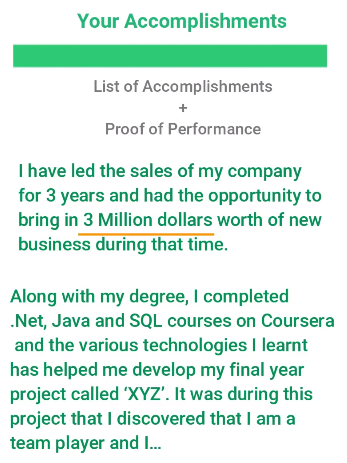
# Interview Question: Tell Me About Yourself

#1 Experience summary, education …



**#2 Your accomplishment.**



**#3 Why you want this job & how you are fit for this job ?**

**“this” keyword**

1. The this keyword allows you to reuse functions with different contexts. Said differently, **the “this” keyword allows you to decide which object should be focal when invoking a function or a method.**
2. Every function in JavaScript has a prototype property that references an object.

var VS let VS const

If you declare a variable withour ‘var’, let or const keyword, then JS engine is going to hoist it to top. Since does not have any keyword, JS engine will take it to the Top i.e., global scope.

That is why it will get attached with global scope.

Therefore it is recommended to never use a variable without var/let/const.

**var:**

function scoped

undefined when accessing a variable before it's declared

can be reassigned

**let:**

block scoped

ReferenceError when accessing a variable before it's declared

can be reassigned

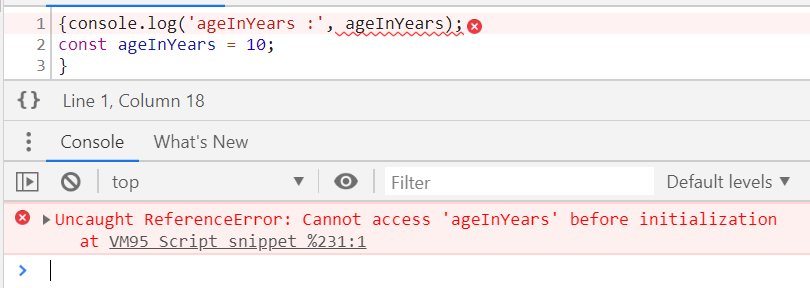
**const:**

block scoped [anything inside { } ]

ReferenceError when accessing a variable before it's declared

can't be reassigned

If we try to access a variable before it is declared with const keyword.



Execution Context, Scoping and Hoisting

Just like functions, modules or packages allows you to manage the complexity of writing code, Execution context allows JavaScript Engine to manage the complexity of interpreting the code.

Global Execution context: It is the first execution context gets created when JS engine runs your code.

**Rules**

#1. **Function declarations are hoisted** **before any statement** is hoisted or executed by JS Engine, which means if we write:

function getName() { return ‘Hennry’; }

then this function name with function body is going to be hoisted.

#2. **Variable declarations are hoisted, after function declaration**, which means if you write

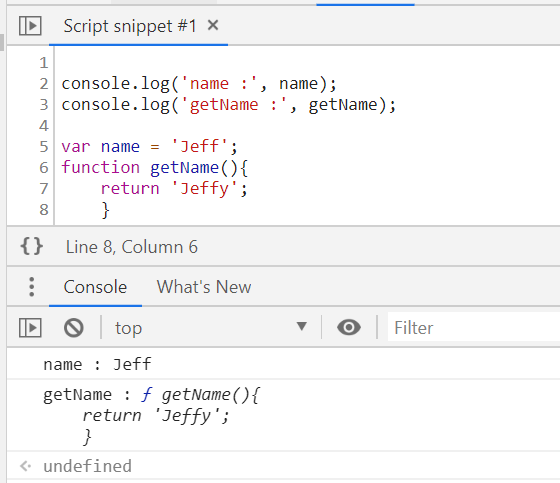
var a = 10;

Then, it is going to be interpreted as below in lexing phase,

var a; // Value of ‘a’ is Undefined

Below is interpreted during execution phase, when the interpreter reaches that particular line of code.

a =10; // Value of ‘a’ is 10



Scope Chain

It is a chain of nested scopes. It is created when we define a function or block of code inside another function or block of code.

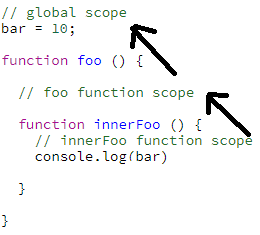
To Determine the Scope Chain, we need to look at the code first.

At top: there is global scope, which is window object in the browse.

Then there are functions which have their own scopes.

# Resolving Scope:

When resolving a variable, inner functions first look at their own scope. If the variable cannot be found in its own scope it will **climb up the scope chain** and looks for the variable name in the environment where the function was defined. This look like this:



Slice()

The **slice()** method returns a shallow copy of a portion of an array into a new array object selected from begin to end (end not included) where begin and end represent the index of items in that array. It returns a new array containing the extracted elements. The original array will not be modified.

var animals = ['ant', 'bison', 'camel', 'duck', 'elephant'];

console.log(animals.slice(2));

// expected output: Array ["camel", "duck", "elephant"]

console.log(animals.slice(2, 4));

// expected output: Array ["camel", "duck"]

console.log(animals.slice(1, 5));

// expected output: Array ["bison", "camel", "duck", "elephant"]

[] vs new Arrar()

### **Array Methods**

We talked in depth above about how if you want to share methods across instances of a class, you should stick those methods on the class’ (or function’s) prototype. We can see this same pattern demonstrated if we look at the Array class. Historically you’ve probably created your arrays like this

const friends = []

Turns out that’s just sugar over creating a new instance of the Array class.

const friendsWithSugar = []

const friendsWithoutSugar = new Array()

**One thing you might have never thought about is how does every instance of an array have all of those built in methods (splice, slice, pop, etc)?**

* Well as you now know, it’s because those methods live on Array.prototype and when you create a new instance of Array, you use the new keyword which sets up that delegation to Array.prototype on failed lookups.

We can see all the array’s methods by simply logging Array.prototype.

console.log(Array.prototype)

/\*

concat: ƒn concat()

constructor: ƒn Array()

copyWithin: ƒn copyWithin()

entries: ƒn entries()

every: ƒn every()

fill: ƒn fill()

filter: ƒn filter()

find: ƒn find()

findIndex: ƒn findIndex()

forEach: ƒn forEach()

includes: ƒn includes()

indexOf: ƒn indexOf()

join: ƒn join()

keys: ƒn keys()

lastIndexOf: ƒn lastIndexOf()

length: 0n

map: ƒn map()

pop: ƒn pop()

push: ƒn push()

reduce: ƒn reduce()

reduceRight: ƒn reduceRight()

reverse: ƒn reverse()

shift: ƒn shift()

slice: ƒn slice()

some: ƒn some()

sort: ƒn sort()

splice: ƒn splice()

toLocaleString: ƒn toLocaleString()

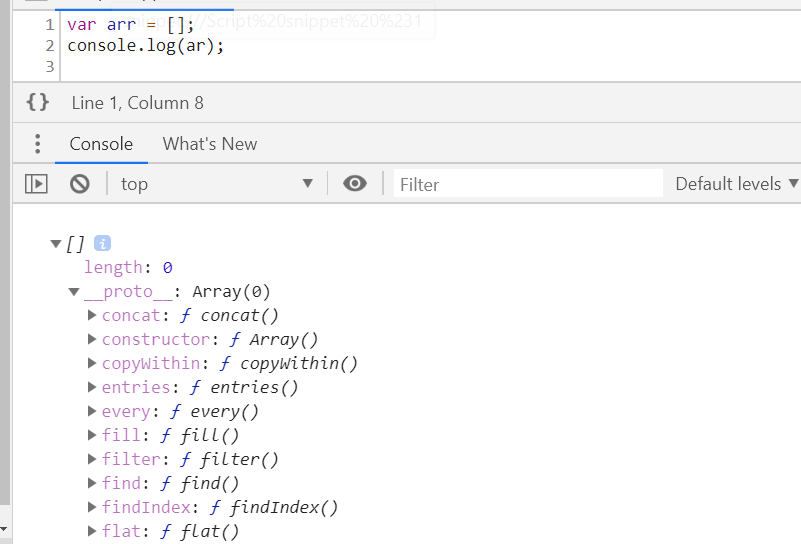
toString: ƒn toString()

unshift: ƒn unshift()

values: ƒn values()

\*/

The exact same logic exists for Objects as well. All object will delegate to Object.prototype on failed lookups which is why all objects have methods like toString and hasOwnProperty.



Sorting an Array

The sort() method sorts an array alphabetically:

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort();        // Sorts the elements of fruits

## **Reversing an Array**

The reverse() method reverses the elements in an array.

You can use it to sort an array in descending order:

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort();        // First sort the elements of fruits  
fruits.reverse();     // Then reverse the order of the elements

Numeric Sort

By default, the sort() function sorts values as **strings**.

However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".

To fix this, we use Callback function passed as argument to sort() function.

var points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return b - a});

1. If the result is negative a is sorted before b.
2. If the result is positive b is sorted before a.
3. If the result is 0 no changes are done with the sort order of the two values.

**Example:**

1. The compare function compares all the values in the array, two values at a time (a, b).
2. When comparing 40 and 100, the sort() method calls the compare function(40, 100).
3. The function calculates 40 - 100 (a - b), and since the result is negative (-60),  the sort function will sort 40 as a value lower than 100.

Array Methods:

1. Push()

method adds a new element to an array (at the end)

Returns the new array length

1. Pop()

This method removes the last array element

Returns the value that was "popped out"

1. Shift() :

this method removes the first array element

Returns the string that was "shifted out"

1. Unshift() :

add an element at beginning of the array

Returns the new array length.

1. Splice()

method can be used to remove / add new items to an array:

Reurns an array with the deleted items

The first parameter: position where new elements should be **added** (spliced in).

The second parameter: **how many** elements should be **removed**.

The rest of the parameters are omitted. No new elements will be added.

var fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(0, 1);        // Removes the first element of fruits

1. Delete

delete fruits[0]; // Changes the first element in fruits to **undefined**

1. Slice()

It creates a new array. It does not remove any elements from the source array.

var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
var citrus = fruits.slice(1, 3); // Orange,Lemon

var lastThree = fruits.slice(1); // Orange,Lemon,Apple,Mango

1. Concat()

The concat() method does not change the existing arrays. It always returns a new array.

The concat() method can take any number of array arguments.

Argument Object

The arguments object is a local variable available within all **non-**[**arrow**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) functions. You can refer to a function's arguments inside that function by using its arguments object. It has entries for each argument the function was called with, with the first entry's index at 0.

For example, if a function is passed 3 arguments, you can access them as follows:

arguments[0] // first argument

arguments[1] // second argument

arguments[2] // third argument

Each argument can also be set or reassigned:

arguments[1] = 'new value';

The arguments object is not an [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array). It is similar, but does not have any Array properties except [length](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/length). For example, it does not have the [pop()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/pop) method. However, it can be converted to a real Array:

var args = Array.prototype.slice.call(arguments);

// Using an array literal is shorter than above but allocates an empty array

var args = [].slice.call(arguments);

As you can do with any Array-like object, you can use ES2015's [Array.from()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from) method or [spread syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax) to convert arguments to a real Array:

var args = Array.from(arguments);

var args = [...arguments];

The [typeof](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof) operator returns 'object' when used with arguments

console.log(typeof arguments); // 'object'

The type of individual arguments can be determined by indexing arguments:

console.log(typeof arguments[0]); // returns the type of the first argument

Argument without rest, default or destructure

If arguments DOESN’T contain – rest, default or destructured arguments, then the values in the arguments object **do** change in sync with the values of the argument variables.

function func(a) {

arguments[0] = 99; // updating arguments[0] also updates a

console.log(a);

}

func(10); // 99

function func(a) {

a = 99; // updating a also updates arguments[0]

console.log(arguments[0]);

}

func(10); // 99

Argument with rest, default or destructure

If arguments DOES contain – rest, default or destructured arguments, then the values in the arguments object **do** change in sync with the values of the argument variables.

function func(a = 55) {

arguments[0] = 99; // updating arguments[0] does not also update a

console.log(a);

}

func(10); // 10

// An untracked default parameter

function func(a = 55) {

console.log(arguments[0]);

}

func(); // undefined

var arr1 = ["Cecilie", "Lone"];  
var arr2 = ["Emil", "Tobias", "Linus"];  
var arr3 = ["Robin", "Morgan"];  
var myChildren = arr1.concat(arr2, arr3);

1. toString()

converts an array to a comma separated string

All JavaScript objects have a toString() method.

var fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.toString(); // Banana,Orange,Apple,Mango

for..of vs for..in

Both for..of and for..in statements iterate over lists; the values iterated on are different though, for..in returns a list of keys on the object being iterated, whereas for..of returns a list of values of the numeric properties of the object being iterated.

Here is an example that demonstrates this distinction:

Example 1:

let list = [4, 5, 6];

for (let i in list) { // I contains property names

console.log(i); // "0", "1", "2",

}

for (let i of list) { I contains property values

console.log(i); // "4", "5", "6"

}

Example 2:

Object.prototype.objCustom = function() {};

Array.prototype.arrCustom = function() {};

let iterable = [3, 5, 7];

for (let i in iterable) {

console.log(i); // logs: 0, 1, 2, "arrCustom", "objCustom"

}

for (let i in iterable) {

if (iterable.hasOwnProperty(i)) {

console.log(i); // logs: 0, 1, 2,

}

}

for (let i of iterable) {

console.log(i); // logs: 3, 5, 7

}

Value, denumerable : they are property descriptor.

Arrays : are iterable

Objects: not iterable by default => that is why for …of fails on it

An iterator is built in object in several data types like:

Arrays, String, Maps, Sets, NodeLists have built-in iterator object

**Index.html**

Originally it served the purpose of indexing the files in a webserver directory. Now it is the default file the server looks for when in a directory. This and the order of default files to look for can be customised in the server settings.

When opening your site's root directory in a browser, index.html will be opened by default.

If you have no index.html page, you will get a directory listing of all of your files on that server (live or localhost), acting just like a file explorer on your local machine.

Responsive Web Design

<head>

<meta name="viewport" content="width= device -width, initial-scale=1.0">

</head>

Viewport means we are specifying about the area of web page in which the content is visible to the user.

This means that the browser will (probably) render the width of the page at the width of its own screen. So if that screen is 320px wide, the browser window will be 320px wide, rather than way zoomed out and showing 960px (or whatever that device does by default, in lieu of a responsive meta tag).

**Note:** [don't](http://blog.javierusobiaga.com/stop-using-the-viewport-tag-until-you-know-ho) use a responsive meta tag if your website isn't specifically designed to be responsive and work well at that size, as it will make the experience worse.

There are [more attributes](http://www.inmotionhosting.com/support/edu/website-design/create-responsive-template/viewpoint-meta-tag-website-responsive-template) this tag supports:

**content="width=device-width:** You tell the mobile, tablet or desktop that the contend width needs to be equal to device width and,

**initial-scale=1.0:** You are not trying to scale the screen up or down to fit your web page on screen.

**Note:** The meta tag should be added in the head tag in HTML document.

A Responsive tags has the following attributed:

* **width**: Width of the virtual viewport of the device.
* **height**: Height of the virtual viewport of the device.
* **initial-scale**: Zoom level when the page is first visited.
* **minimum-scale**: Minimum zoom level to which a user can zoom the page.
* **maximum-scale**: Maximum zoom level to which a user can zoom the page.
* **user-scalable**: Flag which allows the device to zoom in or out.(value= yes/no).

How to make image responsive?

Max-width: 100%

Height: auto;

Media Queries

Media Queries are a new technique introduced in CSS3 that change the presentation of content based on different viewport sizes.

We use a handful of [media queries](https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries) to create sensible breakpoints for our layouts

Media Queries consist of a media type, and if that media type matches the type of device the document is displayed on, the styles are applied.

The most commonly used query is max-width. Any width that is less than the max-width specified, all of the CSS within the query will take effect.

@media [<media-query-list>](https://developer.mozilla.org/en-US/docs/Web/CSS/@media#media-query-list) {

<group-rule-body>

}

Main.css---------------------------------------------------------------------------------------------------------------

#container {

width: 960px;

margin: 0 auto;

}

#left-column{

width: 700px;

float: left;

background: orange;

}

#right-column{

width: 260px;

float: left;

background: yellow;

}

/\*To make conatiner and left and right column to adjust their size when screen width < 960px\*/

@media screen and (max-width: 959px){

#container {

width: 100%;

}

#left-column{

width: 70%;

}

#right-column{

width: 30%;

}

img {

width: 100%;

}

}

/\*To make left and right column to stack over each other when screen width < 640px\*/

@media screen and (max-width:640px){

#left-column{

width: 100%;

}

#right-column{

width: 100%;

}

}

/\*To make our container to not squeeze further when screen width < 320px\*/

@media screen and (max-width:320px){

#container {

width: 320px;

}}

Media.html---------------------------------------------------------------------------------------------------------

<!DOCTYPE html>

<html>

<head>

<title> Learning </title>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<link rel="stylesheet" href="main.css"/>

</head>

<body>

<div id="container">

<img src="banner.png"/>

<section id="left-column">

<p> This page is reponsive and this is the left column in this web page. This page is reponsive and this is the left column in this web page</p>

</section>

<aside id="right-column">

<p> This page is reponsive and this is the left column in this web page. This page is reponsive and this is the left column in this web page</p>

</aside>

</body>

</html>

----------------------------------END--------------------------------

Margin

/\* Apply to all four sides \*/

margin: 1em;

margin: -3px;

/\* vertical | horizontal \*/

margin: 5% auto;

/\* top | horizontal | bottom \*/

margin: 1em auto 2em;

/\* top | right | bottom | left \*/

margin: 2px 1em 0 auto;

/\* Global values \*/

margin: inherit;

margin: initial;

margin: unset;

[CSS: Width and Max-Width](https://stackoverflow.com/questions/6456468/css-width-and-max-width)

From my understanding of the properties:

if width > **max-width** use max-width

if max-width > **width** use width

If the width goes above max-width, keep it at max-width. If the width is below max-width, keep it on width.

It will never go over the max-width, that doesn't mean it can't stay under, the max keyword obviously indicates a limit and not a rule.

**Use Case:**

You might want to limit the width of a modal, right? Kinda gives it that "modal" look on larger screens. Let's say 600px sounds right. But, you want to make sure it doesn't bust outside of its parent element. For example, avoid causing horizontal scrolling on a mobile screen.

How would you do that?

1. width: 600px; max-width: 100%;
2. width: 100%; max-width: 600px;

This might be a useful way to think about it: **if the element would render wider than what max-width says it can be, max-width wins.**

Scenarios:

* **Parent is 1000px wide**
  + Width says element should be 600px wide. That doesn't break the max-width rule, so **600px** it is!
  + Width says element should be 1000px wide. That breaks max-width rule, so forces element down to **600px**.
* **Parent is 320px wide**
  + Width says element should be 600px wide. That breaks the max-width rule which says element can only be at most 320px wide, so **320px** it is!
  + Width says element should be 320px wide. That doesn't break the max-width rule, so **320px** it is!

Whether the parent element is wider or narrower, these different rulesets end up agreeing with each other.

[What does !important mean in CSS?](https://stackoverflow.com/questions/9245353/what-does-important-mean-in-css)

A rule that has the !important property will always be applied no matter where that rule appears in the CSS document.

So, if you have the following:

.class {

color:red !important;

}

.outerClass .class {

color:blue;

}

The rule with the important will be the one applied. !important appeared in CSS1 so every browser supports it (IE4 to IE6 with a partial implementation, IE7+ full).

box-sizing

The box-sizing property can be used to adjust this behavior:

1. **content-box (**default)

It gives you the default CSS box-sizing behavior. If you set an element's width to 100 pixels, then the element's content box will be 100 pixels wide, and the width of any border or padding will be added to the final rendered width, making the element wider than 100px.

For example,

.box {

width: 350px;

border: 10px solid black;

}

renders a box that is 370px wide.

width = width of the content*, and*

height = height of the content*.*

1. **border-box**

It tells the browser to account for any border and padding in the values you specify for an element's width and height. If you set an element's width to 100 pixels, that 100 pixels will include any border or padding you added, and the content box will shrink to absorb that extra width. This typically makes it much easier to size elements.

For example,

.box {

width: 350px;

border: 10px solid black;

}

 renders a box that is 350px wide, with the area for content being 330px wide.

width = border + padding + width of the content*, and*

height = border + padding + height of the content*.*

## **Example**

This example shows how different box-sizing values alter the rendered size of two otherwise identical elements.

**HTML**

<div class="content-box">Content box</div>

<br>

<div class="border-box">Border box</div>

**CSS**

div {

width: 160px;

height: 80px;

padding: 20px;

border: 8px solid red;

background: yellow;

}

.content-box {

box-sizing: content-box;

/\* Total width: 160px + (2 \* 20px) + (2 \* 8px) = 216px

Total height: 80px + (2 \* 20px) + (2 \* 8px) = 136px

Content box width: 160px

Content box height: 80px \*/

}

.border-box {

box-sizing: border-box;

/\* Total width: 160px

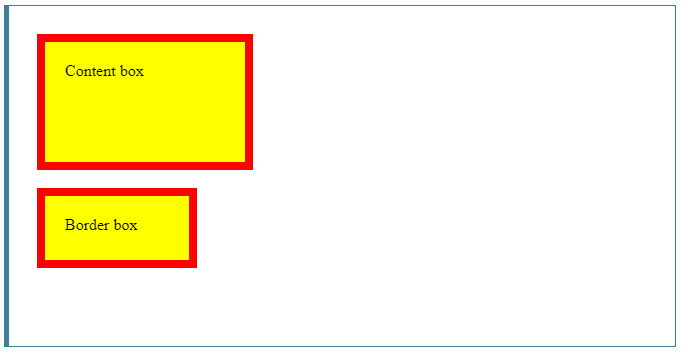
Total height: 80px

Content box width: 160px - (2 \* 20px) - (2 \* 8px) = 104px

Content box height: 80px - (2 \* 20px) - (2 \* 8px) = 24px \*/

}

**Result**



## **CSS Specificity**

If there are two or more conflicting CSS rules that point to the same element, the browser follows some rules to determine which one is most specific and therefore wins out.

The universal selector (\*) has low specificity, while ID selectors are highly specific!

The following list of selector types increases by specificity:

1. [Type selectors](https://developer.mozilla.org/en-US/docs/Web/CSS/Type_selectors) (e.g., h1) and pseudo-elements (e.g., ::before).
2. [Class selectors](https://developer.mozilla.org/en-US/docs/Web/CSS/Class_selectors) (e.g., .example), attributes selectors (e.g., [type="radio"]) and pseudo-classes (e.g., :hover).
3. [ID selectors](https://developer.mozilla.org/en-US/docs/Web/CSS/ID_selectors) (e.g., #example). **(highest specificity)**

When an important rule is used on a style declaration, this declaration overrides any other declarations.

Using !important, however, is **bad practice** and should be avoided because it makes debugging more difficult by breaking the natural [cascading](https://developer.mozilla.org/en-US/docs/Web/CSS/Cascade) in your stylesheets.

How to center a box inside another box?

Index.html-------------------------------------------------------------------------------------------------------

<!DOCTYPE html>

<html>

<head>

<title> Learning </title>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<link rel="stylesheet" href="main.css"/>

</head>

<body>

<div class="out">

<div class="inner">

</div>

</div>

</body>

</html>

Style.css-------------------------------------------------------------------------------------------------------

.out {

width: 300px;

height: 300px;

background: yellow;

position: relative;

margin: auto; /\* Brings the outer box horizontally center of screen \*/

}

.inner {

width: 100px;

height: 100px;

background: red;

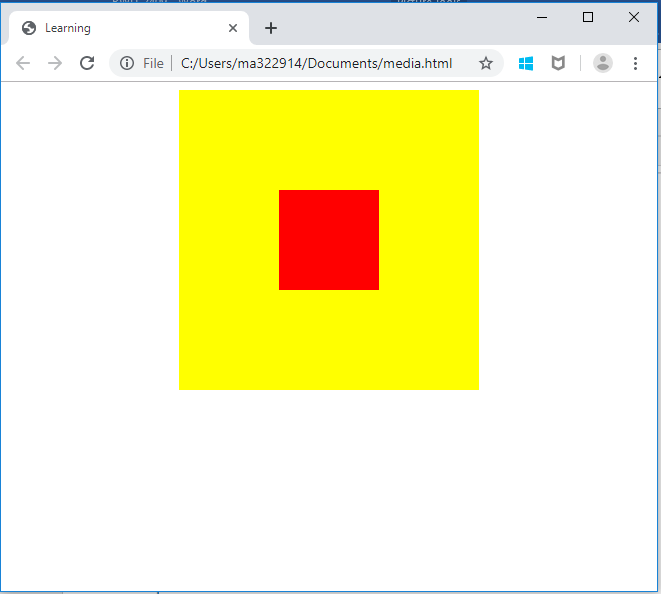
position: absolute;

top: 50%;

left: 50%;

transform: translate(-50%, -50%);

}



CSS | translate() Function

The **translate() function** is an inbuilt function which is used to reposition an element in a horizontal and vertical direction.

**Single**[**<length-percentage>**](https://developer.mozilla.org/en-US/docs/Web/CSS/length-percentage)**value**

A [<length>](https://developer.mozilla.org/en-US/docs/Web/CSS/length) or [<percentage>](https://developer.mozilla.org/en-US/docs/Web/CSS/percentage) that specifies a 2D translation, with the same translation along both the X and Y axes.

**Two**[**<length-percentage>**](https://developer.mozilla.org/en-US/docs/Web/CSS/length-percentage)**values**

Two [<length>](https://developer.mozilla.org/en-US/docs/Web/CSS/length) or [<percentage>](https://developer.mozilla.org/en-US/docs/Web/CSS/percentage) that specify the X and Y axis translation values (respectively) of a 2D translation.

**Three values**

Two [<length-percentage>](https://developer.mozilla.org/en-US/docs/Web/CSS/length-percentage) and single [<length>](https://developer.mozilla.org/en-US/docs/Web/CSS/length) values that specify the X, Y, and Z axis translation values (respectively) of a 3D translation.

**none**

Specifies that no translation should be applied.

Pseudo Class and Elements

: pseudo- class [ :hover ]

: : pseudo element [ : : after, : :before ] //after means, after existing content

To write tooltip:

**HTML ---------------------------------------------**

<p> Hello </p>

**CSS ---------------------------------------------**

p:hover::after {

content: "Tooltip";

font-size: 10px;

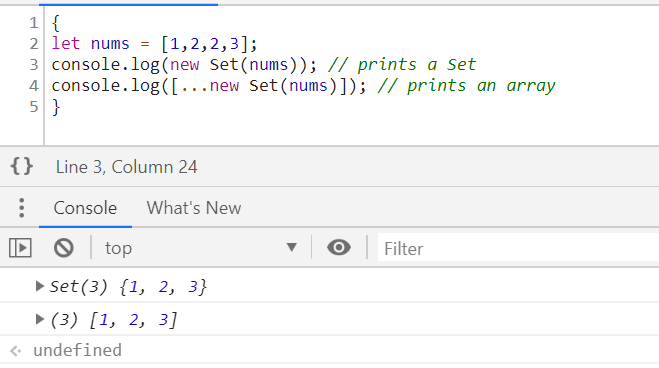
border: 2px white dotted;

}

# Tricky JavaScript Interview Questions and Answers

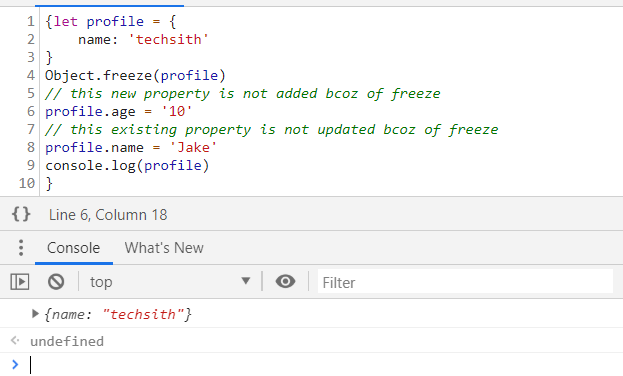
Q. How to remove duplicates from an array?

Ans. A set does not keep duplicates, so we can create a set using array which has duplicate elements.



Q. How to prevent an object from being added new properties?

Ans. Object.freeze(<object\_name>);



Q. How to prevent an object from being added new properties, but allow to modify existing ones?

Ans. Object.seal(<object\_name>);

