More JavaScript Concepts

Const

ES6 introduced two new keywords: **let** and **const.**

Const is used to declare **read-only variables**, ie, they ***can't be reassigned*** and ***redeclared.***
For eg;

```
const a = 1;
console.log(a);
```

This will print the value of the **constant** variable as usual. But what if we tried to reassign a value to it?

```
const a = 1;
console.log(a);
a = 10;
```

This will produce the following error:

```
a = 10;
 ^
TypeError: Assignment to constant variable.
```

So, you can read the value **as many times** as you want, but you **can't** change its value. The **const declaration** creates a **read-only reference** to a value. It means that the variable identifier **can't be reassigned**.
Variables created using the **var** keyword can be **redeclared** and **reassigned**.
For eg;

```
var animal = 'cat';
animal = 'dog';
//This is allowed
```

Unlike **var, const** variables ***must be initialized when they are declared,*** else it will **produce an error**.
For example, something like this:

```
const name;
name = "Robert";
```

```
console.log(name);
```
will result in this error:
```
const name;
     ^^^^
SyntaxError: Missing initializer in const declaration
```

So, the correct way of declaring **const** variables is to initialize them with a value. You **can't defer** this step.

- **Hoisting**

  As you know that the variables defined using the **var keyword** are ***hoisted to the top*** and can be ***initialized later*** on.
  ```
  console.log(name); //prints -> undefined
  var name;
  name = 'Sam';
  console.log(name); //prints -> Sam
  //This is allowed
  ```

  But this concept works differently with variables declared using the **const keyword.** The **const** variables ***are hoisted to the top***, but **they're not initialized**, unlike **var**.

  Since **const** variables ***can't be used unless they're initialized***, this will result in an error.

  For example, this
  ```
  console.log(a);
  const a = 30;
  ```

  will produce error:
  ```
  console.log(a);
              ^
  ReferenceError: Cannot access 'a' before initialization
  ```

Let

We'll need to understand Javascript scopes before diving into the **let keyword.**

- **Global Scope:**

  The variables created outside any function have **Global scope.** These variables can be accessed anywhere in the program.

  For eg.,

```
var firstName = 'John'

function printName() {
   //firstName can be used here
   console.log(firstName);
}
```

- **Function Scope:**

  The variables created inside a function have **function scope.** These variables are also known as the ***local variables*** of the function. The variables having function scope can ***only be accessed inside the function in which they were declared***.

  For eg.,

```
function printName() {
   var name = "John";
   //name can be used only inside this function
}
// name can't be used here
console.log(name);
```

  The ***console.log()*** statement will cause this error:

```
console.log(name);
            ^
ReferenceError: name is not defined
```

  Before ES6, Javascript only had Global Scope and Function Scope. Variables cannot have block scope, ie., the variables created inside a block **{}** can be accessed from outside this block.

```
{
   var name = 'John'
}
//name can be used here (outside the block)
```

```
console.log(name + 'Doe');
```

ES6 introduced the concept of **Block scope.** The variables created using the **let keyword have block scope.** These variables can't be accessed from outside the block.

For eg.,

```
{
   let name = 'John';
}

//name cannot be used here
console.log(name);
```

The above code show this error:

```
console.log(name);
            ^

ReferenceError: name is not defined
```

because the ***name variable*** has block scope and it can't be accessed from outside the block in which it was declared.
However, the variables created in the **global scope** using the **let** and **var** keywords will behave similarly, as they both will have a global scope.

```
var a = 10; //global scope
let b = 20; //global scope
```

The **let variables** behave **similarly** to **the const variables** in the case of hoisting. Using let variables before their declaration will cause an error.
For eg.,

```
name = 'John Doe';
let name;
```

Will give this error:

```
name = 'John Doe';
    ^
```

```
ReferenceError: Cannot access 'name' before initialization
```

Let vs Var

Let's see what are the differences between **let** and **var** when used with **for loop.**

Consider the program below:

```
for (var i = 1; i <= 5; i++) {
    setTimeout(function() {
        console.log(i);
    }, 500);
}
```

What do you think will be the output?
The code will print:

```
6
6
6
6
6
```

It's because the **console.log(i)** statement will be executed after 500ms before which the value of the variable **i** will become **5**. As the variable is declared using **var**, it has a global scope and thus, it will have the value **5** after the loop ends.

But what if we use **let** instead of **var**?

```
for (let i = 1; i <= 5; i++) {
    setTimeout(function() {
        console.log(i);
    }, 500);
}
```

The code will print:

```
1
2
3
4
5
```

This is happening because variables created using **let** have block scope and therefore, there is a **different instance** of **i** for ***each iteration of the loop.*** You can say that every interval of the loop has a separate value of **i.**

Loops in JavaScript

These are the different loops available in JavaScript:

- **for -** It's used to iterate over a block of code a specified number of times.
- **for...in -** It's used to iterate through properties of an object
- **for...of -** It's used to iterate over iterable objects like String, Array, NodeList, etc.
- **while -** It's used to iterate over a block of code until a specified condition is true.
- **do...while -** It's also used to iterate over a block of code until a specified condition is true.

- **for loop**

  Syntax:

```
for(initialization, condition, updateCounter){
    //code to be executed
}
```

  1. ***Initialization*** *is* ***executed once*** *before the execution of the code inside the loop*
  2. ***Condition*** *is checked before* ***every iteration*** *of the loop.*
  3. ***UpdateCounter*** *is used to update (increase or decrease) the value of the initialized variables and it's* ***executed every time*** *after every iteration of the loop.*

  Once the condition of the loop is satisfied, the loop stops execution of the code.

  For eg.,

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}
```

- **for...in loop**

  The for...in loop can be used to iterate through the properties of an object.

Syntax:

```
for (const key in object) {
    //code to be executed
}
```

For example;

```
var person = {
    firstName: "Alex",
    lastName: "Mercer",
    age: 29
}
for (i in person) {
    console.log(i);
}
```

Output:

```
firstName
lastName
age
```

You can also use the **for...in** loop with arrays only if the **index order is not important.**

For Eg.,

```
var arr = [1, 2, 3, 4, 5];
for (i in arr) {
    console.log(i);
}
```

***Note**: The array index order varies with the implementation, so in the **for...in** loop, the array values may not be accessed the way you expected! Therefore, it's advised not to use the **for...in loop** to iterate over arrays.*

- **for...of loop**

This loop can be used to iterate over **iterable objects** like **Arrays, Strings** and **more**.

Syntax:

```
for (variable of iterable) {
```

```
    statement
}
```

Here, **variable** is the *variable* which will be used to iterate over the **iterable object.** During **every iteration** of the loop, ***different values/properties will be assigned to this variable.*** **Iterable** is the object on which iteration will be performed.

For eg.,

```
var arr = [1, 2, 3, 4, 5];
for (i of arr) {
    console.log(i);
}
```

Output:

```
1
2
3
4
5
```

Similarly, you can also iterate over a String:

```
var iterable = "hello";
for (i of iterable) {
    console.log(i);
}
```

- **while loop**

The **while loop** executes a block of code as long as a specific condition is true.

Syntax:

```
while (condition) {
    //code to be executed
}
```

For eg.,

```
var i = 0;
```

```
while (i < 10) {
    console.log(i);
    i++;
}
```

The above code will **print numbers from 0 to 9** and it stops execution when the value of **i** becomes **equal to 10.**

Note: *Don't forget to update the value of the iterator variable, else the loop will continue to execute the block of code an infinite number of times which will eventually crash your browser.*

- **do...while loop**

The **do...while loop** is a variation of the **while loop.** This loop continues to execute a specified statement ***until the specified condition evaluates to false.*** Here the condition is checked after execution of the code block, which results in the **execution of the code at least once.**

Syntax:

```
do {
    //code to be executed
} while (condition);
```

For eg.,

```
var i = 0;
do {
    i++;
    console.log(i);
} while (i < 5);
```

Output:

```
1
2
3
4
5
```

As the execution of the code takes place before the condition is checked, **do...while** loop executes the code at least once even if the condition is false:

```
var i = 5;
do {
    console.log(i);
} while (i > 5);
```

The above code prints **5** once after which the condition is checked and the loop terminates