51+ CODE SNIPS

CODING BUGS NOTES GALLERY

JAVASCRIPT

Tips For Professional

CODING BUGS NOTES GALLERY

eBook by Manjurhusain Datardi

Index

JavaScript Tips

JavaScript is one of the most popular scripting or programming language.

In 1995, Brendan Eich from Netscape designed and implemented a new language for the Netscape Navigator browser. It was initially named ***Mocha***, then ***LiveScript***, and finally ***JavaScript***.

JavaScript is everywhere.

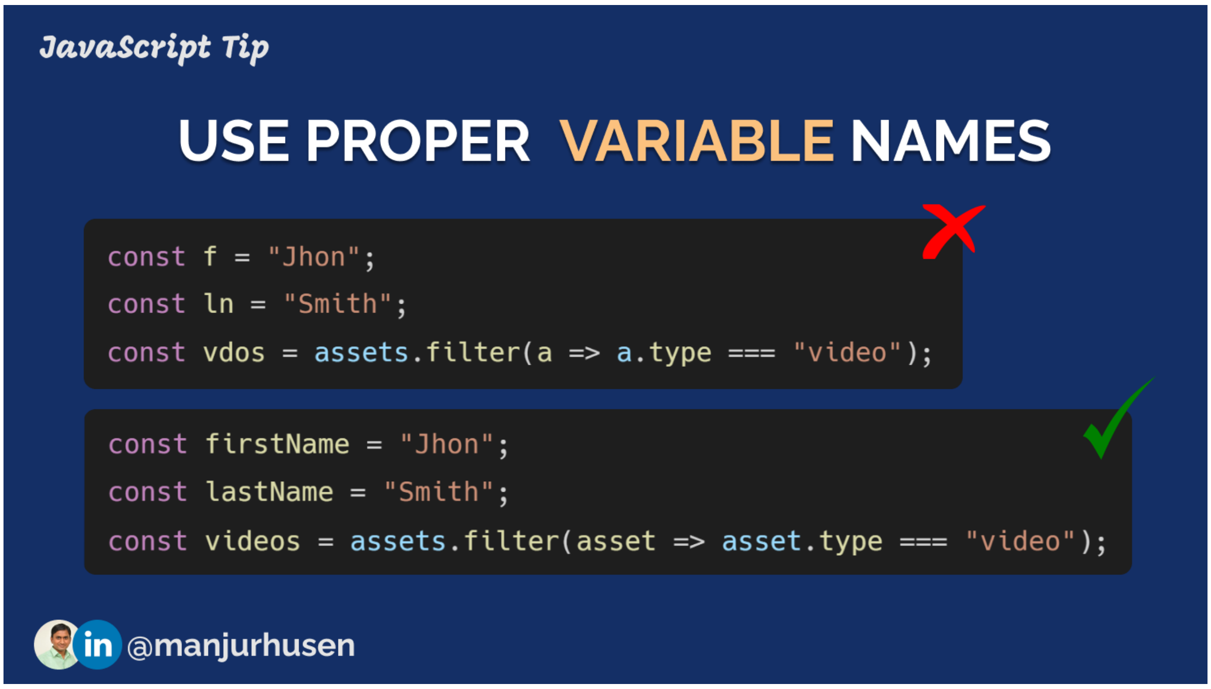
- More than 94% of websites use JavaScript.
- JavaScript completes its ninth year in a row as the most commonly used programming language. (2021 StackOverflow developer survey)

I have used the following two images in some code snippets with different meanings in different examples.

Image	**Meaning**
❌	Code is okay Can improve code Incorrect way

✓	Better code Improved code Correct way

Use proper variable names

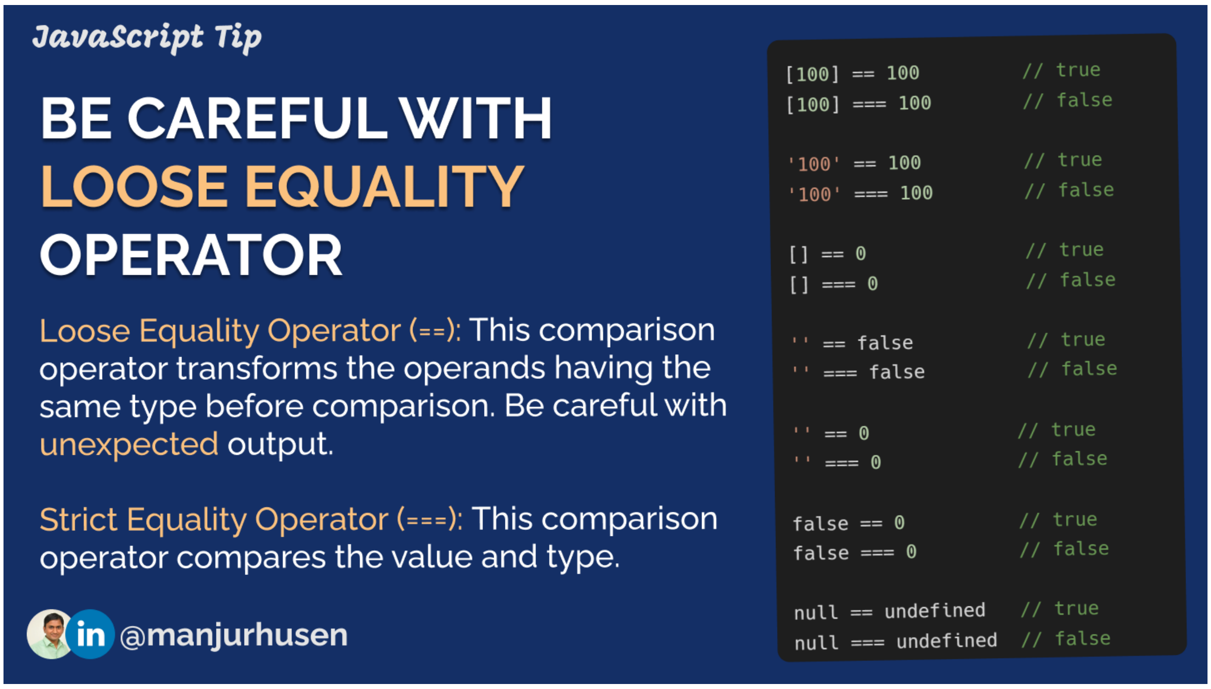


```
const f = "Jhon";
const ln = "Smith";
const vdos = assets.filter(a => a.type === "video");
```

```
const firstName = "Jhon";
const lastName = "Smith";
const videos = assets.filter(asset => asset.type === "video");
```

- Use the specific naming convention. Mostly used camel-case naming convention.
- The variable name should be concise and descriptive.
- It should explain the purpose.
- It is easy to pronounce.

Be careful with comparison using the loose equality operator


JavaScript Tip

BE CAREFUL WITH LOOSE EQUALITY OPERATOR

Loose Equality Operator (==): This comparison operator transforms the operands having the same type before comparison. Be careful with unexpected output.

Strict Equality Operator (===): This comparison operator compares the value and type.

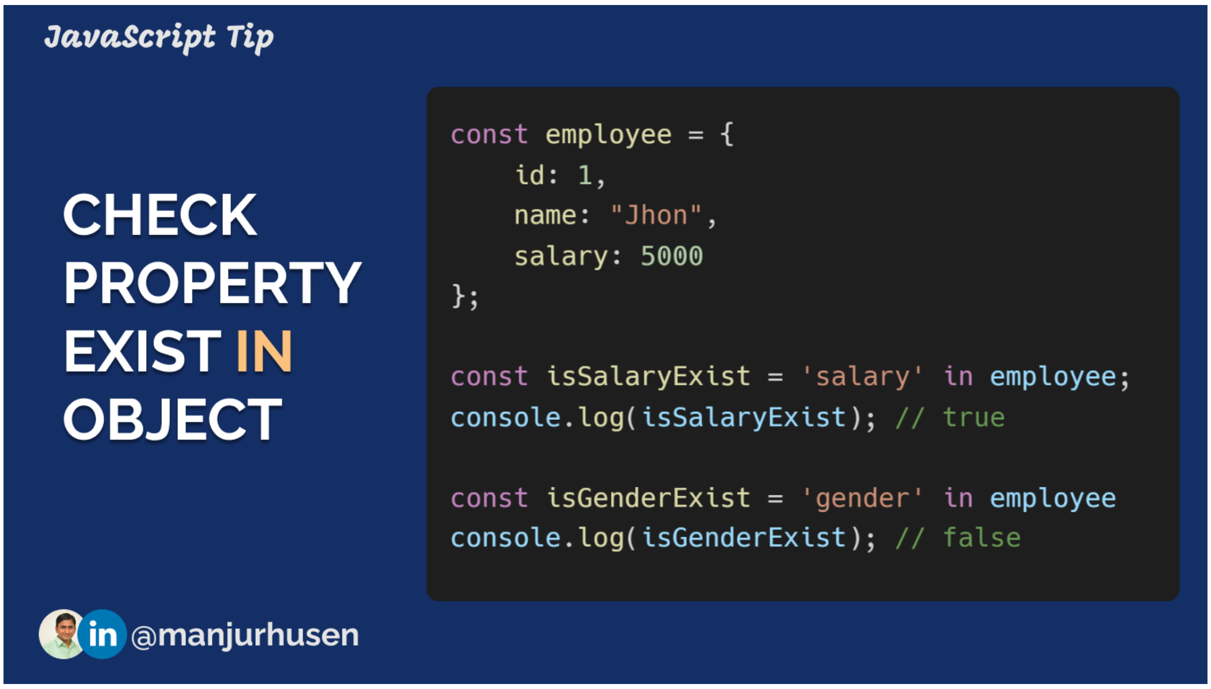
@manjurhusen

```
[100] == 100              // true
[100] === 100             // false

'100' == 100              // true
'100' === 100             // false

[] == 0                   // true
[] === 0                  // false

'' == false               // true
'' === false              // false

'' == 0                   // true
'' === 0                  // false

false == 0                // true
false === 0               // false

null == undefined         // true
null === undefined        // false
```

Loose Equality Operator (== OR !=) performs the automatic type conversion before comparison if needed.

Like in the above example, you can get unexpected output with Loose Equality Operator.

Check property exists in an object

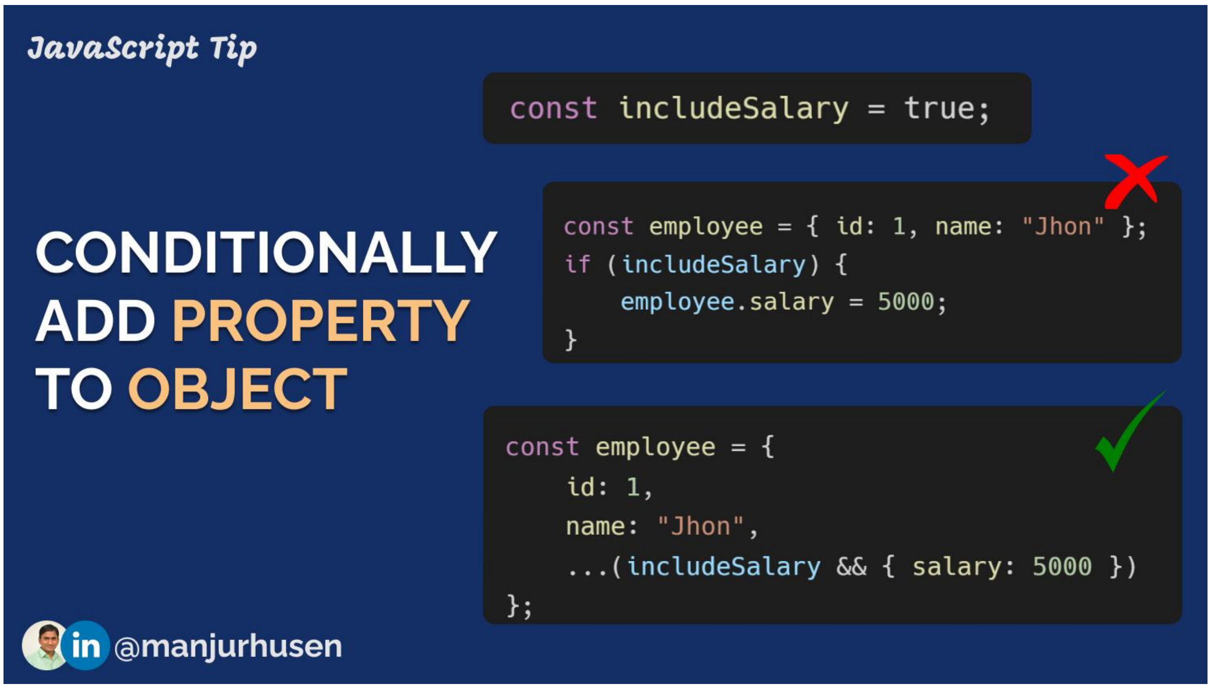


```
const employee = {
    id: 1,
    name: "Jhon",
    salary: 5000
};

const isSalaryExist = 'salary' in employee;
console.log(isSalaryExist); // true

const isGenderExist = 'gender' in employee
console.log(isGenderExist); // false
```

The ***in*** operator returns the boolean value true/false.

The ***in*** operator returns true if a property exists in the object or its prototype chain.

Conditionally add a property to an object



```
const includeSalary = true;
```

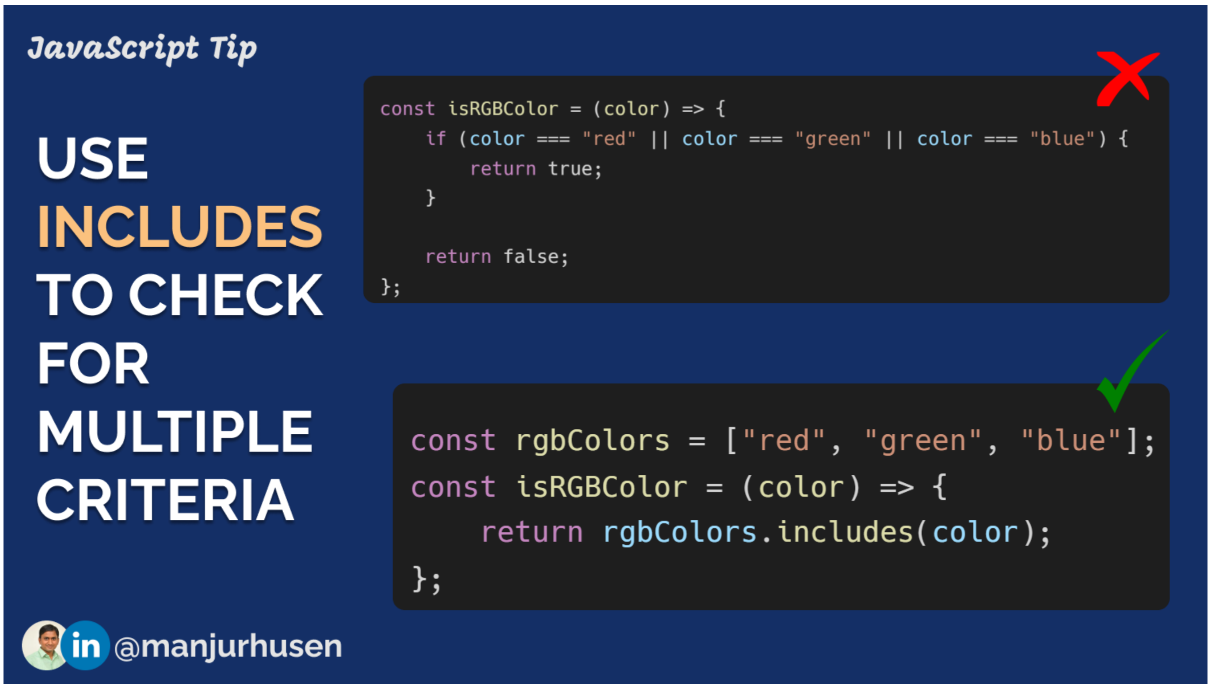
```
const employee = { id: 1, name: "Jhon" };
if (includeSalary) {
    employee.salary = 5000;
}
```

```
const employee = {
    id: 1,
    name: "Jhon",
    ...(includeSalary && { salary: 5000 })
};
```

Use ***spread operator (...)*** to spread an object into another object conditionally.

Use condition with **&&** operator to add a new property to an object. It will add a property to an object if the condition match.

Use includes to check for multiple criteria



```
const isRGBColor = (color) => {
    if (color === "red" || color === "green" || color === "blue") {
        return true;
    }

    return false;
};
```

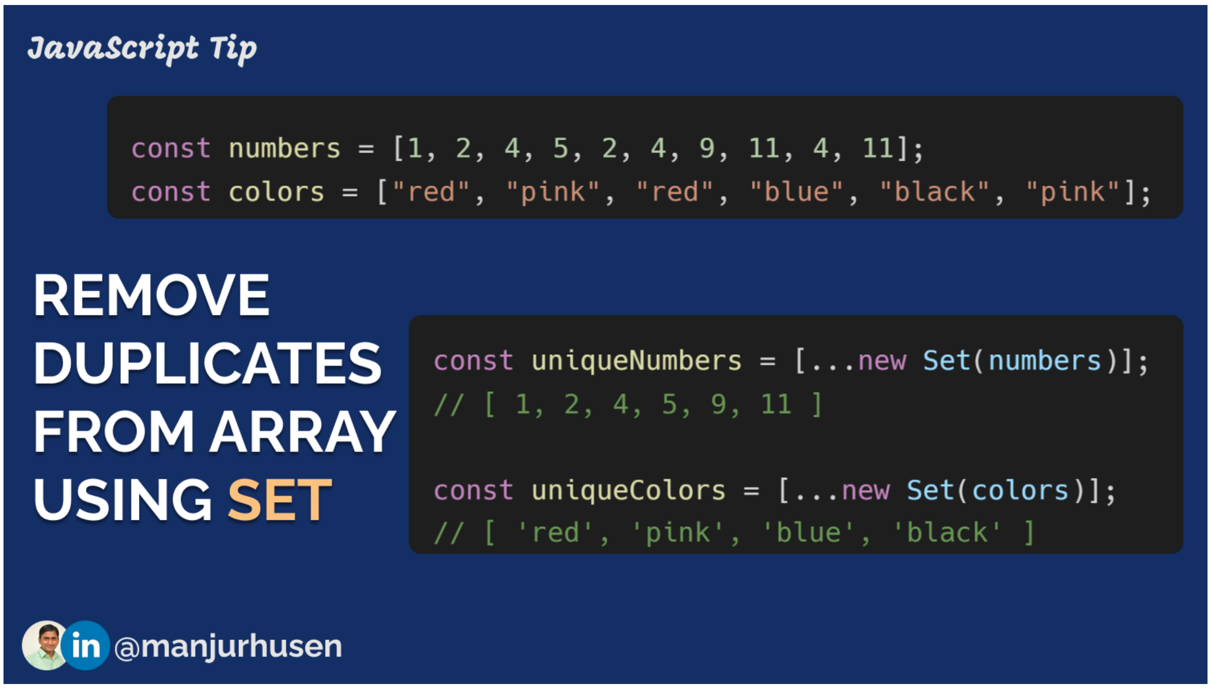
```
const rgbColors = ["red", "green", "blue"];
const isRGBColor = (color) => {
    return rgbColors.includes(color);
};
```

The ***includes()*** method determines whether an array includes a certain value among its entries. It returns true if a value exists, otherwise, it returns false.

Instead of extending the statement with more **||** (**OR**) conditions, rewrite the conditional by using the ***includes*** method.

More readable and concise alternative.

Remove duplicates from an array using Set



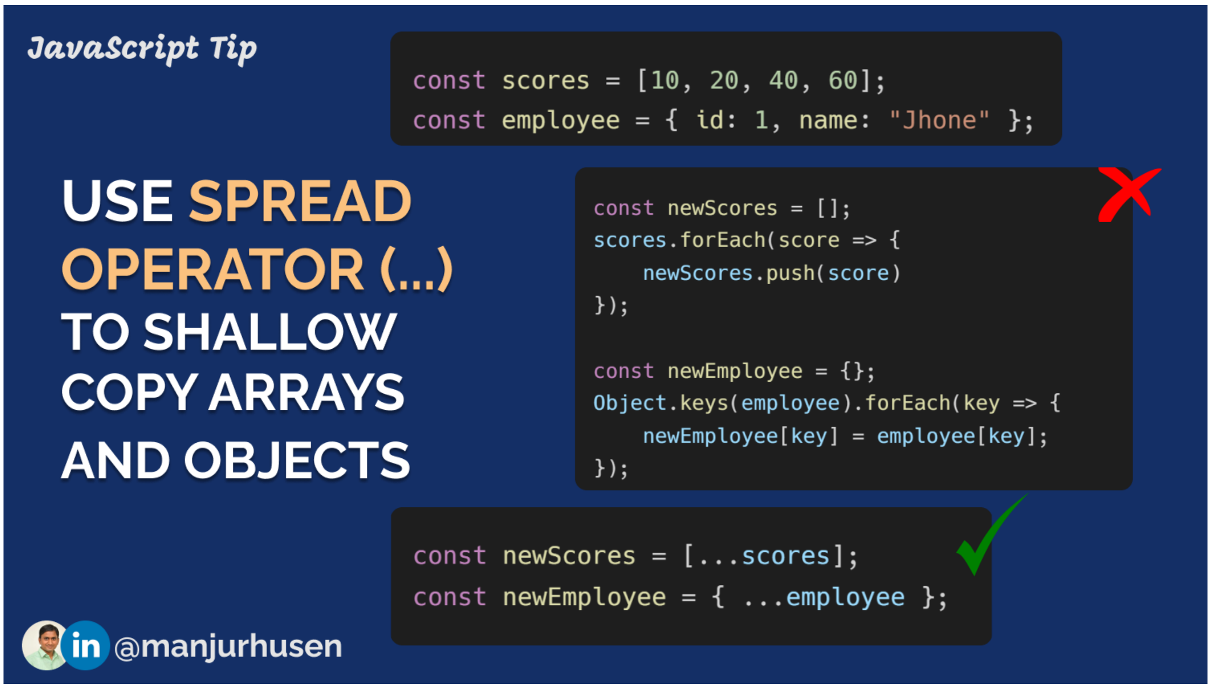
```
const numbers = [1, 2, 4, 5, 2, 4, 9, 11, 4, 11];
const colors = ["red", "pink", "red", "blue", "black", "pink"];
```

```
const uniqueNumbers = [...new Set(numbers)];
// [ 1, 2, 4, 5, 9, 11 ]

const uniqueColors = [...new Set(colors)];
// [ 'red', 'pink', 'blue', 'black' ]
```

Set is a new data object introduced in ES6. The Set only lets you store unique values of any type. When you pass an array to a ***new Set(array)***, it will remove duplicate values.

The ***spread syntax (...)*** is used to include all the items of the Set to a new array.

Use spread operator to shallow copy arrays and objects


JavaScript Tip
USE SPREAD OPERATOR (...) TO SHALLOW COPY ARRAYS AND OBJECTS
@manjurhusen

```
const scores = [10, 20, 40, 60];
const employee = { id: 1, name: "Jhone" };
```

```
const newScores = [];
scores.forEach(score => {
    newScores.push(score)
});

const newEmployee = {};
Object.keys(employee).forEach(key => {
    newEmployee[key] = employee[key];
});
```

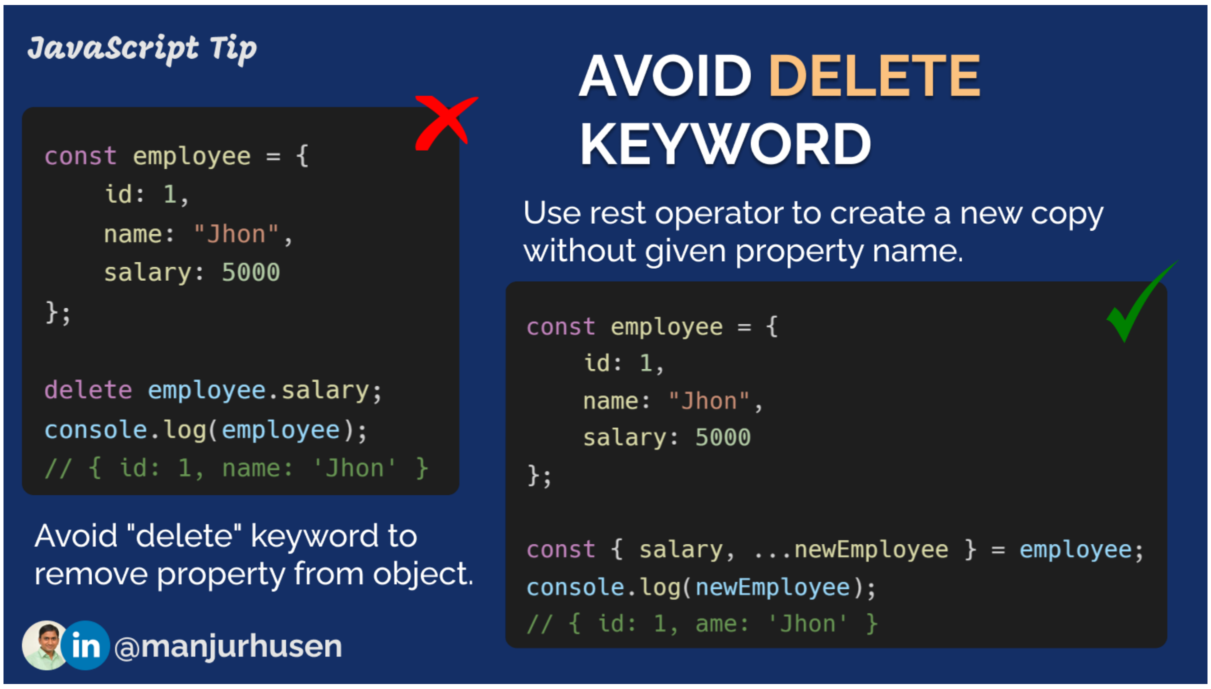
```
const newScores = [...scores];
const newEmployee = { ...employee };
```

Use the ***spread operator (...)*** to create a shallow copy of the object and array.

The *spread operator (...)* allows us to make copies of the original data (whether it is an array or object) and create a new copy of it.

It is an easy and clean way.

Avoid delete keyword



JavaScript Tip

AVOID DELETE KEYWORD

```
const employee = {
    id: 1,
    name: "Jhon",
    salary: 5000
};

delete employee.salary;
console.log(employee);
// { id: 1, name: 'Jhon' }
```

Avoid "delete" keyword to remove property from object.

Use rest operator to create a new copy without given property name.

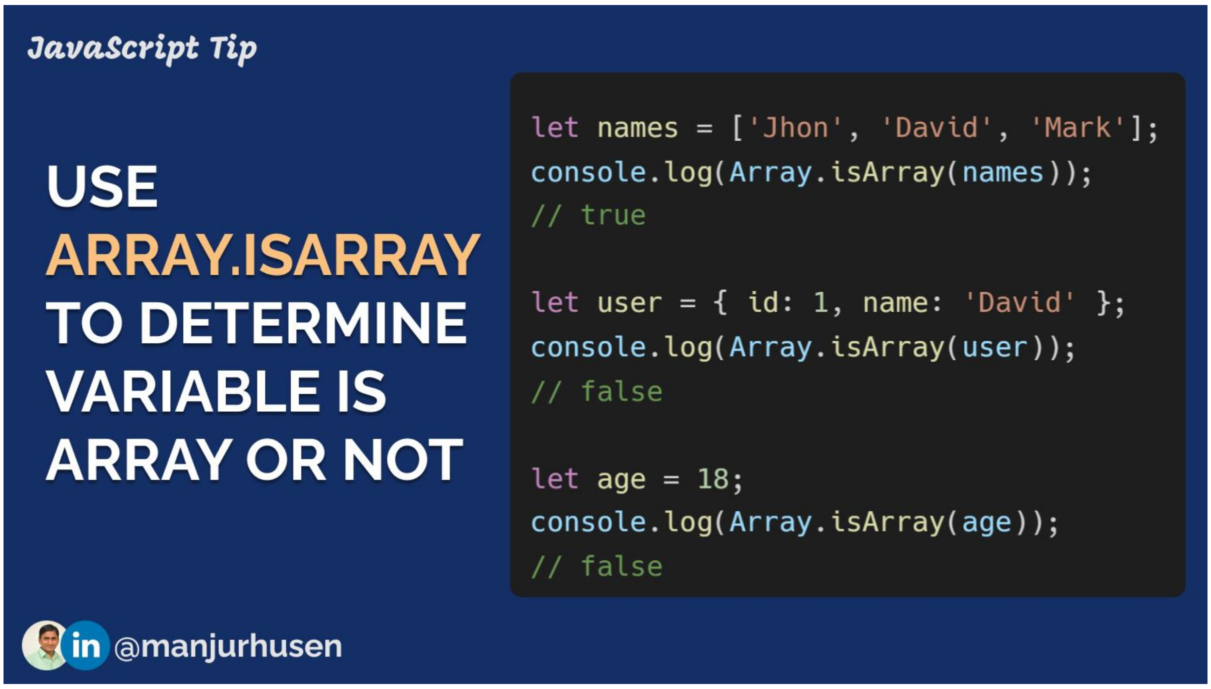
```
const employee = {
    id: 1,
    name: "Jhon",
    salary: 5000
};

const { salary, ...newEmployee } = employee;
console.log(newEmployee);
// { id: 1, ame: 'Jhon' }
```

@manjurhusen

Avoid a ***delete*** keyword to remove a property from an object. This way mutates the original object and hence leads to unpredictable behavior and makes debugging difficult.

A better way to delete a property without mutating the original object is by using the ***rest operator (...)***. Use the *rest operator (...)* to create a new copy without the given property name.

Use Array.isArray to determine the array

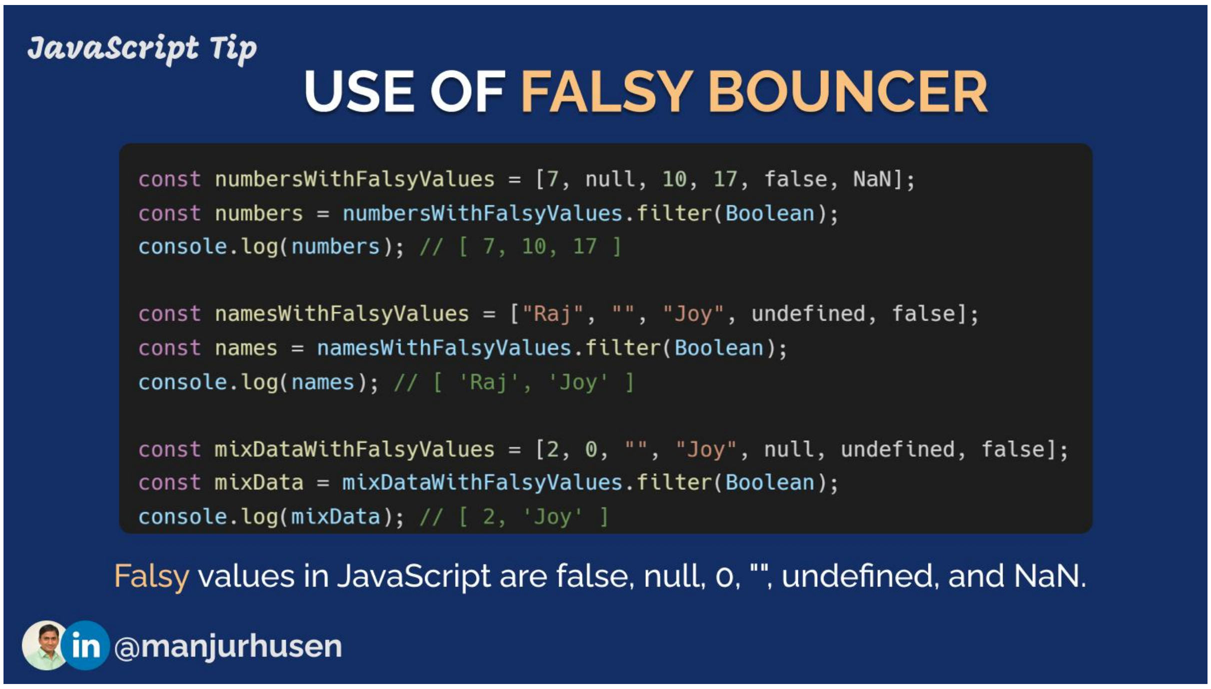


```
let names = ['Jhon', 'David', 'Mark'];
console.log(Array.isArray(names));
// true

let user = { id: 1, name: 'David' };
console.log(Array.isArray(user));
// false

let age = 18;
console.log(Array.isArray(age));
// false
```

The ***Array.isArray()*** method determines if the given argument is an Array or not.

- Returns *true* if the value is Array.
- Returns *false* if the value is not Array.

Use of falsy bouncer


JavaScript Tip

USE OF FALSY BOUNCER

```
const numbersWithFalsyValues = [7, null, 10, 17, false, NaN];
const numbers = numbersWithFalsyValues.filter(Boolean);
console.log(numbers); // [ 7, 10, 17 ]

const namesWithFalsyValues = ["Raj", "", "Joy", undefined, false];
const names = namesWithFalsyValues.filter(Boolean);
console.log(names); // [ 'Raj', 'Joy' ]

const mixDataWithFalsyValues = [2, 0, "", "Joy", null, undefined, false];
const mixData = mixDataWithFalsyValues.filter(Boolean);
console.log(mixData); // [ 2, 'Joy' ]
```

Falsy values in JavaScript are false, null, 0, "", undefined, and NaN.

@manjurhusen

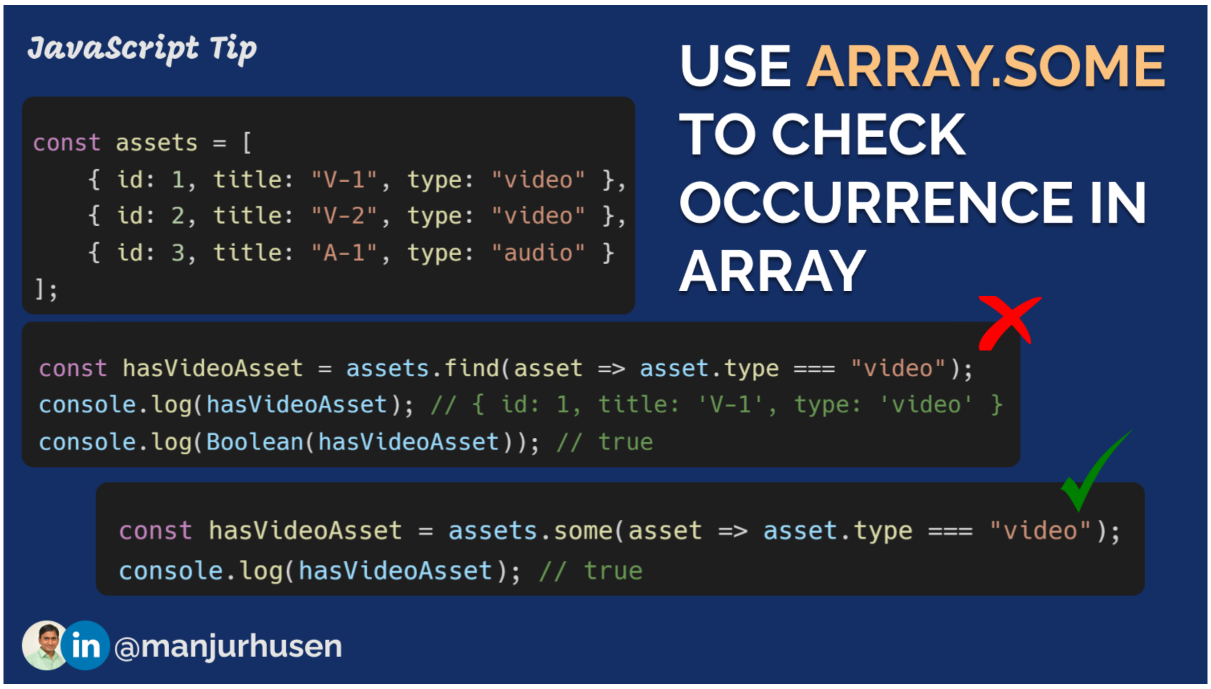
A falsy value is a value that is considered false when examined as a Boolean.

Falsy Bouncer means removing all falsy values from an array.

Falsy values in JavaScript are *false, null, 0, undefined, NaN, and "" (empty string).*

Pass the ***Boolean*** to ***Array.filter*** as the first argument and it will serve as a falsy bouncer.

Use Array.some to check occurrence in array



```
const assets = [
    { id: 1, title: "V-1", type: "video" },
    { id: 2, title: "V-2", type: "video" },
    { id: 3, title: "A-1", type: "audio" }
];
```

```
const hasVideoAsset = assets.find(asset => asset.type === "video");
console.log(hasVideoAsset); // { id: 1, title: 'V-1', type: 'video' }
console.log(Boolean(hasVideoAsset)); // true
```

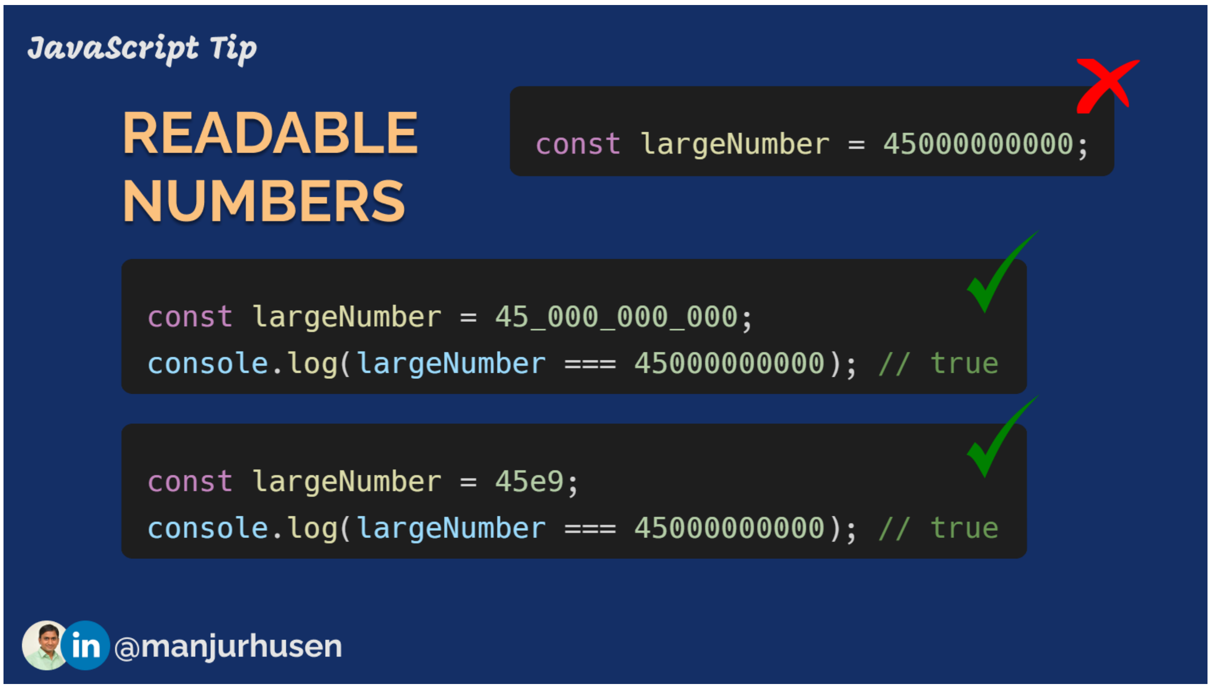
```
const hasVideoAsset = assets.some(asset => asset.type === "video");
console.log(hasVideoAsset); // true
```

If we want to check only occurrence means value exist or not then use Array.some instead of Array.find.

The ***some()*** method checks if any array items pass a test implemented by the provided function. If the function returns true, *some()* returns true and stops.

The *some()* method does not change the original array.

Readable numbers



```
const largeNumber = 45000000000;
```

```
const largeNumber = 45_000_000_000;
console.log(largeNumber === 45000000000); // true
```

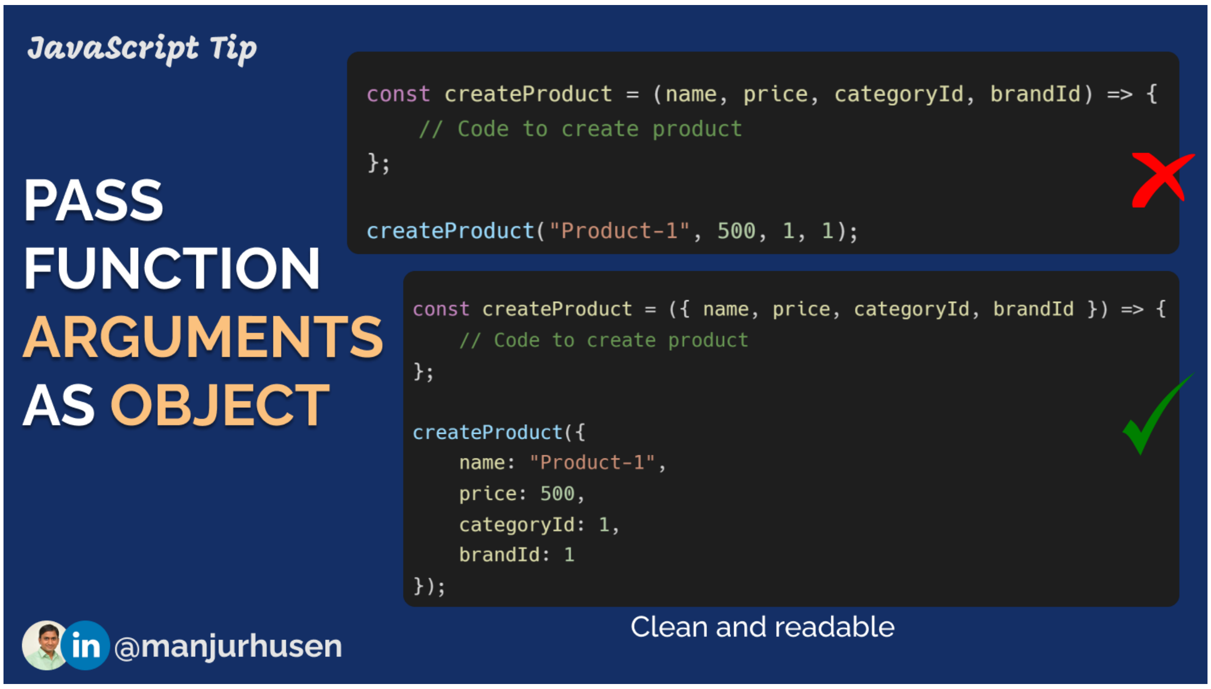
```
const largeNumber = 45e9;
console.log(largeNumber === 45000000000); // true
```

When working with large numbers it can be hard to read them out.

The ***Numeric Separators*** allow us to use ***underscore*** *(_)* as a separator in numeric literals, for example, you can write 50000 as 50_000.

This feature improves readability.

Pass function arguments as an object



```
const createProduct = (name, price, categoryId, brandId) => {
    // Code to create product
};

createProduct("Product-1", 500, 1, 1);
```

```
const createProduct = ({ name, price, categoryId, brandId }) => {
    // Code to create product
};

createProduct({
    name: "Product-1",
    price: 500,
    categoryId: 1,
    brandId: 1
});
```

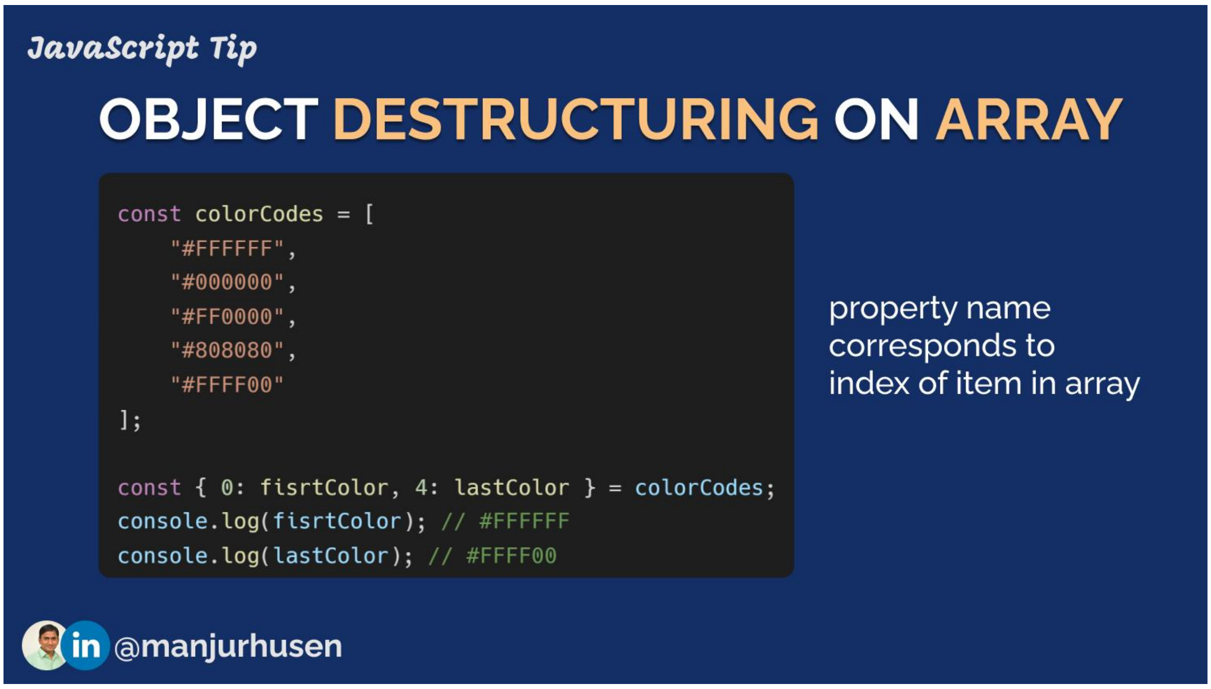
Clean and readable

Parameters are part of a function definition. A JavaScript function can have any number of parameters. When we invoke a function and pass some values to that function, these values are called function arguments.

If a function has more than 1 parameter, it is hard to figure out what these arguments mean when the function is called. When you pass the arguments, the order is important.

A better way is to create a function with object (with properties) parameters like in the example. When we pass the argument contained in an object it is pretty much clear from the names of the properties. Also, the order of properties doesn’t matter anymore.

Object destructuring on arrays



JavaScript Tip

OBJECT DESTRUCTURING ON ARRAY

```
const colorCodes = [
    "#FFFFFF",
    "#000000",
    "#FF0000",
    "#808080",
    "#FFFF00"
];

const { 0: fisrtColor, 4: lastColor } = colorCodes;
console.log(fisrtColor); // #FFFFFF
console.log(lastColor); // #FFFF00
```

property name corresponds to index of item in array

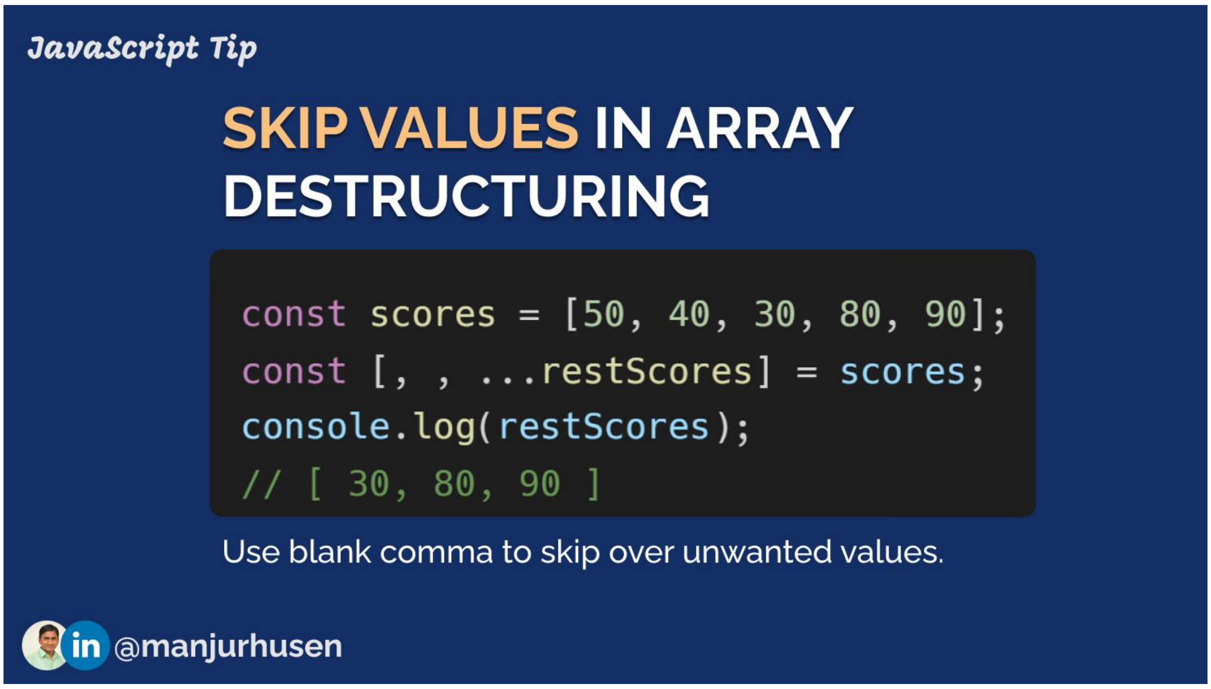
@manjurhusen

The destructuring assignment provides a clean way to extract values from arrays and objects. Array destructuring is a way that allows us to extract an array’s value into new variables.

Each item in the array has an index. The property name corresponds to the index of the item that returns the value like in the example.

It is an easy way to get a specific item from an array in a single line of code.

Skip values in array destructuring


JavaScript Tip

SKIP VALUES IN ARRAY DESTRUCTURING

```
const scores = [50, 40, 30, 80, 90];
const [, , ...restScores] = scores;
console.log(restScores);
// [ 30, 80, 90 ]
```

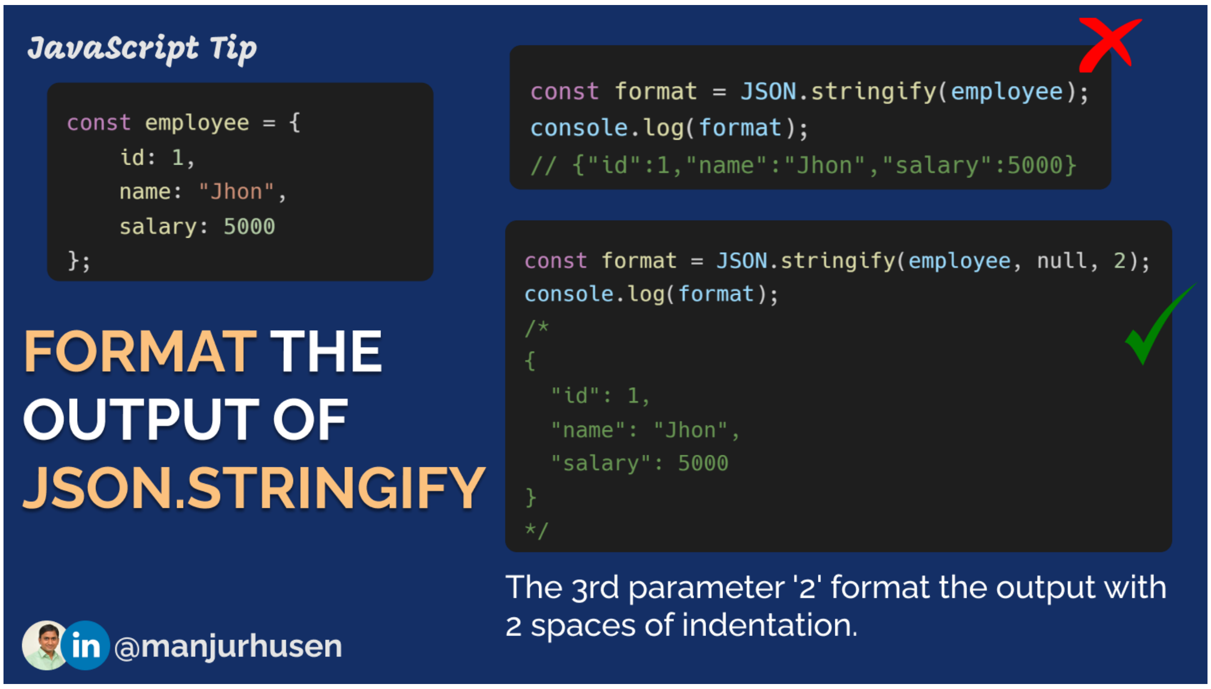
Use blank comma to skip over unwanted values.

Destructuring means breaking down a complex structure into simpler parts.

Array destructuring is a way that allows us to extract an array’s value into new variables. Sometimes we don't need some values from the array means we want to skip those values. During the destructuring arrays, if you want to skip some values, use an **empty *placeholder* comma**.

This is a clean way to skip values.

Format the output of JSON.stringify



```
const employee = {
    id: 1,
    name: "Jhon",
    salary: 5000
};
```

```
const format = JSON.stringify(employee);
console.log(format);
// {"id":1,"name":"Jhon","salary":5000}
```

```
const format = JSON.stringify(employee, null, 2);
console.log(format);
/*
{
  "id": 1,
  "name": "Jhon",
  "salary": 5000
}
*/
```

The 3rd parameter '2' format the output with 2 spaces of indentation.

The ***JSON.stringify()*** method converts a JavaScript object to a JSON string.

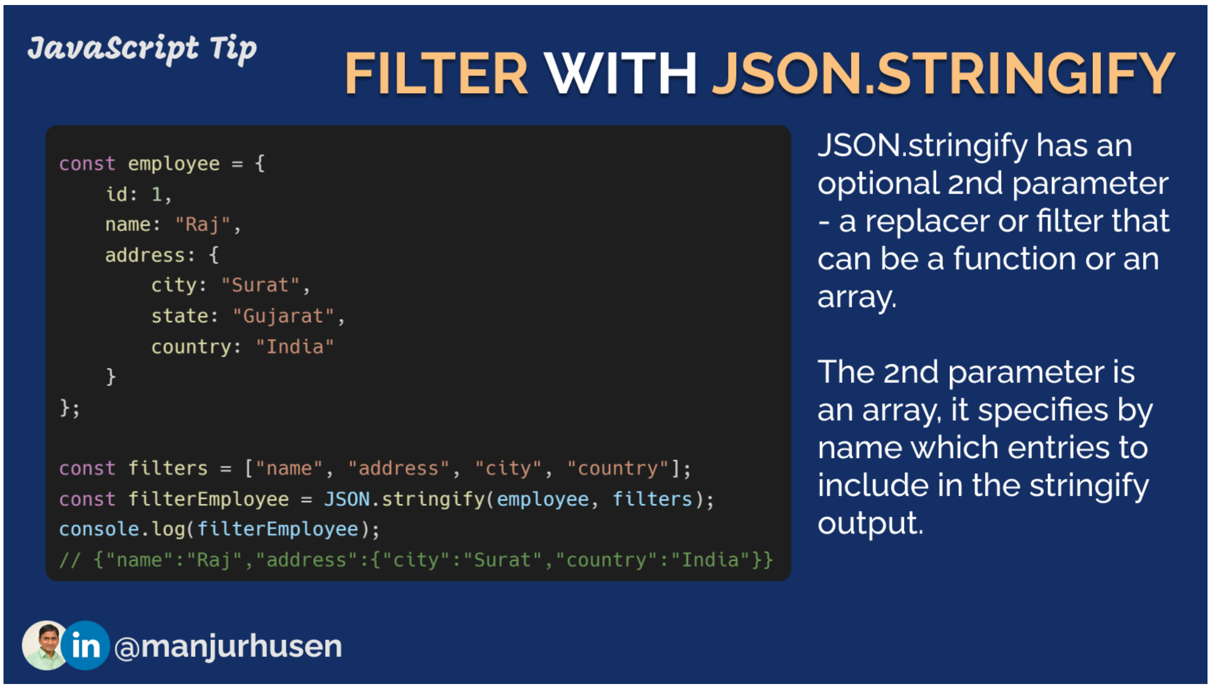
The 3rd parameter to *JSON.stringify()* is called ***spacer***.

You can pass String or Number value to insert whitespace in the returned string.

If the 3rd parameter is a Number, it indicates the number of spaces for indenting purposes.

If the 3rd parameter is a String, the string is used as whitespace.

Filter with JSON.stringify


JavaScript Tip

FILTER WITH JSON.STRINGIFY

```
const employee = {
    id: 1,
    name: "Raj",
    address: {
        city: "Surat",
        state: "Gujarat",
        country: "India"
    }
};

const filters = ["name", "address", "city", "country"];
const filterEmployee = JSON.stringify(employee, filters);
console.log(filterEmployee);
// {"name":"Raj","address":{"city":"Surat","country":"India"}}
```

JSON.stringify has an optional 2nd parameter - a replacer or filter that can be a function or an array.

The 2nd parameter is an array, it specifies by name which entries to include in the stringify output.

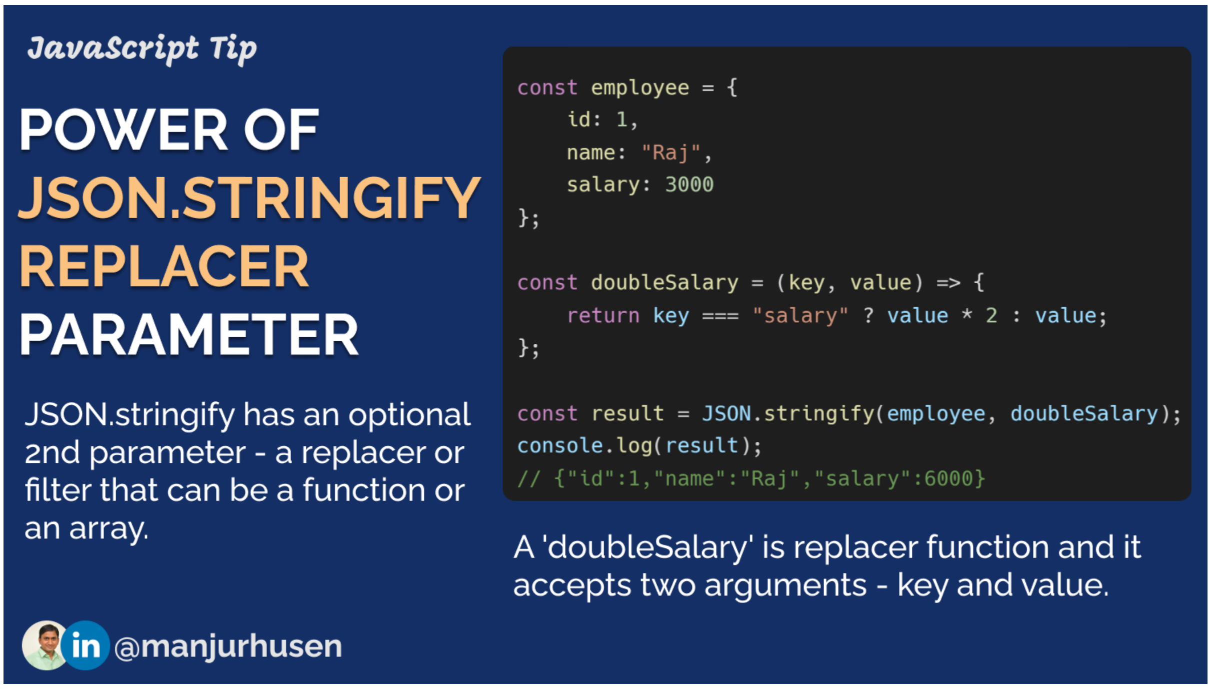
@manjurhusen

The ***JSON.stringify()*** method converts a JavaScript object to a JSON string.

The 2nd parameter to *JSON.stringify()* is a ***replacer*** or ***filter*** that can be a function or an array.

When 2nd parameter is passed as an array, it works as a filter and includes only those properties in the JSON string which are defined in an array.

Power of JSON.stringify replacer parameter


JavaScript Tip

POWER OF JSON.STRINGIFY REPLACER PARAMETER

JSON.stringify has an optional 2nd parameter - a replacer or filter that can be a function or an array.

```
const employee = {
    id: 1,
    name: "Raj",
    salary: 3000
};

const doubleSalary = (key, value) => {
    return key === "salary" ? value * 2 : value;
};

const result = JSON.stringify(employee, doubleSalary);
console.log(result);
// {"id":1,"name":"Raj","salary":6000}
```

A 'doubleSalary' is replacer function and it accepts two arguments - key and value.

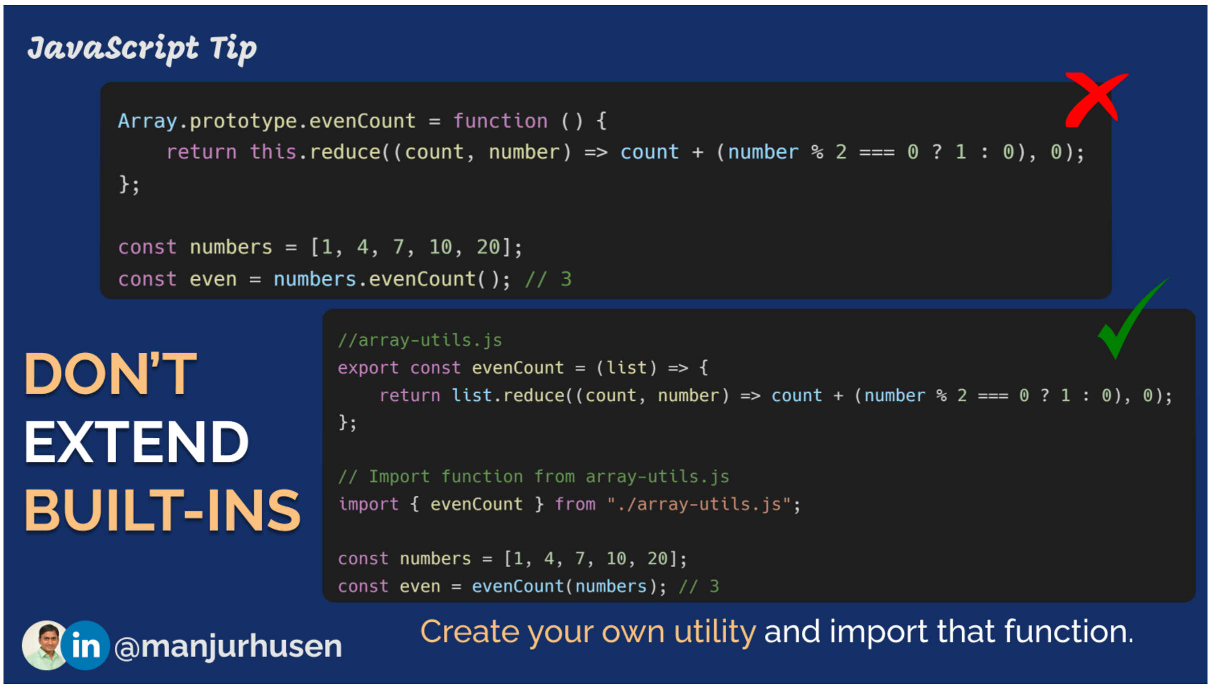
@manjurhusen

The *JSON.stringify()* method converts a JavaScript object to a JSON string.

The 2nd parameter to ***JSON.stringify()*** is a ***replacer*** or ***filter*** that can be a function or array.

When 2nd parameter is passed as a replacer function, it alters the behavior of the stringification process. As a function, it takes two parameters, the key and the value being stringified.

Don’t extend built-ins


JavaScript Tip

```
Array.prototype.evenCount = function () {
    return this.reduce((count, number) => count + (number % 2 === 0 ? 1 : 0), 0);
};

const numbers = [1, 4, 7, 10, 20];
const even = numbers.evenCount(); // 3
```

DON’T EXTEND BUILT-INS

```
//array-utils.js
export const evenCount = (list) => {
    return list.reduce((count, number) => count + (number % 2 === 0 ? 1 : 0), 0);
};

// Import function from array-utils.js
import { evenCount } from "./array-utils.js";

const numbers = [1, 4, 7, 10, 20];
const even = evenCount(numbers); // 3
```

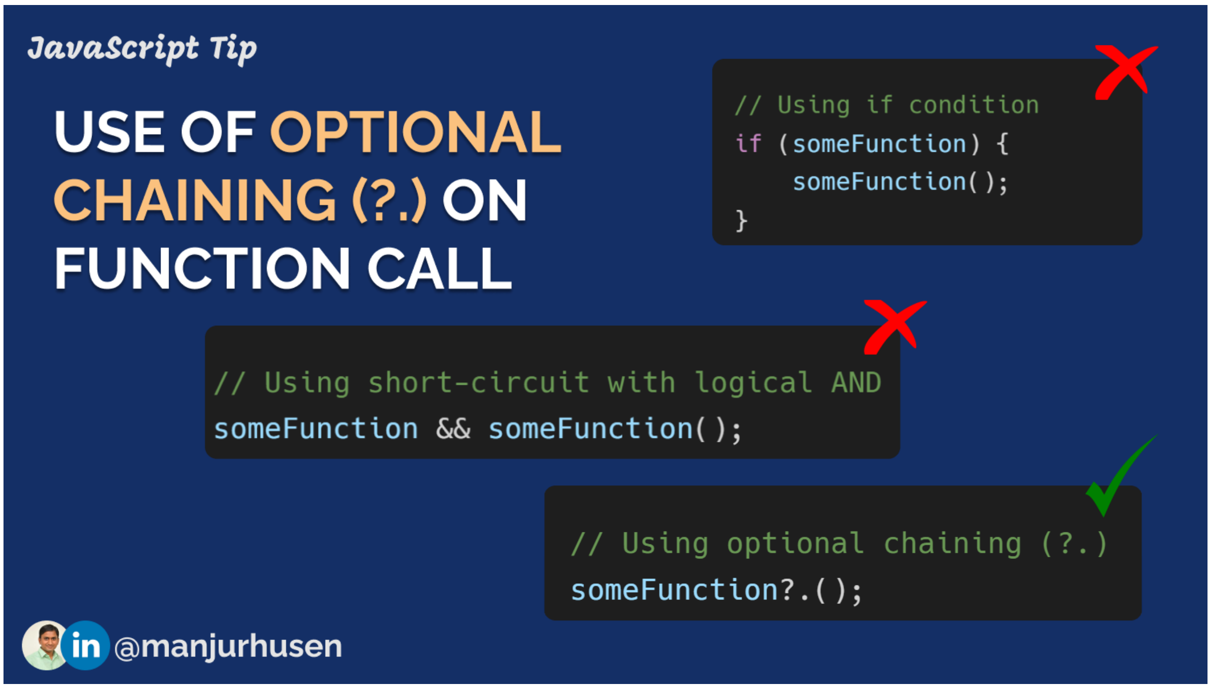
@manjurhusen

Create your own utility and import that function.

Extending built-in Objects/types or Array is not a good practice in JavaScript.

A better way is to create your own utility library and use it.

Use of optional chaining on function call



```
// Using if condition
if (someFunction) {
    someFunction();
}
```

```
// Using short-circuit with logical AND
someFunction && someFunction();
```

```
// Using optional chaining (?.)
someFunction?.();
```

The ***optional chaining operator (?.)*** is a safe and concise way to access properties that are potentially null or undefined.

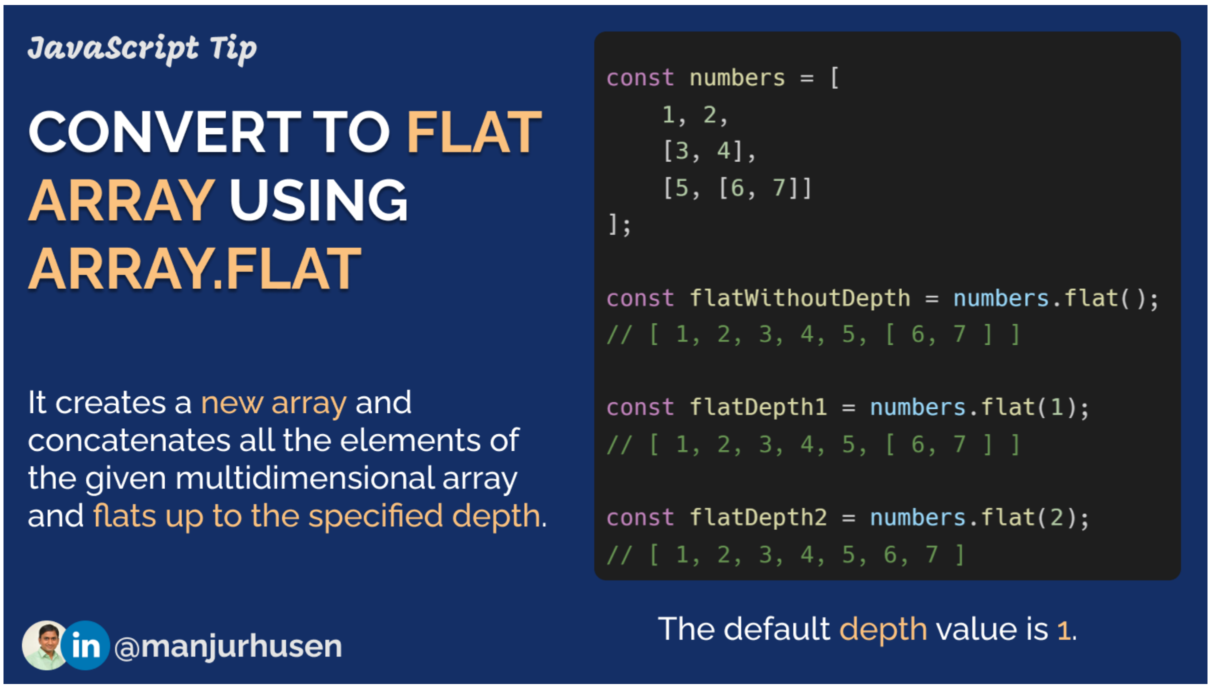
The *chaining operator (.)* throws an error if a reference is null or undefined.

The *optional chaining operator (?.)* will return undefined if a reference is null or undefined.

Just like with properties, we can use the optional chaining operator with methods also.

Less code and clean way.

Convert to a flat array using Array.flat



JavaScript Tip

CONVERT TO FLAT ARRAY USING ARRAY.FLAT

It creates a new array and concatenates all the elements of the given multidimensional array and flats up to the specified depth.

```
const numbers = [
    1, 2,
    [3, 4],
    [5, [6, 7]]
];

const flatWithoutDepth = numbers.flat();
// [ 1, 2, 3, 4, 5, [ 6, 7 ] ]

const flatDepth1 = numbers.flat(1);
// [ 1, 2, 3, 4, 5, [ 6, 7 ] ]

const flatDepth2 = numbers.flat(2);
// [ 1, 2, 3, 4, 5, 6, 7 ]
```

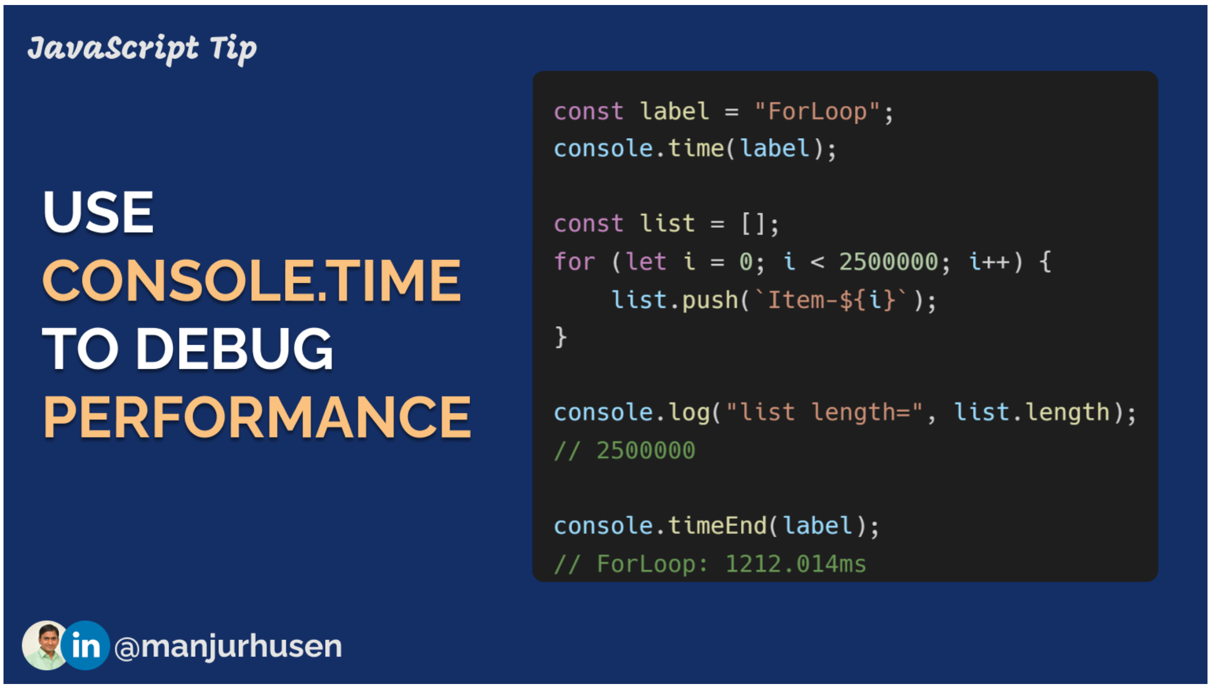
The default depth value is 1.

@manjurhusen

Flattening an array is the process of reducing the number of dimensions of an array to a lower number.

The ***flat()*** method creates a new array with all items of subarray concatenated into it recursively up to the specified *depth*.

Use console.time to debug performance

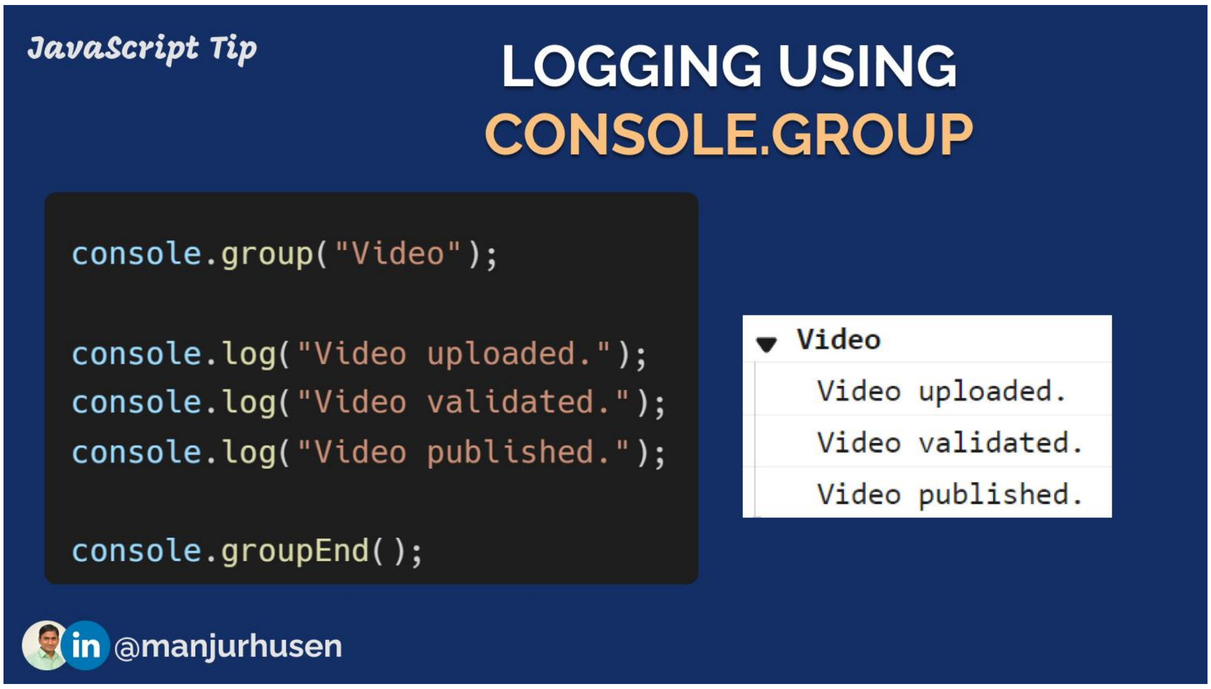


```
const label = "ForLoop";
console.time(label);

const list = [];
for (let i = 0; i < 2500000; i++) {
    list.push(`Item-${i}`);
}

console.log("list length=", list.length);
// 2500000

console.timeEnd(label);
// ForLoop: 1212.014ms
```

The console object has ***time()*** and ***timeEnd()*** methods. These two methods help us to analyze the performance of our code.

The *console.time()* method starts a timer to track how long an operation takes. You can give each timer a unique name. When you call *console.timeEnd()* with the same name, the browser will output the time in milliseconds.

Logging using console.group



```
console.group("Video");

console.log("Video uploaded.");
console.log("Video validated.");
console.log("Video published.");

console.groupEnd();
```

```
▼ Video
    Video uploaded.
    Video validated.
    Video published.
```

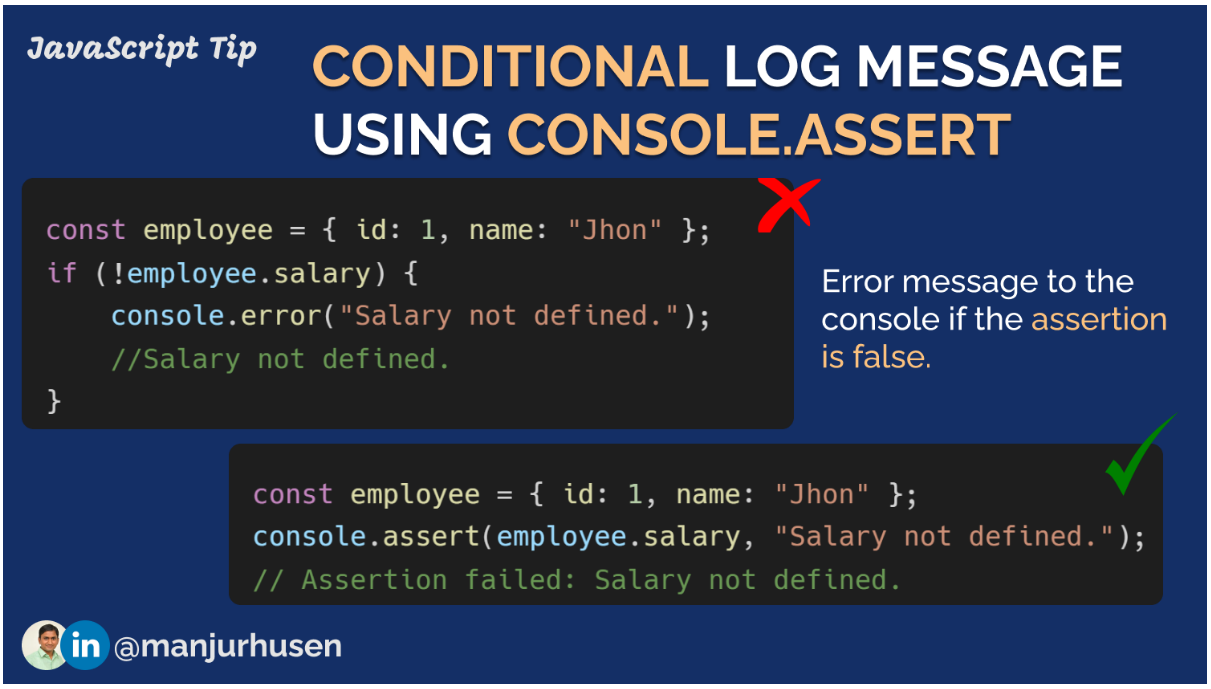
The console object has ***group()*** and ***groupEnd()*** methods.

The *console.group()* method starts a new inline group in the web console log. This method takes an optional argument *label*.

The *console.groupEnd()* method ends the group.

It organizes your messages and improves visibility.

Conditional log message using console.assert



JavaScript Tip

CONDITIONAL LOG MESSAGE USING CONSOLE.ASSERT

```
const employee = { id: 1, name: "Jhon" };
if (!employee.salary) {
    console.error("Salary not defined.");
    //Salary not defined.
}
```

Error message to the console if the assertion is false.

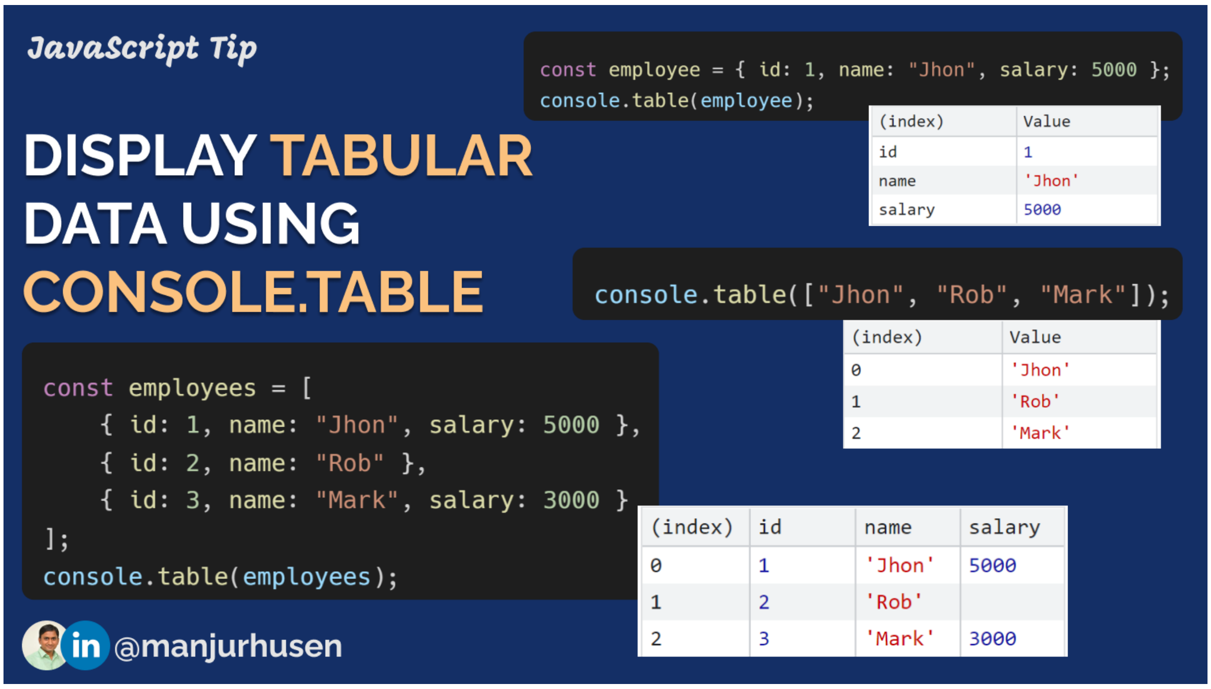
```
const employee = { id: 1, name: "Jhon" };
console.assert(employee.salary, "Salary not defined.");
// Assertion failed: Salary not defined.
```

@manjurhusen

The console object has an ***assert()*** method which helps to log an error message conditionally.

The *console.assert()* method writes an error message to the console if the assertion is false. If the assertion is true, nothing happens.

Display tabular data using console.table



```
const employee = { id: 1, name: "Jhon", salary: 5000 };
console.table(employee);
```

(index)	Value
id	1
name	'Jhon'
salary	5000

```
console.table(["Jhon", "Rob", "Mark"]);
```

(index)	Value
0	'Jhon'
1	'Rob'
2	'Mark'

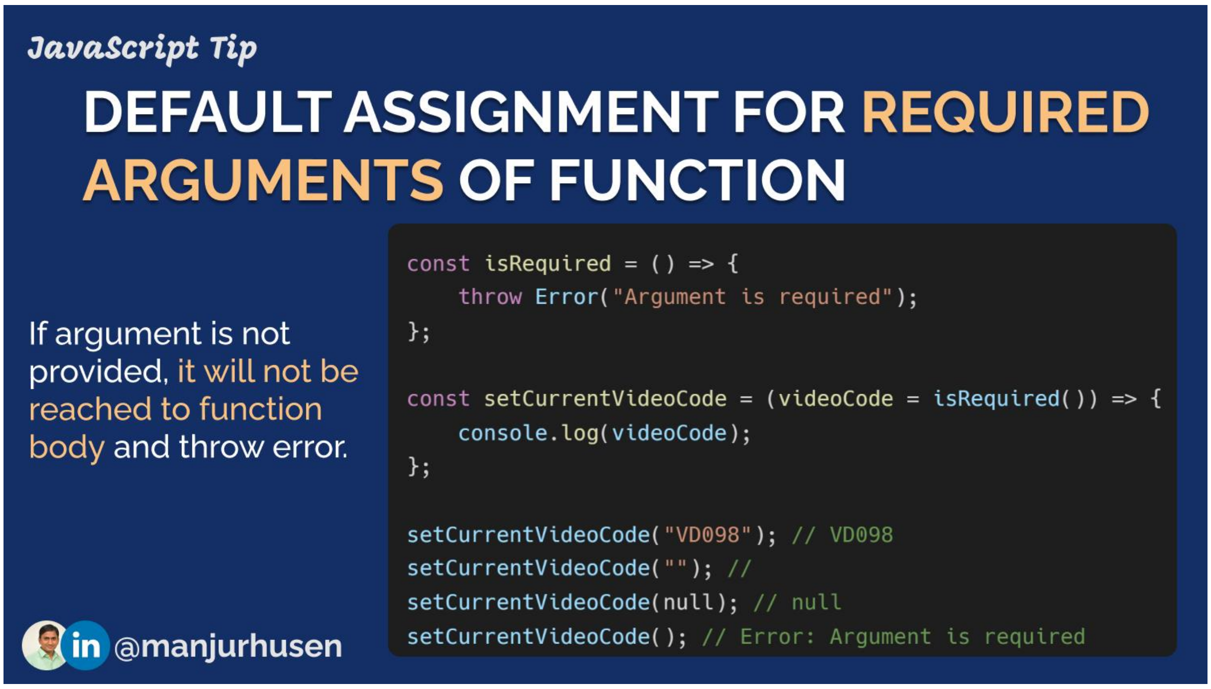
```
const employees = [
    { id: 1, name: "Jhon", salary: 5000 },
    { id: 2, name: "Rob" },
    { id: 3, name: "Mark", salary: 3000 }
];
console.table(employees);
```

(index)	id	name	salary
0	1	'Jhon'	5000
1	2	'Rob'	
2	3	'Mark'	3000

The console object has a ***table()*** method which allows you to display arrays and objects to the console in tabular form.

The *console.table()* method provides better data visualization.

Default assignment for required arguments of the function


JavaScript Tip
DEFAULT ASSIGNMENT FOR REQUIRED ARGUMENTS OF FUNCTION
If argument is not provided, it will not be reached to function body and throw error.
in @manjurhusen

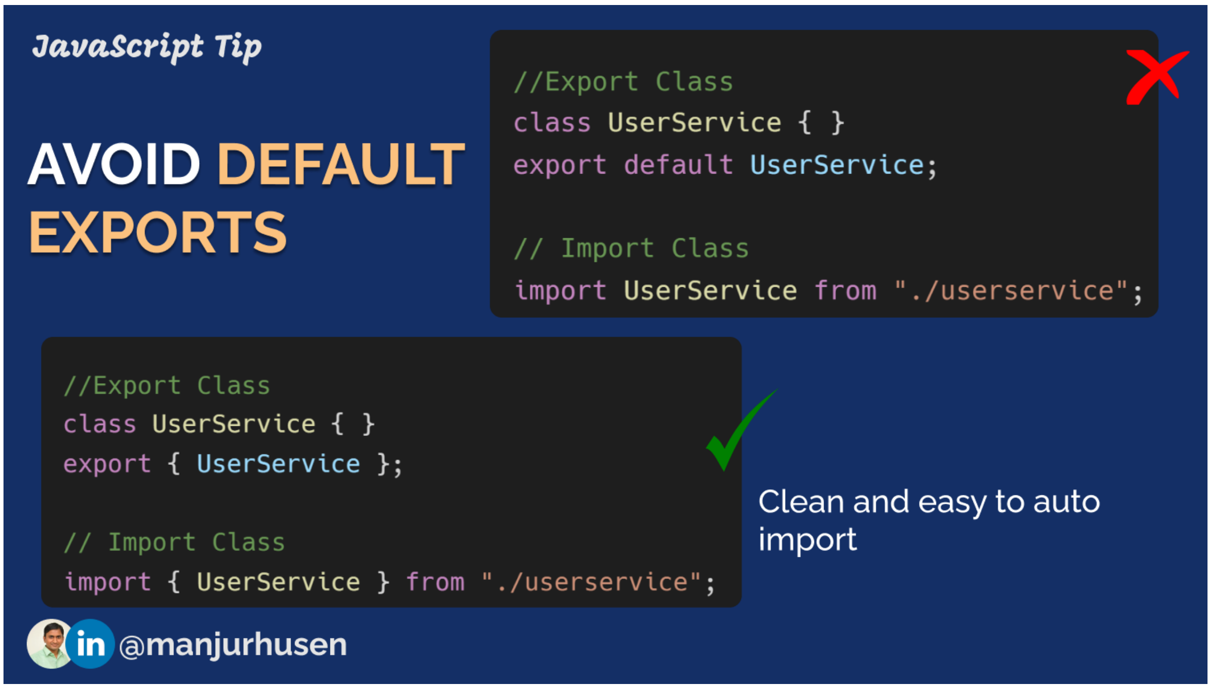
```
const isRequired = () => {
    throw Error("Argument is required");
};

const setCurrentVideoCode = (videoCode = isRequired()) => {
    console.log(videoCode);
};

setCurrentVideoCode("VD098"); // VD098
setCurrentVideoCode(""); //
setCurrentVideoCode(null); // null
setCurrentVideoCode(); // Error: Argument is required
```

You can use ***default parameters*** to make the function arguments required.

If you don't provide the parameter, it will default to the function which throws an error.

Note that null is considered a value, so passing null will not result in a default assignment.

Avoid default exports



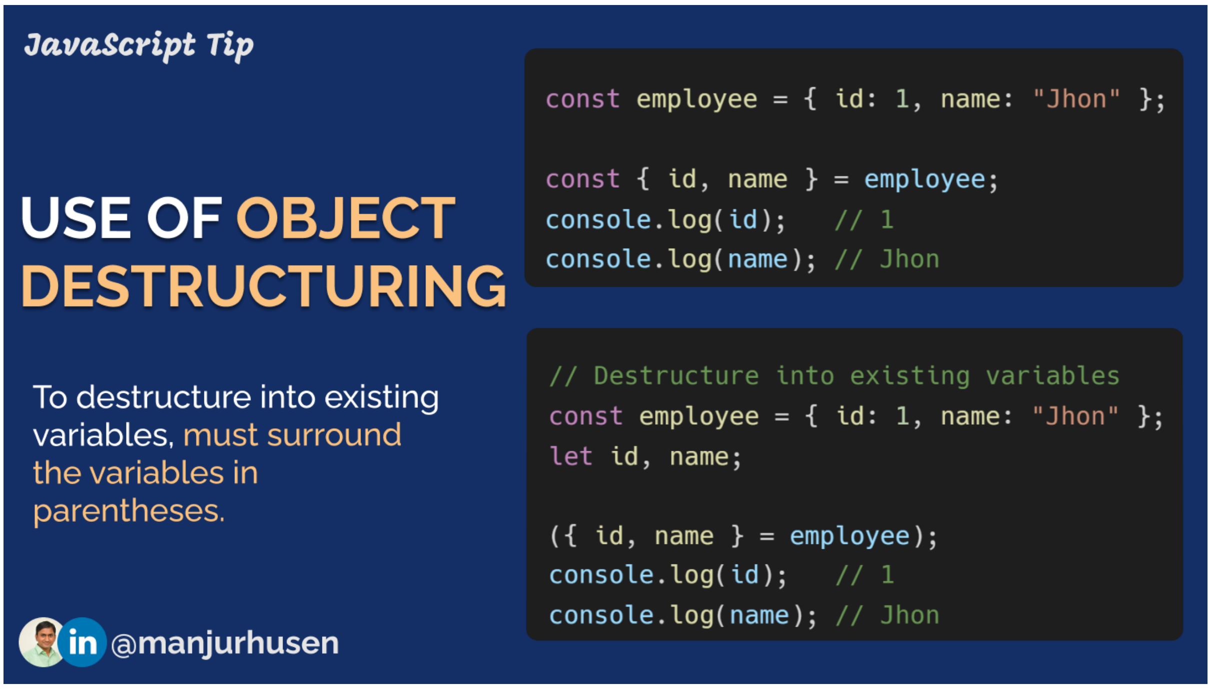
```
//Export Class
class UserService { }
export default UserService;

// Import Class
import UserService from "./userservice";
```

```
//Export Class
class UserService { }
export { UserService };

// Import Class
import { UserService } from "./userservice";
```

Problems with default exports are:

- Discoverability is very poor for default exports.
- Difficult to analyze by automated tools or provide code autocompletion.
- Horrible experience for CommonJS.
- TypeScript auto-import struggles.
- Default exports make large-scale refactoring impossible.

Use of object destructuring


JavaScript Tip

USE OF OBJECT DESTRUCTURING

To destructure into existing variables, must surround the variables in parentheses.

```
const employee = { id: 1, name: "Jhon" };

const { id, name } = employee;
console.log(id);   // 1
console.log(name); // Jhon
```

```
// Destructure into existing variables
const employee = { id: 1, name: "Jhon" };
let id, name;

({ id, name } = employee);
console.log(id);   // 1
console.log(name); // Jhon
```

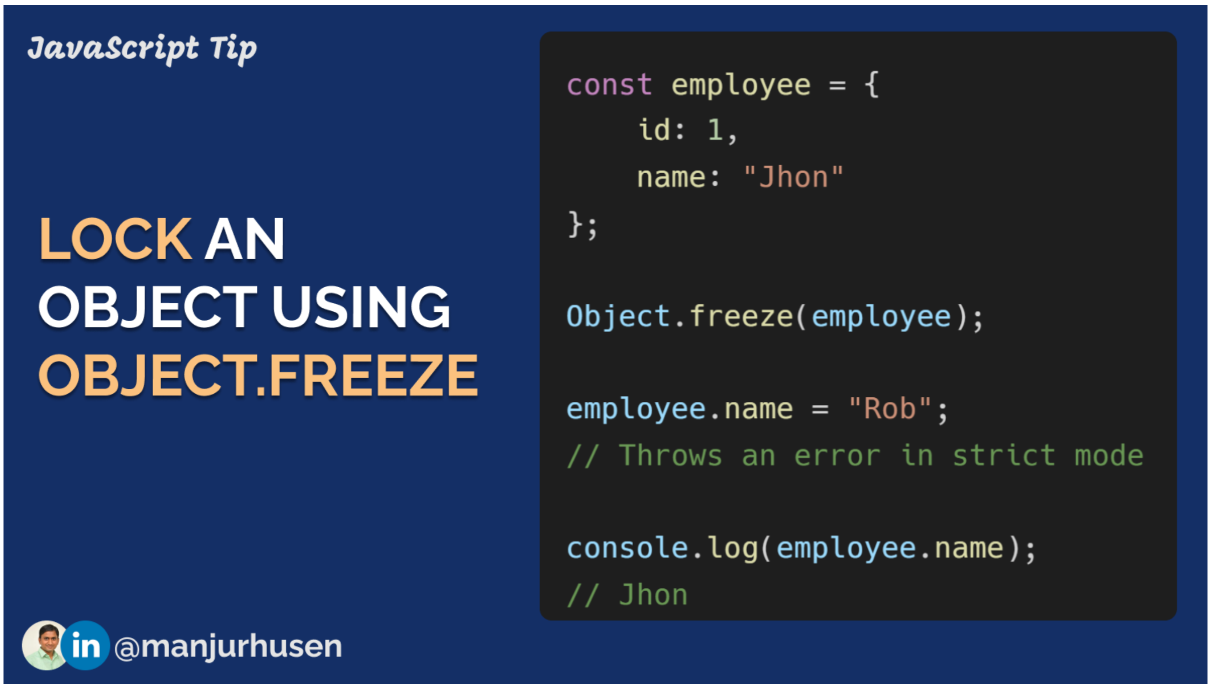
@manjurhusen

Object destructuring provides a unique way to neatly extract an object’s value into new variables.

To assign values to variables, declare the variables in curly brackets and assign the object like in code snippet.

To destructure into existing variables must surround the variables with *parentheses*.

Lock an object using the Object.freeze

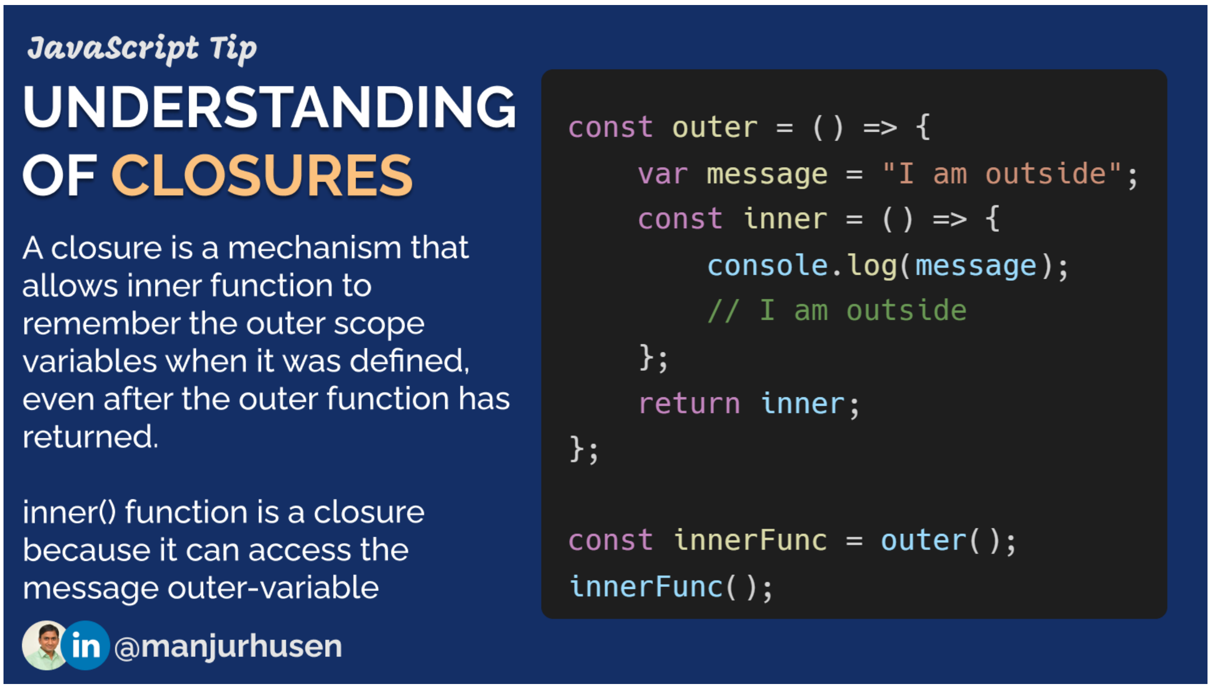


```
const employee = {
    id: 1,
    name: "Jhon"
};

Object.freeze(employee);

employee.name = "Rob";
// Throws an error in strict mode

console.log(employee.name);
// Jhon
```

The ***Object.freeze()*** method freezes an object. A frozen object can no longer be changed.

This method prevents new properties from being added and modification of existing properties.

Understanding of closures


JavaScript Tip

UNDERSTANDING OF CLOSURES

A closure is a mechanism that allows inner function to remember the outer scope variables when it was defined, even after the outer function has returned.

inner() function is a closure because it can access the message outer-variable

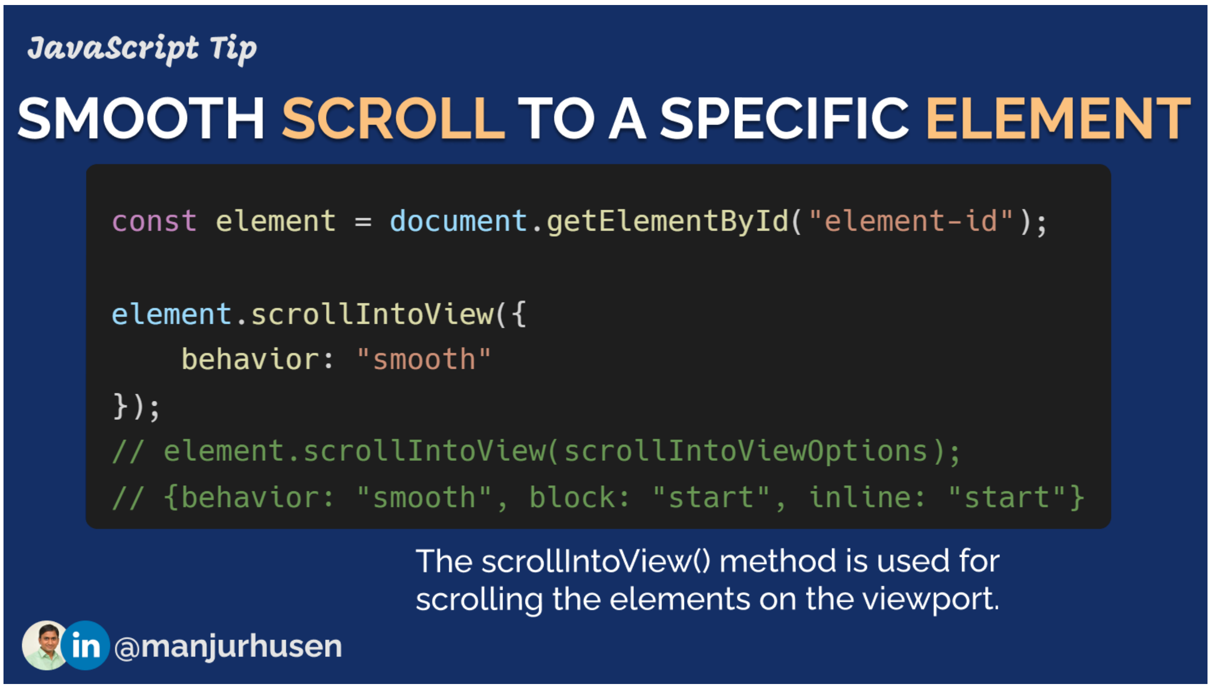
```javascript
const outer = () => {
    var message = "I am outside";
    const inner = () => {
        console.log(message);
        // I am outside
    };
    return inner;
};

const innerFunc = outer();
innerFunc();
```

A ***closure*** is a mechanism that allows the inner function to remember the outer scope variables when it was defined, even after the outer function has returned.

The closure has three scope chains:

- It can access its own scope means variables defined between its curly brackets ({ }).
- It can access the outer function’s variables.
- It can access the global variables.

Smooth scroll to a specific element


JavaScript Tip

SMOOTH SCROLL TO A SPECIFIC ELEMENT

```
const element = document.getElementById("element-id");

element.scrollIntoView({
    behavior: "smooth"
});
// element.scrollIntoView(scrollIntoViewOptions);
// {behavior: "smooth", block: "start", inline: "start"}
```

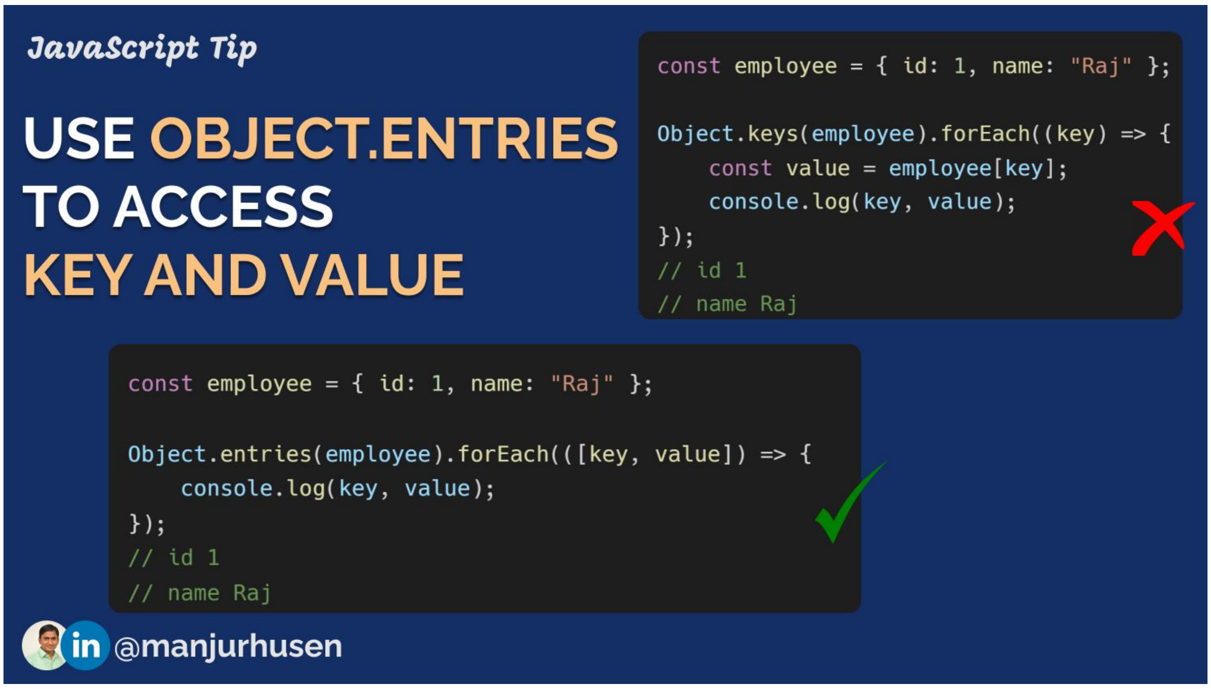
The scrollIntoView() method is used for scrolling the elements on the viewport.

@manjurhusen

The ***Element.scrollIntoView()*** method scrolls the specified element into the viewing portion of the window.

It provides the *behavior* option for smooth scrolling.

Use Object.entries to access key and value



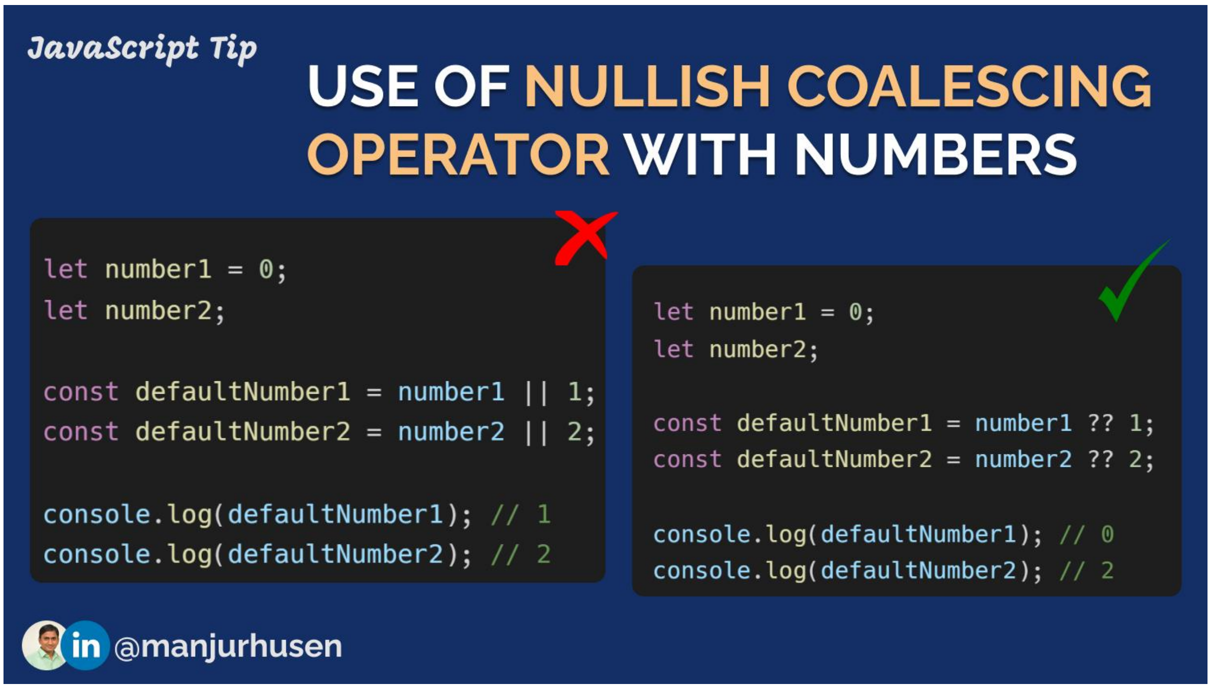
```
const employee = { id: 1, name: "Raj" };

Object.keys(employee).forEach((key) => {
    const value = employee[key];
    console.log(key, value);
});
// id 1
// name Raj
```

```
const employee = { id: 1, name: "Raj" };

Object.entries(employee).forEach(([key, value]) => {
    console.log(key, value);
});
// id 1
// name Raj
```

The ***Object.entries()*** method is used to return an array of a given object's own enumerable property [key, value] pairs.

The order of the properties is the same as in an object.

Use of nullish coalescing operator with numbers



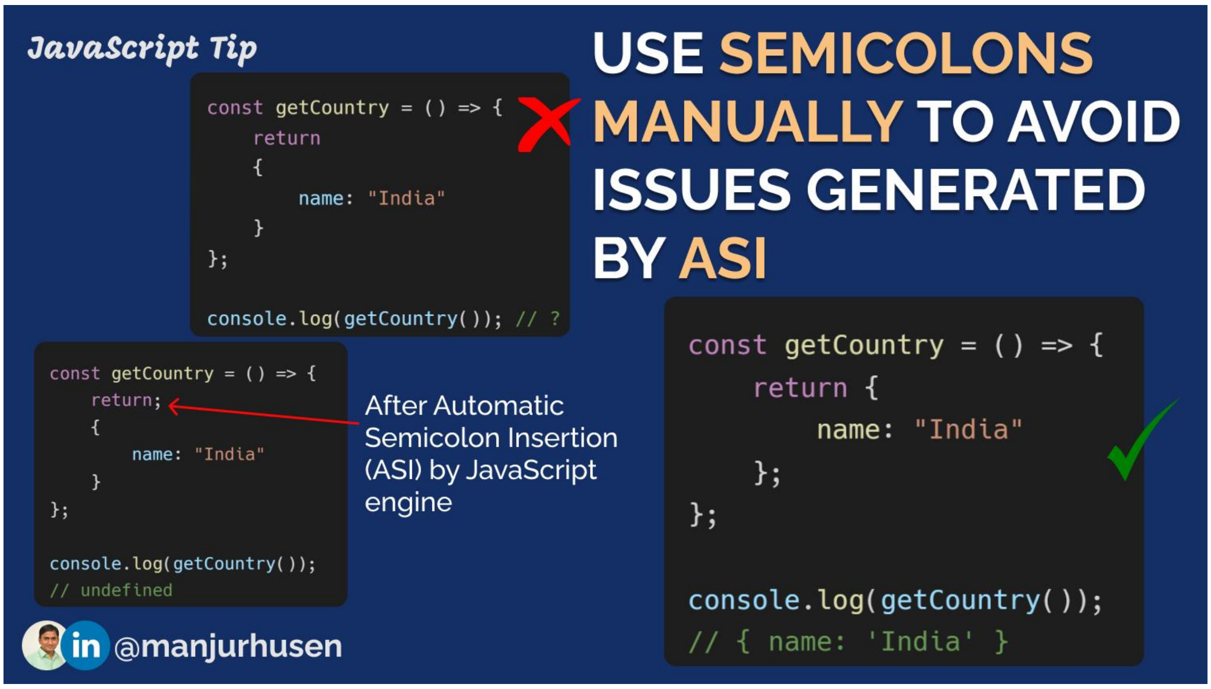
```
let number1 = 0;
let number2;

const defaultNumber1 = number1 || 1;
const defaultNumber2 = number2 || 2;

console.log(defaultNumber1); // 1
console.log(defaultNumber2); // 2
```

```
let number1 = 0;
let number2;

const defaultNumber1 = number1 ?? 1;
const defaultNumber2 = number2 ?? 2;

console.log(defaultNumber1); // 0
console.log(defaultNumber2); // 2
```

A ***Nullish*** value is a value that is either null or undefined.

The ***Nullish Coalescing Operator (??)*** is a logical operator that accepts two values and returns the second value if the first one is null or undefined and otherwise returns the first value.

Use semicolons manually to avoid issues generated by ASI


JavaScript Tip
USE SEMICOLONS MANUALLY TO AVOID ISSUES GENERATED BY ASI
After Automatic Semicolon Insertion (ASI) by JavaScript engine
@manjurhusen

```
const getCountry = () => {
    return
    {
        name: "India"
    }
};

console.log(getCountry()); // ?
```

```
const getCountry = () => {
    return;
    {
        name: "India"
    }
};

console.log(getCountry());
// undefined
```

```
const getCountry = () => {
    return {
        name: "India"
    };
};

console.log(getCountry());
// { name: 'India' }
```

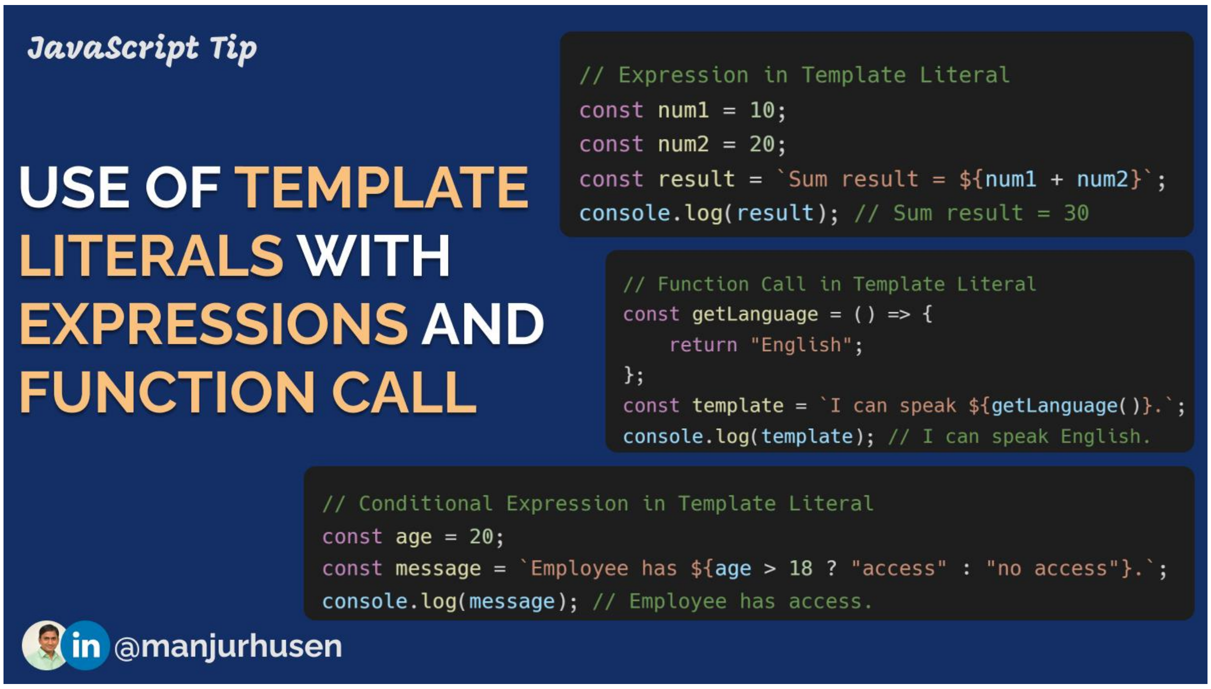
ASI stands for *Automatic Semicolon Insertion*.

In JavaScript, semicolons are optional. JavaScript Engine automatically inserts a semicolon, where it is required.

If the code is not formatted correctly like in the above example, JavaScript Engine will add a semicolon to the end of the return statement and consider that no value is returned. So, it returns as undefined.

You should not depend on the ASI. If ASI fails and you are missing semicolons, the code will fail.

Use of template literals with expressions and function call



```javascript
// Expression in Template Literal
const num1 = 10;
const num2 = 20;
const result = `Sum result = ${num1 + num2}`;
console.log(result); // Sum result = 30
```

```javascript
// Function Call in Template Literal
const getLanguage = () => {
    return "English";
};
const template = `I can speak ${getLanguage()}.`;
console.log(template); // I can speak English.
```

```javascript
// Conditional Expression in Template Literal
const age = 20;
const message = `Employee has ${age > 18 ? "access" : "no access"}.`;
console.log(message); // Employee has access.
```

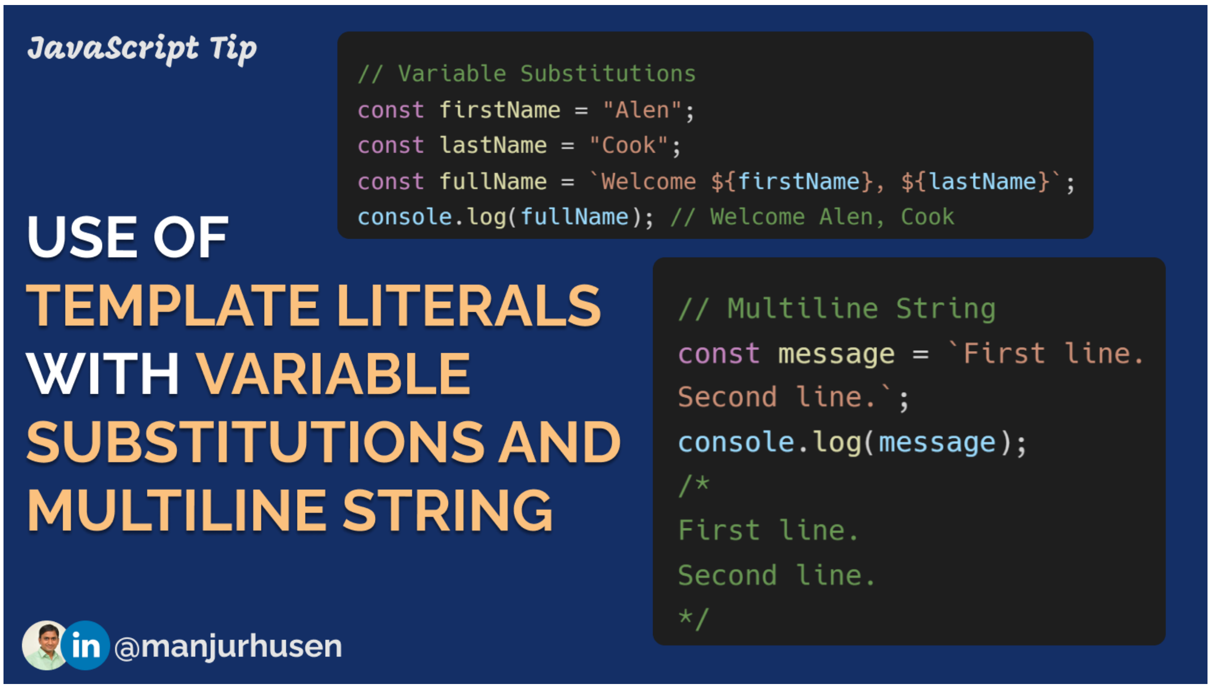
Template Literals use back-ticks (``) instead of single ('') or double ("") quotes.

Template literals provide an easy way to interpolate variables and expressions into strings.

Template literals allow *expressions* and *functions* in strings.

Using template literal means not only less code but higher readability also.

Use of template literals with variable substitutions and multiline string


JavaScript Tip
USE OF TEMPLATE LITERALS WITH VARIABLE SUBSTITUTIONS AND MULTILINE STRING
@manjurhusen

```
// Variable Substitutions
const firstName = "Alen";
const lastName = "Cook";
const fullName = `Welcome ${firstName}, ${lastName}`;
console.log(fullName); // Welcome Alen, Cook
```

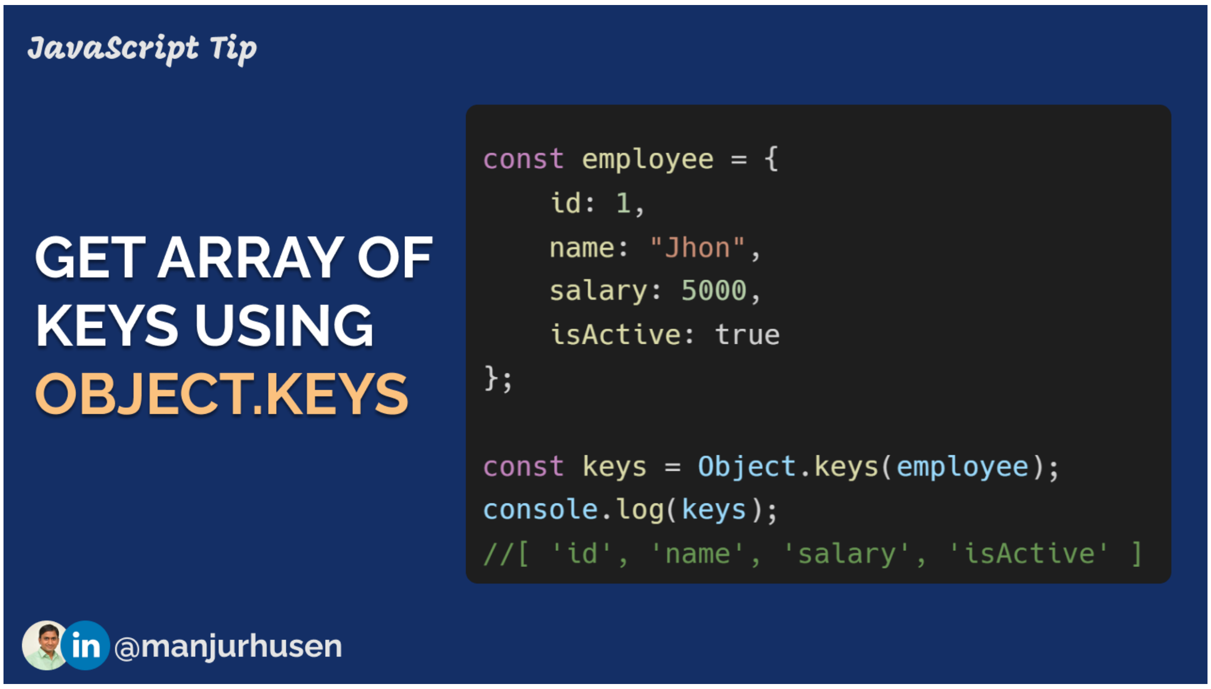
```
// Multiline String
const message = `First line.
Second line.`;
console.log(message);
/*
First line.
Second line.
*/
```

Template Literals use back-ticks (``) instead of single ('') or double ("") quotes.

Template literals provide an easy way to interpolate *variables* and *expressions* into strings. You can do it using the *${...}* syntax.

Template literals make *multiline* strings much simpler.

Get an array of keys using Object.keys


JavaScript Tip

GET ARRAY OF KEYS USING OBJECT.KEYS

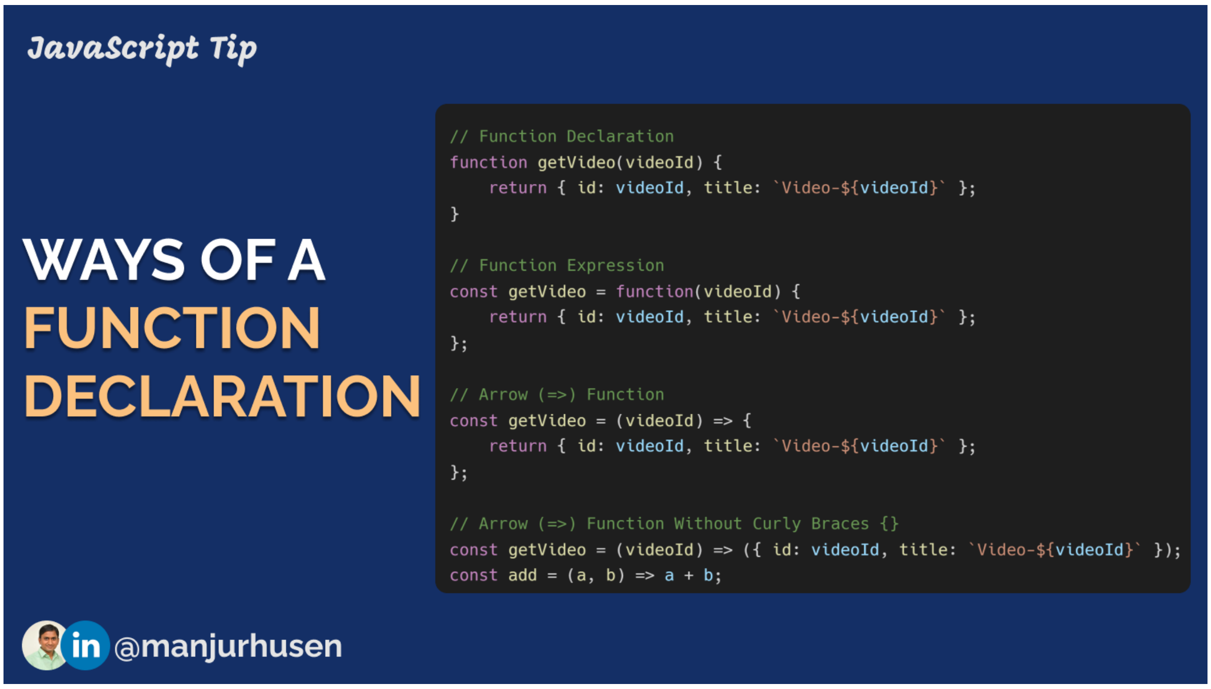
```
const employee = {
    id: 1,
    name: "Jhon",
    salary: 5000,
    isActive: true
};

const keys = Object.keys(employee);
console.log(keys);
//[ 'id', 'name', 'salary', 'isActive' ]
```

@manjurhusen

The ***Object.keys()*** returns an array of a given object's own enumerable property names.

The ordering of the properties is the same as that when looping over them manually.

Ways of a function declaration



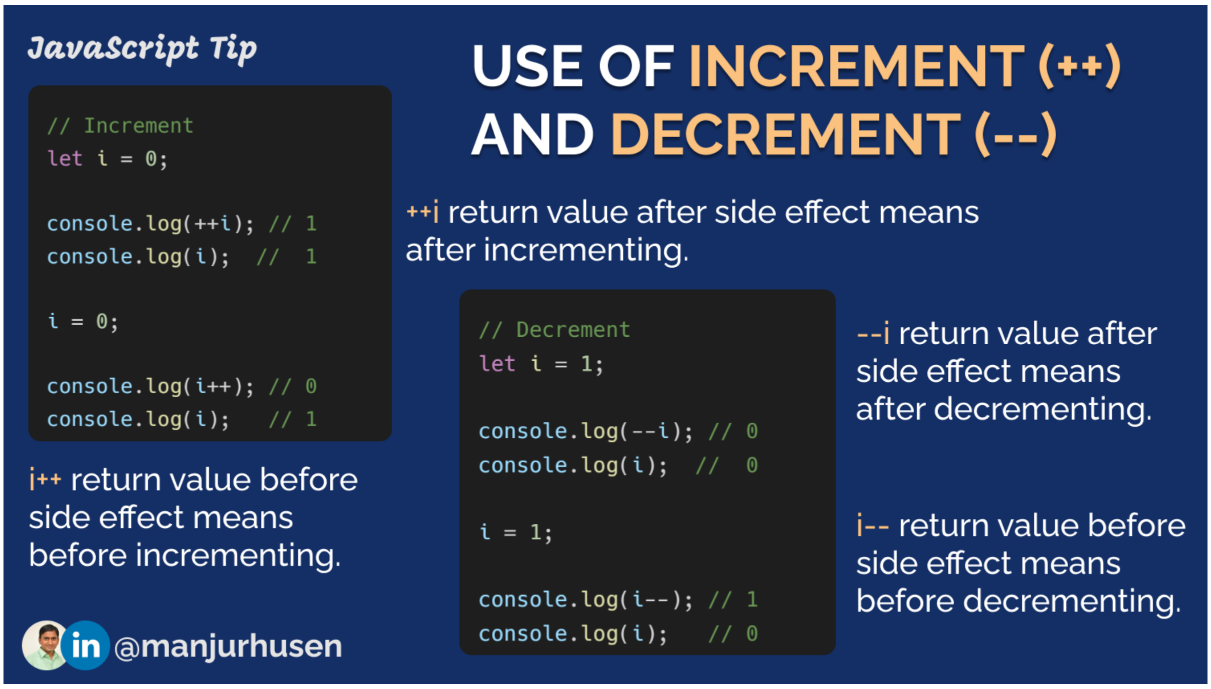
```javascript
// Function Declaration
function getVideo(videoId) {
    return { id: videoId, title: `Video-${videoId}` };
}

// Function Expression
const getVideo = function(videoId) {
    return { id: videoId, title: `Video-${videoId}` };
};

// Arrow (=>) Function
const getVideo = (videoId) => {
    return { id: videoId, title: `Video-${videoId}` };
};

// Arrow (=>) Function Without Curly Braces {}
const getVideo = (videoId) => ({ id: videoId, title: `Video-${videoId}` });
const add = (a, b) => a + b;
```

Functions are one of the fundamental building blocks in JavaScript.

Following are the different ways to write functions.

- Function declaration
- Function Expression
- Arrow (=>) function
- Arrow (=>) function without curly braces ({}) – (Use only for a single statement of code)

Use of increment (++) and decrement (--)


JavaScript Tip

USE OF INCREMENT (++) AND DECREMENT (--)

```
// Increment
let i = 0;

console.log(++i); // 1
console.log(i);   //  1

i = 0;

console.log(i++); // 0
console.log(i);    // 1
```

++i return value after side effect means after incrementing.

i++ return value before side effect means before incrementing.

```
// Decrement
let i = 1;

console.log(--i); // 0
console.log(i);   //  0

i = 1;

console.log(i--); // 1
console.log(i);    // 0
```

--i return value after side effect means after decrementing.

i-- return value before side effect means before decrementing.

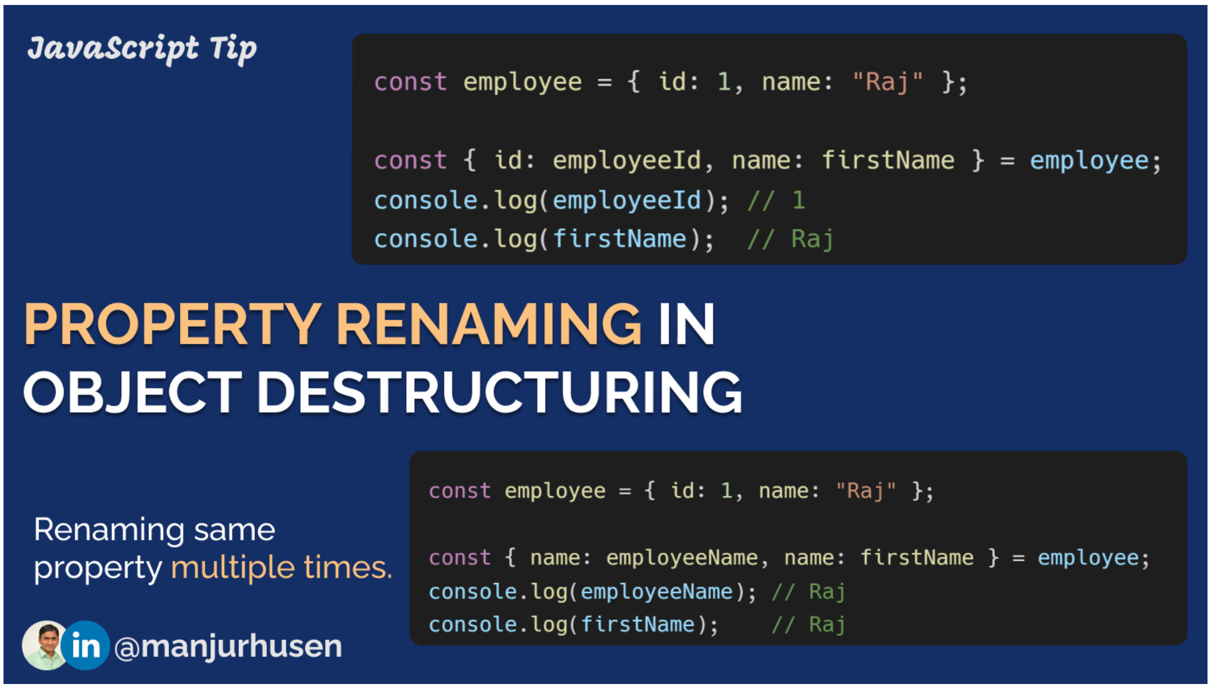
The ***increment operator (++)*** adds one (+1) to its operand and returns a value. The increment (++) operator can be used before or after the operand.

Increment Syntax: ***i++*** or ***++i***

The ***decrement operator (--)*** subtracts one (-1) to its operand and returns a value. The decrement (--) operator can be used before or after the operand.

Decrement Syntax: ***i--*** or ***--i***

Property renaming in object destructuring


JavaScript Tip

```
const employee = { id: 1, name: "Raj" };

const { id: employeeId, name: firstName } = employee;
console.log(employeeId); // 1
console.log(firstName);  // Raj
```

PROPERTY RENAMING IN OBJECT DESTRUCTURING

Renaming same property multiple times.

```
const employee = { id: 1, name: "Raj" };

const { name: employeeName, name: firstName } = employee;
console.log(employeeName); // Raj
console.log(firstName);    // Raj
```

@manjurhusen

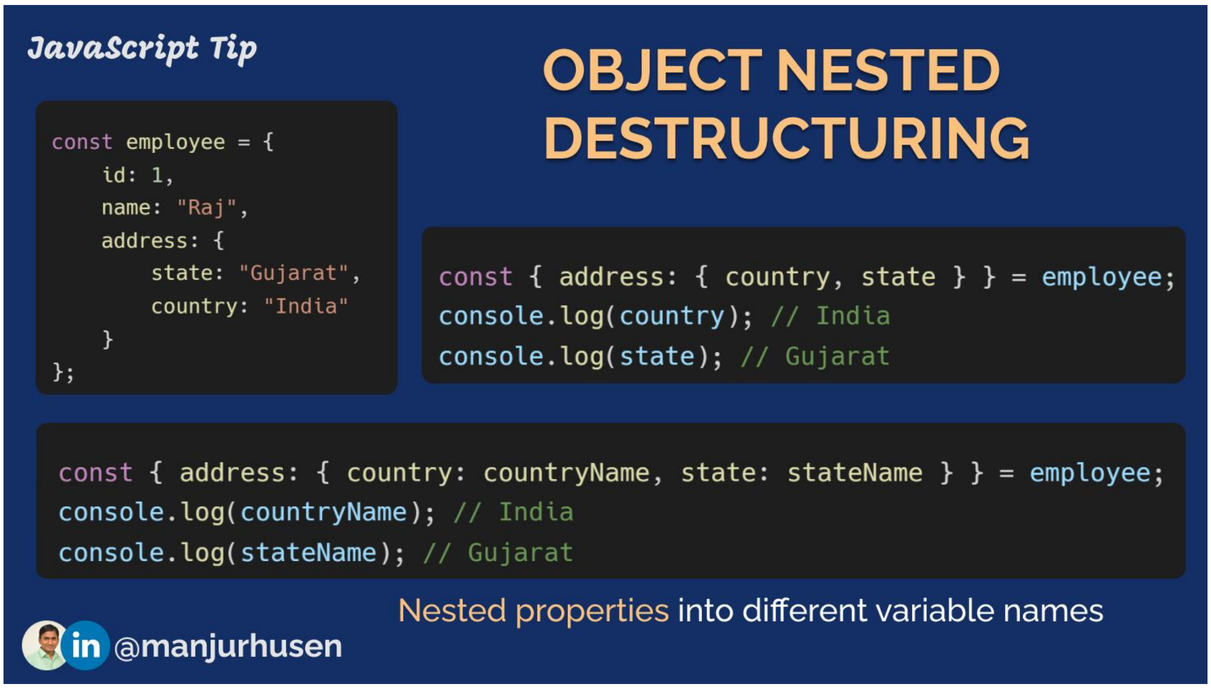
Object destructuring provides a unique way to neatly extract an object’s value into new variables.

Sometimes an object contains some properties, but you want to access it and rename it.

When renaming a variable in object destructuring, the left-hand side will be the original field in the object, and the right-hand side will be the name you provide to rename it to.

It is also possible to destructure the same property multiple times into different variable names like in code snippet.

Object nested destructuring


JavaScript Tip
OBJECT NESTED DESTRUCTURING
Nested properties into different variable names
@manjurhusen

```
const employee = {
    id: 1,
    name: "Raj",
    address: {
        state: "Gujarat",
        country: "India"
    }
};
```

```
const { address: { country, state } } = employee;
console.log(country); // India
console.log(state); // Gujarat
```

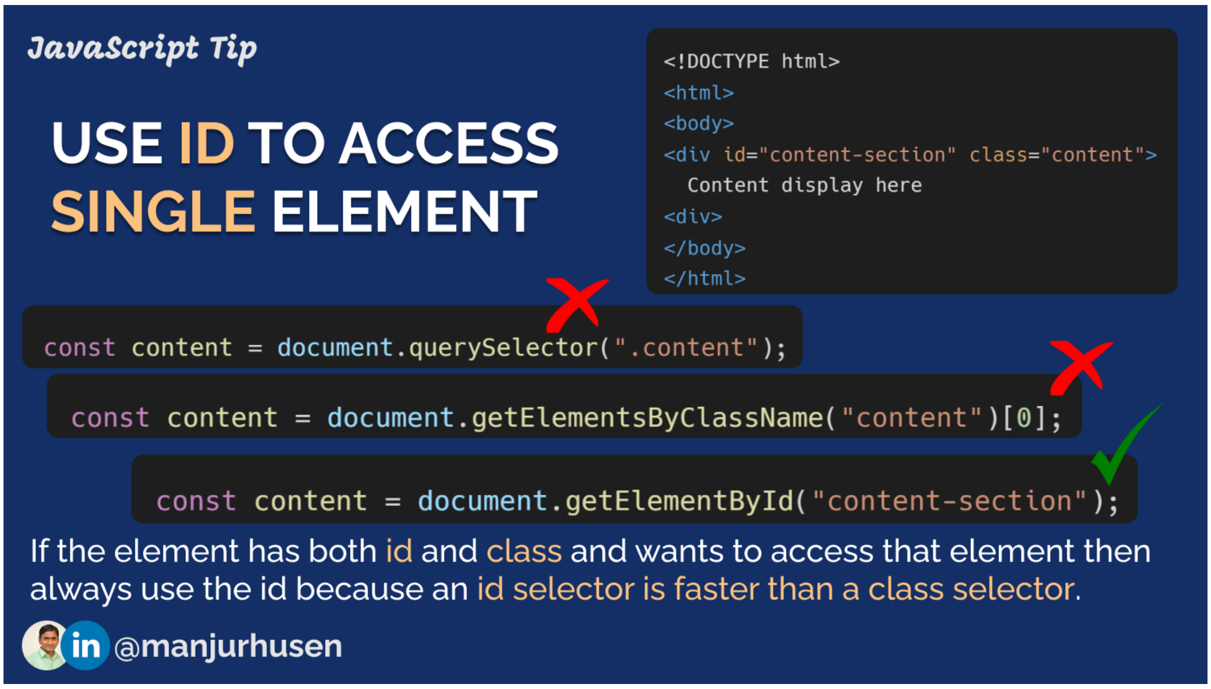
```
const { address: { country: countryName, state: stateName } } = employee;
console.log(countryName); // India
console.log(stateName); // Gujarat
```

With ***destructuring***, we can quickly extract properties or data from objects into separate variables.

You need to give a complete path when you have to destructure a nested property.

Destructuring an object does not modify the original object.

Use id to find a single element


JavaScript Tip

USE ID TO ACCESS SINGLE ELEMENT

```
<!DOCTYPE html>
<html>
<body>
<div id="content-section" class="content">
  Content display here
<div>
</body>
</html>
```

```
const content = document.querySelector(".content");
```

```
const content = document.getElementsByClassName("content")[0];
```

```
const content = document.getElementById("content-section");
```

If the element has both id and class and wants to access that element then always use the id because an id selector is faster than a class selector.

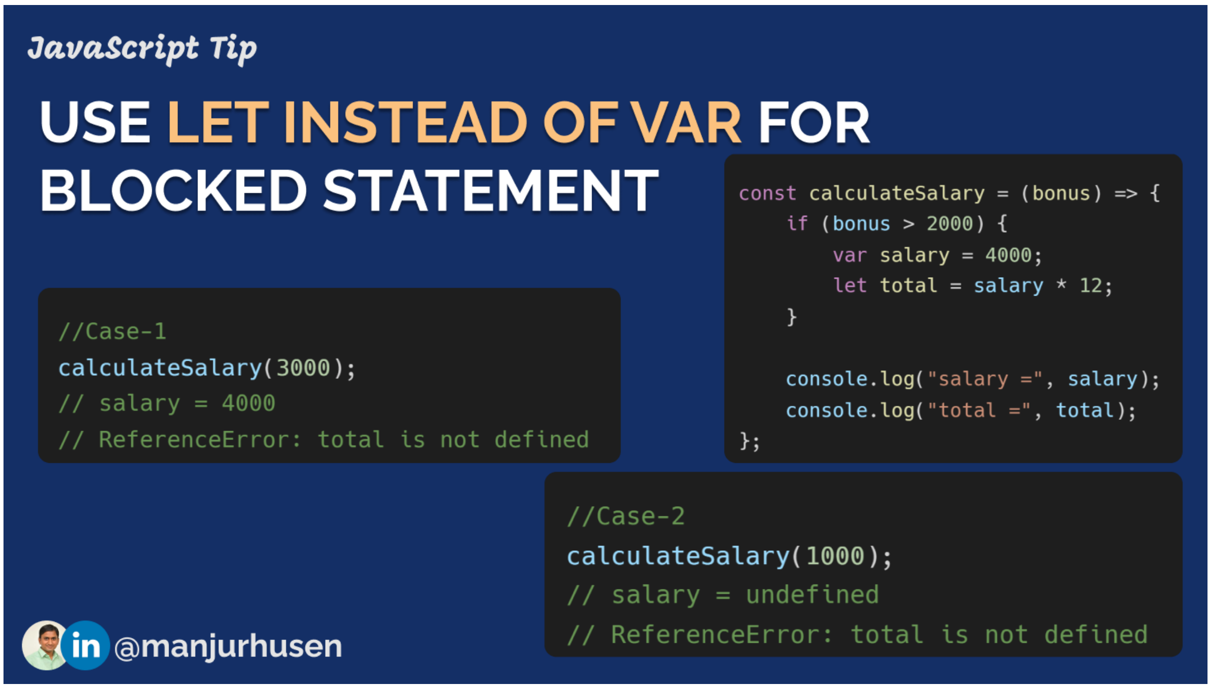
Never use the same id for multiple elements on the same HTML page.

The ***getElementById()*** method returns an element object.

The ***getElementById()*** method returns null if the element does not exist.

When you want to access any element, please use element-id if exists. Access element by id is faster than class.

Use let instead of var for blocked statement


JavaScript Tip

USE LET INSTEAD OF VAR FOR BLOCKED STATEMENT

```
const calculateSalary = (bonus) => {
    if (bonus > 2000) {
        var salary = 4000;
        let total = salary * 12;
    }

    console.log("salary =", salary);
    console.log("total =", total);
};
```

```
//Case-1
calculateSalary(3000);
// salary = 4000
// ReferenceError: total is not defined
```

```
//Case-2
calculateSalary(1000);
// salary = undefined
// ReferenceError: total is not defined
```

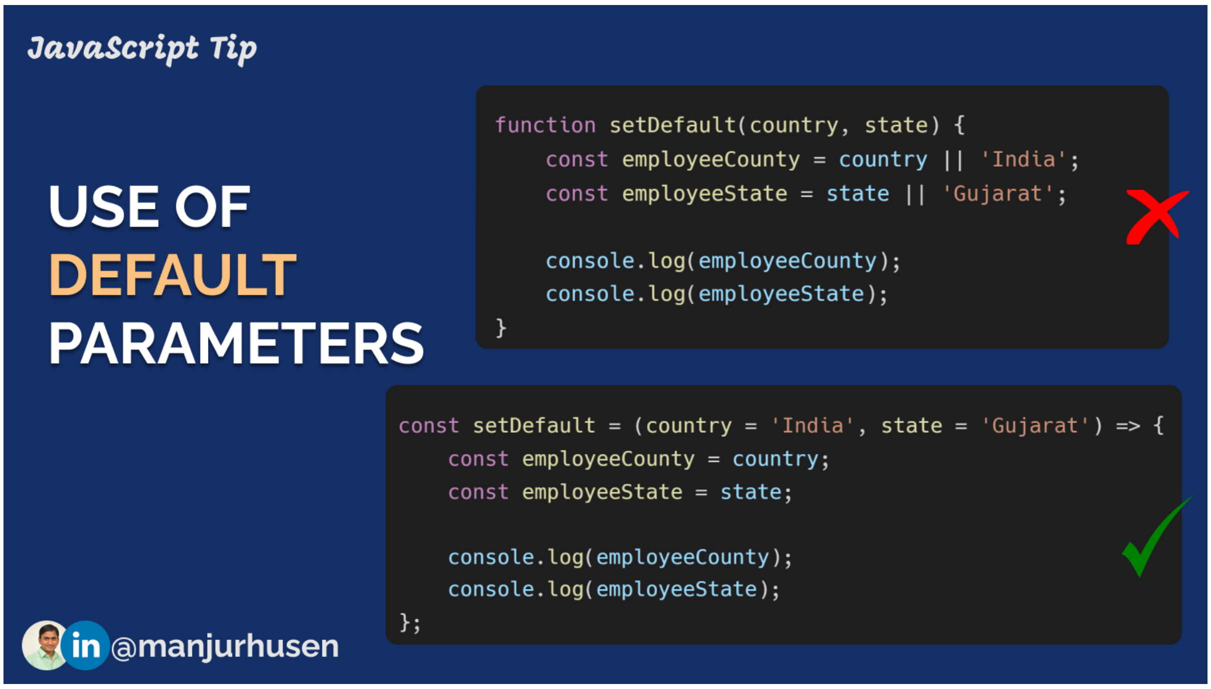
@manjurhusen

Scope means where these variables are available for use. The ***var*** declarations are globally scoped or function/locally scoped.

Using ***var*** is the oldest method of variable declaration in JavaScript. A variable declared using *var* is function scoped when it is declared within a function.

A ***let*** variable is scoped to the immediate enclosing block denoted by *curly braces* ({ }). You cannot access the *let* variable outside of its scope. The above code snippet shows the behavior of *var* and *let* variable.

Use of default parameters



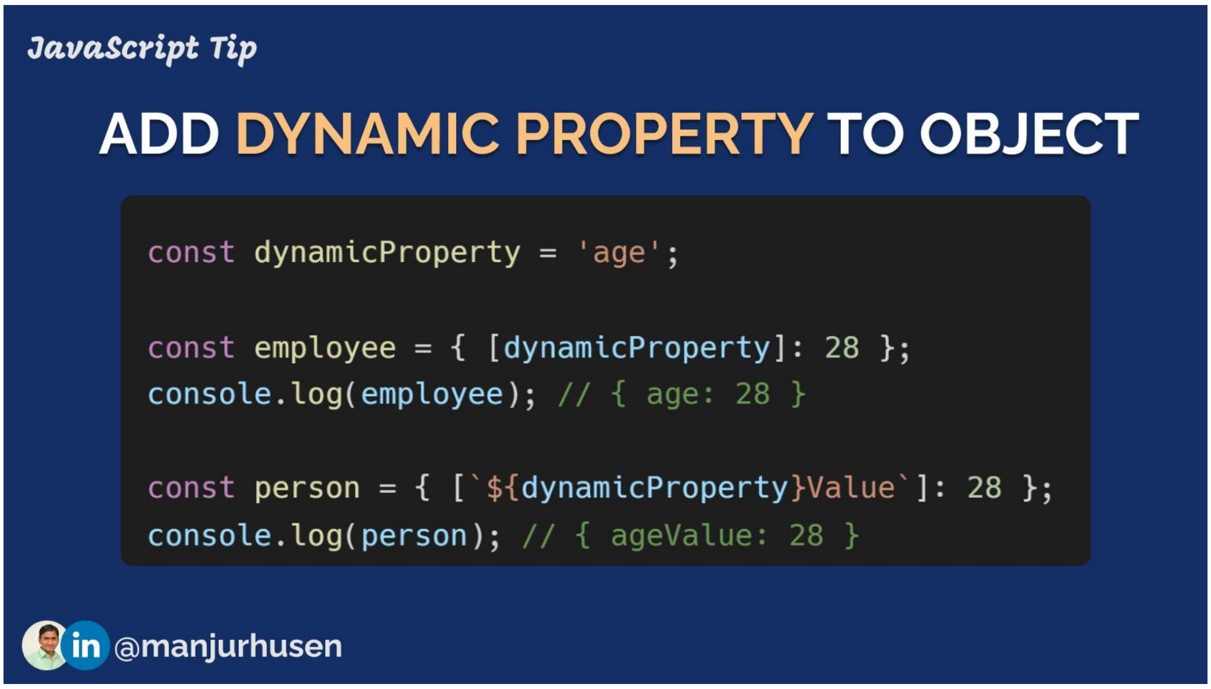
```
function setDefault(country, state) {
    const employeeCounty = country || 'India';
    const employeeState = state || 'Gujarat';

    console.log(employeeCounty);
    console.log(employeeState);
}
```

```
const setDefault = (country = 'India', state = 'Gujarat') => {
    const employeeCounty = country;
    const employeeState = state;

    console.log(employeeCounty);
    console.log(employeeState);
};
```

Default function parameters allow named parameters to be initialized with *default values* if no value or *undefined* is passed.

ES6 provides an easier way to set the *default values* for the function parameters. Use the *assignment operator (=)* and the default value after the parameter name to set a default value for that parameter.

Add dynamic property to an object

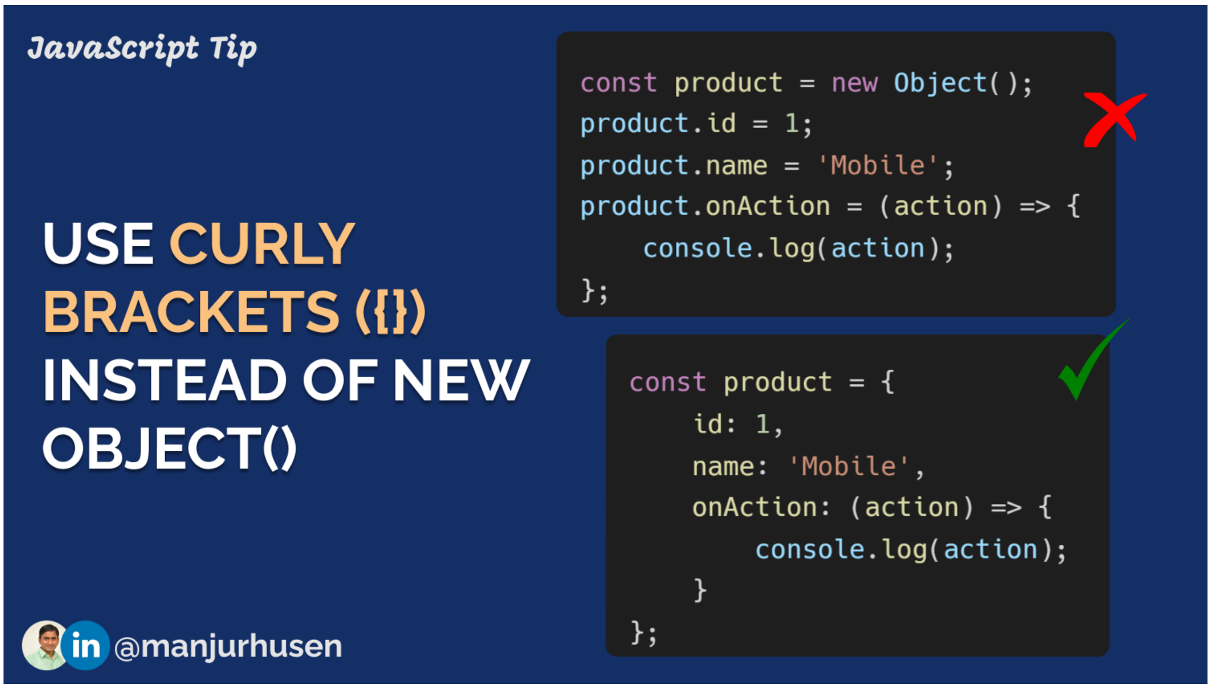


```javascript
const dynamicProperty = 'age';

const employee = { [dynamicProperty]: 28 };
console.log(employee); // { age: 28 }

const person = { [`${dynamicProperty}Value`]: 28 };
console.log(person); // { ageValue: 28 }
```

ES6 provides an easy way to create a *dynamic property* in an object.

We can simply pass the property name in *square brackets ([])* which we want to make property in the object.

Use curly brackets ({}) instead of new Object()



```
const product = new Object();
product.id = 1;
product.name = 'Mobile';
product.onAction = (action) => {
    console.log(action);
};
```

```
const product = {
    id: 1,
    name: 'Mobile',
    onAction: (action) => {
        console.log(action);
    }
};
```

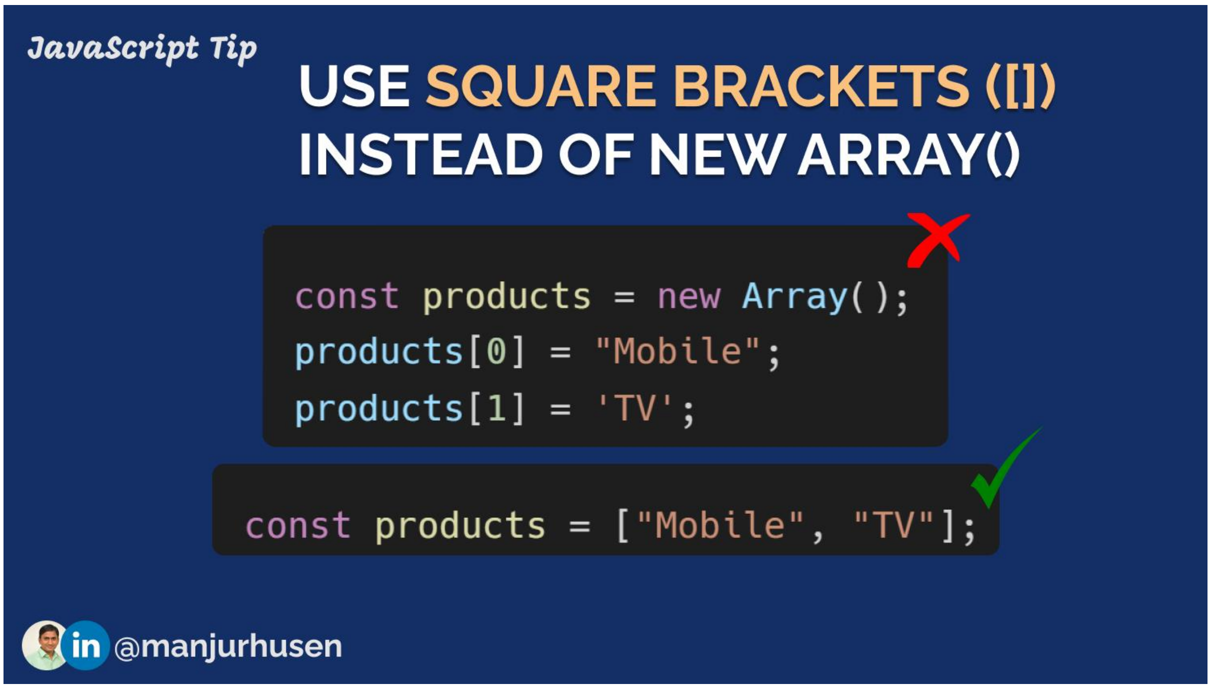
Objects can be initialized using *new Object(), Object.create()*, or using the *literal notation*.

You can use curly brackets ({}) to declare objects in JavaScript. Creating a new object this way is called ***object literal notation***.

The advantage of the literal notation is, that you are able to quickly create objects with properties inside the *curly brackets ({})*. You notate a list of key: value pairs delimited by commas.

Better and clean way.

Use square brackets ([]) instead of new Array()



```
const products = new Array();
products[0] = "Mobile";
products[1] = 'TV';
```

```
const products = ["Mobile", "TV"];
```

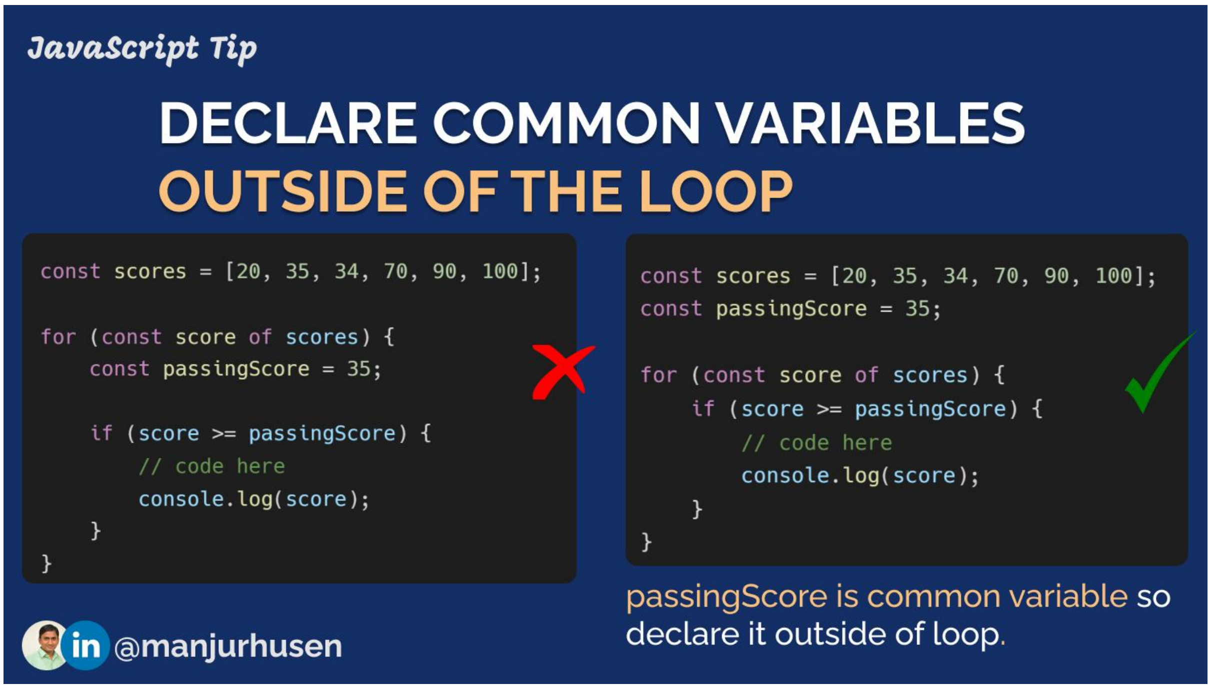
Arrays can be created using the *new Array()*, but in the same way, they can be created using *literal notation* also.

You can use *square brackets ([])* to declare arrays in JavaScript. Creating an array this way is called array literal notation.

The advantage of the array literal notation is, that you are able to quickly create arrays.

Better and clean way.

Declare common variables outside of the loop



JavaScript Tip

DECLARE COMMON VARIABLES OUTSIDE OF THE LOOP

```
const scores = [20, 35, 34, 70, 90, 100];

for (const score of scores) {
    const passingScore = 35;

    if (score >= passingScore) {
        // code here
        console.log(score);
    }
}
```

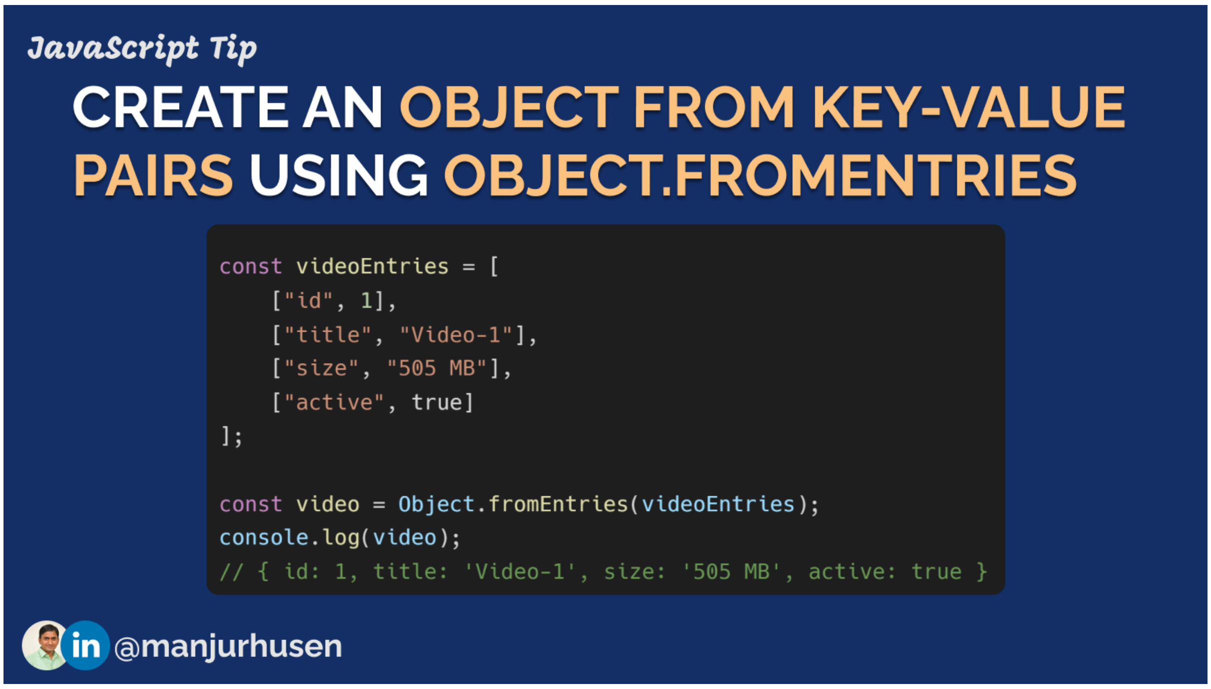
```
const scores = [20, 35, 34, 70, 90, 100];
const passingScore = 35;

for (const score of scores) {
    if (score >= passingScore) {
        // code here
        console.log(score);
    }
}
```

passingScore is common variable so declare it outside of loop.

@manjurhusen

Variables that are not going to reassign in the loop must be declared outside of the loop, otherwise, they will be created again and assigned the same value every time.

Create an object from key-value pairs using Object.fromEntries

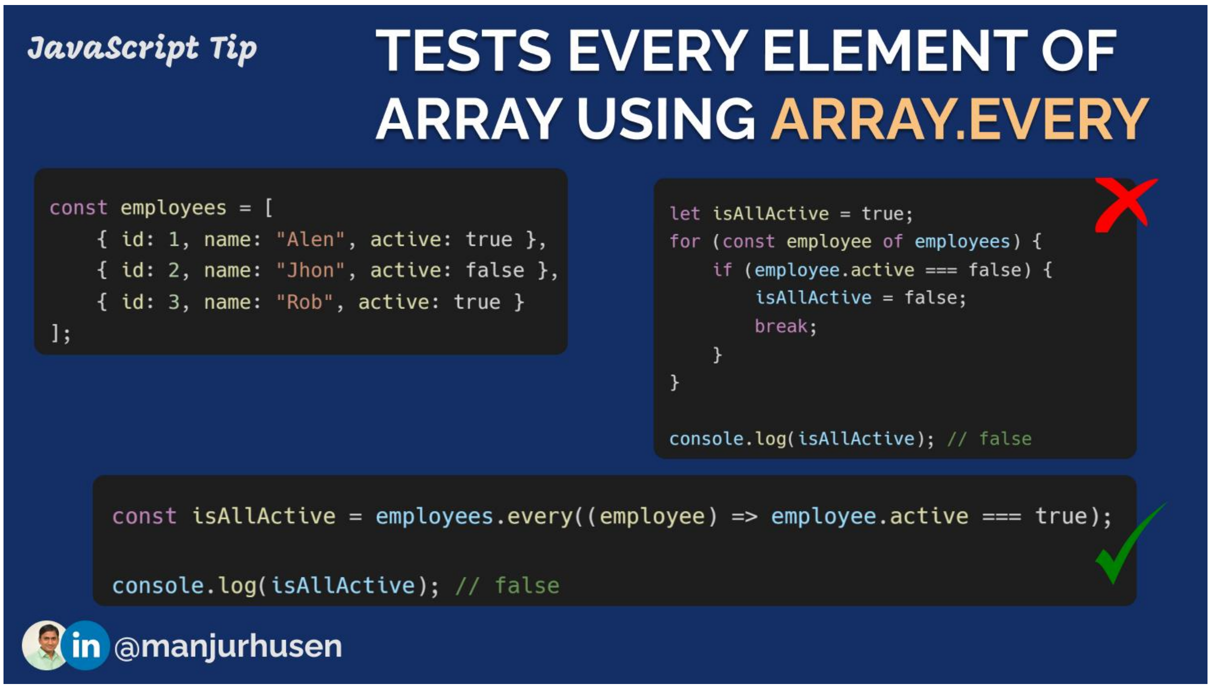

JavaScript Tip
CREATE AN OBJECT FROM KEY-VALUE PAIRS USING OBJECT.FROMENTRIES
@manjurhusen

```
const videoEntries = [
    ["id", 1],
    ["title", "Video-1"],
    ["size", "505 MB"],
    ["active", true]
];

const video = Object.fromEntries(videoEntries);
console.log(video);
// { id: 1, title: 'Video-1', size: '505 MB', active: true }
```

The ***Object.fromEntries()*** method transforms a list of key-value pairs into an object.

Object.fromEntries() performs the reverse of *Object.entries()*.

Tests every element of the array using Array.every



```
const employees = [
    { id: 1, name: "Alen", active: true },
    { id: 2, name: "Jhon", active: false },
    { id: 3, name: "Rob", active: true }
];
```

```
let isAllActive = true;
for (const employee of employees) {
    if (employee.active === false) {
        isAllActive = false;
        break;
    }
}

console.log(isAllActive); // false
```

```
const isAllActive = employees.every((employee) => employee.active === true);

console.log(isAllActive); // false
```

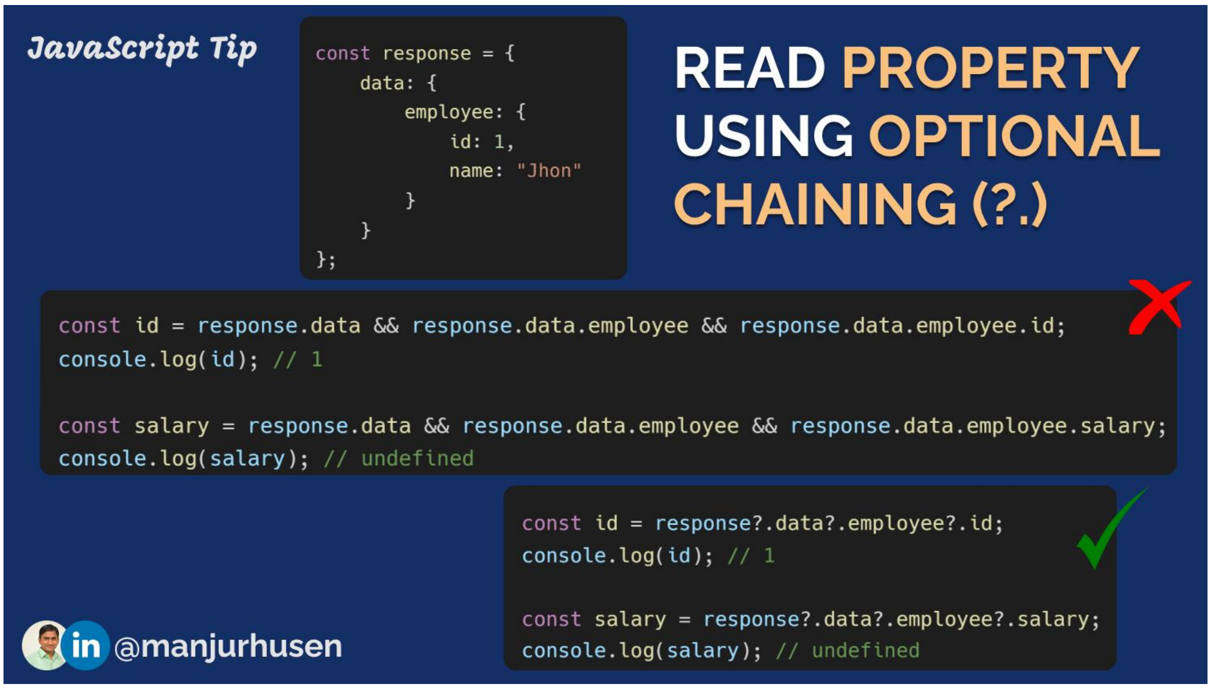
The Array ***every()*** method checks whether all the array elements pass the test implemented by the provided function.

It returns true if the function returns true for all elements.

It returns false if the function returns false for one element. When *every()* finds a false result, it will stop the loop and continue no more which improves the performance.

The *every()* method does not change the original array.

Read property using optional chaining (?.)



```
const response = {
    data: {
        employee: {
            id: 1,
            name: "Jhon"
        }
    }
};
```

```
const id = response.data && response.data.employee && response.data.employee.id;
console.log(id); // 1

const salary = response.data && response.data.employee && response.data.employee.salary;
console.log(salary); // undefined
```

```
const id = response?.data?.employee?.id;
console.log(id); // 1

const salary = response?.data?.employee?.salary;
console.log(salary); // undefined
```

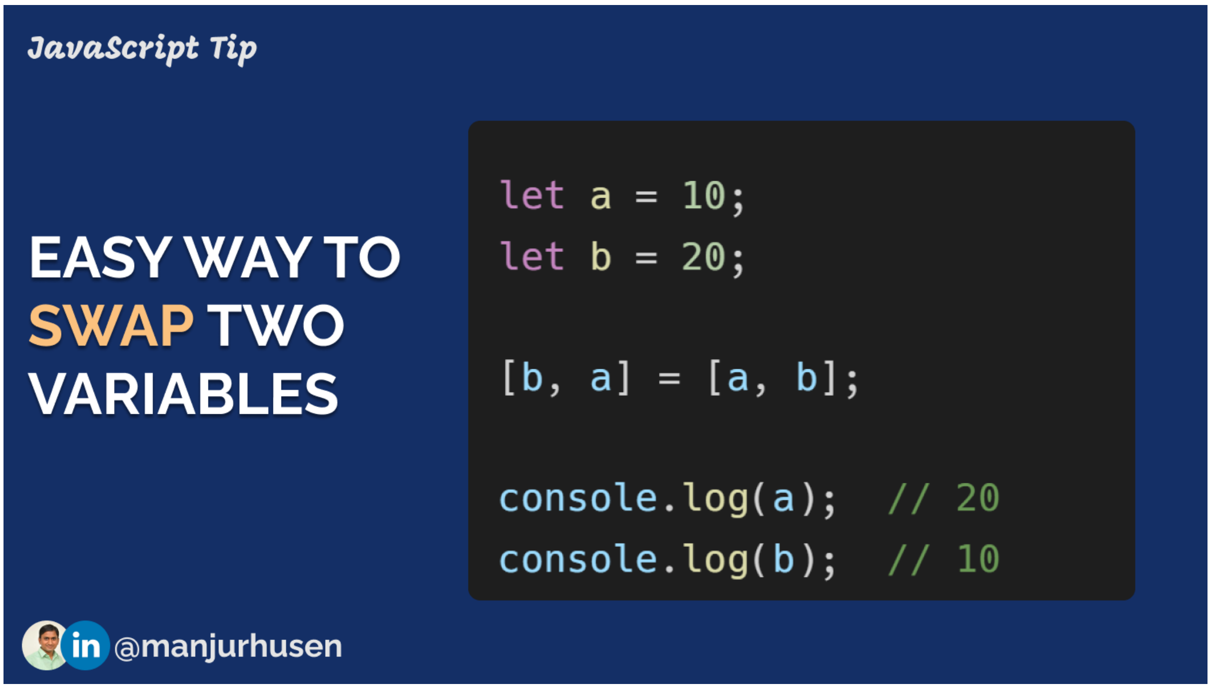
The ***optional chaining operator (?.)*** is a secure way to read nested object properties, even if an intermediate property doesn’t exist.

The *optional chaining operator (?.)* stops the evaluation if the value before *?.* is nullish (undefined or null) and returns undefined.

It prevents writing boilerplate code.

Less and clean code.

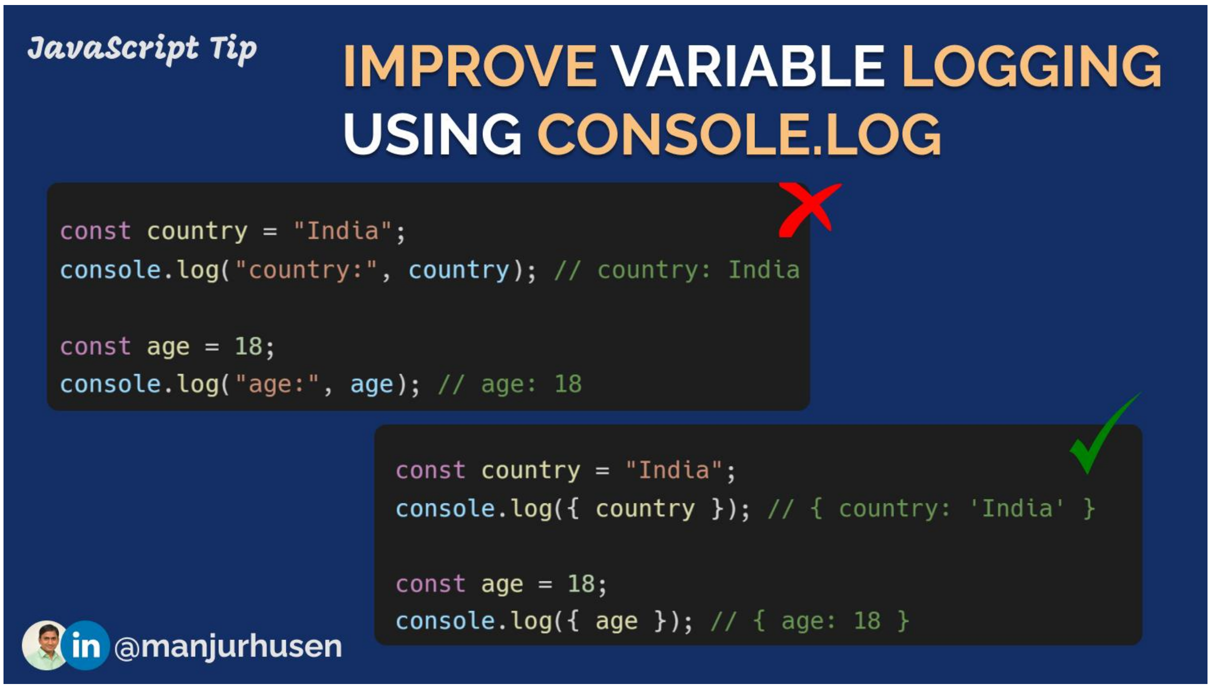
Easy way to swap two variables



```javascript
let a = 10;
let b = 20;

[b, a] = [a, b];

console.log(a);  // 20
console.log(b);  // 10
```

Use *destructuring assignment* approach because it is short and expressive. Swapping is performed in just one line statement. It works with any data type like numbers, strings, booleans, or objects.

Improve variable logging using console.log



```
const country = "India";
console.log("country:", country); // country: India

const age = 18;
console.log("age:", age); // age: 18
```

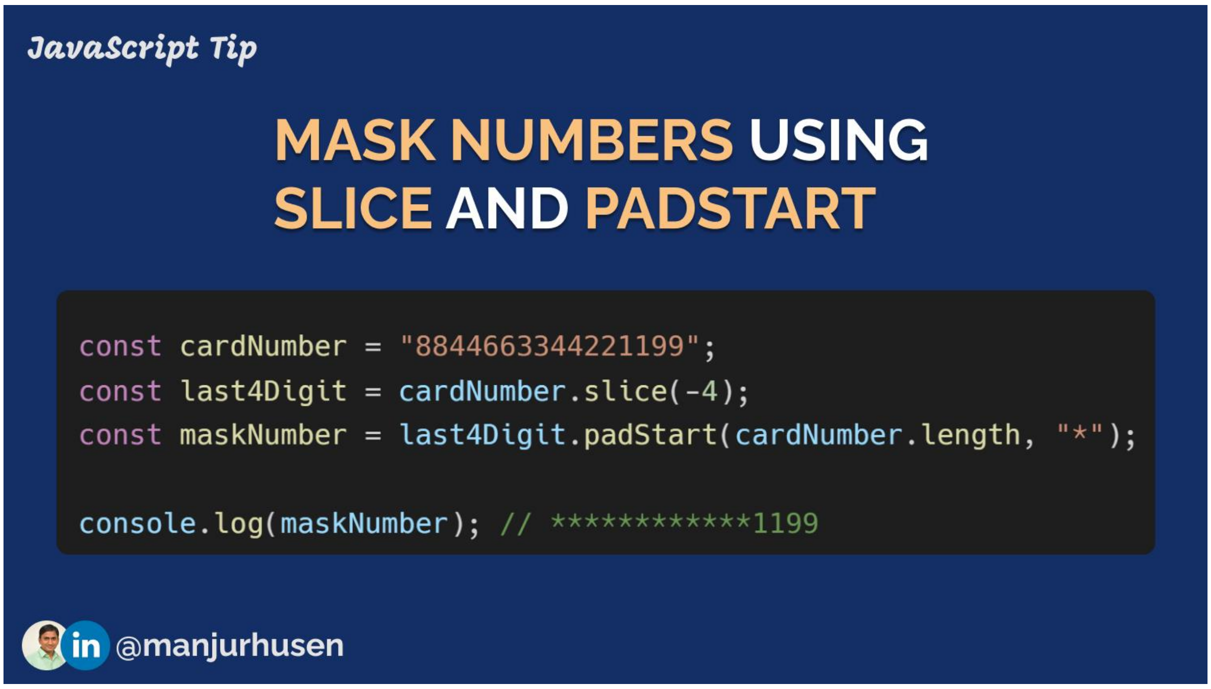
```
const country = "India";
console.log({ country }); // { country: 'India' }

const age = 18;
console.log({ age }); // { age: 18 }
```

In JavaScript, we use ***console.log()*** to log the variables or messages. Sometimes it is difficult to understand what variable corresponds to a log in the console when too many variable logs.

To log the variable, wrap the variable into a pair of *curly brackets {variable-name}.*

It will improve readability.

Mask numbers using slice and padStart



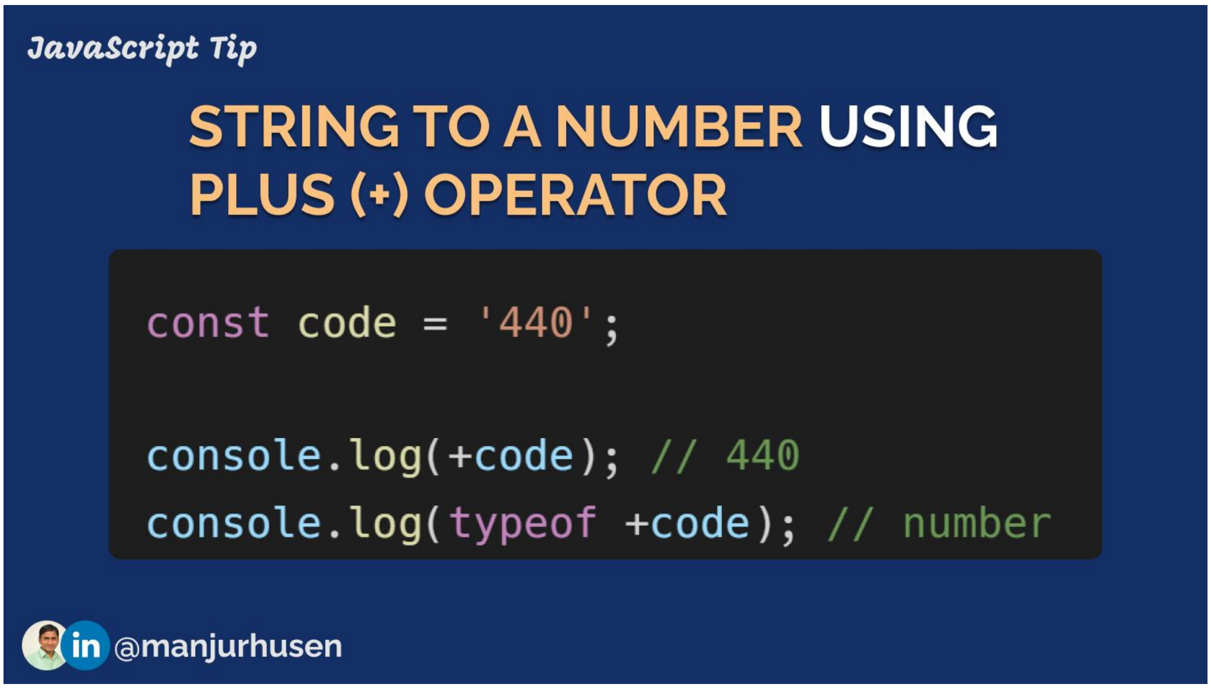
```
const cardNumber = "8844663344221199";
const last4Digit = cardNumber.slice(-4);
const maskNumber = last4Digit.padStart(cardNumber.length, "*");

console.log(maskNumber); // ************1199
```

The ***slice()*** method returns selected elements in an array, as a new array. Negative numbers select from the end of the array.

The ***padStart()*** method pads the current string with another string until the resulting string reaches the given length. The padding is applied from the start of the current string.

Masking is possible with less code.

String to a number using the plus (+) operator



```
const code = '440';

console.log(+code); // 440
console.log(typeof +code); // number
```

The ***unary plus operator (+)*** is the fastest and preferred way of converting something into a number.