

Topic: **Apis, Exception handling, this call apply bind and constructor function**

API- application programming interface

API stands for Application Programming Interface

It is a collection of communication protocols and subroutines used by various programs to form the communication between them.

In simple words It allows communication between two different software systems.

How api works

Client initiates the request via the api uri (uniform resource identifier)

The api makes a call to the server after receiving the request

Then the server sends the response back to API with the information

Finally, the API transfers data to the client

Types of apis

- **Browser APIs:** Interact with browser and web page elements (dom api), storage api
- **Server Apis:** Provide data and services from a server (restfull api)
- **Third-party Apis:** Offered by external services (social media apis)
- **Rest apis:** REST APIs are designed to make server-side data readily available by representing it in simple formats such as JSON and XML.
- **Soap apis(Simple Object Access Protocol):** SOAP uses XML exclusively to format data, making it more verbose and complex compared to REST.
- **GraphQL:** Allows clients to request exactly the data they need, great for complex data relationships. The client controls what data to retrieve, reducing over-fetching or under-fetching of data.
- **gRPC (Google Remote Procedure Call):** gRPC is a modern, high-performance, open-source RPC (Remote Procedure Call) framework developed by Google suitable for microservices and performance-critical systems.
- **WebSockets:** WebSocket is a protocol that allows for real-time, two-way communication between the client and server over a single TCP connection. WebSocket can send data in JSON, XML, or any other format, but the communication is typically binary, Real-time, bi-directional communication for applications like chat and gaming.

Error handling methods in js

In JavaScript, try...catch statements are used to handle exceptions (errors) that occur in your code. By using these statements, you can ensure that your code continues to run even if an error occurs.

Error — It is an object that is created to represent a problem that occurs often with userinput or establishing a connection.

Why we need try catch

// **case -1** -----> both statements will execute

```
console.log("hi hello");
console.log("you have reached the end");
```

// **case -2** ----->It interrupts the program from 163 line to end because console.lag is not a method it

```
console.lag("hi hello");
console.log("you have reached the end");
```

To overcome this we need **try catch methods**

Syntax:

```
try {
  // Code that may throw an error
} catch (err) {
  // Code to handle the error
} finally {
  // Code that will always run, regardless of error
}
```

Explanation

try block: Contains the code that may throw an error. If no errors occur, the code inside the catch block is skipped.

catch block: Contains code that will execute if an error occurs in the try block. The err parameter contains the error object.

finally block (optional): Contains code that will run after the try and catch blocks, regardless of whether an error was thrown or not.

//try block contains error catch block takes one parameter err means error it will store the error caused by try

Ex-1:

```
try{
  console.lag("hi hello");
}
catch(err){
  console.log(err);
}
```

We can also use

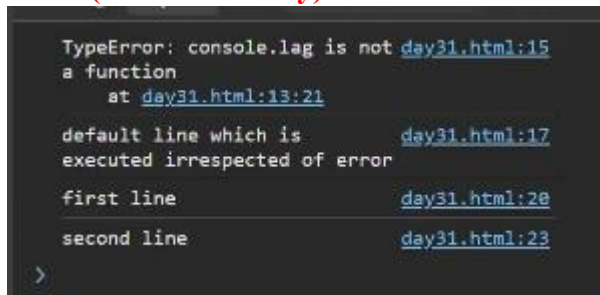
console.error(err)

Ex-2

```
try{
  console.lag("hi hello");
}
catch(err){
  console.error(err);
}
finally{
  console.log("this will always run");
}
```

```
}  
console.log("you have reached the end");
```

Ex-3: (if we use finally)



Throw statement in js

The throw statement is used to raise an exception in JavaScript. When an exception is thrown, the **normal flow of code execution is stopped**, and **control is passed to the nearest enclosing catch block**. If no catch block is found, the script will terminate.

Example:

```
num = prompt("entry");  
try {  
  if (num <= 0) {  
    throw new Error("The number must be positive.");  
  }  
}  
catch (err) {  
  console.error(err);  
}  
console.log("you have reached the end");
```

This program prompts the user to enter a number, checks if the number is positive, and handles errors if the number is not positive. It uses a try...catch block to manage potential exceptions and ensures that a final message is logged to indicate the end of the program.

Try Block:

```
if (num <= 0) { throw new Error("The number must be positive."); }
```

Condition: Checks if the input num is less than or equal to 0.

Error Handling: If the condition is true, an error is thrown with the message "The number must be positive."

Catch Block:

```
catch (err) { console.error(err); }
```

Function: Catches the error thrown in the try block.

Error Logging: Uses console.error(err) to log the error object to the console, which includes the error message and stack trace.

Final Log Statement:

```
console.log("you have reached the end");
```

Purpose: To indicate that the program has reached its end, regardless of whether an error occurred.

Types of errors

1. Syntax Errors

Description: Occur when the code contains **invalid syntax**, which prevents the script from being parsed correctly.

```
if (true {  
  console.log("This is a syntax error");  
}
```

2. Reference Errors

Description: Occur when trying to reference a **variable that is not declared**.

```
console.log(nonExistentVariable);
```

3. Type Errors

Description: Occur when an operation is performed on a **value of an inappropriate type**

```
let num = 5;  
num.toUpperCase(); // TypeError: num.toUpperCase is not a function
```

4. Range Errors

Description: Occur when a numeric variable or parameter is **outside its valid range**

```
let arr = new Array(-1);  
function createArray(size) {  
  if (size > 100) {  
    throw new RangeError("Array size is too large");  
  }  
  return new Array(size);  
}  
createArray(101);
```

5. Eval Errors

Description: Occur due to improper use of the **eval() function**. This error type is **rarely encountered** and is primarily included for **backward compatibility**.

```
eval("alert('Hello')"); // This example won't necessarily throw an EvalError in modern browsers but shows misuse of eval.
```

6. URI Errors

Description: Occur when global **URI handling functions are used incorrectly**, such as `encodeURIComponent()`, `decodeURIComponent()`, and `decodeURIComponent()`.

```
decodeURIComponent("%");
```

This keyword

``this`` is a keyword that refers to the **current calling object** or context in which it is used. It is used to access the properties and methods of the current object.

1.Global Context: When this is used in the global scope (outside of any function), it refers to the global object. In a web browser environment, the global object is `window`.

```
console.log(this === window); // true
```

Function Context: When this is used within a function that is not a method of an object, its value depends on how the function is called. If the function is called as a standalone function, this will typically refer to the global object (or undefined in strict mode).

```
function sayHello() {  
  console.log(this);  
}  
sayHello(); // In a browser, this would log the window object
```

3.Method Context: When this is used within a method of an object, it refers to the object that the method is called on.

//here this refers to the person means object name

```
const person = {  
  name: 'John',  
  greet: function() {  
    console.log('Hello, my name is ' + this.name);  
  }  
};  
person.greet(); // Logs "Hello, my name is John"
```

This in arrow function

Ex-1:

```
let obj={  
  name:"John",  
  age:30,  
  sayHello:()=>{  
    console.log(this.age) //undefined  
  }  
}  
obj.sayHello()
```

Ex-2:

```
let obj={  
  name:"John",  
  age:30,  
  sayHello:()=>{  
    console.log(this)// window  
  }  
}  
obj.sayHello()
```

Understanding this in Arrow Functions

Arrow functions (`() => { }`) in JavaScript behave differently with respect to this compared to regular functions.



```
btn=document.querySelector("button");  
  
btn.addEventListener("click", function(){  
  console.log(this);  
  this.style.color="red"  
})
```

Here are the key points to note:

Lexical Scope: Arrow functions do not have their own this context. Instead, they inherit this from the surrounding (lexical) scope where they are defined.

Global Object: In most cases where obj is defined at the top level (not inside another function or block), this refers to the global object (window in browsers).

4.This refers to the particular event in evenhandlers

```
document.getElementById('myButton').onclick = function() {  
  console.log(this); // Refers to the button element with the ID 'myButton'  
};
```

Changing this with call, apply, and bind

call and apply immediately invoke the function with a specified this value and arguments.

```
function showThis() {  
  console.log(this);  
}  
const obj = { name: "John" };  
  
showThis()// window  
showThis.call(obj); // obj  
showThis.apply(obj); // obj
```

bind

bind returns a new function with a specified this value, without immediately invoking the function.

```
const boundFunction = showThis.bind(obj);  
boundFunction(); // obj
```



call with parameters

```
function hello(city, profession) {  
  console.log("hello my name is " + this.name + " iam from " + city + " my profesion is " + profession);  
}  
  
obj = {  
  name: "john",  
};  
obj2 = {  
  name: "peter",  
};  
  
hello.call(obj, " vizag", " senior trainer");
```

```
hello.call(obj2, "hyd", " developer ");
```

apply

```
function hello(city, profession) {  
  console.log( "hello my name is " + this.name + " iam from " + city + " my profesion is " + profession);  
}
```

```
obj = {  
  name: "john",  
};  
obj2 = {  
  name: "peter",  
};
```

```
hello.apply(obj, [" vizag", " senior trainer"]);  
hello.apply(obj2, ["hyd", " developer "]);
```

bind

```
function hello(city, profession) {  
  console.log( "hello my name is " + this.name + " iam from " + city + " my profesion is " + profession );  
}
```

```
obj = {  
  name: "john",  
};  
var bind1 = hello.bind(obj);  
bind1(" vizag", " senior trainer");
```

Key Differences and Use Cases

call vs apply: The primary difference between call and apply is how arguments are passed:

call passes arguments individually.

apply expects an array of arguments.

Use call when you know the exact arguments to pass, and use apply when you have arguments in an array or array-like object.

bind: Unlike call and apply, bind does not immediately invoke the function. Instead, it creates a new function with the bound this value and optionally prepended arguments. This is useful when you want to create a function with a fixed this value that you can later execute.

Constructor function

It is a old way to create objects

Single same objects

```
function Person(){  
  this.name = "john";  
  this.age = 30;  
}  
  
let person1=new Person();  
console.log(person1);  
  
let person2=new Person();  
console.log(person2);-
```

Multi objects

// constructor function - it is a old way to create objects in js

```
function Person(name, age){  
  this.name = name;
```

```

    this.age = age;
  }

  let person1=new Person("Hemanth", "26");
  console.log(person1);
  let person2=new Person("teja", "25");
  console.log(person2);

```

Prototype

```

function Person(){
  this.name = "john";
  this.age = 30;
}

```

```


Person.prototype.course="trainers"

```

```

let person1=new Person();
console.log(person1.course);
let person2=new Person();
console.log(person2.course)

```



```

function Person(person,age){
  this.person=person;
  this.age=age
}

var obj=new Person("john", 23);
var obj2=new Person("jane",22);

Person.prototype.city="hyderabad";

console.log(obj.city)

```

```

function Person(name, age) {
  this.name = name;
  this.age = age;
}

```

```

function Student(name, age, grade) {
  // Using call to inherit the properties from Person
  Person.call(this, name, age);
  this.grade = grade;
}

```

```

const student1 = new Student('John', 20, 'A');
console.log(student1); // Output: { name: 'John', age: 20, grade: 'A' }

```

Javascript classes

The old way to create objects was using constructor functions.

```

// constructor function
function Person () {
  this.name = "John",
  this.age = 23
}

```

```

// create an object
const person = new Person();

```