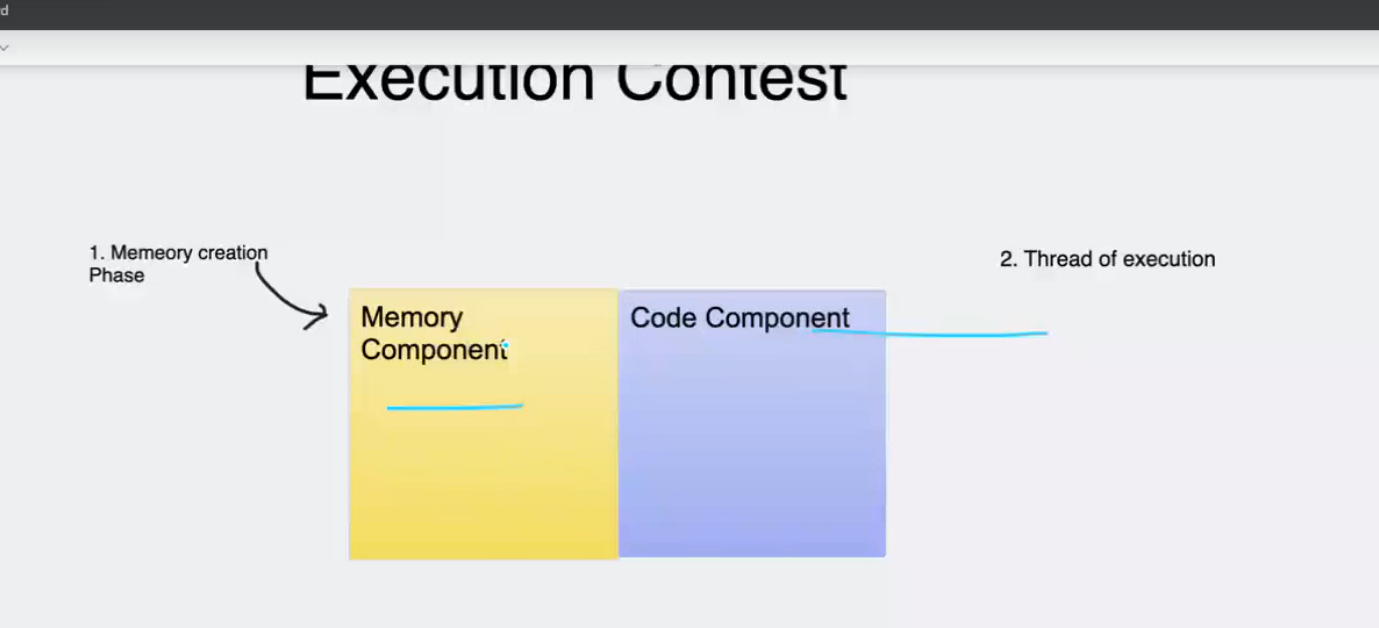
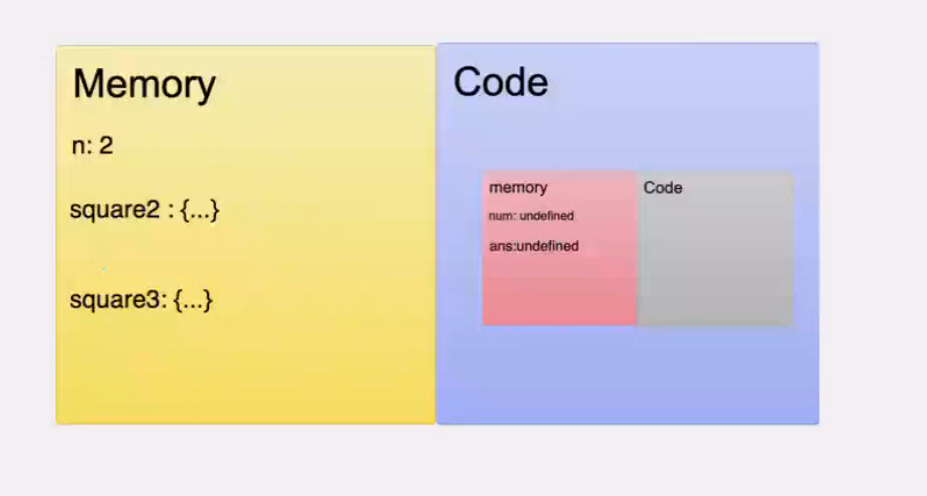
Global execution context --



Everything In javascript is function and object.



Console.log – it is api, it is not part of js

Phase 1)reading phase

Saglyat adhi js purn code run karel varun khaliparyant ani sagle variables ani function store karel memory madhe

Var = undefined ani functions jas chya tas store karel.

Phase 2)executing phase

Parat top to bottom code read karel ani execution start karel

Like var la value assign karel , jar function al tr te ek navin execution context tayar karel ani tithe parat process follow karel, function run zalyavar te delete hoil.

JavaScript is a single-threaded interpreted language. Every browser has its own JavaScript engine. Google Chrome has the V8 engine, Mozilla Firefox has SpiderMonkey, and so on. They all are used for the same goal, because the browsers cannot directly understand JavaScript code.

Let's look at an example so we can learn more:

var n = 5;

function square(n) {

var ans = n \* n;

return ans;

}

var square1 = square(n);

var square2 = square(8);

console.log(square1)

console.log(square2)

In the above code,

1. n is initialized and assigned a value of 5
2. We defined a function square() that accepts an argument n and returns the square of n.
3. We call the square() function and store the returned value in the square1 variable.
4. We call the square() function and store the returned value in the square2 variable.
5. Finally, it outputs both square1 and square2

Behind the scenes, JavaScript is doing so many things. Let's understand all of it.

**What is the Execution Context?**

When the JavaScript engine scans a script file, it makes an environment called the **Execution Context**that handles the entire transformation and execution of the code.

During the context runtime, the parser parses the source code and allocates memory for the variables and functions. The source code is generated and gets executed.

There are two types of execution contexts: **global** and **function**. The global execution context is created when a JavaScript script first starts to run, and it represents the global scope in JavaScript. A function execution context is created whenever a function is called, representing the function's local scope.

**Phases of the JavaScript Execution Context**

There are two phases of JavaScript execution context:

1. **Creation phase**: In this phase, the JavaScript engine creates the execution context and sets up the script's environment. It determines the values of variables and functions and sets up the scope chain for the execution context.
2. **Execution phase**: In this phase, the JavaScript engine executes the code in the execution context. It processes any statements or expressions in the script and evaluates any function calls.

Everything in JS happens inside this execution context. It is divided into two components. One is memory and the other is code. It is important to remember that these phases and components are applicable to both global and functional execution contexts.

**Creation Phase**

Execution Context

Let's take this simple example once again:

var n = 5;

function square(n) {

var ans = n \* n;

return ans;

}

var square1 = square(n);

var square2 = square(8);

console.log(square1)

console.log(square2)

At the very beginning, the JavaScript engine executes the entire source code, creates a global execution context, and then does the following things:

1. Creates a global object that is**window** in the browser and **global** in NodeJs.
2. Sets up a memory for storing variables and functions.
3. Stores the variables with values as undefined and function references.

This is called the creation phase. Here's a diagram to help explain it:

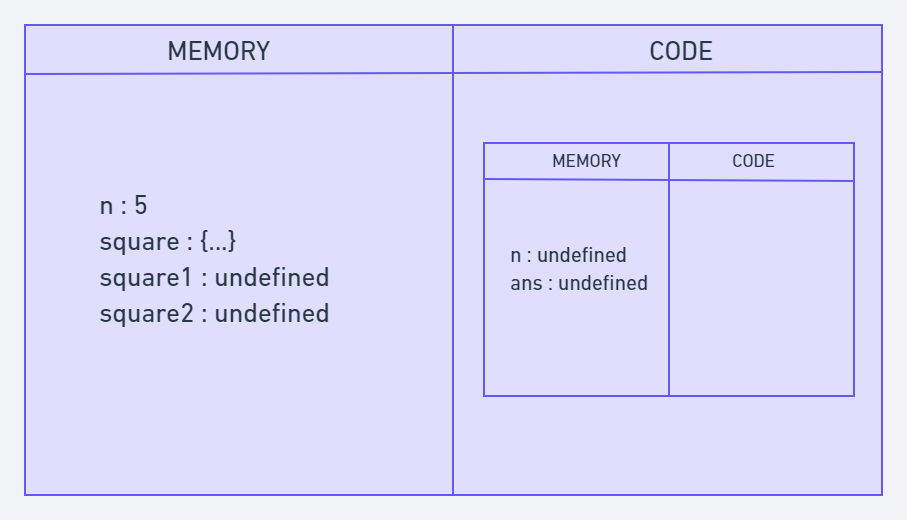
Creation Phase in Execution Context

After this creation phase, the execution context will move to the code execution phase.

**Execution Phase**

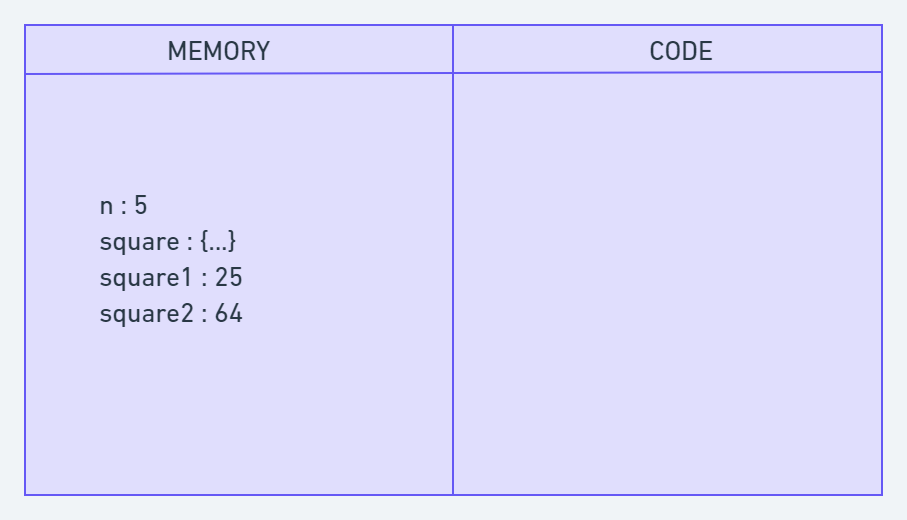
Now, in this phase, it starts going through the entire code line by line from top to bottom. As soon as it encounters **n = 5**, it assigns the value 5 to 'n' in memory. Until now, the value of 'n' was undefined by default.

Then we get to the 'square' function. As the function has been allocated in memory, it directly jumps into the line **var square1 = square(n);**. square() will be invoked and JavaScript once again will create a new function execution context.

Code Execution Phase

Once the calculation is done, it assigns the value of square in the 'ans' variable that was undefined before. The function will return the value, and the function execution context will be destroyed.

The returned value from square() will be assigned on square1. This happens for square2 also. Once the entire code execution is done completely, the global context will look like this and it will be destroyed also.



**What is the Call Stack?**

To keep the track of all the contexts, including global and functional, the JavaScript engine uses a **call stack**. A call stack is also known as an 'Execution Context Stack', 'Runtime Stack', or 'Machine Stack'.

It uses the LIFO principle (Last-In-First-Out). When the engine first starts executing the script, it creates a global context and pushes it on the stack. Whenever a function is invoked, similarly, the JS engine creates a function stack context for the function and pushes it to the top of the call stack and starts executing it.

When execution of the current function is complete, then the JavaScript engine will automatically remove the context from the call stack and it goes back to its parent.

Let's see the following example:

function funcA(m,n) {

return m \* n;

}

function funcB(m,n) {

return funcA(m,n);

}

function getResult(num1, num2) {

return funcB(num1, num2)

}

var res = getResult(5,6);

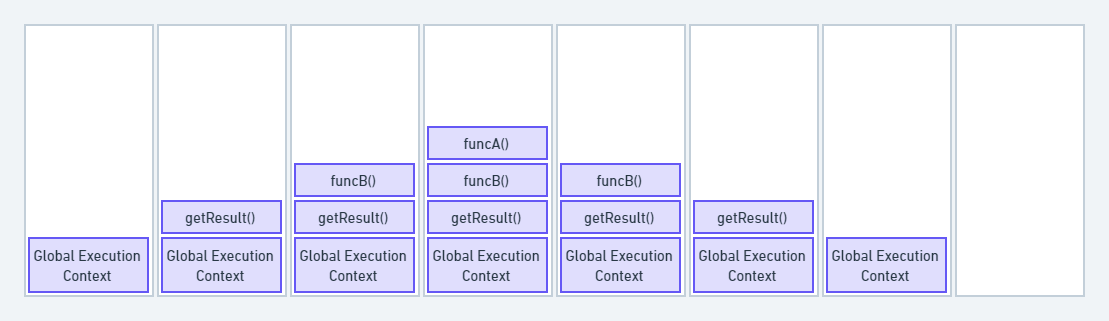
console.log(res); // 30

In this example, the JS engine creates a global execution context that enters the creation phase.

First it allocates memory for funcA, funcB, the getResult function, and the res variable. Then it invokes getResult(), which will be pushed on the call stack.

Then getResult() will call funcB(). At this point, funcB's context will be stored on the top of the stack. Then it will start executing and call another function funcA(). Similarly, funcA's context will be pushed.

Once execution of each function is done, it will be removed from the call stack. The following picture depicts the entire process of the execution:

Call Stack

The call stack has its own fixed size depending on the system or browser. If the number of contexts exceeds the limit, then a stack overflow error will occur. This happens with a recursive function that has no base condition.

function display() {

display();

}

display();

C:\Users\rwiteshbera\Desktop\Javascript\n.js:2

display();

^

RangeError: Maximum call stack size exceeded

Output

**Conclusion**

In conclusion, JavaScript execution context is a crucial part of understanding how JavaScript works behind the scenes. It determines the environment in which code is executed and what variables and functions are available to use.

The creation phase includes creating the global and function execution contexts, creating the scope chain, and allocating memories for the variables and functions. During the execution phase, the JavaScript engine executes the code line by line. This includes evaluating and executing statements.

Hoisting—

Hoisting is a JavaScript feature that moves all variable and function declarations to the top of their scope, even if they are declared later in the code. This means that you can refer to a variable or function before it is declared, but the value will be undefined.

Console.log(a) //undefined

Var a = 10

Let and const la Reference err bhetel.

Let and const bhi hoist hote hai but they didn’t assign any value

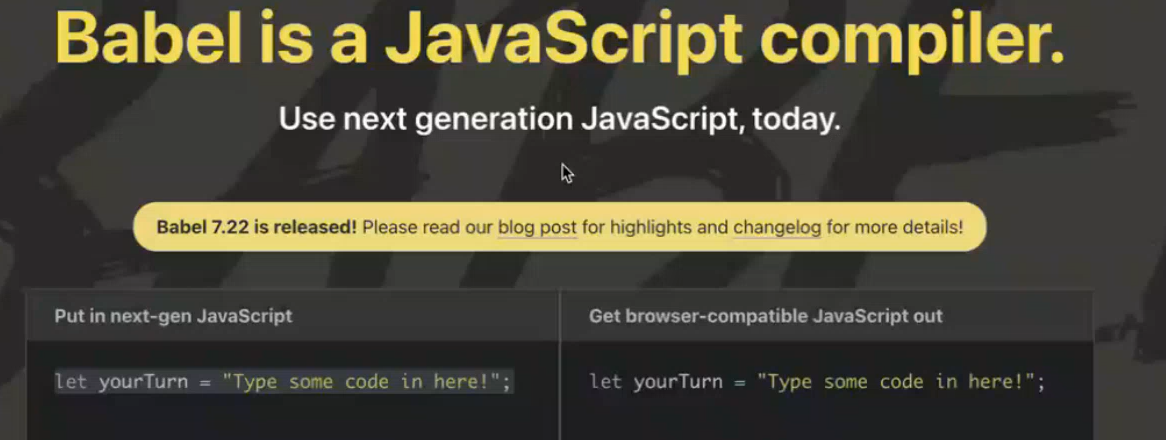
Scope chain --

ES6-2016

Babel – backword compatible

New js ->old js

Babel is a JavaScript compiler that converts modern JavaScript code into a version compatible with all browsers.



Babel is bydefault in react

ternary operator –

Api -application programming interface

Api call - get push delete’

HTTP requests –

**CRUD operations (create, read, update, and delete)**

**GET:**GET request is used to read/retrieve data from a web server. GET returns an HTTP status code of **200 (OK)** if the data is successfully retrieved from the server.

**POST:**POST request is used to send data (file, form data, etc.) to the server. On successful creation, it returns an HTTP status code of **201**.

**PUT:**A PUT request is used to modify the data on the server. It replaces the entire content at a particular location with data that is passed in the body payload. If there are no resources that match the request, it will generate one.

**PATCH:**PATCH is similar to PUT request, but the only difference is, it modifies a part of the data. It will only replace the content that you want to update.

**DELETE:**ADELETE request is used to delete the data on the server at a specified location.

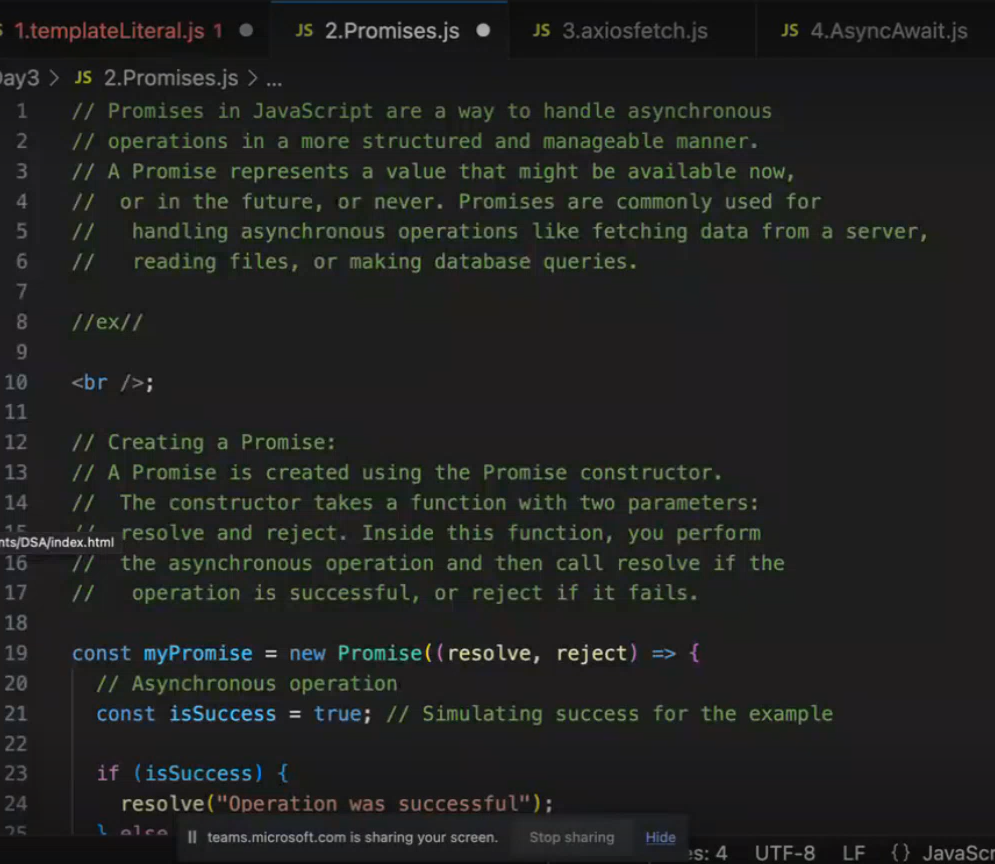
Promises –

promise is an object that represents the eventual completion (or failure) of an asynchronous operation. It is a more modern way of handling asynchronous code in JavaScript than using callbacks.

Uses – fetching api, error handling

Promise is asynchronous A computer screen shot of text

Description automatically generated

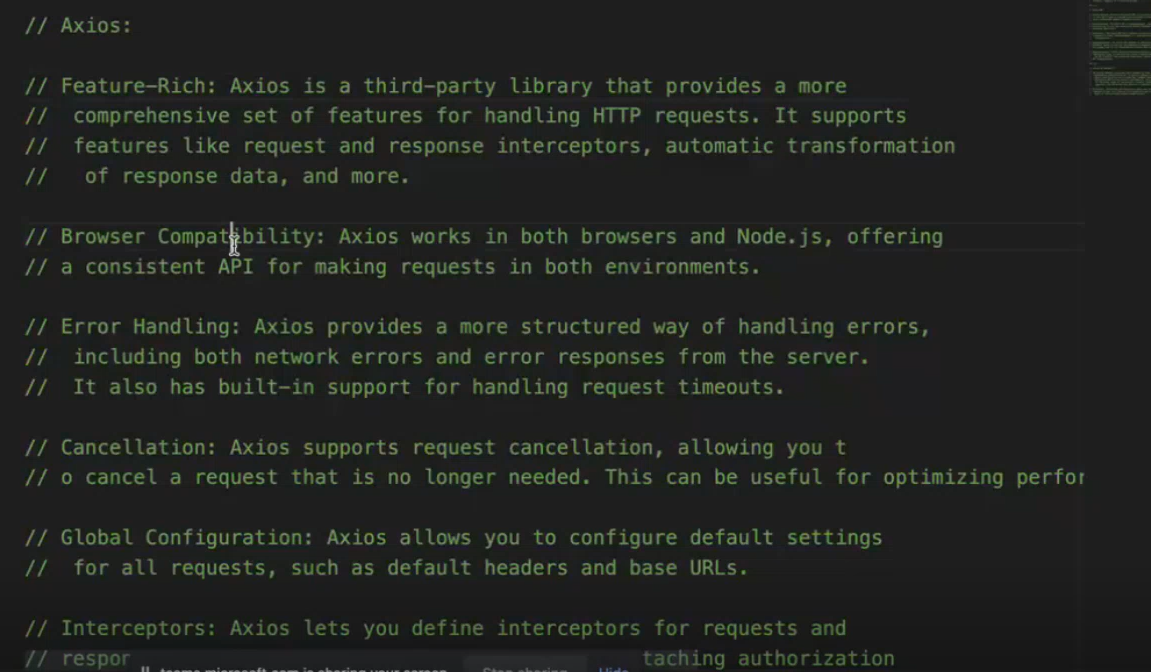


New -- Axios –js library to make http(hypertext transfer protocol)request from browser to web server

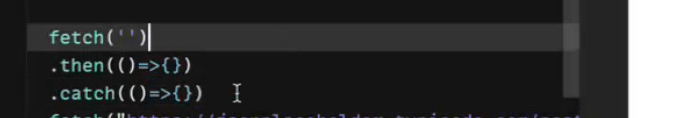
Axios is a promise-based HTTP client for JavaScript. It is a popular library used to make HTTP requests from the browser and the server. Axios is easy to use and has a lot of features, making it a great choice for making HTTP requests in JavaScript.

Here are some of the features of Axios:

* Promise-based: Axios uses promises to handle the asynchronous nature of HTTP requests. This makes it easy to chain requests together and handle errors.
* Configurable: Axios is highly configurable. You can customize the headers, timeouts, and other settings of each request.
* Lightweight: Axios is a lightweight library that is easy to download and install.
* Cross-platform: Axios can be used in the browser and the server.



Old - Fetch –



Best -- Async await --

The async keyword is used to define an asynchronous function. The await keyword is used to wait for the result of an asynchronous operation.

A close-up of a computer code

Description automatically generated

Higher order function -first class citizen

A higher-order function (HOF) is a function that does at least one of the following:

* Takes one or more functions as arguments.
* Returns a function as its result.

All other functions are first-order functions.

Example – map()

Jo dusrya la invoke karto as parameter or argument

Callback function –jo dusra function as parameter mhanun jato

A callback function is a function that is passed as an argument to another function

Callback function is part of HOF only

A white screen with black text

Description automatically generated

Inheritance –1

Inheritance is a mechanism in JavaScript that allows a new class to inherit the properties and methods of an existing class. This can be used to avoid duplicating code and to create more complex and flexible object-oriented programs.

In JavaScript, there are two main types of inheritance:

* **Class inheritance:** This is the most common type of inheritance in JavaScript. It is used to create a new class that inherits the properties and methods of an existing class.
* **Prototype inheritance:** This is a less common type of inheritance in JavaScript. It is used to create a new object that inherits the properties and methods of an existing object.

A computer code on a white background

Description automatically generated

n this example, the Student class inherits from the Person class. This means that the Student class has all the properties and methods of the Person class, plus any additional properties and methods that are defined in the Student class.

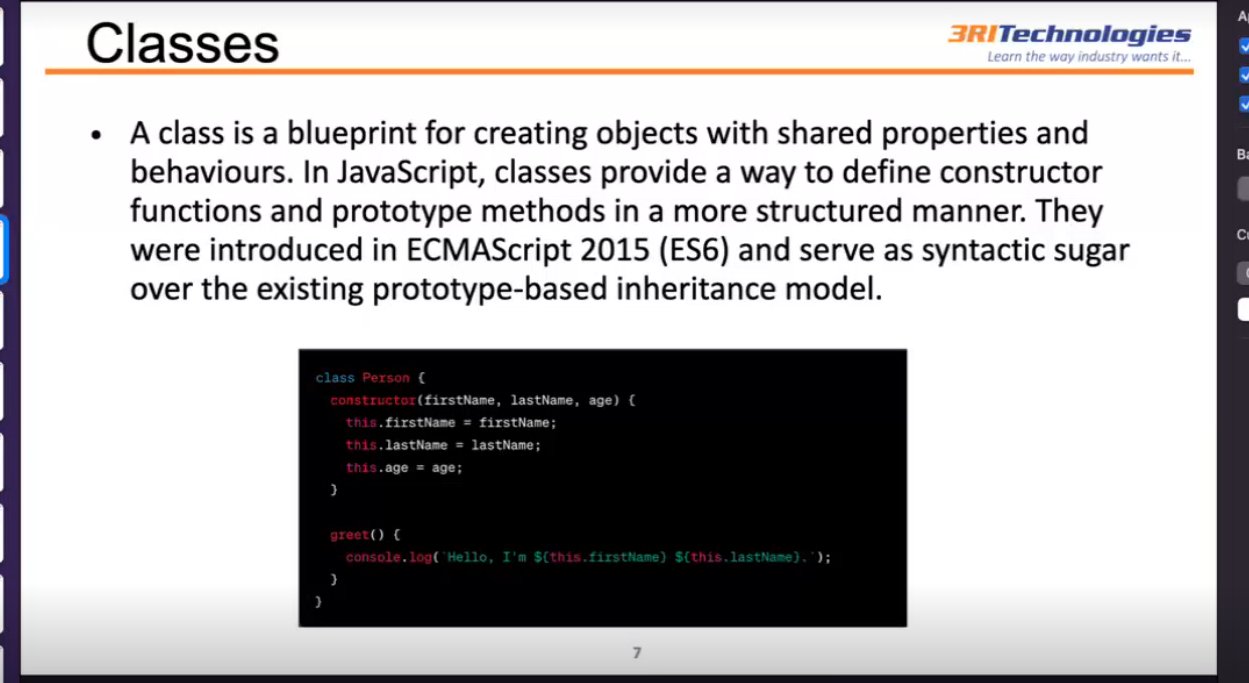
USeeffect –

The useEffect hook in React is used to perform side effects, such as fetching data from an API or subscribing to an event, manually changing DOM. It is similar to the componentDidMount and componentDidUpdate lifecycle methods, but it is more flexible and efficient.

Side effect – can be anything that affect outside of the current scope of the current function that being executed.

Ji goshta pratek veles page refresh hotacha disli pahije tya sathi useeffect vapratat

Class –



Constructor()--The constructor is a method used to initialize an object's state in a class. It automatically called during the creation of an object in a class.

Super()--super is used to call a superclass constructor, super is used to access a superclass field: A subclass can access a field defined in its parent class using the super keyword

extend keyword --  
can create new class with inheriting properties from parent class  
this -  
represents the value where it is called  
super -  
previous class ki props ko access karne ke liye

Prototype()