
    console.log(x) //prints 2
    console.log(y); //throws ReferenceError - out of its scope
    console.log(z); //throws ReferenceError - out of its scope
})();
console.log(x); //throws ReferenceError - out of its scope
```

Ponadto, gdy deklarujesz zmienną globalną ze słowem kluczowym var, to zostaje ona dołączona do kontekstu globalnego (window w przeglądarce i global w Node.js). Nie dzieje się tak w przypadku zmiennych globalnych zadeklarowanych za pomocą let i const.

Warto zapamiętać

- Gdy po prostu przypiszesz wartość do zmiennej, bez deklaracji za pomocą słów kluczowych, zmienna ta zostaje utworzona i dołączona do globalnego kontekstu wykonania (window w przeglądarce i global w Node.js). Robienie tego nie jest jednak zalecane, ponieważ znacznie utrudnia to debugowanie.

```
x = "this gets attached to the global this";
console.log(this.x); //prints the value of x

function testFn() {
    y = "this also gets attached to the global this";
    console.log(this.y) //prints the value of y, this here refers to either window or global
}

testFn();
```

- Zmienne zadeklarowane za pomocą słowa kluczowego var mogą być ponownie zadeklarowane w dowolnym momencie kodu, nawet jeśli są w tym samym kontekście wykonania. Nie dotyczy to zmiennych zdefiniowanych za pomocą słów kluczowych let i const, ponieważ można je zadeklarować tylko raz w ramach ich zasięgu leksykalnego.

```
var x = 1;
var x = 2;
console.log(x); //prints 2
let y = 1; //throws SyntaxError:  Identifier y has already been declared
let y = 2; //throws SyntaxError:  Identifier y has already been declared
const z = 1; //throws SyntaxError: Identifier z has already been declared
const z = 2; //throws SyntaxError: Identifier z has already been declared
```

Może się pojawić wtedy problem, zwłaszcza, jeśli używasz let lub const do deklarowania zmiennej wewnątrz switch.

```
var x = 1;
switch (x) {
    case 0:
        let foo = 20; //throws SyntaxError: Identifier foo has already been declared
        break;
    case 1:
        let foo = 30; //throws SyntaxError: Identifier foo has already been declared
        break;
}
```

Oczywiście można tego uniknąć, używając nawiasów klamrowych wokół case'ów, aby zdefiniować różne bloki, ale prawdopodobnie należałoby to zrefaktoryzować.

```
var x = 1;
switch (x) {
    case 0:
        {
            let foo = 20;
            break;
        }
    case 1:
        {
            let foo = 30;
            break;
        }
}
```

- Kolejną kwestią, o której warto pamiętać to fakt, że mimo iż nie można przypisać ponownie wartości do stałej, to nadal jest ona mutowalna. Dobrze można to zilustrować faktem, że jeśli wartością jest obiekt, właściwości obiektu będzie można modyfikować.

```
const obj = {
    firstName: "Favour"
};
obj.lastName = "Harrison";
console.log(obj); //prints an object having both firstName and lastName properties
```