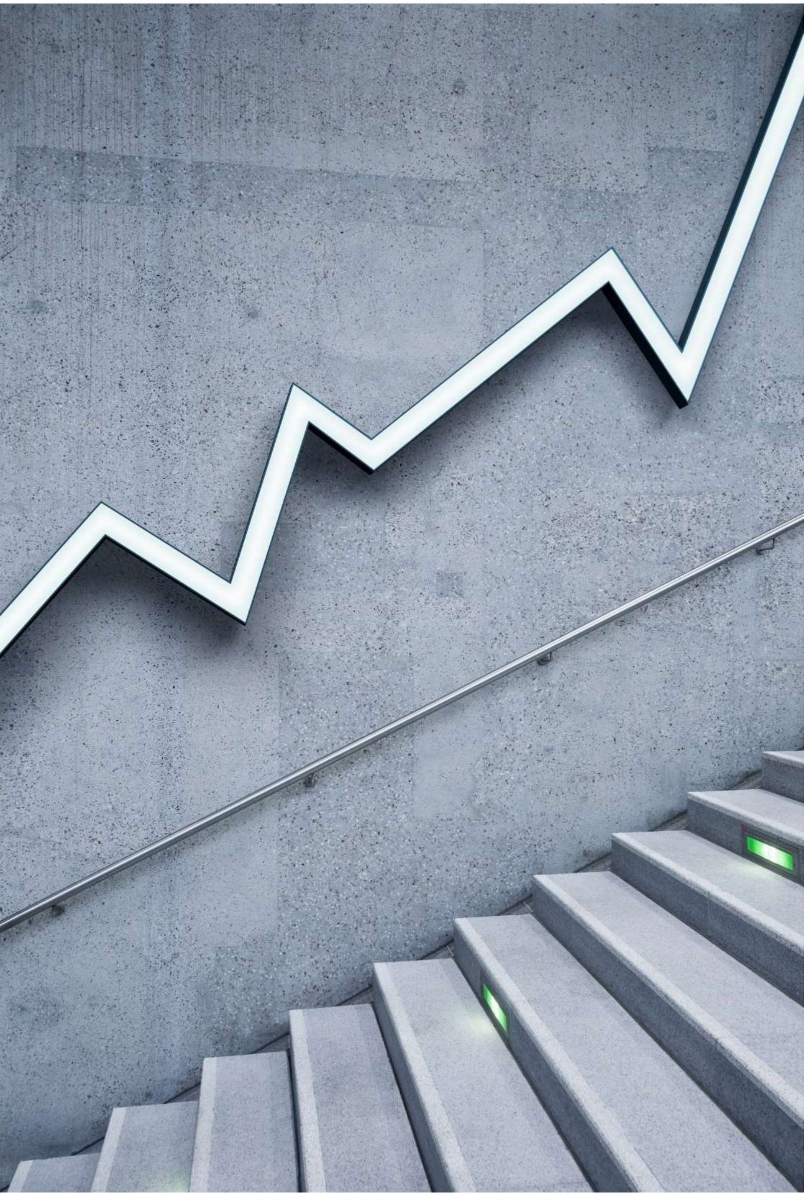

Learning JavaScript

PRAVEEN NAIR

What is JavaScript?

Used to program the behavior of web pages
JavaScript was invented by Brendan Eich in 1995.

JavaScript code is inserted between <script> and </script> tags.

Javascript was developed by Netscape

JavaScript vs VBScript (Microsoft)

Javascript supports all browser, vbscript supports IE

Originally Sun Microsystem and now Oracle

First Program

```
let a=10; // line break also works
let b=20;
let c = a + b;
console.log(c);
```

Comments // and /*

```
let name='John';
     let age=20
   /*
console.log(name)
*/
```

Printing using backtick

```
let n=2;
let s = `Price of an apple is ${n}`;
document.write(s)
```

………………………………..

Also called template literals….try multiline

Math Operators

Addition + (also concatenates string)

Subtraction -

Multiplication *

Division /

Remainder %

Exponentiation **

Comparison Operators

Operator	Description	Comparing	Returns
==	equal to	x == 8	FALSE
		x == 5	TRUE
		x == "5"	TRUE
===	equal value and equal type	x === 5	TRUE
		x === "5"	FALSE
!=	not equal	x != 8	TRUE
!==	not equal value or not equal type	x !== 5	FALSE
		x !== "5"	TRUE
		x !== 8	TRUE
>	greater than	x > 8	FALSE
<	less than	x < 8	TRUE
>=	greater than or equal to	x >= 8	FALSE
<=	less than or equal to	x <= 8	TRUE

Logical Operators

Logical NOT (!)

Logical AND (&&)

Logical OR (||)

Precedence of AND && is higher than OR ||

Conditional branching: if

```
let n = 7
if (n%2==0){
  console.log("Even Number")
}
else{
  console.log("Odd Number")
}
```

while loop

```
while (condition) {
  ...
}
```

For loop

```
for (let i = 0; i < 3; i++) {
  console.log(i);
}
```

Try break and continue

JavaScript Functions

```
function showMsg() {
  console.log( ‘Hello World!' );
}


showMsg();
```

Passing arguments

```
function sum(a, b) {
  c = a + b;
 console.log(c);
}
sum(1, 2);
```

Returning Values

```
function sum(a, b) {
  return a + b;
}
let result = sum(1, 2);
console.log( result ); // 3
```

Variables

let

const (constant, can’t be changed)

Var –

var is function scoped and let is block scoped. Variable declared by let cannot be redeclared

Variables are case-sensitive, try camelCase, titlecase, with dash

Data Types (Primitive/Value type)

1. let n=2;
2. let s = “Hello World”; //double or single quote
3. let flag = true; //true or false - boolean
4. let name; //undefined
5. let cost=null;

Type conversion

```
let value = true;
console.log(typeof value); // Boolean
value = String(value);

let numStr="34";
num = Number(numStr); // becomes a number 123

console.log(Boolean(num))

/* Values that are intuitively “empty”, like 0, an empty string, null,
undefined, and NaN, become false. Other values become true.*/
```

Data Types (Reference Type)

1. Objects
2. Arrays
3. Functions

Function Expressions

```
let sayHello = function() {
        console.log( "Hello World" );
     };

sayHello();
```

Arrow functions

```
let result = (a, b) => {
             let c = a + b
             return c
     };


let result = function(a, b) {
      let c = a + b
      return c;

   };


console.log(result(3, 2) );
```

Arrow Functions Recap

```
let a = b = 10
const fnc = () => a + b  //no braces and return is needed

let a = b = 10
const fnc = () => return a + b  //can't use return without braces

let a = 10;
const fnc = x => x + 20; //no brackets for x needed
console.log(fnc(a));

let a = (b = 10);
const fnc = () => {
  return a + b;  //return is needed if braces are used
};
```

Objects – Keyed Collections

```
let student = {
  name: "Smitha",
  age: 30
};
console.log(student.name) // student[“house address”]
console.log(student.age)
Console.log(student)
```

Objects – Spread Operator

```
const student = {
   name:"John",
   age:21
}
//spread operator
const obj = {...student,city:"NYC"}
console.log(obj)
```

Objects – return

```
let a = b = 10
      const fnc = () => ({name:'John'})  //need backet if returning obj
      console.log(fnc());
```

Array Items – foreach, map

```
let fruits = ["apple", "mango", "orange"];

    fruits.forEach((value,index,arr) => {

      console.log(value,index,arr);

    });
let fruits = ["apple", "mango", "orange"];

    fruits.map((value, index, arr) => {

      console.log(value, index, arr);

    });
```

Array Items – Spread Operator

```
const names = ["Vivek","Shivam","Aman"]

const arr = [...names,"Suresh"]

console.log(arr)
```

Array Items – filter and find

```
let score = [34, 12, 67, 89, 30];
     let result = score.filter((v) => {
       return v > 40;
     });
     console.log(result);
let empnum = [1003, 1005, 1006, 1034];
     let result = empnum.find((v) => {
       return v == 1003;
     });
     console.log(result);
```

Reduce method

```
function calculateTotal() {

  let total = cart.reduce((sum, item) => sum + item.price * item.quantity, 0);

  return total;
```

Array Items – some method

```
let marks = [10,60,80,40]
        let result = marks.some((value)=>{
            return value > 30
        })
        console.log(result)
```

Array Items – every method

```
let marks = [10,60,80,40]
    let result = marks.every((value)=>{
      return value > 30
    })
    console.log(result)
```

console.table

```
let array = [“a”,”b”]

console.table(array)
```

Array of Objects

```
<script>
    const products = [
      { name: "Product 1", price: 300 },
      { name: "Product 2", price: 100 },
      { name: "Product 3", price: 500 },
    ];
    const cart = [];
    let item = products[0];
    item.quantity = 2;
    item.total = item.quantity * item.price;
    cart.push(item);
    item = products[2];
    item.quantity = 3;
    item.total = item.quantity * item.price;
    cart.push(item);
    console.log(cart);
    let orderValue = cart.reduce((sum, value) => {
      return sum + value.total;
    },0);
    console.log("Order Value is", orderValue);
  </script>
```

(IIFE)immediately invoked function expression

```
(function functionName() {
  console.log("Hello World");
})();
```

```
(function functionName() {
  console.log("Hello World");
})();
```

Array from method

```
<script>
    let arr = Array.from("aeiou");
    document.write(arr);
</script>
```

Ternary/conditional operator ‘?’

```
let isEligible = (age > 18) ? true : false;
```

Try multiple condition

```
condition1
    ? true_expression1
    : condition2
        ? true_expression2
        : else_expression2
```

Switch statement

```
let price = 40;

switch (price) {
  case 30:
    console.log( 'Too Cheap’ );
    break;

  case 40:
    console.log( 'Perfect Price’ );
    break;

  case 50:
    console.log( 'Too Costly’ );
    break;

  default:
    console.log( "I don't know the price" );

}
```

Functions (…args) vs arguments

```
 <script>
  function sum(){
        let sum=0
        for (let i=0;i<arguments.length;i++){
          sum = sum + arguments[i]
        }
        console.log(sum)
      }
      sum(2,3,4,5)
..............................................
function sum(...args) {
        let sum = 0;
        for (let i = 0; i < args.length; i++) {
          sum = sum + args[i];
        }
        console.log(sum);
      }
      sum(2, 3, 4, 5);

 </script>
```

Module Import/Export - multiple

```
 function add(x,y){
    return x+y
}
function subtract(x,y){
    return x-y
}

export {add, subtract}
..................................
 import {add,subtract} from "./calc.js“
       let sum = add(4,5)
       console.log(sum)
       let difference = subtract(8,3)
       console.log(difference)
```

Array Destructuring – Part 1

Vanila JavaScript

```
const numbers = [10, 20, 30];
const first = numbers[0];
const second = numbers[1];
console.log(first, second);
```

ECMAScript

```
const numbers = [10, 20, 30];
const [first, second] = numbers;
console.log(first, second); // 10 20
```

Array Destructuring – Part 2

Skipping Values

```
const [a, , c] = [1, 2, 3];
console.log(a, c); // 1 3
```

Default Values

```
const [x = 5, y = 10] = [1];
console.log(x, y); // 1 10
```

Rest Operator

```
const [first, ...rest] = [1, 2, 3, 4];
console.log(first); // 1
console.log(rest);  // [2,3,4]
```

Object Destructuring – Part 1

Vanila JavaScript

```
const user = {
  name: “John",
  age: 30
};
const name = user.name;
const age = user.age;
```

ECMAScript

```
const user = {
  name: “John",
  age: 30
};
const { name, age } = user;   //based on property names, not position.
console.log(name, age);
```

Object Destructuring – Part 2

Renaming Variables

```
const { name: userName } = user;
console.log(userName);
```

Default Values in Object

```
const { country = "India" } = user;
console.log(country);
```

Object Destructuring – Part 3

Nested Destructuring

```
const student = {
  name: "Rahul",
  marks: {
    math: 90,
    science: 85
  }
};

const { marks: { math } } = student;
console.log(math); // 90
```

Destructuring in Function

Vanila JavaScript

```
function printUser(user) {
  console.log(user.name, user.age);
}
```

ECMAScript

```
function printUser({ name, age }) {
  console.log(name, age);
}
printUser({ name: "Sam", age: 25 });
```

Array Methods - 1

```
let fruits = ["Apple", "Orange", "Mango", "A"];
    // let fruits = new Array("Apple", "Orange", "Mango");
    console.log(fruits[0])
    console.log(fruits.length)
    console.log(fruits.at(-1)) //  or fuits[fruits.length-1]
    fruits.push("cherry","banana") //appends in the end – can add multiple
    console.log(fruits)
    fruits.pop(); // removes last element
    console.log(fruits);
    fruits.shift() // removes first element
    console.log(fruits)
    fruits.unshift("Plum","Pears") //adds in the beginning – can add multiple
    console.log(fruits)
    console.log(fruits.reverse())
    fruits.sort() //for desc - sort and reverse
    console.log(fruits)
```

Array Methods - 2

```
      let num = [4,17,3,5]
      console.log(num.sort(function(a, b){return a-b})) // for ascending
      console.log(num.sort(function(a, b){return b-a})) // for ascending
    console.log(fruits.indexOf("Apple"))
    console.log(fruits.lastIndexOf("Apple")) //if apple appears multiple times
console.log(fruits.includes("Apple"))
      // push/pop run fast, while shift/unshift run slow due to rearrangements
 let num = [4, 17, 3, 5];
      //   delete num[0];
      //   console.log(num);
      num.splice(0, 2);  //modify original array
      console.log(num);
slice // copy without modifing original array
let result = num.slice(1, 4);
```

Builtin Methods (Numbers)

```
    let a = 10.7
    console.log(Math.floor(a)) //rounds down
    console.log(Math.ceil(a)) //rounds up
    console.log(Math.round(a)) //rounds to nearest integer
    let b = 12.345
    console.log(b.toFixed(1)) // returns string, try toFixed(5)
    let str = 10
    console.log(isNaN(str))
    let v1 = "10“
    console.log(parseInt(v1) + 2)
console.log(Number(v1) + 2)
    console.log(Math.random())  // random between 0 and 1
    console.log(Math.max(2,4,7,9,1))
    console.log(Math.min(2,4,7,9,1))
    console.log(Math.pow(2,3)) // 2 to the power 3
```

Builtin Methods (String)

```
let str = "Hello";
 console.log(str[0])
 console.log(str.charAt(0))
console.log(str.length)
 for (let c of "Welcome") {
   console.log(c);
 }
console.log(str.toUpperCase())
console.log(str.toLowerCase())
console.log(str.indexOf('l’))
console.log(str.lastIndexOf('l’))
console.log(str.includes('l’))
console.log(str.startsWith('H’))
console.log(str.endsWith('o’))
console.log(str.slice(1,4))  /try (-4,-1)
console.log(str.substring(1,4)) // try substr - start, length
console.log('a' > 'Z')  // comparing string
```

slice

Date Methods (get)

```
let d = new Date();
console.log(d.getDate()); // 1 to 31
console.log(d.getFullYear());
console.log(d.getMonth());
console.log(d.getDay()); //weekday
console.log(d.getHours());
console.log(d.getMinutes());
console.log(d.getSeconds());
console.log(d.getMilliseconds());
console.log(d.getTime()); // milliseconds since 1/1/1970
console.log(Date.now()); // milliseconds since 1/1/1970
```

Creating Dates in JavaScript

```
let d = new Date()

let d = new Date(2021,10,27,20,12,10) //try yyyy-mm-dd, mm/dd/yyyy

let d = new Date("November 10, 2015 11:13:00");

let d = new Date(1000) // milliseconds starts January 01, 1970

console.log(d);
```

Date Methods (set)

```
let d = new Date()
d.setFullYear(2000)
d.setMonth(0)
d.setDate(1)
d.setHours(1)
d.setMinutes(1)
d.setMilliseconds(1)  //0 – 999
d.setTime(1) // starting 1/1/1970
console.log(d)
```

Hoisting with var

Hoisting is moving declarations to the top. Only declarations are hoisted, not initializations.

……………………………………………………………..

```
console.log(a);
var a = 10;
```

JavaScript interprets it like this:

```
var a;         // declaration is hoisted
console.log(a);  // undefined
a = 10;        // initialization stays in place
```

Hoisting with let and const

```
console.log(b);

let b = 20;
```

Output: ReferenceError

why?

let and const are hoisted BUT placed in Temporal Dead Zone (TDZ). They cannot be accessed before initialization.

Function Hoisting

```
sayHello();
function sayHello() {
  console.log("Hello");
}
```

Function declarations are fully hoisted (both name and body).

Function Expression Hoisting

```
sayHi();
var sayHi = function() {
  console.log("Hi");
};
```

```
TypeError: sayHi is not a function
```

....

Only variable declaration is hoisted
Function body is NOT hoisted

Arrow Function Hoisting

```
greet();
const greet = () => {
 console.log("Hello");
};
output: ReferenceError
```

……………………………………………….

Why?

const is in Temporal Dead Zone.

JavaScript execution

JavaScript execution happens in 2 phases:

1. Creation Phase
Memory is allocated
Variables are set to undefined
Functions are stored fully

2. Execution Phase
Code runs line by line
Assignments happen

Hoisting occurs in the creation phase.

Why promise is needed

```
//asynchronous : occuring at the same time
        const f1 = () => {
          setTimeout(() => {
            return 5;
          }, 5000);
        };

        const f2 = (x) => {
          console.log(x + 6);
        };

        let n1 = f1();
        f2(n1);
```

Use callback to solve the issue

```
const f1 = (fnc) => {
    setTimeout(() => {
     fnc(5);
    }, 5000);
   };

   const f2 = (x) => {
    console.log(x + 6);
   };

   f1(f2);
```

Use promise and .then

```
const f1 = () => {
      return new Promise((resolve, reject) => {
        setTimeout(() => {
          resolve(5); //use resolve instead of return
        }, 5000);
      });
    };

    const f2 = (x) => {
      console.log(x + 6)
    };

    f1().then((a) => f2(a));
```

Async/await

```
 const f1 = () => {
        return new Promise((resolve, reject) => {
          //   resolve(5);
          reject("Something went wrong");
        });
      };
      const f2 = () => {
        console.log("Function 2");
      };
      const f3 = async () => {
        try {
          let n1 = await f1();
          f2(n1);
        } catch (err) {
          console.log(err);
        }
      };
f3()
```

Promise - Real World Example

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const user = {
        name: "John",
        email: "john@gmail.com",
        role: "student",
      };
      resolve(user);
    }, 2000);
  });
}
function display({name,email,role}){
console.log(`Hello ${name}`)
}

fetchData()
  .then((data) => display(data))
  .catch((err) => console.log(err));
```

Promise.all – Waits for all promises to resolve. If any one fails, it immediately rejects.

```
function fetchStudentDetails() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const user = {
        name: "John",
      };
      resolve(user);
    }, 2000);
  });
}
function fetchStudentMarks() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const marks = {
        math: 80,
      };
      resolve(marks);
    }, 3000);
  });
}
function display(data) {
  console.log(data);
}
Promise.all([fetchStudentDetails(), fetchStudentMarks()])
  .then((data) => display(data))
  .catch((err) => console.log(err));
```

Using with asyn/await

```
async function getData() {
  try {
    const results = await Promise.all([
      fetchStudentDetails(),
      fetchStudentMarks(),
    ]);
    console.log(results);
  } catch (error) {
    console.log(error);
  }
}
getData();
```

Other Promise methods

Promise.race – Returns the first promise that settles (resolve or reject).

Promise.any – Returns the first fulfilled promise. Ignores rejections unless all fail.

Promise.allSettled() – Waits for all promises, whether resolved or rejected.

Fetch with async await & JSON

```
const url = "https://jsonplaceholder.typicode.com/users/";
const showUsers = async () => {
  try {
    const response = await fetch(url);
    const json = await response.json();
    // console.log(json);
    for (let user of json) {
      console.log(user.name);
    }
  } catch (error) {
    console.log(error);
  }
};
showUsers();
```

JavaScript Object Notation (JSON)

JSON is a text-based data format used to represent structured data.

```
{
  "name": “John",
  "age": 30,
  "isTrainer": true
}
```

....

Keys must be in double quotes

Strings must use double quotes

No functions allowed

No undefined values

Data must be valid types

No trailing comma.

JSON.stringify

```
student = {name:"john",age:20,pass:true}
student = JSON.stringify(student)
console.log(student)
```

expected output:

```
{"name":"john","age":20,"pass":true}
```

JSON.parse

```
let student = '{"name":"john", "age":20}'

console.log(student.name)

let obj = JSON.parse(student)

console.log(obj.name)
```

Closure (access to outer variable)

A closure is a function that has access to variables from its outer scope even after the outer function has executed.

```
function outer() {
  let count = 0;

  function inner() {
    count++;
    console.log(count);
  }

  return inner;
}

const counter = outer();

counter(); // 1
counter(); // 2
counter(); // 3
```

Closure – real world use case - Data Privacy

```
function existingUser() {
  let password = "12345";
  function checkPassword(input) {
    return input === password;
  }
  return checkPassword;
}

const checkPassword = existingUser();
console.log(checkPassword("12345"));
```

Error Handling – reference error

```
try{
            console.log(a)
        }
 catch(err){
            console.log(err)
            console.log(err.message)
            console.log(err.name)
        }
```

Error Handling – eval & syntaxError

if we try to evaluate code with a syntax error.

```
try{
        let name="document.write(Hello World')"
        let result = eval(name)
        console.log(result)
     }
     catch(err){
        console.log(err)
        console.log(err.message)
        console.log(err.name)
     }
```

Error Handling – typeError

```
try {
      let num = 34
      console.log(num.toLowerCase());
    } catch (err) {
      console.log(err);
      console.log(err.message);
      console.log(err.name);
    }
```

for…in Loop - Arrays

```
const employees = ["Amit", "Ravi", "John"];
for (let x in employees) {
  console.log(x);
}

for (let x in employees) {
  console.log(employees[x]);
}
```

for…in Loop - Objects

```
const employees = {
  emp1: "Amit",
  emp2: "Ravi“
};

for (let x in employees) {
  console.log(x);              // key
  console.log(employees[x]);   // value
}
```

for…of Loop - Arrays

```
const employees = ["Amit", "Ravi", "John"];
for (let x of employees) {
  console.log(x);
}
```

for…of Loop - Objects

```
const employees = {
  emp1: "Amit",
  emp2: "Ravi“
};

for (let x of employees) {
  console.log(x); //Error
}

for (let key of Object.keys(employees)) {
  console.log(key, employees[key]);
}
```

"use strict" in JavaScript

JavaScript originally allowed many unsafe behaviors. Strict mode removes or restricts them.

```
"use strict";
x = 10; //  Error
…………………………………….
"use strict";
function sum(a, a) { return a+a} // SyntaxError
………………………………………..
"use strict";
let x = 10;
delete x; //  Error
……………………………………..
"use strict";
let x = 010; // Leading zero not allowed Error
```

this keyword inside an object

```
const employee = {
  name: "Amit",
  greet: function () {
    console.log(this.name);
  }
};
employee.greet(); // Amit
…………………………………………………
greet: () => {
    console.log(this.name);  // will throw error
  }
……………………………………
greet() {

    console.log(this.name); //correct way

  }
```

this keyword – real world example

```
const cart = {

  items: [],

    addItem(product) {
    this.items.push(product);
    console.log(`${product} added to cart.`);
  },
  showItems() {
    console.log("Cart Items:", this.items);
  }
};
cart.addItem("Laptop");
cart.addItem("Mobile");
cart.showItems();
```

this with new Keyword

```
function Employee(name) {
  this.name = name;
}

const emp1 = new Employee("Amit");
console.log(emp1.name); // Amit
```

Example without constructor or prototype

```
const emp1 = {
  name: "Amit",
  greet: function () {
    console.log("Hello " + this.name);
  }
};

const emp2 = {
  name: "Ravi",
  greet: function () {
    console.log("Hello " + this.name);
  }

};

emp1.greet();

emp2.greet();
```

Example with constructor without prototype

```
function Employee(name) {
  this.name = name;

  this.greet = function () {
    console.log("Hello " + this.name);
  };

}
const emp1 = new Employee("Amit");
const emp2 = new Employee("Ravi");

emp1.greet(); // Hello Amit
emp2.greet(); // Hello Ravi
……………………………………………………
emp1 → greet() function copy #1
emp2 → greet() function copy #2
```

Example with prototype

```
function Employee(name) {
  this.name = name;
}
Employee.prototype.greet = function () {
  console.log("Hello " + this.name);
};
const emp1 = new Employee("Amit");
const emp2 = new Employee("Ravi");

emp1.greet(); // Hello Amit
emp2.greet(); // Hello Ravi
………………………………………………..
emp1 → shared greet()
emp2 → shared greet()
```

Nullish coalescing operator '??'

```
let height = 0;


console.log(height || 100); // 100    returns truthy value
console.log(height ?? 100); // 0
```

What is Babel?

Babel is a JavaScript compiler (transpiler) that converts modern JavaScript into older JavaScript so it can run in older environments.

Modern Code (ES6)

```
const greet = (name) => `Hello ${name}`;
```

After Babel (ES5)

```
var greet = function(name) {
  return "Hello " + name;
};
```

Thank You

- PRAVEEN NAIR