Java Script

It's better to put < script > element below all the elements in the body section. There are two reasons to do so … The browser start parsing the file from top to bottom and if you have put the script code at top then browser will keep parsing that for long and your page will not show any thing and this will create a bad user experience… so it's better to not put the code at the top. Although there are some exceptions also.But still most of time we can go with this format;

And second thing all these operation of javascript will work on the elements ...so it is better to have the js code later than elements;

Type of Notation should use :

```
Camel notation : oneTwoThree
Pascal notation : ( for constructor functions)->OneTwoThird
```

Node is a program which contain javascript in google's v8 engine.

Variable in JS

We can use let to declare the variables in the javaScript;

For sure there are some rules while defining the name

1.) There should be no number in the beginning;
2.) No hyphen (-) should be used
3.) Cannot use reserved words (keywords)

A good software engineering ethic: We should also use meaning full name;

let variable_name=34;
If you don't initialize then there would be an error : undefined error;

```
Primitive Type:  number, string, bool, undefined, null
```

Java-Script is a dynamic type

Falsy and Truthy in JS;
Falsy means those which are equivalent to false -
Undefined, ' ' ,null, 0, false, NaN

Anything that is not falsy is truthy.

Factory Function :

```
// Create object by Factory Functions;
function createObj(name){
    return{
        name,
        view(){
            console.log('name is : ',name);
        }
    };
}
const obj1=createObj('sourav',20);
obj2.view();
-----------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
name is :  sourav
```

And the another way is using the constructor function :

```
// Constructor Function;
function Circle(radius){
    this.radius = radius;
    this.draw = function(){
        console.log('Ok I have draw a circle');
    }
}

const obj = new Circle(1);
// new is an 'operator' and it will return a 'this' pointer to the
object
obj.draw();
-----------------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
Ok I have draw a circle
```

Strictly saying there is no difference between these two pattern - so we can use any one of them.

Dynamic nature of Object;

It means that once you create object in JS you can always add new properties and new methods and as well as delete any one of them at any time ;

```
const obj = {
    name: 'Sourav'
}

obj.age = 20;
obj.view = function (){
    console.log('abc');
}

console.log(obj);

Output:
{ name: 'Sourav', age: 20, view: [Function] }

delete obj.age;
delete obj.view;
console.log(obj);

Output:
{ name: 'Sourav' }

// Looking at the constant object it may seem that how we can change this … but fine we are not changing this to some new object ... but we are only adding properties and functions to the current object;

So if try to change:

obj = {}
console.log(obj);

Then there will this in the output:
-------------------------------------------------------------
Output:
obj = {}
    ^
TypeError: Assignment to constant variable.
```

Constructor Properties:

Suppose this example:

```
let x = {};
// Inside JS engine change it to let x= new Object()
// So it is created by Object constructor function;
console.log(x.constructor);
```

```
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
[Function: Object]
```

Functions are objects:

```
// circle which we have created as a function is internally an
object in javascript

function Circle(radius){
    this.radius=radius;
    this.draw = function (){
        console.log('Circle drawn');
    }
}
// Internally this get converted to
//              |
//              |
//              V

const Obj = new Function('radius', `
    this.radius=radius;
    this.draw = function (){
    console.log('Circle drawn');
}
`)

const obj = new Obj(1);

console.log(obj.radius);
```

Also there are some other methods along with this object .
They are apply , call;
Object.call (this: arg, arguments.. explicitly)
Object.apply (this: arg, [arguments ,...]);

Value Type vs Reference Type:

```
 Value type :  Number, String , Boolean, undefined, null , symbols; These are copied by the value
 Reference type : Objects, Functions, Array : These are copied by the reference
let x = 10;
let y = x;
x=20;
console.log(x,y);
-----------------------------------------------------------
Output : 20, 10
```

Now Reference type (or an object there)

```
let x = { value: 10}; // This object is not stored in the variable x but instead x stores the address of the object ... and when we do let y = x then the address of that location is copy to the y
let y = x; // now x and y are pointing to the same location
x.value= 20 ;
console.log(x,y);

Output:
{ value : 20 } , { value : 20 }
```

See with an example: When we pass with reference and value ;

```
let x = 10;
function f(x){
    x++;
}
f(x);
console.log(x);
```

```
Output :  10
```

But if we pass an object - :

```
let x = {value: 10};
function f(x){
    x.value++;
}
f(x);
console.log(x);
-----------------------------------------------------------------
Output:
{ value: 11 }
```

Enumerating properties of an Object :

```
const circle = {
    radius : 1,
    draw(){
         console.log( ' Circle of radius : ', radius);
    }
}
for (let key in circle){
    console.log(key, circle[key]);
}
-----------------------------------------------------------------
OUtput:
radius 1
draw [Function: draw]
```

But on the other hand if we use for (let key of circle) then this will result in an error … The reason is that we can only use them on iterables … such as [array] and [maps] .

So

```
for (let key of circle){
    console.log(key);
}
Output:
for (let key of circle){
                 ^
```

```
TypeError: circle is not iterable
```

However we want to do it with ‘for’ we have a method which returns an array and since arrays are iterable so we can use for.

```
for (let key of Object.keys(circle)){
    console.log(key);
}
----------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
radius
draw
```

Let’s look on this method more i.e (Object.keys(circle))
Earlier we have seen that this **object is a built-in constructor function**

```
// So somewhere in JS build-in we have constructor function
like this;

function Object() { }

// and whenever we create an object using object literal
syntax
// internally that is converted to this constructor function

// Object literal syntax like this :

const obj = { value : 10 , view : function ( ){
console.log(value);}}
// internally this is translated to a call to the constructor
function
// so that looks like :

obj = new Object ( );

// also we learn that every function in JS is an object ... so
they have properties and methods that we can access

// Beside of object.keys( ) which return a string of array of
the properties and methods of an objects we also have some
other
```

```
// Object.entries() - it returns a { key: value } pair
```

If we wanna see whether a given object has a given method or properties we can use ‘in’ operator.

if(‘radius’ in Circle) console.log(radius);

Cloning an Object :

```
const obj = {
    name: 'Sourav',
    age: 20,
    view(){
        console.log(name, age);
    }
}

const obj1 = {...obj } // ... means we spread.. we are
spreading all the properties and method of the obj in { }

// another way

 const obj2 = Object.assign({color:'red'},obj);

 console.log('obj1 : ', obj1);
 console.log('obj2 : ', obj2);
```

Garbage Collection :

We do not need to explicitly free memory which was allocated earlier… We have JS Garbage Collector and it works itself and clean all those variables which are not in current use and deallocate memory .

String Object and primitive :

```
// String primitive
let name = 'sourav';
console.log(typeof name);

JS engine convert the string primitive to string object
So we can evaluate all the methods that are provided with the
string Object ;
// String Object
const fName = new String('sourav');
```

```
console.log(typeof fName);
```

Example are :

```
let name = 'Sourav Sharma';
console.log('length of  ',name +' is '+ name.length);
console.log('1st character : ',name[0],' and 7th
Character',name[6]);
let store= name.includes('ou');
console.log(store);
-------------------------------------------------------------
OUtput:
PS C:\Users\Sourav Sharma\javascript> node index.js
length of   Sourav Sharma is 13
1st character :  S  and 7th Character
true
```

```
let name = ' Sourav Sharma ';

console.log('index of "a" = ' + name.indexOf('a')); // return
first index from starting;

let name1 = name.replace('Sharma', 'A'); // This returns a new
string ... not make any changes to the original one.
console.log('Replacing Sharma with A =' + name1);

let name2= name.toUpperCase();
console.log('upper Case: =' + name2);

let name3= name.trim(); // trims all the white spaces from
left and right;
console.log(name3);

console.log(name.trimRight());

// Incase if you wanna have single quote in your string;
// Then you have to use escape character;

console.log('my name is \'sourav \' \n and now we are on new
line');
```

```
console.log(name.split('o'));
```

```
-------------------------------------------------------------
OUtput:
index of "a" = 5
Replacing Sharma with A = Sourav A
upper Case: = SOURAV SHARMA
Sourav Sharma
 Sourav Sharma
my name is 'sourav '
 and now we are on new line
[ ' S', 'urav Sharma ' ]
```

Template Literals

```
// Template Literals; like we have object literals {} ,
boolean literals : true, false;
// Suppose you have to send mail to different users: and we
need not to worry about formats:
// like using /n , \' and other

name = 'Sourav';

const msg = `
Hi ${name}
Thanks for supporting cp community!

Regards Rachit Jain.
`;

console.log(msg);
```

```
-------------------------------------------------------------
Output:
Hi Sourav
Thanks for supporting cp community!

Regards Rachit Jain.
```

Date Object in JS:

```
let date =new Date();
console.log(date);
-----------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
2020-01-03T14:50:15.085Z
```

Array in JS

Constant with the declaration doesn't stop us modifying our array - like add new elements, and delete some elements… It only stop to modify it to some brand new array;

Array in JS are objects:

```
const arr = [1,2,3];

// Insert at end;
arr.push(4,5);

// Insert at beginning;
arr.unshift(-1,0);

// Insert at any position ;
arr.splice(2,0,'a','b');

console.log(arr);
-----------------------------------------------------------
OUtput:
[-1, 0, 'a', 'b', 1, 2,  3, 4,   5]
```

Finding elements (Primitive)

```
const arr = [1,2,3,4,5,4,4,4];

const find = arr.indexOf(4); // returns first index at which 4 is founded
// we can also pass second argument which mean from where to start

console.log(arr.indexOf(4,4));

if(find==-1){
  console.log('Not found!');
}
else {
  console.log('Found at index : ',find);

}

const lastFind = arr.lastIndexOf(4);
console.log(lastFind);

// to check whether an element is without checking for return -1 ;

We have new method;

console.log(arr.includes(1));

console.log(arr.sort());
-----------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
5
Found at index :  3
7
true
[ 1, 2, 3, 4, 4, 4, 4, 5]
```

Finding elements in Reference Types:

```
const courses = [
  {id: 3, name: 'CS-3003'},
  {id: 4, name: 'CS-3004'}
];

const course = courses.find(function(i){
  return i.name==='CS-3003';
});

console.log(course);
-----------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
{ id: 3, name: 'CS-3003' }

// If not found then it will return undefined
-----------------------------------------------------------------

:In case if we used findIndex() It will return index of first
appearance :
And in case we don't find the element then it will return -1;

const course = courses.findIndex(function(i){
  return i.name==='abcdef';
});
console.log(course);
-----------------------------------------------------------------
Output :
-1;
```

Arrow function

```
Whenever you want to pass a function as a call back function(
or an argument to a different function ) : like we have done
in previous

// course.find( function(){ } ) , we can use the arrow
function syntax ( => ) see how we can do that:
```

```
const courses = [
  {id: 3, name: 'CS-3003'},
  {id: 4, name: 'CS-3004'}
];

const course = courses.findIndex( (obj) => {
  return obj.name==='abcdef';
});
```

If you have a single parameter in the call-back function then even you can remove parentheses from there... so code will be more cleaner and if you don't have any parameter then pass this : ()=> an empty parentheses and if you have one line of code inside and it is returning something then you can remove return keyword from there and { } these also; So your code will appear:

Remove elements from array:

```
const arr = [1, 2, 3, 4];

// Remove from end:
let store = arr.pop();

// Remove from beginning:
store = arr.shift();

// From any position :
arr.splice(1,1);

console.log(arr);
-------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
[ 2 ]
```

Emptying an array

```
const arr = [1, 2, 3, 4];

// First thing:
arr = [];

// Second :
arr.length(0);

//Third:
arr.splice(0,arr.length());

// Another loops:
// although it is not good approach
while(arr.length()){
  arr.pop();
}
```

Combining two arrays

```
const arr = [1, 2, 3, 4];
const arr2 = [5,6,7,8,9];


const arrNew = arr.concat(arr2); // Both array remain
unaffected and result is in new array :
console.log(arrNew);

const slice = arrNew.slice(2,4); // [x,y) : WHERE x is
starting index (0 based indexing) and y is excluded :
console.log(slice);
In slice if we don't pass any x,y  arrNew.slice() then the
whole array is copied and if we don't pass the second i.e y
then whole array from x index (included) will be copied into
new location :
```

```
------------------------------------------------------------
OUtput:
PS C:\Users\Sourav Sharma\javascript> node index.js
[ 1, 2, 3, 4, 5, 6, 7, 8, 9]
[ 3, 4 ]
```

An important point :

```
const arr = [ {id: 4} ];
const arr2 = [5,6,7,8,9];

Now here we are concatenate the array where it contains object
i.e { id: 1} : and till now we know objects in JS are not
copied but they are passed by reference so the newArr will not
contain the copy of these values but only a reference: So if
we change at original location then changes will reflect at
both locations : because both arr and arrNew both are pointing
to the same location on their 0th index; So If they are
primitive type they are copied by value else by reference;

const arrNew = arr.concat(arr2);

arr[0].id = 'sourav';
console.log(arr);
console.log(arrNew);
------------------------------------------------------------
OUtput:
PS C:\Users\Sourav Sharma\javascript> node index.js
[ { id: 'sourav' } ]
[ { id: 'sourav' }, 5, 6, 7, 8, 9 ]
```

A bit more flexible is spread operator :

```
                    Spread operator IN (ES6)
```

```
const arr = [1, 2, 3, 4];
const arr2 = [5, 6, 7, 8, 9];
// Spread operator ... : we are spreading the elements of arr,
arr2
const arrNew = [...arr, 'Sourav Sharma', ...arr2];
// We can also add elements in between easily
console.log(arrNew);

Output :
PS C:\Users\Sourav Sharma\javascript> node index.js
[ 1, 2, 3, 4, 'Sourav Sharma', 5, 6, 7, 8, 9 ]
```

Let's see how to iterate on the array :

Iterate on array

```
const arr = [1, 2, 3, 4];
// We are using
 forEach(  function(element){console.log(element); })
arr.forEach( value => console.log(value));
// If want to do it with index:

arr.forEach( (value, index) => console.log(index,value));
```

Joining array

```
const arr = [1, 2, 3, 4];
const joined = arr.join(' SD ');
console.log(joined);

const msg = 'This is a message';
const str = msg.split(' ');
console.log(str);

// Now let's join them back :
```

```
const strJoin = str.join('-');
console.log(strJoin);

// This technique is useful in urls because url does not
contain white spaces : but a user when search for any thing
for sure he/she leaves white spaces :
```

Sort array

```
const arr = [28,11,34,56,78];
arr.sort();
console.log(arr);

arr.reverse();
console.log(arr);
```

But working with object's is a little bit different :

```
const arr = [
  {id: 1, name: 'Sourav'},
  {id: 2, name: 'Abcdef'}
]
//arr.sort( (a,b) => b.name < a.name ? 0 : -1 )
function sortArray(){
  arr.sort( (a,b) => {
    if(a.name <b.name)return -1;
    if(a.name > b.name)return 1;
    return 0; // if (a.name === b.name)
  })
  console.log(arr);
}
sortArray();
// see if we change the name of id: 2
arr[0].name ='abcdef';
sortArray();
-------------------------------------------------------------
OUtput:
PS C:\Users\Sourav Sharma\javascript> node index.js
```

```
[ { id: 2, name: 'Abcdef' }, { id: 1, name: 'Sourav' } ]
[ { id: 1, name: 'Sourav' }, { id: 2, name: 'abcdef' } ] //
see Sourav appear before because ASCII(a)> ASCII(S): and if we
want to ignore this we can :
const name1 = arr[0].name.toUpperCase;
const name2 = arr[1].name.toUpperCase;
```

Testing the elements of the array

```
const arr = [ 1, -2, 3, 4, 5];

let allPositive = arr.every( value => value>=0);
console.log(allPositive);

let atLeastOnePositive = arr.some( value => value>=0);
console.log(atLeastOnePositive);
```

Filtering elements

```
const arr = [ 1, -2, 3, 4, 5];

let storeArr = arr.filter( value => value>=0);

console.log(storeArr);
```

Mapping an array

```
const arr = [1, -1, 2, 3];

const allPositive = arr.filter( n => n>=0);

const obj = allPositive.map( n => ({ value: n}));

console.log(obj);
```

```
// Both filter and map returns a new array : they don't modify
the existing one :
// and these functions can be chained :

// arr
//   .filter( n => n>=0)
//   .map( n => ({value:n}))
//   .filter( n => n.value>1);
//   .map( n => n.value);
// So Our final ans will be an array = [2,3];
```

Reducing an array

```
const arr = [1, 1, 2, 3];

let xorValue = arr.reduce((accumulator,currValue)
=>accumulator ^ currValue ,0); // this 0 which is the 2nd
parameter is acting as an initial value for ans = 0;

// If we don't initialize accumulator then it will initialize
with the first element
console.log(xorValue);
```

Exercises:

1. Counting Occurence of an element in an array : using one line reduce method:

```
const arr = [1, 2, 3, 4, 1, 1, 1];
const ans = countOccurence(arr,1);
console.log(ans);

function countOccurence( arr, search ){
    if(arr.length === 0) return undefined;
    return arr.reduce( (a,b)=> (a>b) ? a: b);
}
-------------------------------------------------------------
Output:
4
```

Question 2: Pick all the movies released in year : 2018 and have rating > 4 and sort them in descending order:

```
const movies = [
  {title: 'a', year: 2018 ,rating: 4.5},
  {title: 'b', year: 2018 ,rating: 4.7},
  {title: 'c', year: 2018 ,rating: 3},
  {title: 'd', year: 2017 ,rating: 4.5},
];

go(movies);
function go(movies){
  const store = movies
                  .filter( n=> n.rating>4 && n.year===2018)
                  .sort((a,b)=> a.rating > b.rating ? -1 : 1)
                  .map( n=> n.title);
  console.log(store);

}
```

Function in JS

1. Function Declaration Vs Expression

```
Function Declaration : {Putting semicolon is not necessary}
function f(){
  console.log('I am a declared function ');
}
// Anonymous(no name) Function expression: {Need semicolon}
As we are dealing with an expression (like var x = 4;)
We know function in JS are object : so setting ok to a
function is similar setting it to an object
let ok = function(){console.log('I am function expression');};

 Named Function expression :
```

```
let ok2 = function named(){ console.log(' I am named
functional expression ')};

// Calling these type of function is similar as normal call to
function
ok();
ok2();

let ok3 = ok; // Now because function in JS is object to ok is
not storing the whole function ... the only thing it has it's
reference and now ok3 is also referring to the same location;

ok3();
------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
 I am function expression
 I am named functional expression
 I am function expression
```

Important

The major difference between the Function Declaration and Function Expression :
1) If we do function declaration syntax: then we can call the function even before we define it where as if we use function expression syntax to define a function we can't do the same **[We got a reference error]** - This is similar to using a variable without defining it or declaring:

Now why this is happening : When JS Engine execute this code - **It will move all the function declaration at the top - This is what we call Hoisting .** In JS automatically JS Engine move all the function declaration to the top.

Argument in JS-Dynamic in type and size

```
function sum( a,b ){
  console.log(arguments.callee);
  return a+b;
}
console.log(sum(1,3));
console.log(sum('sourav','-sharma'));
```

```
console.log(sum(1)); // 1 + undefined = NaN
console.log(sum()); // undefined + undefined = NaN
console.log(sum(1,3,4,5)); // Only the first two will be read
console.log(sum(1.4 ,3));
-----------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
4
sourav-sharma
NaN
NaN
4
4.4
```

But Suppose if we want the arguments - dynamic in JS then what to do?
Well every function in JS have a special object called: **'arguments'** [key : value] where key are indexes

```
function sum(){
  var ans = 0;
  for(let i of arguments){
    ans+=i;
  }
  return ans;
}

console.log(sum(1,2,3,4,5,6,7,8,9,10));
-----------------------------------------------------------------
Output: 55
```

A **better way to implement above**

The "Rest Operator" - I am always last in arguments

```
function sum(addThis,... args){
  console.log(args); // Returns an array
  const res = args.reduce( (a,b) => a+b);
  return res + addThis;
}
console.log(sum(100,1,2,3,4,5));
Output:
```

```
[1,2,3,4,5]
115
```

Imp Point : You can't pass any argument after the rest parameter. And this is also the reason to call it rest : it contains rest* of the arguments (0 or more)

Default parameter

```
function interest ( principal, rate =4.5 , year =5){
  return principal*rate/100* year;
  // Similar like C++ : if we give a parameter default value : then all those which are present to the right of that must have default value either it become confusing for JS engine … although there is a trick … to pass that argument undefined in the call -- but it is not recommended
}
console.log(interest(10000));
-----------------------------------------------------------
Output:
2250
```

A **Special** type of method called : Getter and Setter

Getter and Setter - Changes method to properties

```
const person= {
  firstName : 'Sourav',
  lastName : 'Sharma',

  get fullName(){
    return `${person.firstName} ${person.lastName}`;
  },

  set fullName(value){
    const name = value.split(' ');
    this.firstName = name[0];
    this.lastName = name[1];
  }
};
```

```
person.fullName = 'Sourav Rajkumar';
console.log(person.fullName);

-------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
Sourav Rajkumar
```

Error Handling in JS

Try and Catch

The ‘minor’ difference between **Error** and **Exception** is

const e = new Error() : This is simple error object in JS
But the moment we throw this error this become **Exception**

```
const person= {
  firstName : 'Sourav',
  lastName : 'Sharma',

  get fullName(){
    return `${person.firstName} ${person.lastName}`;
  },

  set fullName(value){
    if(typeof value !== 'string')
      throw new Error('This is not a string');
    const name = value.split(' ');
    if(name.length !== 2 )
      throw new Error('Enter first Name and Second Name');
    this.firstName = name[0];
    this.lastName = name[1];
  }
}

try{
  person.fullName = '';
}
```

```
catch(e){
  alert(e);
}
```

```
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
Error: Enter first Name and Second Name
```

Local Vs Global Scope

```
function start(){
  for(let i=0;i<5;++i){
  }
  // console.log(i); //i is only accessible inside this block;So this will result in Reference Error.
}

start();

function global(){
  console.log(i);

  for(var i=0;i<5;++i){
    console.log(i);
    if(i==4){
      var color = 'red';
    }
  }
  console.log(color);
  console.log(i); // It is accessible : out-side also : so var remain active in the function (after declare point : If you console it earlier then 'undefined' will be output)
  // This is one of the weirdest things in JS .Earlier JS only contain var to declare variables ... but after the ES6 or (ES2015) : we also have 'let' and 'const' :These create block scope variable
  // Whereas var create function scope variables
```

```
}
global();
```

NOTE:

Another big problem with **var** is that : If we declare them globally , then they are accessible through **window object** in browsers.What are its consequences :Since window object has only one instance Suppose if you are using some third party library … and if that also have some variable name as you defined globally then that will overwrite your variable.So we should avoid doing so.

And also one more thing if you declare function globally then they are also accessible through window … We will later see how to fix this by using modules

So avoid using var keywords

IMP

The This Keywords

```
// Imp Note: (method -> obj) : mean if inside method you use
this then it refer to object
// [function -> global (window)] : mean if used inside normal
function then it refer to the window object or global object

const obj = {
  title : 'C Programming',
  tags : ['a','b','c'],
  view(){
    console.log(this); // here this refer to the current
object in which it is used :
  },
  showTags(){
      this.tags.forEach( function(tags){
          console.log(this.title,tags); // *goto
      },this); // here as a second parameter we are passing
  }            // the object which will this refer
};
// * -> here we are inside anonymous callback function ( which
is a normal function ( so this point to global object ))
```

```
// It's not a method inside the object obj ... and because
this is a normal function so this refer to the global object
i.e window
// Imp: If we use this.tags.forEach( tags =>
console.log(this.title,tags); ) : this will work fine
obj.showTags();This happens because arrow function use this
value of their container function
```

Lets see how what happen if we use it with normal function

```
function f(){
  console.log(this);
}
f();
// this here refer to the global object
-------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
Object [global] { global: [Circular],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
[Symbol(util.promisify.custom)]: [Function] },
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(util.promisify.custom)]: [Function]
  }
}
-------------------------------------------------------------
```

But if we use **Constructor function :Then see what happen**

```
function f(value){
  this.value = value;
  console.log(this);
}
```

```
const g = new f(45); // If you look output : here 'this' is
not referring to global object

// why ? This happens so because we have seen new create and
empty object { } and now the 'this' it creates will point to
current empty object

------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
f { value: 45 }
```

Changing the value of 'This'

```
// Approach 1. But not so good approach

const obj = {
  title : 'a',
  tags : [1, 2, 3, 4],
  viewTitle(){
    const self = this;
    this.tags.forEach( function(tags){
      console.log(self.title,tags);
    })
  }
}

//obj.viewTitle();

// Approach 2. Using apply
function playVideo(a, b){
  console.log(this);
}

playVideo.call({name: 'Sourav' }, 'a', 'b') ;// First
parameter : this will now reference this object
// If we call it simple then it will return a global object
```

```
playVideo.apply({name: 'Sharma'}, ['a', 'b']);

// The only difference between apply and call is that if you have argument in call if we can pass
// it simply like see above [ I added] , but in the case of apply we have to put them in the array

const g = playVideo.bind( { name: 'Souraavv'}); // This bind method donot call out playVideo method... it returns a new function and set this to point to the newly created object i.e { name : Souraavv}
// As this return a new function
g();
-----------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
{ name: 'Sourav' }
{ name: 'Sharma' }
{ name: 'Souraavv' }
```

Now we will use bind along with callback function (as they are object) and bind is a method with that object.

```
const obj = {
  title : 'a',
  tags : [1, 2, 3, 4],
  viewTitle(){
    this.tags.forEach( function(tags){
      console.log(this.title,tags);
    }.bind(this));
  }
}

obj.viewTitle();
-----------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
a 1
```

```
a 2
a 3
a 4
```

Best Way To Use 'This' inside the callback function

But a **good way to do is using arrow Function (In ES6 they inherits this value from the containing function)**

```
const obj = {
  title : 'a',
  tags : [1, 2, 3, 4],
  viewTitle(){
    this.tags.forEach(  tags => {
       console.log(this.title,tags);
    });
  }
};

obj.viewTitle();
-----------------------------------------------------------------
OUtput:
PS C:\Users\Sourav Sharma\javascript> node index.js
a 1
a 2
a 3
a 4
```

Exercise 1:

```
// Question : We are passing any no. of argument and we have
to return their sum

function sum( ... args){
  return args.reduce( (a,b) => a+b);
}

console.log(sum(1,3,5,6));
```

```
// Second we are passing an array as an argument
const arr = [1, 3, 5, 6];
function sum2(... args){
  if( args.length === 1 && Array.isArray(args[0]))
   args = [... args[0]];
  return args.reduce( (a,b) => a+b);
}

console.log(sum2(arr));
```

Exercise 2.

```
const circle = {

  radius: 1,
  get area(){
    return this.radius * this.radius * Math.PI;
  },

  set area(rad){
    this.radius = rad;
  }
};

console.log(circle.area);
```

Exercise 3:

```

const numbers = [1, 2, 3, 4];

try{
const count = countOccurrences(true, 1);
}
catch(e){
  console.log(e.message);
}


function countOccurrences(array, searchElement) {
 if(! Array.isArray(array) )
```

```
  throw new Error('This is not an array');
else{
  return array.reduce((accumulator, current) => {
    const occurrence = (current === searchElement) ? 1 : 0;
    console.log(accumulator, current, searchElement);
    return accumulator + occurrence;
  }, 0);
}
}
```

Output:

```
PS C:\Users\Sourav Sharma\javascript> node index.js
Error: This is not an array
```

Object Oriented Programming JavaScript

Object Oriented Programming in JS
Four basic things need to understand before

1.) **Encaptulation**
2.) **Abstraction**
3.) **Inheritance**
4.) **Polymorphism**

Abstraction : Let's see how to introduce this in our code.Let's start with its definition. It means that we should hide the details which are not necessary for the user : Like implementation and access to some properties. In short we only want essential to show to user.

Some Definition : **Closure** It tell what a function can access … what are the scopes. A function which is inside some other function can access all the variables and other function defined in the scope of its parent function.Don't get confused with the scope and closure , they are different.

To introduce the abstraction , we will not make those properties and methods but instead we will those simple local variable (using **let keyword** instead of using **'this.propertyName'**) and similar for the function we will use let.See below illustrated.

Private Properties and Method

```
function Circle(radius){
  this.radius = radius;

  let color = 'red'; // This is local to the scope of Circle
  let optimumLocation = function(x,y){
    //.. do it's calculation
  }

  this.draw = function(){
    let x, y;
    y = this.radius;
    optimumLocation(1,2); // This is what we call as closure : Since optimumLocation is accessible so we say it is in closure of its parent function
    console.log('draw');
  }
}

// Note: Their is strict difference between 'closure' and 'scope'
// Scope : is temporary and it dies as the function call ends . All the variable will recreated and reinitialize
// Closure stay there : mean the variable color and optimumLocation will continue to stay in the memory ... they will preserve there state because they are part of the closure of the draw function

const obj = new Circle(10);
console.log(obj);
```

So we have two private members (although we should not refer to them as the member because they are no more members now but are local variables) i.e color and optimumLocation.

Now we have seen how to convert properties to private. But now how we will display them outside if we want. As they are no more accessible outside. So now we are going to introduce the concept (again) getter and setter

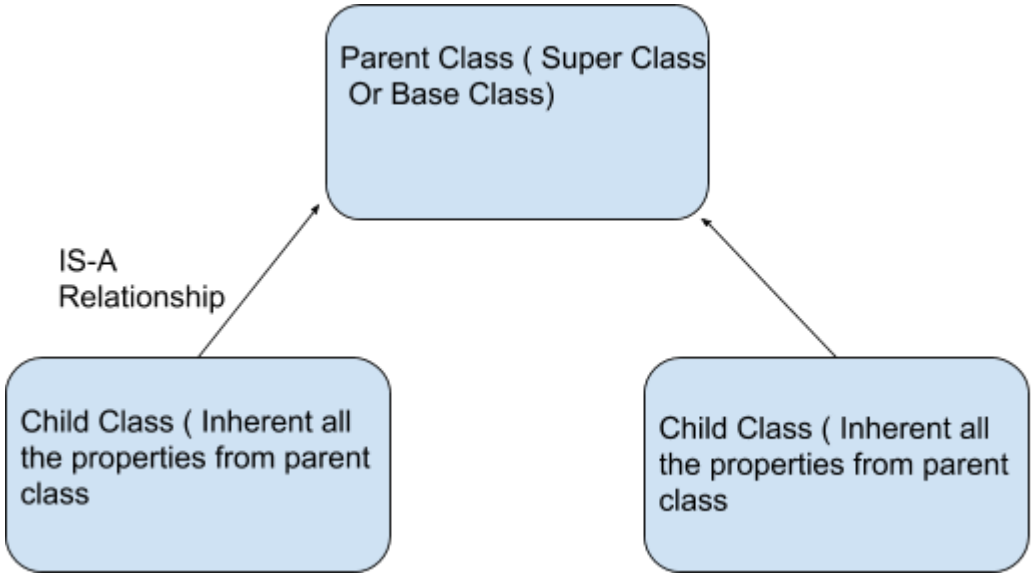
```
function Circle(radius){
  // Local Variables;
  let color = 'red';
  let optimumLocation = { x:1, y:2 };

  this.radius = radius;
  this.draw = function(){
    console.log('draw');
  }

  Object.defineProperty(this,'optimumLocation',{
    get: function(){
      return optimumLocation;
    },
    set: function(value){
      if(!value.x || ! value.y)
       throw new Error('This is invalid input');
      optimumLocation = value;
    }
  });
}

const obj = new Circle(28);

console.log(obj.optimumLocation);
obj.optimumLocation = {x: 2,y:3};
-----------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
{ x: 2, y: 3}
```

Implementing StopWatch

```
function StopWatch(){
    let starting,ending,running,duration =0;
    this.start = function (){
```

```
        if(running)
         throw new Error('It is already started!');
        running = true;
        starting = new Date();
    };
    this.stop = function(){
        if(!running)
         throw new Error('This is not started yet ... we can\'
t stop it');
        running = false;
        ending = new Date();
        const time = ( ending.getTime() - starting.getTime()
)/1000;
        duration+= time;
    };

    this.reset = function(){
        starting = ending = null;
        duration =0;
        running = false;
    };

    Object.defineProperty(this,'duration',{
        get: function(){
            return duration;
        }
    });
}
const obj = new StopWatch();
obj.start();
var i =0 ;
while(i<1000000000){
  i++;
}
obj.stop();
console.log(obj.duration);
Output:
PS C:\Users\Sourav Sharma\javascript> node temp1.js
0.873
```

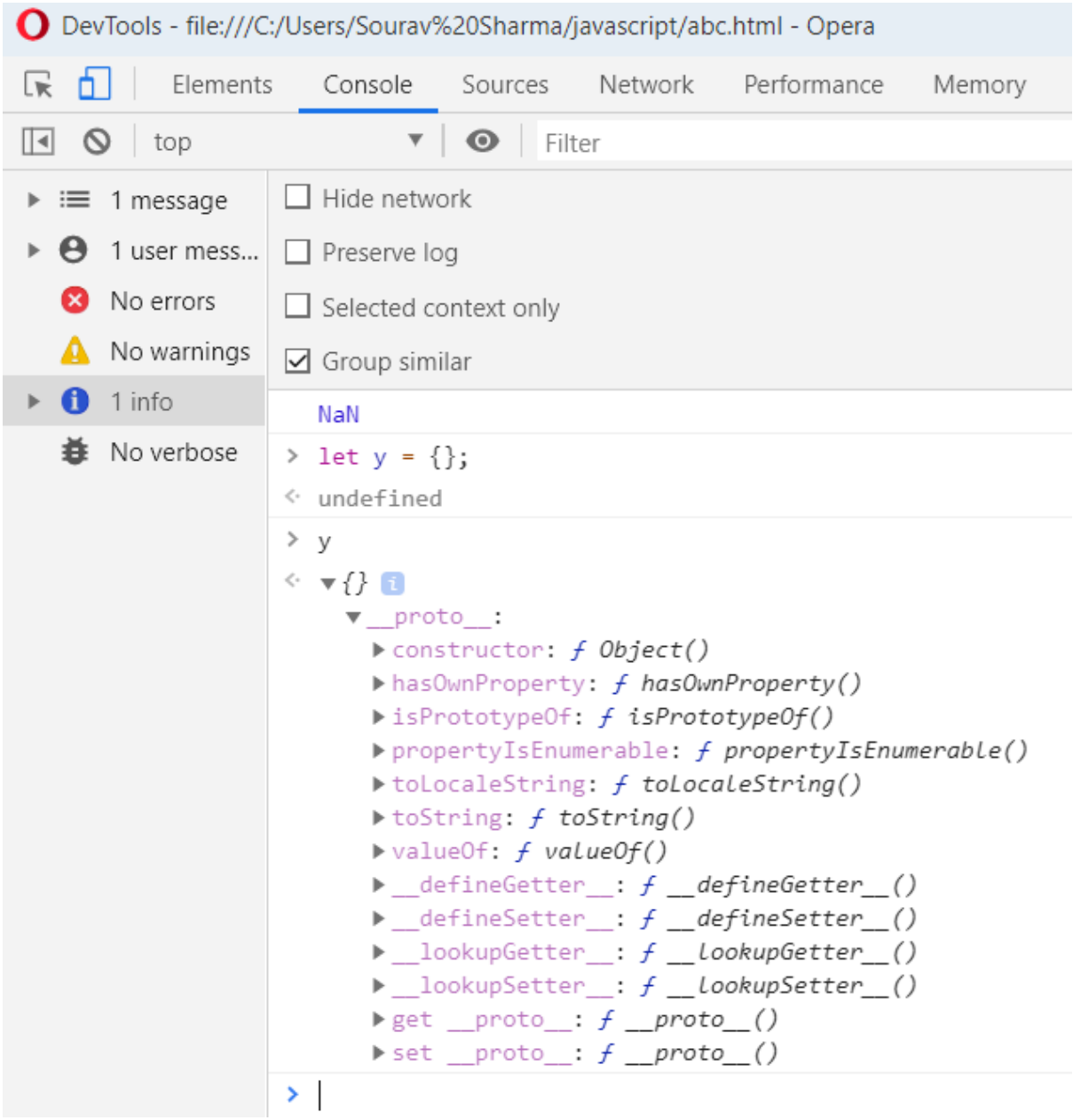
Inheritance



But in JS we don't have classes we only have object. So we implement inheritance using objects only.
The diagram which we can see above is refer to Classical Inheritance. **And there is another term also called** Prototypical Inheritance.

Prototypical Inheritance

Every object in JS (except 1) have a prototype (root object)



see above. We declare a variable y as an empty object and if see y has a **prototype.**

When accessing a property or method on an object and if it is not present in the object then JS Engine look for it in its parent (prototype)
A prototype is also a simple object in memory.Every object in JS has a prototype except the root object.

Multi-level Inheritance

See we can see that prototype which array refer has another prototype and that prototype has another prototype (see that is same as the above example : in Prototype example).

IMP: **Object created by a given constructor will have the same prototype.**

Property Descriptor

```
const person = { name : 'Sourav' };

Object.defineProperty(person,'name',{
  // By default all these properties are by default true;
  writable: false, // will not allow to change the value
  enumerable : true, //If false: name will not shown in keys
of object : Object.keys(person)
  configurable: false // will not allow to delete name from
person object
});

person.name = 'Sharma';
delete person.name;
console.log(Object.keys(person));
console.log(person);

-----------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
[ 'name' ]
{ name: 'Sourav' }
```

----------Imp----------

Every constructor has a prototype property : like

Array.prototype for syntax like const arr = [ ]; (because array inside JS Engine are created using Array constructor function (a built-in **constructor** function).and this constructor object also have a prototype
Object.prototype => let a = { };

and this is exactly the same as the __proto__ property of the object created like Array.prototype === arr.__proto__ ;

Prototype vs Instance Member

```
function Circle(radius){
  this.radius = radius;
```

```
  this.draw = function(){
    console.log('draw');
  }
}
const obj = new Circle(1);
const obj2 = new Circle(2);
```

See we can thousands of method inside this Circle object and similarly we can create thousands of object and we do so we have a lot of copies of draw function (unnecessary) with each created object (and unnecessary occupance of space). So what we can do ? We will use prototyping. Now we will take draw method out of the Circle object and put it in to its prototype. And then we will just have a single copy of this method.

Lets see how we will do that:

Prototype vs Instance Member

```
function Circle(radius){
  // Instance Members
  this.radius = radius;
}
// Circle.prototype === obj.__proto__; These both are pointing to the same object in the memory i.e we called it CircleBase

 // prototype is also an object and in JS we know objects are dynamic so we safely change their property
 // So using Circle.prototype we can safely access the CircleBase Object

 Circle.prototype.draw = function(){
  // Prototype Members
  console.log('draw');
 }
const obj = new Circle(1);
// See the diagram of DevTools ... we can see that now draw is a part of proto and because of prototypical inheritance we can still access it;
obj.draw();
const obj2 = new Circle(2);
```

```
Live reload enabled.
> obj
< ▼Circle {radius: 1}
    radius: 1
  ▼__proto__:
    ▶draw: ƒ ()
    ▶constructor: ƒ Circle(radius)
    ▶__proto__: Object
>
```

So typically in JS we have two types of methods and parameter :
One are **Instance Type or Member** and other are **Inheritance type or Member**

We can also override the function in the prototype : like the **toString()**

```
// We can refer both prototype methods inside instance method
and vice versa

function Circle(radius){
  this.radius = radius;
  this.move = function(){
    this.draw();
    console.log('move');
  }
}

Circle.prototype.draw = function(){
  // this.move(); // Commented this because it will cause
infinite loop
  console.log('draw');
}
Circle.prototype.toString = function(){
  console.log(`This is override function : with radius
${this.radius}`);
}

const obj = new Circle(28);
//obj.draw();
obj.move();
obj.toString();
```

```
------------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
draw
move
This is override function : with radius 28
```

It doesn't matter if you first create the obj and then modify the prototype . The draw method will still be available on the obj methods.Because since we have only one object in the memory (CircleBase) as soon as we make changes they are immediately applied.

Iterate Instance member and Prototype member

Object.key( ) : Only return instance (owner) methods;
for..in : return all property (instance + prototype);

```
function Circle(radius){
  this.radius = radius;

  this.move = function(){
    console.log('move');
  }
}

Circle.prototype.draw = function(){
  console.log('draw');
}

const obj = new Circle(28);

console.log(Object.keys(obj)); // Output: [ 'radius', 'move' ]
only instance member

// What about for..in loop
// It will return all the prototype member + instance member
for(let key in obj){
  console.log(key);
}
```

```
// Note : Some time instance is also alias as owner.We can check using obj.hasOwnProperty('draw'): will return false and true on the remaining two.
-------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
[ 'radius', 'move' ]
radius
move
draw
```

Avoid Extending the Built-in Object

You should avoid to modify the built-in objects in JS
So here is **Rule : Don't Modify objects you don't own!**

One more thing moving things to prototype method will optimize the methods.

But there are also some examples where it is not good to optimize… I mean it's not bad to be optimize your methods. But sometimes doing so we mesh up thing in the objects. Like we may disturb the **abstraction.**

Example : Prototyping is always good

```
function StopWatch(){
    let starting,ending,running,duration =0;
    Object.defineProperty(this,'duration',{
        get: function(){
            return this.duration;
        },
        set: function(val){
            duration =val;
        }
    });
    Object.defineProperty(this,'starting',{
        get: function(){
            return this.starting;
        }
    });
```

```
    Object.defineProperty(this,'ending',{
        get: function(){
            return this.ending;
        }
    });
    Object.defineProperty(this,'running',{
        get: function(){
            return this.running;
        }
    });
}

const obj = new StopWatch();

StopWatch.prototype.stop = function(){
    if(!this.running)
     throw new Error('This is not started yet');
    this.running = false;
    this.ending = new Date();
    const time = ( this.ending.getTime() -
this.starting.getTime() )/1000;
    this.duration+= time;
};

StopWatch.prototype.reset = function(){
    this.starting = ending = null;
    this.duration =0;
    this.running = false;
};

StopWatch.prototype.start = function (){
    if(this.running)
     throw new Error('It is already started!');
    this.running = true;
    this.starting = new Date();
};

const obj = new StopWatch();
```

```
Obj.duration = 10; // Oh we can modify it from outside and now it is useless object.
```

See a user can now set the duration from the outside … which surely we never want. Because in this way our object will present a lie to user.And this is not good to the ethics of **abstraction**.And why we have to do so because we have moved the methods to prototype and now in order to access all the properties we have to make them available outside. Although we don't need to optimize this problem because we know we are not going to create thousands of **stopWatch** objects.

A well said quote : **" Premature Optimization is the root of all evils ".**

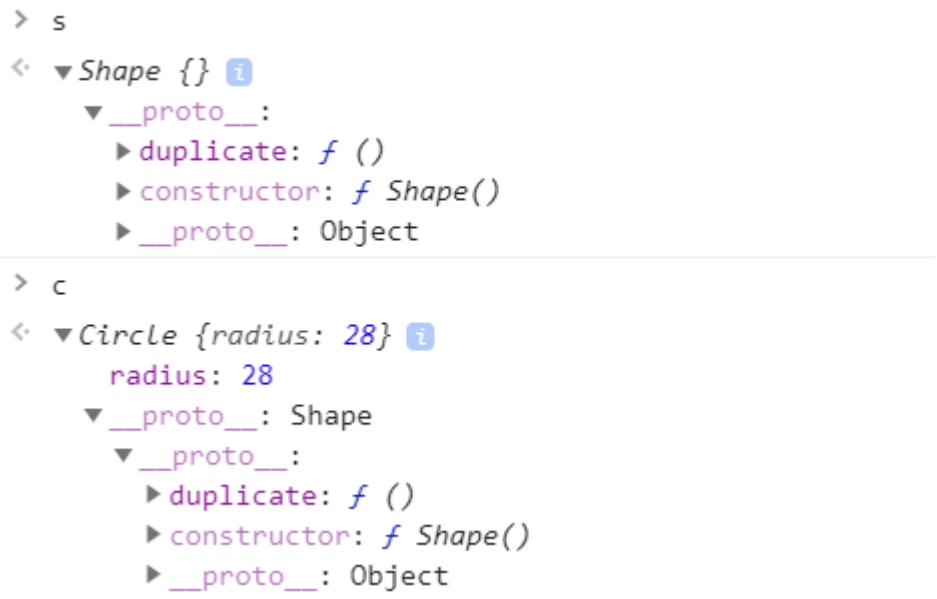
Creating your own prototypes

Suppose later on some sunny day we want to add a square object and that will also have duplicate method same as the Circle.duplicate method , now for sure we don't want to add this again to the Square base or (Square.prototype) . So we make a common object Shape from which both circle and square will inherit and we will define duplicate in Shape's prototype (no need to repeat in both square and circle) .

```
function Shape(){
//..
}
Shape.prototype.duplicate = function(){
    console.log('duplicate');
}

function Circle(radius){
    this.radius = radius;
}

Circle.prototype.draw = function(){
    console.log('draw');
}

Circle.prototype.draw = function(){
    console.log('draw');
}
// Earlier Circle.prototype = Object.create(Object.prototype);
// which was implicit inheritance
Circle.prototype = Object.create(Shape.prototype); //
Object.create() creates an object which inherit from the
ShapeBase
```

```
// and now we want Circle.prototype to inherent from that
object ;

const s = new Shape();
const c = new Circle(28);
```

Here after : __proto__ : name (here name show the parent of the current)so don't get confused:

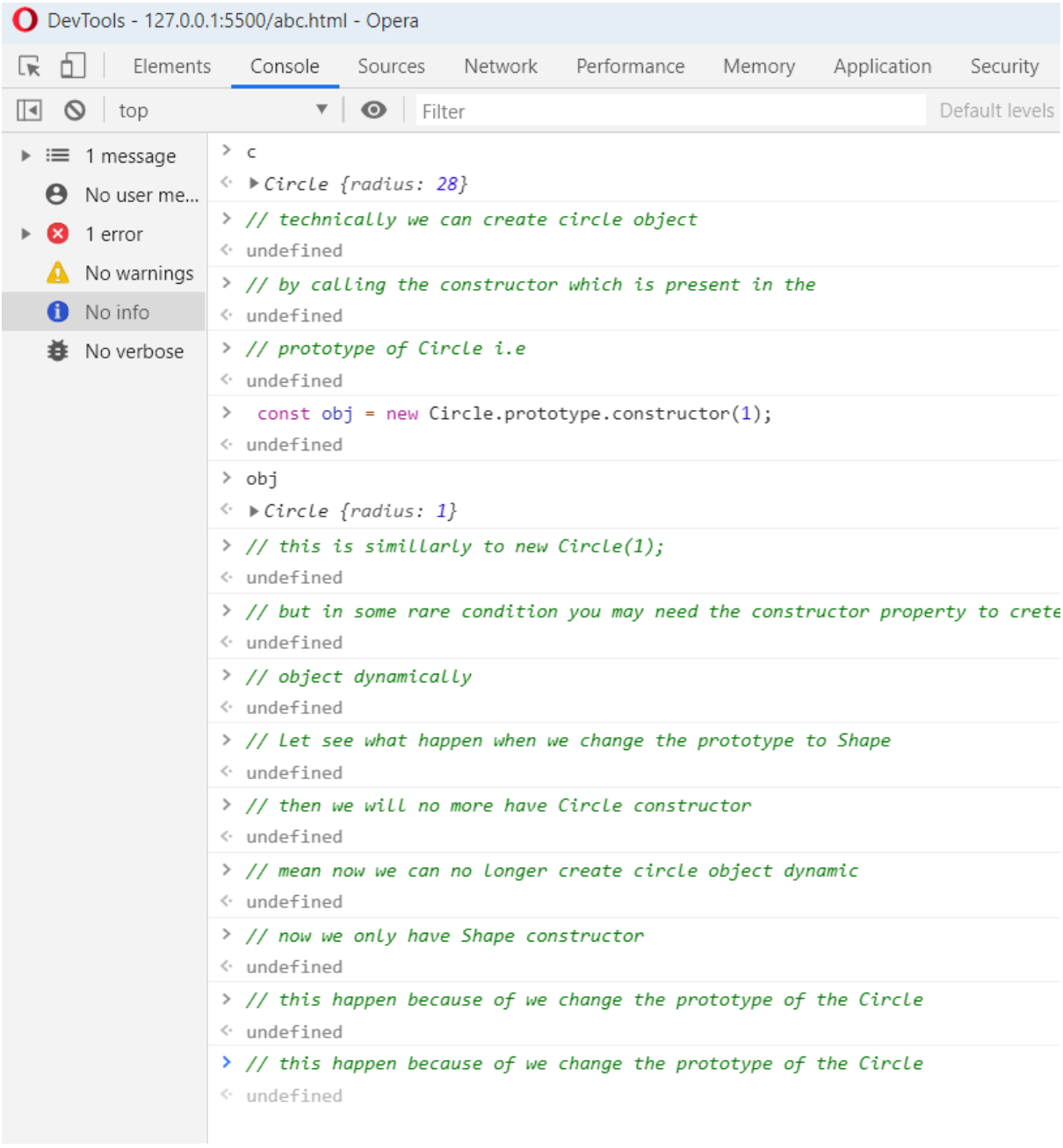


See in Circle the first __proto__ : Shape = It means that the parent of CircleBase is the Shape (fine).
And now Circle also have duplicate method (because of the line :
Circle.prototype = Object.create(Shape.prototype);

There are some problems with this implementation:
See the diagram first and then the code:

```
Circle.prototype = Object.create(Shape.prototype);
Circle.prototype.constructor = Circle;
```

See the second line that is new that we have added.
This line will help us to reset the constructor back to the Circle;
So this is good practice to include this line after you change the prototype of the some object;



Calling the Super Constructor

Let's see how to call the constructor of the Super Constructor:
One simple but that doesn't work: (We will also why it doesn't work):

```
// In this approach we will simply call the Shape in the
Circle object
function Shape(color){
    this.color = color;
};

function Circle(radius,color){
```

```
    Shape(color);
    this.radius = radius;
};

const c = new Circle(28,'red'); // Carefully look at this line and remember how it works

// When we use new it creates an empty object like this {} and let 'this' to point to this object and remember what happen if we don't use 'new' then 'this' be default point to the Global Object ( in browser)
// And the reason it doesn't work  see we call Shape without the new and 'this' in the Shape is now by default to the global
// So we don't set it with the Circle object but instead with the window object; so if you do window.color on console - you will get "red";

// WHAT CAN BE a POSSIBLE SOLUTION?
// For sure if you are thinking that we can put 'new' with the Shape inside the circle then it will create another Shape object and set it's color to the red

// Solution : So we need to call Shape function and set 'this' to point to new instance of circle object
console.log(c);
-----------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
Circle { radius: 28 }
```

Proceeding with the Solution :

Remember our old friend ‘call’ function Object.call({ },arguments)

```
// Solution :

function Shape(color){
    this.color = color;
};
```

```
function Circle(radius,color){
    Shape.call(this,color); // Solution line We are calling
the Shape ( since it is object also)
    // What this line is doing it is binding the Shape object
with 'this' i.e Circle
    this.radius = radius;
};

const c = new Circle(28,'red');
console.log(c);
```

Output:

```
PS C:\Users\Sourav Sharma\javascript> node index.js
Circle { color: 'red', radius: 28 }
```

Intermediate function Inheritance

```
function Shape(color){
    this.color = color;
};

Shape.prototype.duplicate = function (){
    console.log('duplicate');
}

function Circle(radius){
    this.radius = radius;
};

Circle.prototype.draw = function(){
    console.log('draw');
}
// this extend function we call intermediate function
inheritance
function extend(Child, Parent){
    Child.prototype = Object.create(Parent.prototype);
    Child.prototype.constructor = Child;
}
```

```
function Square(side){
    this.side= side;
};
extend(Circle,Shape);
extend(Square,Shape);
```

Method overriding

It may happen that sometimes some child may not need the same implementation of some method which was in parent object. So in that case we will override that method in the child object.

```
function extend(Child, Parent){
    Child.prototype = Object.create(Parent.prototype);
    Child.prototype.constructor = Child;
};

function Shape(color){
    this.color = color;
};

Shape.prototype.duplicate = function (){
    console.log('duplicate in Shape');
};

function Circle(radius){
    this.radius = radius;
};

extend(Circle,Shape);

// It's important to write these lines after the above line : because if we write them earlier then they will vanish as we are modifying the prototype of Circle;

Circle.prototype.duplicate = function(){
 console.log('duplicate in Circle');
};
```

```
const c = new Circle(28);
c.duplicate();
-------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
duplicate in Circle
```

But sometimes we may need the parent implementation also : For that

```
Shape.prototype.duplicate.call(Circle); // this will now call
to the original parent duplicate
```

Poly(many)Morphism(form)

```
function extend(Child, Parent){
    Child.prototype = Object.create(Parent.prototype);
    Child.prototype.constructor = Child;
};

function Shape(color){
    this.color = color;
};

Shape.prototype.duplicate = function (){
    console.log('duplicate in Shape');
};

function Circle(radius){
    this.radius = radius;
};

function Square(side){
    this.side = side;
};
extend(Square,Shape);
extend(Circle,Shape);
```

```
Square.prototype.duplicate= function(){
    console.log('duplicate in Square');
};
Circle.prototype.duplicate = function(){
 console.log('duplicate in Circle');
};


const shapes = [
    new Square(28),
    new Circle(18)
];

// depending on the type of object different duplicate will be
called - This is Polymorphism
for(let key of shapes){
    key.duplicate();
}

-------------------------------------------------------------
OUtput:
PS C:\Users\Sourav Sharma\javascript> node index.js
duplicate in Square
duplicate in Circle
```

Composition is more preferred than **Inheritance.** The reason with inheritance is that it make sometimes basic project complex.So use it wisely.

Composition in JS can be achieved with the help of **Mixins:** where we don't have any hierarchy.

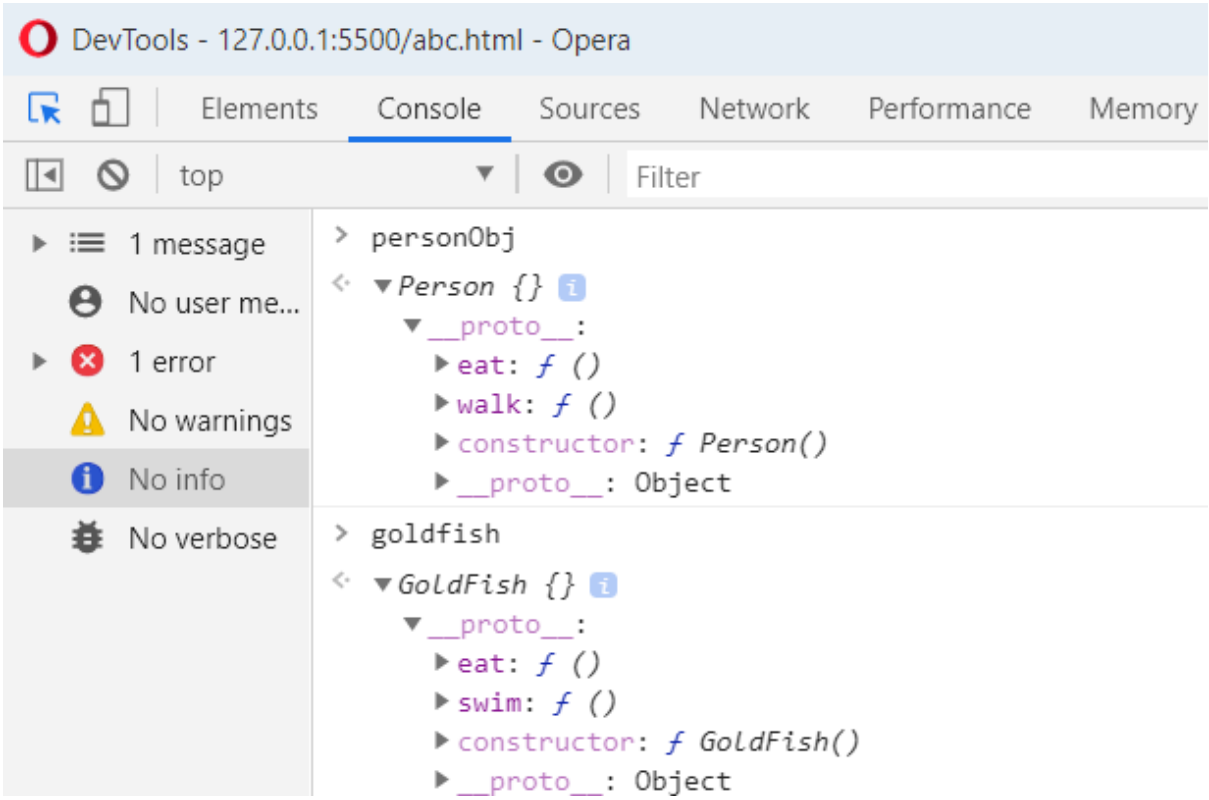
Mixins

```
function mixin(target,... args){
    Object.assign(target,... args);
}
// Defining a feature as an object
const canEat = {
```

```
    eat : function(){
        this.hunger--;
        console.log('eating');
    }
};

// Defining another feature as another object
const canWalk = {
    walk: function(){
        console.log('walking');
    }
};


const canSwim = {
    swim: function(){
        console.log('swim');
    }
}

// here we copy all the method and parameter to the {} object
defined here ;
//const person = Object.assign({},canEat,canWalk);
//console.log(person);


// another thing that we can do we can add them to the object;

function Person(){
};

// here we no need to return ... but instead we are changing
the prototype of the Person so now next time when we create a
person object we can by default have all these property

const personObj = new Person();

function GoldFish(){
    //..
```

```
}

const goldfish = new GoldFish();
mixin(GoldFish.prototype,canSwim,canEat);
mixin(Person.prototype,canWalk,canEat);
```

Output:

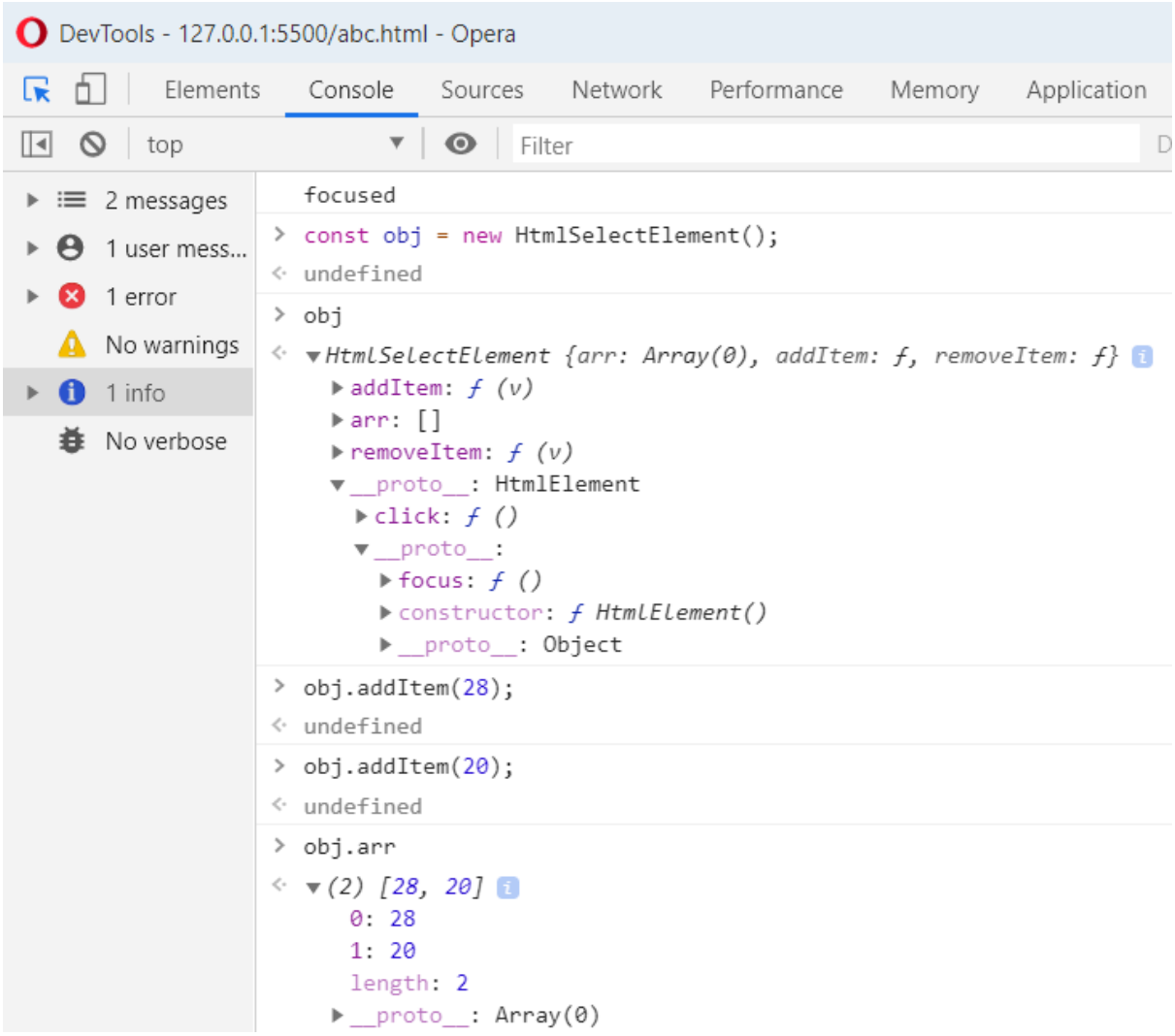


Exercise 1: Prototypical Inheritance

```
function HtmlElement(){
    this.click= function(){
        console.log('clicked');
    }
};

HtmlElement.prototype.focus = function(){
    console.log('focused');
};

function HtmlSelectElement(arr=[]){
    this.arr = arr;
```

```
    this.addItem = function(v){
        this.arr.push(v);
    };
    this.removeItem =function(v){
        this.arr.splice(this.item.indexOf(item),1);
    };
}
//baseHtmlSelectElement
HtmlSelectElement.prototype = new HtmlElement();
//baseHtmlElement

HtmlSelectElement.prototype.constructor = HtmlSelectElement;
```

Output:



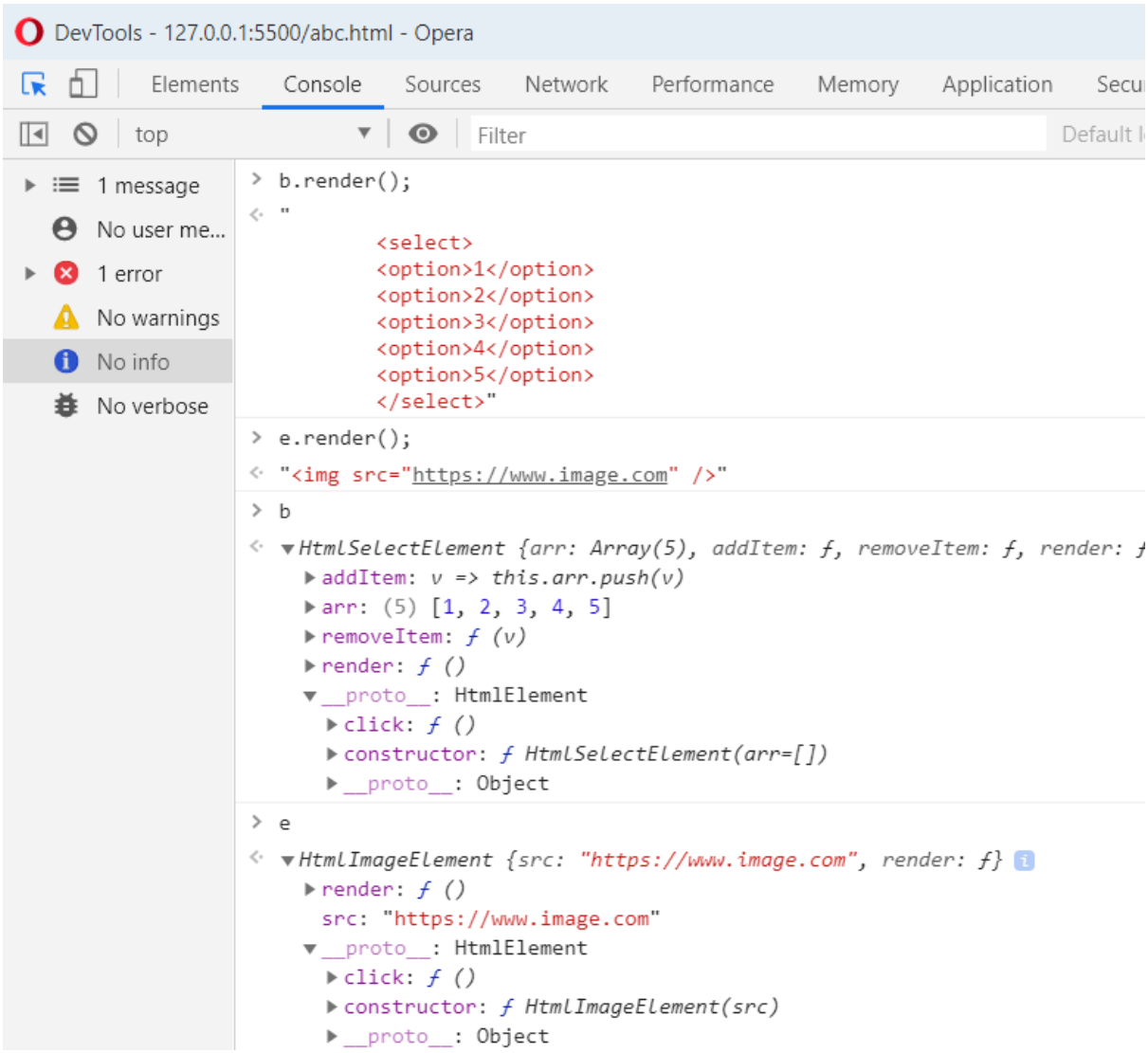
Exercise 2: Polymorphism

```
function HtmlElement(){
    this.click= function(){
```

```
        console.log('clicked');
    }
};

HtmlElement.prototype.focus = function(){
    console.log('focused');
};

function HtmlSelectElement(arr=[]){
    this.arr = arr;

    this.addItem = v => this.arr.push(v);
    this.removeItem =function(v){
        this.arr.splice(this.item.indexOf(item),1);
    };

    this.render = function(){
        return `
        <select> ${this.arr.map(items =>  `
        <option>${items}</option>`).join('')}
        </select>`;
    }
}
//baseHtmlSelectElement
HtmlSelectElement.prototype = new HtmlElement();
//baseHtmlElement

HtmlSelectElement.prototype.constructor = HtmlSelectElement;


function HtmlImageElement(src){
    this.src =src;
    this.render = function(){
        return `<img src="${this.src}" />`
    }
}

HtmlImageElement.prototype= new HtmlElement();
HtmlImageElement.prototype.constructor = HtmlImageElement;
```

```
const b = new HtmlSelectElement([1,2,3,4,5]);
const e = new HtmlImageElement('https://www.image.com');
```

Output



ES6 (ECMAScript-2015)

Content

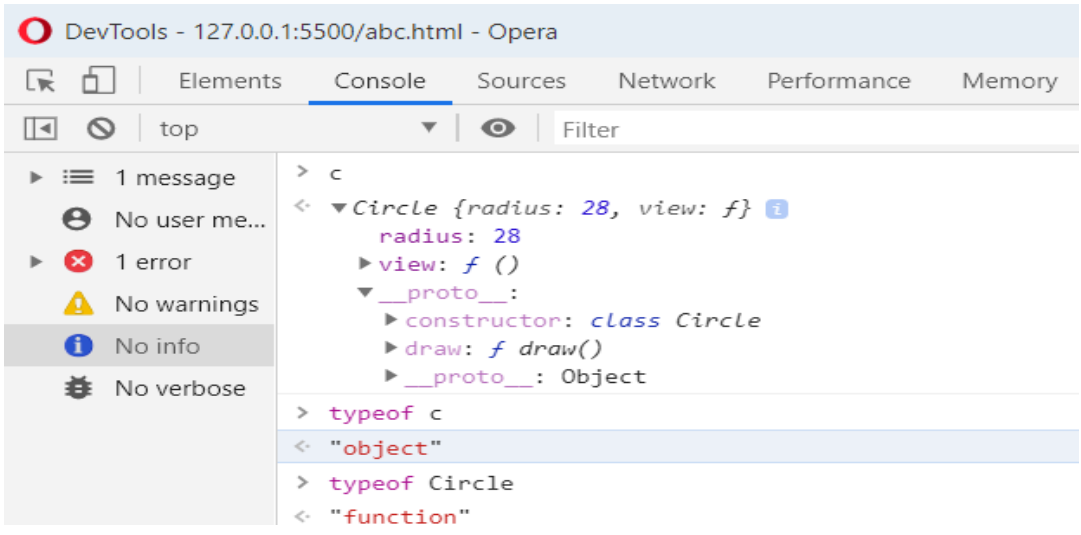
A new way in ES6 to create objects. These classes are not classes in C++,C#, Java They are typically syntactic sugar over constructor function.
So that's the reason we first look for the basic i.e Prototypical Inheritance and how it works and this new one is easier in syntax.

```
function Circle(radius){
    this.radius = radius;
    this.draw = function(){
        console.log('draw');
    }
}
```

Let's see how to write the above code in ES6.

```
class Circle {
    // Every thing which is defined inside constructor appears
as the instance methods;
    constructor(radius){
        this.radius = radius;
        this.view = function(){
            console.log('view');
        }
    }
//Everything inside the body i.e methods are 'PrototypeMethod'
    draw(){
        console.log('draw');
    }
}

const c = new Circle(28);// remember new if you miss:typeerror
```

Output:



Hoisting in Classes

Unlike the function declaration which are hoisted , Classes are not hoisted. See below given example will show and reference error;

```
const c = new Circle(1);

// Class Declaration
class Circle {
}

// Class Expression
const square = class {
};
-----------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
C:\Users\Sourav Sharma\javascript\index.js:1
const c = new Circle(1);
          ^

ReferenceError: Cannot access 'Circle' before initialization
    at Object.<anonymous> (C:\Users\Sourav
Sharma\javascript\index.js:1:11)
```

Which one to use? (Class Declaration or Class Expression) : It is generally recommended to use Class Declaration syntax.

Static Method

Static method are shared with all the objects and they are not separately allocated to each object when we created an object unlike other instance method(non-Static method).

To call these methods we need not to construct object. We can only call them from Class itself , they are not available to any object.

```
class Circle{
    constructor(r){
        this.radius = r;
    }
```

```
    // Instance Method
    draw(){
     //..
    }

    // Static Method
    static parse(str){
        const radius = JSON.parse(str).radius; // Assuming
it's a valid json object
        return new Circle(radius); // returning a circle
object with that radius
    }
}


const circle = Circle.parse('{"radius":28 }'); // this static
method return Circle object;

console.log(circle);

Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
Circle { radius: 28 }
```

Let me show you one more example:
Math in JS is a built in object and this object gives us a bunch of utility function.
Now we never do new Math();
Instead we directly access these methods and these methods must be static so without defining an object of math class.

```
const Circle = function(){
    this.draw = function(){
        console.log(this);
    }
};

const c = new Circle();
```

```
// Method Call : When we call a method of an object then this
bind with this object;
c.draw();
console.log('------------');
const d = c.draw;
// A simple function call : When we call like this then this
will bind to the Global Object ( in Node) and window object in
( Browsers );

d();

Output:
Circle { draw: [Function] }
------------
Object [global] {
  global: [Circular],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
[Symbol(util.promisify.custom)]: [Function] },
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(util.promisify.custom)]: [Function]
  }
}
```

But if we use '**strict mode**' in JS then our code will run in more strict environment… more error and exception will be thrown ;

So what will happen if we introduce 'use strict' mode in the above code

Then output will be :`'use strict';`

```
Circle { draw: [Function] }
----------
Undefined
```

Now the behaviour or ‘this’ keyword will change . So when we enable strict mode if we call a method as a function ‘this’ by default will no longer point to the global object and it will set to undefined.**And the reason for doing so is to prevent us from accidentally changing global objects: That’s bad practice.**

Let’s see how ‘this’ in ES6 classes behave :

```
class Circle{
    draw(){
        console.log(this);
    }
}

const obj = new Circle();
const draw = obj.draw;
draw();
```

What output you are expecting ?
Output :

```
undefined
```

And why so? **Because by default the body of classes are executed in ‘strict mode’.**

Private Members Using Symbols - Abstraction in ES6

```
// 1. Using ES6 - Symbols
// In ES6 we have a new type( primitive )called symbol
const _radius = Symbol();
// Symbol will generate a unique identifier : remember this is
not a constructor function ( so we can not use new with it )

class Circle{
    constructor(radius){
        this[_radius]= radius; // Public by default
    }
}

const c= new Circle(28);
console.log(c);
// this is a trick now using which we can access the symbols:
const key = Object.getOwnPropertySymbols(c)[0];
```

```
console.log(c[key]);
```

Another method

Now technically we can access these private property / method if we access to the weakmaps. But later we will see **module** and then we will see how hide these methods and property. And only export class.

Private Member and Property using WeakMaps (a new type in ES6)

```
const _radius = new WeakMap();
// A 'WeakMap' is essentially a dictionary where 'keys' are objects and value can be anything , and the reason we call them weakMaps is the keys are weak : so if there are no reference to these keys then they will be garbage collector
const _move = new WeakMap();

class Circle{
    constructor(radius){
        // If we want to set this property then we will use 'set' and if want then 'get'
        _radius.set(this,radius); // this is key and radius is value
        _move.set(this, function(){
            console.log('move',this);
        });
        //    Imp:
// If you forget concept related then go to 'article this keyword'; SD: this if used inside this call back function then it refers to the global object ( window ) because we have seen class code is executed in strict mode and now this point to the 'undefine' so that we can't make any changes accidentally ;

// On the other hand if we use 'arrow function' i.e () => then this will inherit this of the container in which it is and at
```

```
this moment it is in constructor container and 'this' here
points to the object itself i.e Circle Object so in that case
this will output : Circle
    }


    draw(){
        console.log(`Consoling private property - radius :
`,_radius.get(this)); // and this will return the value
associated with radius
        console.log(`consoling private method- move: `);
         _move.get(this)(); // here what is value ? It's a
function so this will return a function and to call that
function we are using "()" ahead
    }
}

const c = new Circle(1); // if you console c , then it will
not show radius (private property)
c.draw();
```

Output: (if we use callback function)

```
PS C:\Users\Sourav Sharma\javascript> node index.js
Consoling private property - radius :  1
consoling private method- move: move undefined
```

Output: (If we use arrow function)

```
PS C:\Users\Sourav Sharma\java script> node index.js
Consoling private property - radius :  1
consoling private method- move: move Circle {}
```

Another way to write the above code : where we only have one WeakMap() and not multiple

```
const template = new WeakMap();

class Circle{
    constructor(radius){
```

```
        // as value can be any : Object (so we are passing);
        template.set(this,{
            radius: radius,
            move : ()=> {console.log('move',this);}
        });
    }
    // Now accessing them become different:
    draw(){
        console.log(template.get(this).radius); //
template.get(this) will return a ??.. value and that value
here is ??.. Object : so it return an object i.e { radius:
radius, move: ()=>{}} and inorder to access the element of
this object we have to use '.' again (or remember we can also
use ['nameOfProperty']).
    }
}

const c = new Circle(28);
c.draw();
-----------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
28
```

Getter and Setter in ES6

Remember our old friends **getter** and **setter** having ~~weird syntax~~ beautiful simple syntax to implement ,now we are going to make it more simpler in ES6.

We will use get and set as a keyword in the prefix of a property name.Let's see with an example

```
const _radius = new WeakMap();

class Circle {
    constructor(radius){
        _radius.set(this,radius);
    }
```

```
    get radius(){
        console.log(_radius.get(this));
    }

    set radius(value){
        if(value<0)throw new Error('Invalid Radius');
        _radius.set(this,value);
    }
}

const c = new Circle(28);

c.radius;
try{
c.radius = 1;
}
catch(e){
    console.log(e);
}
c.radius;
```

Output:

```
PS C:\Users\Sourav Sharma\javascript> node index.js
28
1
```

Inheritance in ES6 - Simple

Inheritance in ES6 is much simpler.

```
class a{
    constructor(name){
        this.name=name;
    }
    getName(){
        console.log(this.name);
    }
}
class Shape extends a{
```

```
    constructor(name,color){
        super(name);
        this.color = color;
    }
    move(){
        console.log('move and color is: ',this.color);
    }
}

class Circle extends Shape {
    constructor(name,color,radius){
        super(name,color); // This line should be first
because first we need to construct parent class and then only
we can construct child class
        this.radius = radius;
    }
    draw(){
        console.log('draw');
    }
}

const c = new Circle('sourav','red',28);
c.draw();
c.move();
c.getName();

// Imp Notes:
// If we don't call constructor of the super class (parent
class) we get this error :
// ReferenceError: Must call super constructor in derived
class before accessing 'this' or returning from derived
constructor
```

Output:

```
draw
move and color is:  red
sourav
```

Method Overriding

```
class Parent{
    move(){
        console.log('called :: parent move');
    }
}

class Child extends Parent{
    move(){
        super.move(); // will call to Parent.move();
        console.log('called :: child move');
    }
}

const c = new Child();
c.move();
```

Output:

```
called :: parent move
called :: child move
```

Exercise: Stack( ) Class Implementation.

```
const _p = new WeakMap();
const _arr = new WeakMap();

class Stack{
    constructor(){
        _p.set(this,-1);
        _arr.set(this,[]);
    }
    count(){
        console.log(`Total elements: `,_p.get(this)+1);
    }
    push(value){
        _p.set(this,_p.get(this)+1);
```

```
        _arr.get(this)[_p.get(this)]=value;
    }
    top(){
        if(_p.get(this)<0){
            throw new Error('Stack is already empty');
        }
        console.log(`Top most element :
`,_arr.get(this)[_p.get(this)]);
    }
    pop(){
        if(_p.get(this)<0)
            throw new Error('Stack is already empty');
        _p.set(this,_p.get(this)-1);
    }
    show(){
        console.log('Elements in stack are : ');
        const temp= [..._arr.get(this)]
        for(let i=_p.get(this);i>=0;--i)
        console.log(temp[i]);
    }
}

const s = new Stack();
try{
    console.log(s);
    s.push(1);
    s.push(2);
    s.push(3);
    s.push(4);
    s.top();
    s.show();
    s.count();
    console.log('popped one element');
    s.pop();
    s.show();
}
catch(e){
    console.log(e);
}
```

```
------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
Stack {}
Top most element :  4
Elements in stack are :
4
3
2
1
Total elements:  4
popped one element
Elements in stack are :
3
2
1
```

ES6__TOOLING

Modules

It is good to have single file because having K_LOC in single file is worst.
So we have to make modules.
The main reason to make module:

1) Maintainability
2) Reuse
3) Abstract [imp one] (Hide)

Earlier we have seen that we are using WeakMap(); to create private property ;
But still there were loopholes where we can access items which are private outside ;

```
const c = new Circle(28);
console.log(_radius.get(c));
```

So we want a **proper abstraction** . So what we do we will put private members in different module.

History note ~~which is not important at all~~ [imp]:
In ES5 there was no concept of modules ,so clever developer create some method which implement the functionality of the modules

1) AMD: Asynchronous Module Definition (used in browser application)
2) UMD: Universal Module Definition (can be used in both browser and nodeJS)
3) **CommonJS(imp in real):** used in NodeJS.

But in ES6(used in browser) by default modules are there

CommonJS Module

circle.js

```
// By default the code in module is private : until we
explicitly export it

const _radius = new WeakMap(); // Implementation Detail

// Public Interface
class Circle {
    constructor(radius){
        _radius.set(this,radius);
    }
    draw(){
        console.log(`Circle with radius
${_radius.get(this)}`);
    }

}

// we have a keyword 'module' which refer to current module
and this has property called exports( which is an object )

module.exports = Circle;
/* new ES6 code
const _Circle = Circle;
export { _Circle as Circle };
*/
```

```
// Imp: If we have only one object to export we can simplify
the above line :
// module.exports = Circle;
// if we have multiple class we can have module.exports.Square
= Square;

// Now we will import these to index.js and then simply access

// Now here is the interesting part : In circle module we are
only exporting Circle class so _radius is not accessible in
our other module : this is an implementation detail
// Circle class which we are importing is called Public
INterface and other is IMplementation detail
```

Index.js

```
// Common JS Module

const Circle = require('./circle'); // IN path: './' - refer
current folder ,no need to add extension
// we can also use part of CommonJS : require('./circle')


const c = new Circle(10);
c.draw();

----------------------------------------------------------------
Output:
PS C:\Users\Sourav Sharma\javascript> node index.js
Circle with radius 10
```

ES6 module

Export from circle.js

```
const _radius = new WeakMap();
export class Circle{
    constructor(radius){
        _radius.set(this,radius);
```

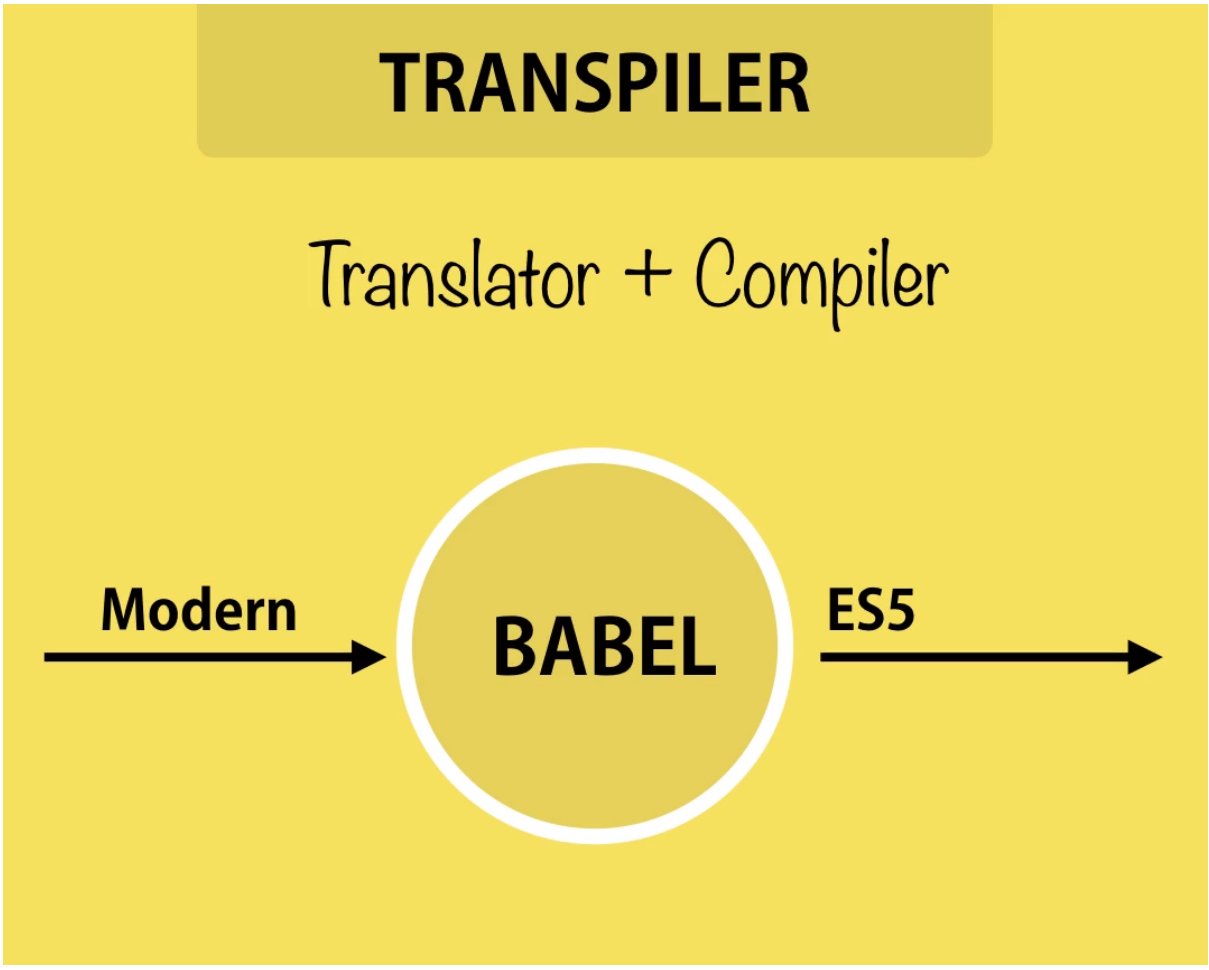
```
    }

    draw(){
        console.log('circle class '+ _radius.get(this));
    }
}
```

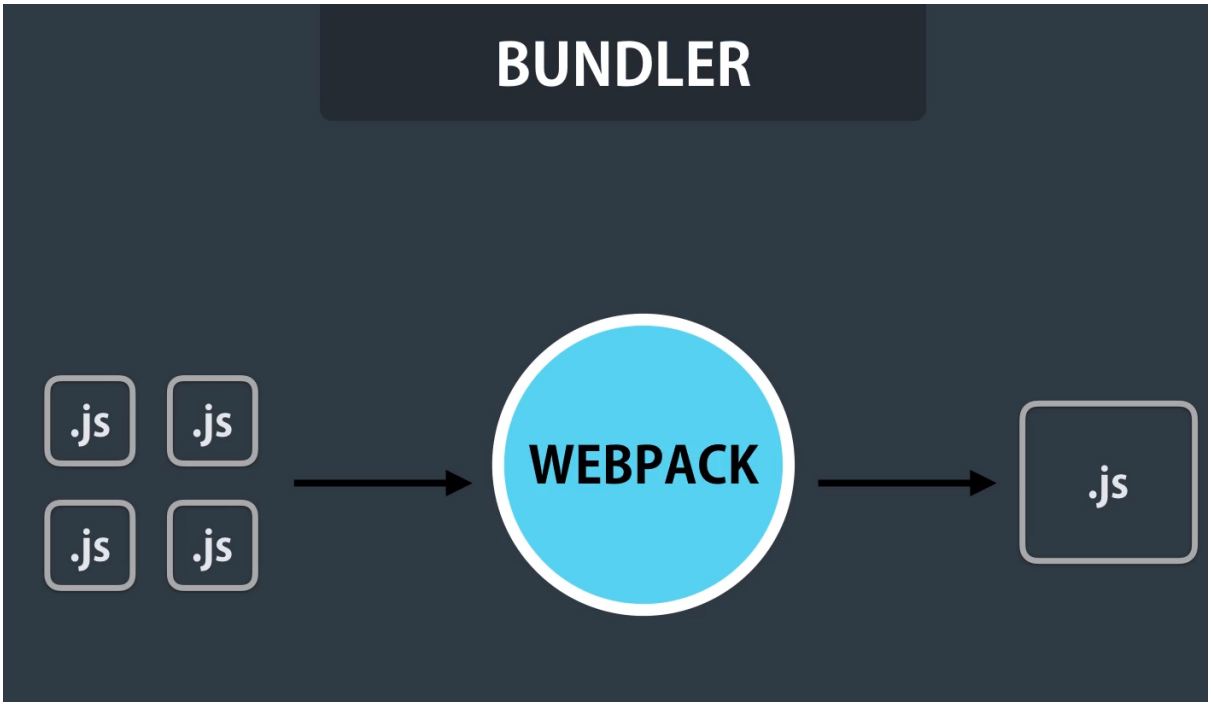
And import in index.js

```
import {Circle} from './circle.js';

const c = new Circle(10);
c.draw();
----------------------------------------------------------------
Output:
Circle class 10
```

ES6 Tools

Only helpful if we are developing browser application … not worry if you are building application in NodeJS





f