

More JavaScript Features

Scope in JavaScript determines the accessibility of variables. The two types of scope are *local* and *global*:

Global and Local Scoping In JavaScript

As we already learned, variables hold the data or information that we can access or reassign. The two main type of scooping in JavaScript is local and global variables.

Local variables: When you use JavaScript, local variables are variables that are defined within functions. They have local scope, which means that they can only be used within the functions that define them.

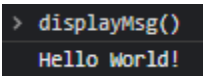
Global Variables: global variables are variables that are defined outside of functions. These variables have global scope, so they can be used by any function without passing them to the function as parameters.

Function scope: variables declare inside a function is local to the function and not affect the variables outside of the function. If there is a variable declared outside the function and we want to use the same variable inside a function, it will refer the outside variable inside the function.

Example 1) function scope

```
let msg = 'This is a outside message'

function displayMsg(){
  let msg = 'Hello World!'
  console.log(msg)
}
```



```
> displayMsg()
Hello World!
```

If we remove the line `let msg = 'Hello World!'` in the function, the function scope the msg value from the global variable

```
let msg = 'This is a outside message'

function displayMsg(){
  console.log(msg)
}
```

```
> displayMsg()  
This is a outside message
```

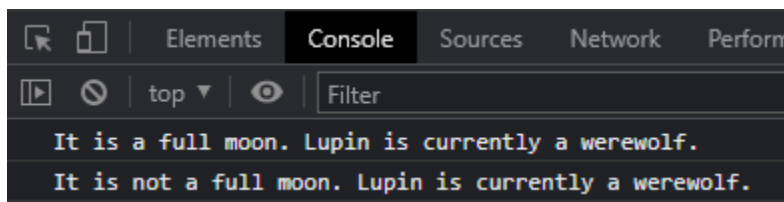
Now, if we update the variable inside the function as:

```
let msg = 'This is a outside message'  
  
function displayMsg(){  
  msg = 'Hello World!'  
  console.log(msg)  
}
```

```
> msg  
◀ 'This is a outside message'  
> displayMsg()  
Hello World!  
◀ undefined  
> msg  
◀ 'Hello World!'
```

Example 2) block scope. global and local variables: using var instead of const or let

```
var fullMoon = true;  
  // Initialize a global variable  
var species = "human";  
  
if (fullMoon) {  
  // Initialize a local variable  
  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}.`);
```



The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays two log messages: 'It is a full moon. Lupin is currently a werewolf.' and 'It is not a full moon. Lupin is currently a werewolf.' The first message is on the top line, and the second is on the bottom line. The console interface includes a filter input and a 'top' button.

Avoid using a global scope as much as possible. To do so, we can convert a global variable into a local variable by placing the function inside a function. This anonymous function runs as soon as the script loads, and it will also keep variables out of the global scope. One way to avoid global scooping, beside changing the variable's name, is to change var to *const* and *let*

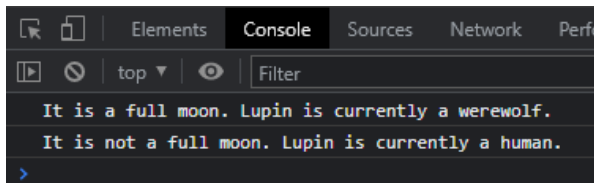
```

const 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}.`);

```



*Use of **const** and **let***

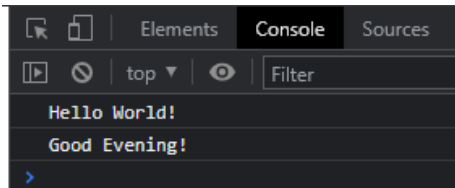
const, constant, variables' are variables whose values cannot be changed later.

Example 3) without constant

```

var msg = "Hello World!";
console.log(msg);
msg = "Good Evening!";
console.log(msg);

```

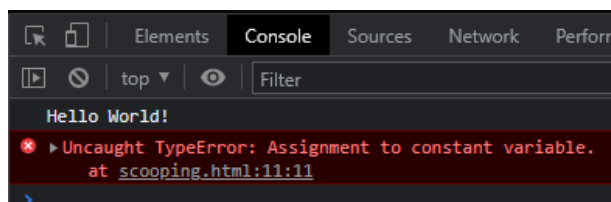


Changing the previous code with **const**

```

const msg = "Hello World!";
console.log(msg);
msg = "Good Evening!";
console.log(msg);

```



The **let** statement declares a block-scoped local variable, optionally initializing it to a value.

Example 4) without **let**

```
var i = 1;
if (i === 1) {
  var i = 2;
  console.log(`x inside if statement ${i}`);
}
console.log(`x outside if statement ${i}`);
```

```
x inside if statement 2
x outside if statement 2
```

Changing the previous code with **let**

```
let i = 1;
if (i === 1) {
  let i = 2;
  console.log(`x inside if statement ${i}`);    // expected output: 2
}
console.log(`x outside if statement ${i}`);    // expected output: 1
```

```
x inside if statement 2
x outside if statement 1
```

Example 5) without **let**

```
<script type="text/javascript">
  for (var i = 0; i < 10; i++) {
    console.log(i);
  }
  console.log(`The last value of i is ${i}`);
```

```
0
1
2
3
4
5
6
7
8
9
The last value of i is 10
>
```

Save function in a variable

```
const sum = function(num1,num2){  
    return num1+num2;  
}
```

To call this function, we just need to call the variable name and enters the arguments values.

```
> sum  
↵ f (num1,num2){  
    return num1+num2;  
}  
  
> sum(3,5)  
↵ 8
```

Example 6) define a function that returns the square of a number. Define the function as a function expression, stored in a variable called **square**.

```
let square = function(num){  
    return Math.pow(num,2)  
}
```

Higher order functions

Functions that operates on or with other functions. It happens when it pass a function as a variable, argument, in another function, and return the inner function result.

Example 7) define a function that will run twice the function **rolldie()**

```
function callTwice(rep){  
    rep();  
    rep();  
}  
  
function rolldie(){  
    const roll = Math.floor(Math.random()*6+1)  
    console.log(roll)  
}  
  
callTwice(rolldie)
```

Returning functions

Example 8) factory functions

```
function makeBetweenFunc(min,max){  
  return function (num){  
    return num>=min && num<=max;  
  }  
}
```

```
> makeBetweenFunc()  
< f (num){  
    return num>=min && num<=max;  
  }  
  
> const child = makeBetweenFunc(1,18)  
< undefined  
  
> child  
< f (num){  
    return num>=min && num<=max;  
  }  
  
> child(6)  
< true  
  
> child(60)  
< false  
  
>
```

Methods

A method is a function that is placed as a property in an object. Every method is a function but that every function is a method.

Example 9)

```
const myMath = {  
  square:function (num){return num*num},  
  double:function(num){return num+num}  
}
```

```
> myMath  
< {PI: 3.14159, square: f, double: f}  
  
> myMath.double(8)  
< 16
```

Newest version of JS do not require to write the function keyword in a method:

```
const myMath = {  
  square (num){return num*num},  
  double (num){return num+num}  
}
```

Example 10) define an object called square, which will hold methods that have to do with geometry of squares. It should contain two methods: area and perimeter:

- area should accept the length of a side and then return the side squared.
- Perimeter should accept the length of a side and return that side multiplied by 4

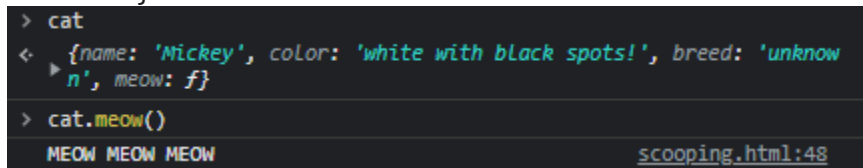
```
const square = {  
  area (side){return side*side},  
  perimeter(side){return side*4}  
}
```

this in method

use the keyword *this* to access other properties on the same object

example)

```
const cat={  
  name: 'Mickey',  
  color: 'white with black spots!',  
  breed: 'unknown',  
  meow(){ console.log('MEOW MEOW MEOW')  
}
```



```
> cat  
< {name: 'Mickey', color: 'white with black spots!', breed: 'unknown', meow: f}  
> cat.meow()  
MEOW MEOW MEOW  
scooping.html:48
```

```
const cat={  
  name: 'Mickey',  
  color: 'white with black spots!',  
  breed: 'unknown',  
  meow(){ console.log(this.name)  
}  
}
```

```

> cat
< {name: 'Mickey', color: 'white with black spots!', breed: 'unknown', meow: f}
> cat.meow()
Mickey
scooping.html:48

```

Example) this

Define an object called hen. It should have three properties:

- Name should be set to 'Helen'
- eggCount should be set to 0
- layAnEgg should be a method which increments the value of eggCount by 1 and return the string 'EGG'

```

const hen = {
  name: 'Helen',
  eggCount: 0,
  layAnEgg(){this.eggCount++; return 'EGG'}
}

```

```

> hen.layAnEgg()
< 'EGG'
> hen.eggCount
< 1

```

Try and catch

```

try{
  hello.toUpperCase();
}catch{
  console.log('ERROR!!!')
}
console.log('AFTER!')
function yell(msg){
  try{
    console.log(msg.toUpperCase().repeat(3));
  }catch(e){
    console.log(e)
    console.log('Please pass a string next time!')
  }
}

```

```

> yell('HELLO')
HELLOHELLOHELLO
< undefined
> yell(123)
TypeError: msg.toUpperCase is not a function
    at yell (scooping.html:67:25)
    at <anonymous>:1:1
Please pass a string next time!

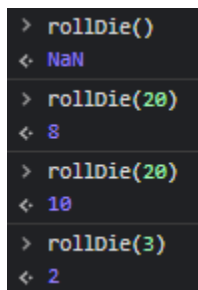
```


Default parameters

Default values in a parameters is set when the argument is not passed as required. In case that the passed argument is not properly passed, the argument will set to the default value in the parameter.

Example)

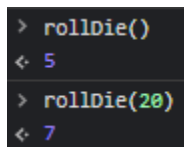
```
function rollDie(numSides){  
    return Math.floor(Math.random()*numSides)+1  
}
```



```
> rollDie()  
↵ NaN  
> rollDie(20)  
↵ 8  
> rollDie(20)  
↵ 10  
> rollDie(3)  
↵ 2
```

Changing the parameter in the rollDie with a default value of 6

```
function rollDie(numSides=6){  
    return Math.floor(Math.random()*numSides)+1  
}
```



```
> rollDie()  
↵ 5  
> rollDie(20)  
↵ 7
```

Since parameters order matters, it is recommended to have a default parameters as last in the order of parameters.

Example)

```
function greeting(msg,name){  
    console.log(`${msg}`, `${name}!`)  
}
```

```
> greeting('Good morning','Peter')
Good morning, Peter!
```

```
function greeting(msg='Hey there',name){
  console.log(`${msg}, ${name}!`)
}
```

```
> greeting('Peter')
Peter, undefined!
```

```
function greeting(name,msg='Hey there'){
  console.log(`${msg}, ${name}!`)
}
```

```
> greeting('Peter')
Hey there, Peter!
```

```
function greeting(name,msg='Hey there', punc='!!'){
  console.log(`${msg}, ${name} ${punc}`)
}
```

```
> greeting('Peter')
Hey there, Peter !!
```

Spread ...

Spread syntax(...) allows an iterable such as an array to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected.

Example)

```
nums =[-34,23,89,-10,0,56,-92]
```

let num1 = Math.max(nums) // since Math.max() takes since values in the parenthesis to compare which one of them is the greatest

```
> num1
< NaN
```

We can change use spread syntax to pass array nums as a set of values in Math.max() method

```
nums = [-34, 23, 89, -10, 0, 56, -92]
```

```
let num1 = Math.max(...nums)
```

```
> num1  
< 89
```

Also, using spread syntax in arrays will look as

```
> console.log(nums)  
▶ (7) [-34, 23, 89, -10, 0, 56, -92]  
← undefined  
> console.log(...nums)  
-34 23 89 -10 0 56 -92
```

Spread in array

Spread allows array to have an independent copy on an array

Example)

```
let colors = ['red', 'blue', 'yellow']
```

```
let animals = ['cat', 'dog']
```

```
let comArray = [...colors];
```

```
> comArray  
← ▶ (3) ['red', 'blue', 'yellow']  
> colors  
← ▶ (3) ['red', 'blue', 'yellow']  
> colors.push('green')  
← 4  
> colors  
← ▶ (4) ['red', 'blue', 'yellow', 'green']  
> comArray  
← ▶ (3) ['red', 'blue', 'yellow']
```

We can also combine arrays using the spread syntax:

```
let comArray = [...colors, ...animals];
```

```
> comArray
< ▶ (5) ['red', 'blue', 'yellow', 'cat', 'dog']
```

Spread string

```
> name = 'Peter'
< 'Peter'
> let spreadName = [...name]
< undefined
> spreadName
< ▶ (5) ['P', 'e', 't', 'e', 'r']
```

...CONTINUE WITH VIDEO

Example) with **let**

```
<script type="text/javascript">
  for (let i = 0; i < 10; i++) {
    console.log(i);
  }
  console.log(`The last value of i is ${i}`);
```

```
0    index.html?name=Huixin+Wu:21
1    index.html?name=Huixin+Wu:21
2    index.html?name=Huixin+Wu:21
3    index.html?name=Huixin+Wu:21
4    index.html?name=Huixin+Wu:21
5    index.html?name=Huixin+Wu:21
6    index.html?name=Huixin+Wu:21
7    index.html?name=Huixin+Wu:21
8    index.html?name=Huixin+Wu:21
9    index.html?name=Huixin+Wu:21
✖ ▶  index.html?name=Huixin+Wu:23
    Uncaught ReferenceError: i is not
    defined
      at index.html?name=Huixin+Wu:23
>
```

Strategy 1: Immediately Invoked Function Expression, IIFE → don't need to call it to run. Automatically runs

```
<p>paragraph 1</p>
<p>paragraph 2</p>
<p>paragraph 3</p>
<script type="text/javascript">
```

```
(function(){
  var myPs = document.querySelectorAll('p');
  for(var i=0; i<myPs.length; i++){
    myPs[i].style.color = "green";
  }
})();
```

Stragery 2: use strict

It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.

The purpose of **"use strict"** is to indicate that the code should be executed in "strict mode".

With strict mode, you can not, for example, use undeclared variables.

Example) without **"use strict"**;

```
<p>paragraph 1</p>
<p>paragraph 2</p>
<p>paragraph 3</p>
<script type="text/javascript">
  (function(){
    myPs = document.querySelectorAll('p');
    for(var i=0; i<myPs.length; i++){
      myPs[i].style.color = "green";
    }
  })();
</script>
```

output

paragraph 1

paragraph 2

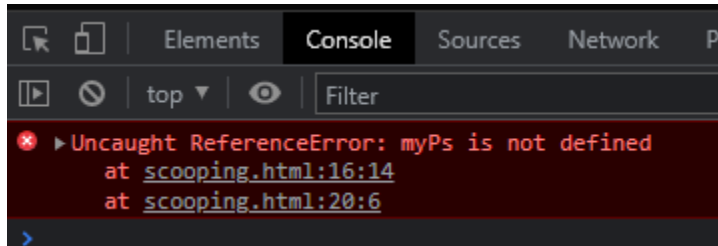
paragraph 3

Example) with **"use strict"**;

```
(function(){
  "use strict";
  myPs = document.querySelectorAll('p');
  for(var i=0; i<myPs.length; i++){
```

```
    myPs[i].style.color = "green";  
  }  
}());
```

Output



<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>

<https://developer.mozilla.org/en-US/docs/Glossary/IIFE>