Interview Questions

1. Solve below

```
for(var i=0;i<3;i++){
    setTimeout(function(){
        console.log("i value is ",i)
    },1000*i)
}
/*
o/p is
i value is  3
i value is  3
i value is  3
*/
//but we need to get 1, 2,3 then
//1. first solution is bind the value scope to the function i.e snapshot bind
copy will helps to maintain I value
for(var i=0;i<3;i++){
    setTimeout((function(num){
        console.log("i value is ",num)
    }).bind("names",i),1000*i)
}
//2. second solution is to use clouses
//The key is to pass a copy of i into the function to be used later...the clouser
for(var i=0;i<3;i++){
    setTimeout(function(num){
        console.log(num);
    },i*1000,i)
}
```

2. Solve below

```
names=["chandra","babu","pothi","reddy"];
/*
for(var i=0;i<3;i++){
    setTimeout(function(){
        console.log(i,names[i]);
    },1000*i)
}
//output will be
3 'reddy'
3 'reddy'
3 'reddy'*/
// if you execute same code in forEach the parameter function will create a
closure.
```

```
names.forEach(function(item,index){
    //this funciton creates a closure means we can access the all array values
    setTimeout(function(){
        console.log(index,item)
    },1000*index)
});
//output will be
/*
0 'chandra'
1 'babu'
2 'pothi'
3 'reddy'
*/
```

3. What is the difference between for…in and for….of

For—In : Loop through property names

For---Of : Loop through property values

```
var person = {id:1,name:"chandra",address:"Hyd"};

for(p in person){
    console.log(p);//id name address
    console.log(person[p]);//values will be print
}

for(r of person){
  console.log(r);//this will get errror because person is not iterable
}
```

For-Of will work only for iterables but For-In will work for arrays and Objects

```
var names = ["chandra","babu","pothi","reddy"]
for(r of names){
  console.log(r);//prints all the values of array
}

for(p in names){
    console.log(p);//indexes i.e 0 1 2 3 will print
}
```

//for-In looping all enumerable properties only.If enumerable is false then it will ignore

```
var names = ["chandra","babu","pothi","reddy"]
names.elf="Jscript";
Object.defineProperty(names,"ent",{value:"Treebreed",enumerable:true});
for(let p in names){
    console.log(p);
    //for...In is looking through all of enumerable properties inside of
object...If enumerable is true..then it will print...otherwise it will
ignore...by default enumerable is true.
}

let middleEarth = {
    'towns':["chennai","Hyd"],
    'villages':["vp1","kdp"]
};
middleEarth.creater = "chandra babu";
Object.defineProperty(middleEarth,"age",{value:"3rd",enumerable:true});

for(let p in middleEarth){
    console.log(p)
}

//for in looping all properties called enumerables
```

Then, what is Enumerable? → If type false, that property is not inherited.

Enumerable property means ~~not changeable~~. By default all are true

true → Inherited

false → not inherit

```
Object.defineProperty(middleEarth,"age",{value:"3rd",enumerable:true});

middleEarth.age = "4th";
console.log(middleEarth.age);//prints 3rd
```

prints 4th

4. Array.of and TypedArray.of

The **Array.of()** method creates a new Array instance with a variable number of arguments, regardless of number or type of the arguments.
The difference between **Array.of()** and the **Array** constructor is in the handling of integer arguments: **Array.of(7)** creates an array with a single element, 7,
whereas **Array(7)**creates an empty array with a length property of 7 (**Note:** this implies an array of 7 empty slots, not slots with actual undefined values).

```
Array.of(7);       // [7]
Array.of(1, 2, 3); // [1, 2, 3]
```

```
Array.of(7);       // [7]
Array.of(1, 2, 3); // [1, 2, 3]

Array(7);          // [ , , , , , , , ]
Array(1, 2, 3);    // [1, 2, 3]
```

The *TypedArray*.**of()** method creates a new typed array with a variable number of arguments. This method is nearly the same as Array.of()

```
Uint8Array.of(1);               // Uint8Array [ 1 ]
Int8Array.of('1', '2', '3'); // Int8Array [ 1, 2, 3 ]
Float32Array.of(1, 2, 3);    // Float32Array [ 1, 2, 3 ]
Int16Array.of(undefined);    // IntArray [ 0 ]
```

Some subtle distinctions between Array.of() and *TypedArray*.of():

- If the this value passed to *TypedArray*.of is not a constructor, *TypedArray*.of will throw a TypeError, where Array.of defaults to creating a new Array.
- *TypedArray*.of uses [[Put]] where Array.of uses [[DefineProperty]]. Hence, when working with Proxy objects, it calls handler.set to create new elements rather than handler.defineProperty.

5. Array.from vs TypedArray.from

The **Array.from()** method creates a new, shallow-copied Array instance from an array-like or iterable object.

```
console.log(Array.from('foo'));
// expected output: Array ["f", "o", "o"]

console.log(Array.from([1, 2, 3], x => x + x));
// expected output: Array [2, 4, 6]
```

The ***TypedArray*.from()** method creates a new typed array from an array-like or iterable object. This method is nearly the same as Array.from().

```
// Set (iterable object)
var s = new Set([1, 2, 3]);
Uint8Array.from(s);
```

```
// Uint8Array [ 1, 2, 3 ]

// String
Int16Array.from('123');
// Int16Array [ 1, 2, 3 ]

// Using an arrow function as the map function to
// manipulate the elements
Float32Array.from([1, 2, 3], x => x + x);
// Float32Array [ 2, 4, 6 ]

// Generate a sequence of numbers
Uint8Array.from({length: 5}, (v, k) => k);
// Uint8Array [ 0, 1, 2, 3, 4 ]
```

6. Object.keys()

Object.keys() will return the enumerable properties in the object.

Object.keys() return only own properties which are enumerable.

7. How we can get the Attributes of object property?

```
var o = {a:1,b:2};
var c = Object.getOwnPropertyDescriptor(o,"a");

console.log(c);
console.log(Object.keys(o));//['a',b'];

var o2 =Object.create(o);
console.log(Object.keys(o2));//[] becuase there is no own properties
```

8.

8. Difference between Iterators and generators?

Generators are some special kind of functions in java script that contains pause and play states.

```
function* AnotherGen(i){
    yield i+1;
    yield i+2;
    yield i+3;
}

function* generate(i){
    yield i;
    yield* AnotherGen();
    yield i+10;
}

var gen = generate(10);

console.log(gen.next());//{ value: 10, done: false }
console.log(gen.next().value);//11
console.log(gen.next().value);
console.log(gen.next().value);
console.log(gen.next().value);
//when evere length complete done returns true.
console.log(gen.next());//{ value: undefined, done: true }
```

What is console.dir?

The output is presented as a hierarchical listing with disclosure triangles that let you see the contents of child objects. In other words, **console.dir** is the way to see all the properties of a specified JavaScript object in **console** by which the developer can easily get the properties of the object.

->Iterable only contains the symbol.iterator function.

->Object is not iterable.

Iterator is useful when we use with the combination of generators.

```
let myArray = [1,2,3,4,5];
let iterator = myArray[Symbol.iterator]();
 //this symbol.iterator is prototype of every iterable here Array
console.log(iterator.next())//{value: 1, done: false}
//this will work similarly Genarator
console.log(iterator.next())
```

```
console.log(iterator.next())
console.log(iterator.next())
console.log(iterator.next())
console.log(iterator.next())//{ value: undefined, done: true }
```

9. Functions:

```
var foo="foo";
function bob(){
    var foo = "foo2";
    console.log(foo);//foo2
}
bob();
console.log(foo);//foo
```

if you don't use var key word in second time then o/p will change

```
var foo="foo";
function bob(){
     foo = "foo2";
    console.log(foo);//foo2
}
bob();
console.log(foo);//foo2
```

Now, convert above into immediate invoked function expression

```
var foo="foo";
function bob(){
    var foo = "foo2";
    console.log(foo);//foo2
}
(bob)();
console.log(foo);
```

Now

```
var foo="foo";
(function bob(){
    var foo = "foo2";
    console.log(foo);//foo2
})();
console.log(foo);
```

4 foo

Now send parameters to IIFE

```
(function bob(name){
    console.log(name);//chandra
})("chandra");
```

In above actual parameter Chandra will send to formal parameter name
Now convert above function to arrow

```
((name)=>{
    console.log(name);//chandra
})("chandra");
```

10.

```
for(var i=0;i<3;i++){
    setTimeout(function(){
        console.log("i value is ",i)
    },1000*i)
}
```

In above for loop each and every iteration it will create functional scope.

Ex: i=0; f(c:i(0)) // in first iteration I scope create and the value will be 0.
I=1; f(c:i(1)// in second iteration I value becomes 1,means because of functional scope the previous iteration value also become 1. Means it will hit same scope. so "Var" will not create for different scope. It will update in same lexical scope. So, now the value will be:
Ist iteration: I:0 f(c:i(1)) here the closure value will be changed to 1.
2nd iteration: I:1 f(c:i(1)) simillarlly final value hit same closure value. So in all iterations the out put will be the latest value.

```
for(let i=0;i<3;i++){
    setTimeout(function(){
        console.log("i value is ",i)
    },1000*i)
}
```

If we use let then it will create separate scope for every iteration…i.e block scope will create. Because of block scope all iterations contains different values:

1st iteration i=0; f(c:i(o)) //here I value zero
2nd iteration i=1; f(c:i(1)) // here i:1 …similarly remaining iterations also will work

11.

```
function add(x) {
    return function(y) {
      if (y) {
        return add(x+y);
      }
      return x;
    };
  }
  console.log(add(4)(5)(10)());
```

12. spread operators and rest parameters

```
var numbers=[1,2,3];
//console.log(numbers);//[1,2,3]
//console.log(...numbers);//1 2 3
//var letters = [numbers,'a','b','c'];
//console.log(letters);//[ [ 1, 2, 3 ], 'a', 'b', 'c' ]
//console.log(letters.length);//4
//var letters1 = [...numbers,'a','b','c'];
//console.log(letters1);//[ 1, 2, 3, 'a', 'b', 'c' ]
//console.log(letters1.length);//6
//function add(a,b,c){
  //  return a+b+c;
//}
//console.log(add(numbers));//1,2,3undefinedundefined
//console.log(add(...numbers));//6
/*
var x= function(...n){
    console.log(n)//[ 1, 2, 3, 4 ]
}
var x= function(){
  console.log(arguments)//{ '0': 1, '1': 2, '2': 3, '3': 4 }  this is object to
convert to actual array then
  console.log(Array.prototype.slice.call(arguments,0));//[ 1, 2, 3, 4 ]
}
*/
var x = function(a,b,...n){
  console.log(n);//[3,4] here 3 and 4 are rest parameters assigned to array n
}
x(1,2,3,4);
```

13. Javascript Curring:

Currying is a technique of evaluating function with multiple arguments, into sequence of function with single argument. ... **Currying** helps you to avoid passing the same variable again and again. It helps to create a higher order function.

→ v, x, y, z length is 4

```
var curry = function(fn,...args){
    var fargs = fn.length;
    console.log("args.length",args.length);
    if(args.length === fargs){
        var result = fn(...args);
        console.log(result);
    }
    else{
        return function(...moreArgs){
            args = args.concat(moreArgs);
            console.log("moreArgs",moreArgs);
            return curry(fn,...args);
        }
    }
}

var adder = function(v,x,y,z){
    return v+x+y+z;
}
curry(adder)(1)(2)(3)(4);
```

↳ more arguments

```
//import or load lodash
var _ = require("lodash");
var abc = function(a, b, c) {
    return a + b + c;
  };
  var curried = _.curry(abc);

  console.log(curried(4)(5)(6));
  // => 15
```

Currying works by natural closure.The closure created by the nested functions to retain access to each of the arguments.So inner function have access to all arguments.
Note: We can achieve the same behavior using bind.The problem here is we have to alter this

```
var addBy2 = abc.bind(this,2);
  var addBy3 = addBy2.bind(this,3);
  console.log(addBy3(5));//8
```

14. Call Bing and apply:

Borrow function from one to other. So we can do multiple inheritance. We can borrow one object prototype to another. We can bind to this. Remember if you have function inside a function, this binding is very difficult.

Call:

```
let add = function(c){
    //here we dont have a & b value inside
    console.log(this.a + this.b + c);//6
}
var obj = {
    a:1,
    b:2
}
//here we call this add function with object
add.call(obj,3)//3 refers to c parameter to add method
```

```
let argsToArray = function(){
    console.log(arguments);//{ '0': 1, '1': 2, '2': 3 } I want to convert to
array
    console.log([].slice.call(arguments));//borrow slice method from array...here
slice is array prototype method so [ 1, 2, 3 ] will print
}
argsToArray(1,2,3);
```

Call super constructor in sub constructor:

```
//create one main construtor and call that in sub constructor
//mammal is main constructor
let mammal = function(legs){
    this.legs = legs;
}
//cat is sub constructor
let cat = function(legs, isDomesticated){
    mammal.call(this,legs);
    this.isDomesticated = isDomesticated;
}
```

```
let lion = new cat(4,false);
console.log(lion);//cat { legs: 4, isDomesticated: false }
```

Apply:

Apply is similar to call but it takes array of arguments.it will pass array of argumets into real arguments.

```
let numArray = [1,2,3];
console.log(Math.min(1,2,3));//1
//now i want to find minimum value in array...so we should convert into arguments
console.log(Math.min.apply(null,numArray));//1
```

Bind:

Bind is also like call and apply we can barrow functionalities from other libraries on your object temporarily but bind your object with that functionality and give you that functionality inside your object.

```
let myObj = {
    asyncGet(cb) {
        cb();
    },
    parse() {
        console.log("aprse called");
    },
    render() {
        /*  this.asyncGet(function(){
              this.parse();//TypeError: this.parse is not a function because this
function dont have parse method....the parse method will be avalibe in outer
scope...
         })
         */
        this.asyncGet(function () {
            this.parse();
        }.bind(this)); //here we are binding outer this scope to asyncGet
function so we can access
    }
};
myObj.render();
```

15. What is this keyword?

The **JavaScript** this **keyword** refers to the object it belongs to. ... In a function, this refers to the global object. In a function, in strict mode, this is undefined. In an event, this refers to the element that received the event. Methods like call(), and apply() can refer this to any object.

```
<script>
//this in global scope
//console.log(this);//Window {postMessage: f, blur: f, focus: f, close: f,
parent: Window, …}
this.table = "window table";
//console.log(window.table);//window table
this.garage = {
    table:"Garage table"
}
console.log(this.garage.table);//Garage table
console.log(window.garage.table);////Garage table
</script>
```

This inside an object:

```
<script>
    this.garage ={
        table:"garage table",
        cleanTable(){
            console.log(`cleaning ${this.table}`)
        }
    }
let johnsRoom = {
    table:"john table",
    cleanTable(){
        console.log(`clenaing ${this.table}`);
    }
};

//console.log(this.johnsRoom.table);//Uncaught TypeError: Cannot read property
'table' of undefined
//console.log(johnsRoom.table)//john table

this.garage.cleanTable();//cleaning garage table
johnsRoom.cleanTable();//clenaing john table
</script>
```

This inside a function:

```
<script>
   this.table = "window table";
   const cleanTable = function(){
       console.log(`cleaning ${this.table}`);//cleaning window table
   }
   cleanTable();
</script>
```

The above one will work without strict mode, i.e. by default every function can access global scope, if we work with strict mode then it won't access this scope, so we should apply some methods like call, apply and bind.

```
<script>
    'use strict';
   this.table = "window table";
   const cleanTable = function(){
       console.log(`cleaning ${this.table}`);//Uncaught TypeError: Cannot read
property 'table' of undefined
   }
   cleanTable();
</script>
```

Now the solution is

```
<script>
    'use strict';
   this.table = "window table";
   const cleanTable = function(){
       console.log(`cleaning ${this.table}`);//cleaning window table
   }
   cleanTable.call(this);
</script>
```

```
<script>
    'use strict';
   this.table = "window table";
   const cleanTable = function(soap){
       console.log(`cleaning ${this.table} using ${soap}`);//cleaning window
table using some soap
   }
```

```
    cleanTable.call(this,"some soap");
</script>
```

```
<script>
    'use strict';
    this.table = "window table";
    const cleanTable = function(soap){
        console.log(`cleaning ${this.table} using ${soap}`)
    }
    this.garage = {
        table:"garage table"
    };
    let johnsRoom = {
        table:"johns table"
    };
    cleanTable.call(this,"some soap");//cleaning window table using some soap
    cleanTable.call(this.garage,"some soap");//cleaning garage table using some
soap
    cleanTable.call(johnsRoom,'some soap');//cleaning johns table using some soap
</script>
```

This inside an inner function:

```
<script>
    'use strict';
    this.table = "window table";
    const cleanTable = function(soap){
        const innerFunction = function(_soap){
            console.log(`cleaning ${this.table} using ${_soap}`)//Cannot read
property 'table' of undefined
        }
        innerFunction(soap)
    }

    cleanTable.call(this,"some soap");
</script>
```

To solve above we have to

1.

```
innerFunction.call(this,soap)//cleaning window table using some soap
```

2.

```
innerFunction.bind(this)(soap);
```

3.

```
<script>
    'use strict';
    this.table = "window table";
    const cleanTable = function(soap){
        const that =this;
        const innerFunction = function(_soap){
            console.log(`cleaning ${that.table} using ${_soap}`)//Cannot read
property 'table' of undefined
        }
        innerFunction(soap)//cleaning window table using some soap
    }

    cleanTable.call(this,"some soap");
</script>
```

4.

```
<script>
    'use strict';
    this.table = "window table";
    const cleanTable = function(soap){
        const innerFunction = (_soap)=>{
            //arrow function will take this scope from outer function, so this
will work
            console.log(`cleaning ${this.table} using ${_soap}`);
        }
        innerFunction(soap);
    }

    cleanTable.call(this,"some soap");
</script>
```

This inside a constructor:

```
<script>
    'use strict';
   let createRoom = function(name){
       this.table = `${name}s table`;
   }
    createRoom.prototype.cleanRoom = function(soap){
        console.log(`cleaning ${this.table} using ${soap}`)
    }

let chandraRoom = new createRoom("chandra");
let pothiRoom = new createRoom("pothi");

chandraRoom.cleanRoom("some soap");
pothiRoom.cleanRoom("some soap");

</script>
```

This inside a class:

```
<script>
    'use strict';
    class createRoom {
        constructor(name) {
            this.table = `${name}s table`;
        }
        cleanRoom(soap) {
            console.log(`cleaning ${this.table} using ${soap}`)
        }
    }
    let chandraRoom = new createRoom("chandra");
    let pothiRoom = new createRoom("pothi");

    chandraRoom.cleanRoom("some soap");
    pothiRoom.cleanRoom("some soap");

</script>
```

Set:

A **set** is a collection of items which are unique i.e no element can be repeated.**Set** in ES6 are ordered : elements of the **set** can be iterated in the insertion order. **Set**can store any types of values whether primitive or objects.

```
let myArr = [1,2,3,3,true];
let mySet = new Set(myArr);
mySet.add('tutorial');
mySet.add({name:"Jscript"});
mySet.delete(1);
console.log(mySet);//Set { 2, 3, true, 'tutorial', { name: 'Jscript' } }
mySet.clear();
console.log(mySet);//Set {}
mySet.add('tutorial');
mySet.add({name:"Jscript"});
console.log(mySet);//Set { 'tutorial', { name: 'Jscript' } } we can add any type
of data in set
console.log(mySet.size);//2
//console.log("entry",mySet.keys(0));
mySet.forEach(function(value){
  //  console.log(value);
})
var getEntriesArry = mySet.entries();

// each iterator is array of [value, value]
// prints [50, 50]
console.log(getEntriesArry.next().value);//[ 'tutorial', 'tutorial' ]
```

WeakSet:

WeakSets are **collections of objects only** and not of arbitrary values of any type.

WeakSets are not iterable objects.

The WeakSet is *weak*: References to objects in the collection are held weakly. If there is no other reference to an object stored in the WeakSet, they can be garbage collected. That also means that there is no list of current objects stored in the collection. WeakSetsare not enumerable.

```
const ws = new WeakSet([{'a':1}]);
console.log(ws);
```

Map:

Map in JavaScript. ... **Map** is a collection of elements where each element is stored as a Key, value pair.**Map** object can hold both objects and primitive values as either key or

value. When we iterate over themap object it returns the key,value pair in the same order as inserted.

Map is a collection of keyed data items, just like an Object. But the main difference is that Map allows keys of any type.

```
//map.js
const x ={};
const a={};
x[a] = 'a';
console.log(x);//{ '[object Object]': 'a' }
console.log(x[a]);//a
```

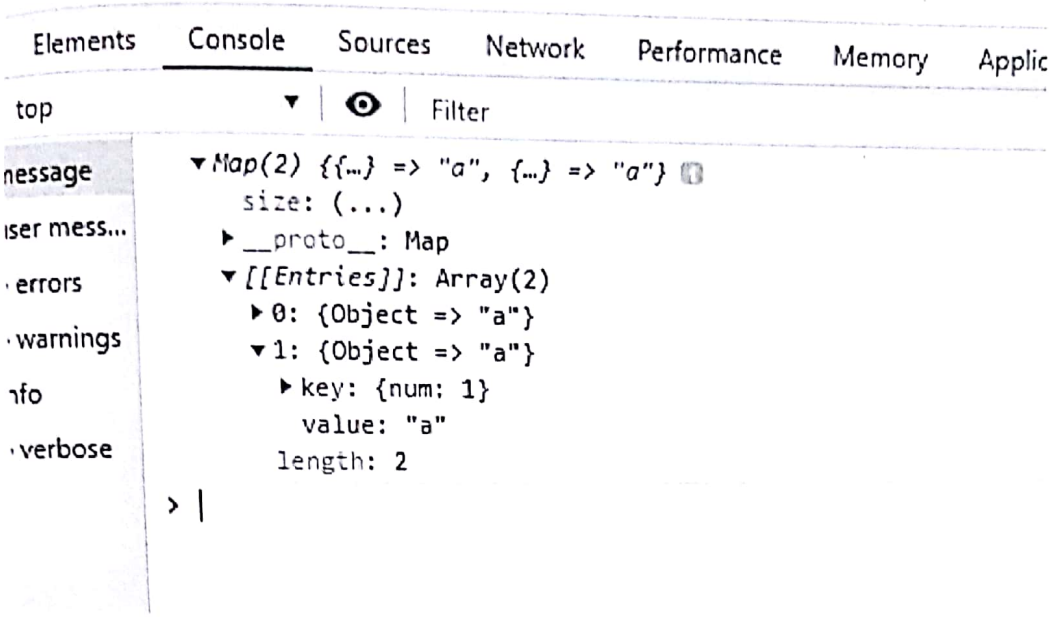
In above, we can access object inside object. But If we have multiple chain object then

```
<script>
    const x ={};
const a={};
const b={num:1}
x[a] = 'a';
x[b] = 'b';
console.log(x);//Object
               //[object Object]: "b"
               //__proto__: Object
//so we can not access b num

</script>
```

So, the maps introduced.

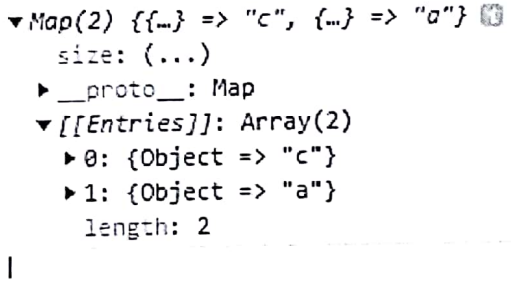
```
<script>
const a={};
const b={num:1}
const myMap = new Map();
myMap.set(a,'a');
myMap.set(b,'a');
console.log(myMap);

</script>
```

Now out put will be:



If I want to add same key with different value,just it will override the value.

```
myMap.set(a,'a').set(b,'a').set(a,'c')
console.log(myMap);
```



So ,map stors only unique values.

Maps are iterables.for off will work…

For off will not work for objects

```
//map.js
let myMap = new Map([['a1',"chandra"],["a2","babu"]]);
myMap.set('a3',"pothi");
console.log(myMap);//Map { 'a1' => 'chandra', 'a2' => 'babu', 'a3' => 'pothi' }
console.log(myMap.get('a1'));//chandra
```

WeakMap:

The **WeakMap** object is a collection of key/value pairs in which the keys are weakly referenced. The keys must be objects and the values can be arbitrary values.

```
<script>
let x={
    a:[1,2]
};
var map = new Map();
map.set(x,'something');
console.log(map);
//let is block scope. it should not have access outside. but here we can acess
and it is not garbage collected
//garbage collected means it should remove once reference removed.
//so weakmap introduced
</script>
```

```
let x={
    a:[1,2]
};
var weakMap = new WeakMap();
weakMap.set(x,'something');

console.log(weakMap);
```

Cookie Jar:

cookies are managed by browsers (HTTP clients) and they allow to store information on the clients' computers which are sent automatically by the browser on subsequent requests.

If your application acts as a client (you connect to remote HTTP servers using the net/httppackage), then there is no browser which would handle / manage the cookies. By this I mean storing/remembering cookies that arrive as Set-Cookie: response headers, and attaching them to subsequent outgoing requests being made to the same host/domain. Also cookies have expiration date which you would also have to check before deciding to include them in outgoing requests.
The http.Client type however allows you to set a value of type http.CookieJar, and if you do so, you will have automatic cookie management which otherwise would not exist or you would have to do it yourself. This enables you to do multiple requests with the net/http package that the server will see as part of the same session just as if they were

made by a real browser, as often HTTP sessions (the session ids) are maintained using cookies.
The package net/http/cookiejar is a CookieJar implementation which you can use out of the box. Note that this implementation is in-memory only which means if you restart your application, the cookies will be lost.
So basically an **HTTP cookie** is a small piece of data sent from a website and stored in a user's web browser while the user is browsing that website.

Cookiejar is a Go interface of a simple cookie manager (to manage cookies from HTTP request and response headers) and an implementation of that interface.

What is REPL ?

Node.js comes with virtual environment called REPL (aka Node shell). REPL stands for Read-Eval-Print-Loop. It is a quick and easy way to test simple Node.js/JavaScript code.

To launch the REPL (Node shell), open command prompt (in Windows) or terminal (in Mac or UNIX/Linux) and type *node* as shown below. It will change the prompt to > in Windows and MAC.

What is Traceur compiler?

Traceur is a JavaScript.next-to-JavaScript-of-today compiler that allows you to use features from the future **today**. Traceur supports ES6 as well as some experimental ES.next features.

Traceur's goal is to inform the design of new JavaScript features which are only valuable if they allow you to write better code. Traceur allows you to try out new and proposed language features today, helping you say what you mean in your code while informing the standards process.

JavaScript's evolution needs your input. Try out the new language features. Tell us how they work for you and what's still causing you to use more boilerplate and "design patterns" than you prefer.

Angular Compilation:

An Angular application consists mainly of components and their HTML templates. Because the components and templates provided by Angular cannot be understood by the browser directly, Angular applications require a compilation process before they can run in a browser.

The Angular Ahead-of-Time (AOT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase *before* the browser downloads and runs that code. Compiling your application during the build process provides a faster rendering in the browser.

Angular offers two ways to compile your application:

Just-in-Time (JIT), which compiles your app in the browser at runtime.

Ahead-of-Time (AOT), which compiles your app at build time.

JIT compilation is the default when you run the ng build (build only) or ng serve (build and serve locally) CLI commands:

ng build

ng serve

For AOT compilation, include the --aot option with the ng build or ng serve command:

ng build --aot

ng serve --aot

The ng build command with the --prod meta-flag (ng build --prod) compiles with AOT by default.

See the CLI command reference and Building and serving Angular apps for more information.

Why compile with AOT?

Faster rendering

With AOT, the browser downloads a pre-compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the app first.

Fewer asynchronous requests

The compiler *inlines* external HTML templates and CSS style sheets within the application JavaScript, eliminating separate ajax requests for those source files.

Smaller Angular framework download size

There's no need to download the Angular compiler if the app is already compiled. The compiler is roughly half of Angular itself, so omitting it dramatically reduces the application payload.

Detect template errors earlier

The AOT compiler detects and reports template binding errors during the build step before users can see them.

Better security

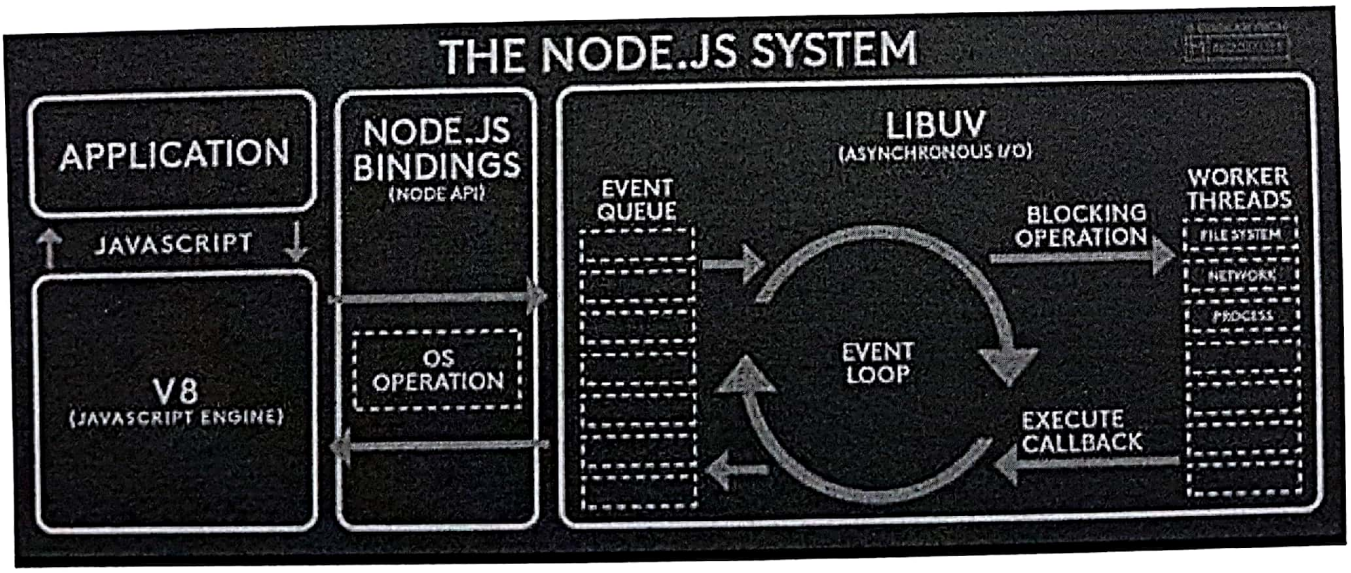
AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

Mongoose virtual:

Virtuals are additional fields for a given model. Their values can be set manually or automatically with defined functionality. A common virtual property is the full name of a person, composed of user's first and last name.

Keep in mind: virtual properties don't get persisted in the database. They only exist logically and are not written to the document's collection.

Libuv:



Q:What is object destructuring?

Destructuring assignment. The two most used data structures in JavaScript are **Object** and Array . ...**Destructuring** assignment is a special syntax that allows us to "unpack" arrays or **objects** into a bunch of variables, as sometimes they are more convenient.

```
var a, b, rest;
[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a); // 10
console.log(b); // 20
console.log(rest); // [30, 40, 50]
```

Q:What is the difference between spread operator and rest parameter?

Both "rest operator" and "spread operator" refer to the same operator (…), used differently. When you see "rest", it's being used to gather up properties. When you see "spread", it's spreading them out.

Spread Operator:

The spread operator will spread all enumerable, own properties of one object to another object.

```
const obj1 = {a:1,b:2};
const obj2={...obj1,c:3,d:4};
console.log(obj2);//{ a: 1, b: 2, c: 3, d: 4 }

//A good use case is creating a shallow copy of an object.
const obj1Copy = {...obj1};
console.log(obj1Copy);//{ a: 1, b: 2 }

//Another use case is merging objects together.
const obj3 = {name:"chandra",address:"AP"};
const merge = {...obj1,...obj2,...obj3};
console.log(merge);//{ a: 1, b: 2, c: 3, d: 4, name: 'chandra', address: 'AP' }
```

Clashing properties

When duplicate properties clash, the order determines the outcome. The property put in last wins. If we spread the object last:

```
const obj1 = { a: 'abc', b: 'def' };
const obj2 = { b: 123, c: 456, ...obj1};
console.log(obj2);//{ b: 'def', c: 456, a: 'abc' }
```

If we spread the object first:

```
const obj1 = { a: 'abc', b: 'def' };
const obj2 = { ...obj1,b: 123, c: 456};
console.log(obj2);//{ a: 'abc', b: 123, c: 456 }
```

Rest Operators:

Rest operator means rest of the parameters.

```
var a, b, rest;
[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a); // 10
console.log(b); // 20
console.log(rest); // [30, 40, 50]
```

What is the difference between Object. assign and spread operator?

1. An alternative approach is to use the object **spread** syntax recently added to the JavaScript specification. It lets you use the **spread** (...) **operator** to copy enumerable properties from one object to another in a more succinct way. The object **spread operator** is conceptually similar to the **ES6** array **spread operator**.

2. Object.assign will create deep copy (clone) but spread create shallow copy.

Diffrence between Object and Map:

Object and **Map** are based on the *same* concept — using key-value for storing data. However, like we always say — *same same but different* — they are indeed quite different from each other, mainly in:

- *Key field*: in **Object**, it *follows the rule of normal dictionary*. The keys **MUST be simple types** — either *integer* or *string or symbols*. Nothing more. But in **Map** it can be **any data type** (an object, an array, etc…). (*try using another object as Object's property key — i dare you :)*)
- *Element order:* in **Map**, original **order of elements (pairs) is preserved**, while in **Object**, it **isn't**.
- *Inheritance*: **Map** is an instance of Object (*surprise surprise!*), but **Object** is definitely **not** an instance of Map.