Analyze the below code. Do you see any issue? If yes, what is that issue?

const object1 = {

a: 10,

b: 20,

c: function () {

console.log(this.a + this.b);

},

};

const func = object1.c;

func();

The issue in this code arises from the use of **this** within the function **c** when it's invoked through **func()**.

When **func()** is called, it loses its connection to **object1**. As a result, **this** inside the function **c** will not refer to **object1**, and attempting to access **this.a** and **this.b** will lead to **undefined** because **this** refers to the global object (or **undefined** in strict mode).

To solve this issue, you can use **.bind()** to explicitly bind **this** to **object1** when assigning **func**:

**const object1 = {**

**a: 10,**

**b: 20,**

**c: function () {**

**console.log(this.a + this.b);**

**},**

**};**

**const func = object1.c.bind(object1); // binding `this` to object1**

**func(); // This will correctly output 30**

Or you can use arrow functions, which lexically bind **this** to the enclosing context:

**const object1 = {**

**a: 10,**

**b: 20,**

**c: () => {**

**console.log(object1.a + object1.b);**

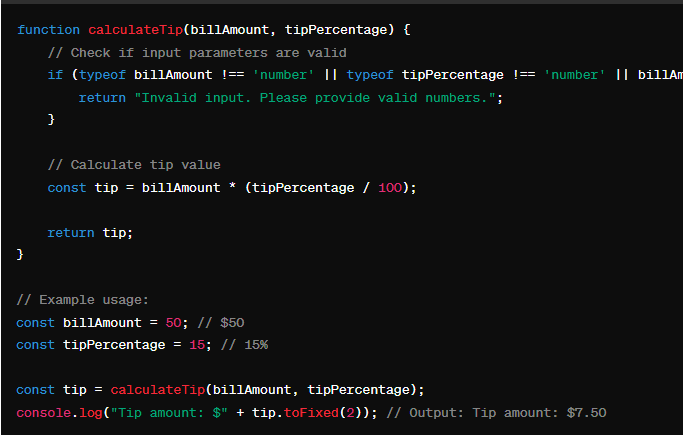
**},**

**};**

**const func = object1.c;**

**func(); // This will also correctly output 30**

Create a JavaScript function that calculates the tip for a given bill amount and tip percentage. Bill amount and tip percentage will be input parameters while output will be calculated tip value.



Implement a simple shopping cart system with features to add items, remove items and calculate the total price. Use objects to represent items, including properties for the item name, price and quantity. Implement features to add items to the cart, remove items and calculate the total cost.

/\*Implement a simple shopping cart system with features to add items, remove items and calculate the total price.

 Use objects to represent items, including properties for the item name, price and quantity.

 Implement features to add items to the cart, remove items and calculate the total cost.\*/

 function Item(name,price,quantity){

    this.name = name;

    this.price = price;

    this.quantity = quantity;

 }

 function shoppingCart(){

    this.items = [];

    this.addItem = function(item){

        this.items.push(item);

    }

    this.removeItem = function(index){

        if(index >= 0 && index <= this.items.length){

            this.items.splice(index,1);

        }

    }

    this.calculateTotal = function(){

        let total = 0;

        this.items.forEach(function(item){

            total += item.price \* item.quantity;

        });

        return total;

    };

 }

// Example usage:

const cart = new shoppingCart();

// Adding items to the cart

const item1 = new Item('Shirt', 20, 2);

const item2 = new Item('Pants', 30, 1);

cart.addItem(item1);

cart.addItem(item2);

// Removing an item from the cart (index 0)

cart.removeItem(0);

// Calculating the total price

const totalPrice = cart.calculateTotal();

console.log('Total Price:', totalPrice);

Find the issue with the below code snippet:

setTimeout(function () {

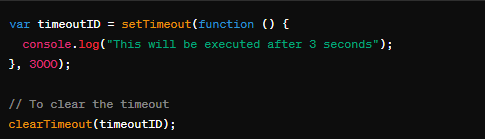
console.log("This will be executed after 3 seconds");

}, 3000);

clearTimeout();

The issue with the provided code snippet is that **clearTimeout()** is being called without specifying which timeout to clear. The **clearTimeout()** function requires a reference to the timeout that was created using **setTimeout()**. Without specifying the timeout ID, the **clearTimeout()** function doesn't know which timeout to clear.

To fix this issue, you need to store the timeout ID returned by **setTimeout()** and then pass it to **clearTimeout()** when you want to cancel it.



What is the issue in the below code:

const fetchData = async () => {

const response = await fetch("https://api.samplewebsite.com/data");

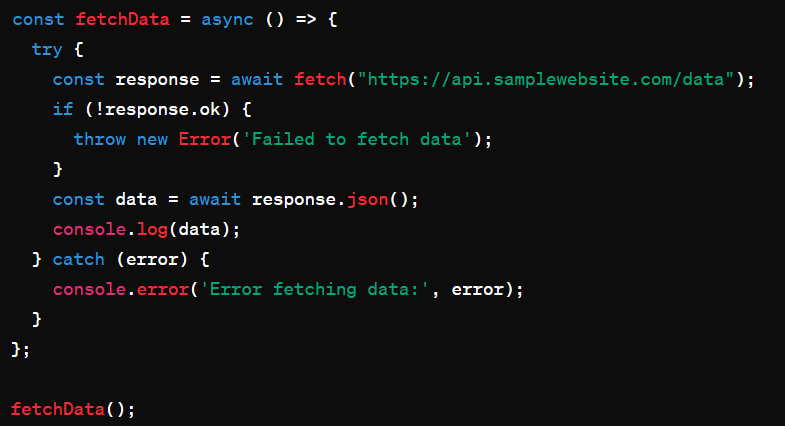
const data = await response.json();

console.log(data);

};

fetchData();

The issue in the provided code is that it assumes the **fetch** request will always be successful. When using asynchronous operations like **fetch**, it's important to handle potential errors that may occur during the network request.



What will be the output of the below code:

const promise1 = Promise.resolve("One");

const promise2 = new Promise((resolve) => setTimeout(() => resolve("Two"), 1000));

const promise3 = Promise.reject("Three");

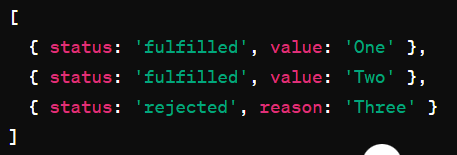
Promise.allSettled([promise1, promise2, promise3]).then((results) => console.log(results));

Here's what will happen in your code:

1. **promise1** is resolved immediately with the value "One".
2. **promise2** is created with a **setTimeout** function to resolve after 1 second with the value "Two".
3. **promise3** is rejected immediately with the reason "Three".

Since **promise1** and **promise3** have immediate resolutions or rejections, they will be settled when **Promise.allSettled()** is called. However, **promise2** takes 1 second to resolve, so it will be settled after **promise1** and **promise3**.

Therefore, the output of the code will be an array containing the results of all promises once they are settled:



What is the issue in the below code:

const obj = {

name: "Conner",

age: 27,

greet: () => {

console.log(`Hey, my name is ${this.name}`);

},

};

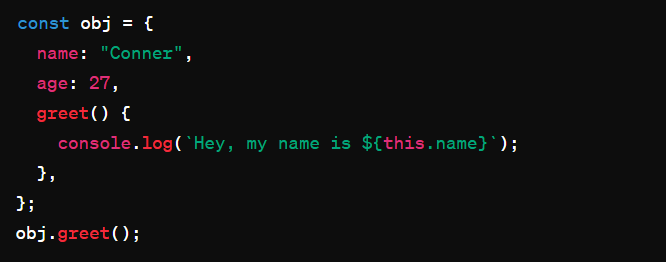
obj.greet();

The issue in the provided code is related to the use of arrow function syntax (**() => { ... }**) for the **greet** method within the **obj** object.

Arrow functions do not have their own **this** context; instead, they inherit **this** from the enclosing lexical context. In this case, since the **greet** method is defined using an arrow function, the **this** inside the **greet** function refers to the **this** value of the enclosing scope where the arrow function is defined.

However, in the global scope or when the object is created, **this** does not refer to the **obj** object. Therefore, **this.name** inside the arrow function resolves to **undefined**, and you'll see output like "Hey, my name is undefined".

To fix this issue, you should use a regular function declaration for **greet** so that **this** refers to the object itself:



What will be the output of below code snippet:

const object1 = {

prop1: "value1",

prop2: {

prop3: "value3",

},

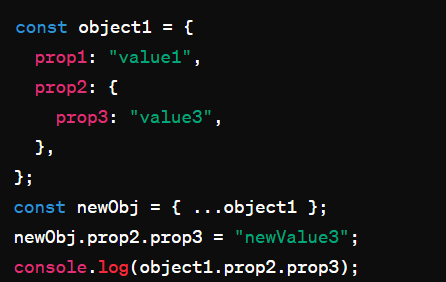
};

const newObj = { ...obj };

newObj.prop2.prop3 = "newValue3";

console.log(object1.prop2.prop3);

ReferenceError: obj is not defined



Since **newObj** is a shallow copy of **object1**, modifying **newObj** will not affect **object1** directly, except for nested objects. Both **newObj** and **object1** have a reference to the same nested object for **prop2**. So, when **newObj.prop2.prop3** is modified, the change is reflected in **object1** as well.

Will the below code return any error? If yes, what will be the error?

class Bird {

constructor(name) {

this.name = name;

}

speak() {

console.log(`${this.name} makes a noise.`);

}

}

class Crow extends Bird{

speak() {

super.speak();

console.log(`${this.name} sings.`);

}

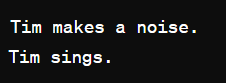
}

const crow = new Crow("Tim");

crow.speak();

The provided code will not return any errors. It defines a class **Bird** with a constructor and a method **speak**, and then extends this class to create a subclass **Crow**. The **Crow** class overrides the **speak** method using the **super** keyword to call the parent class method before adding its own functionality.

When **new Crow("Tim")** is called, it creates a new instance of the **Crow** class with the name "Tim". Then, **crow.speak()** is called, which logs the following output:



This output is a result of calling the **speak** method of the **Crow** class, which in turn calls the **speak** method of the **Bird** class using **super.speak()**, and then adds its own message. So, there will be no errors, and the output will be as described above.

What is wrong with the below code:

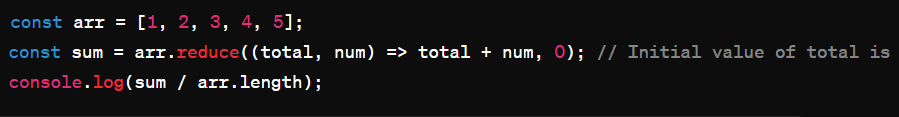
const arr = [1, 2, 3, 4, 5];

const sum = arr.reduce((total, num) => total + num);

console.log(sum / arr.length);

The issue with the provided code is that the **reduce()** function used to calculate the sum is missing the initial value parameter. When you use **reduce()** without providing an initial value, it will start from the first element of the array, which means it will use the first element of the array as the initial value for **total**, and the iteration will start from the second element.

However, in this case, since we are computing the sum of all elements of the array, it's crucial to provide an initial value to the **reduce()** function to avoid unexpected behavior, especially when dealing with empty arrays.



By providing an initial value of 0, the summation will start from 0, and each element will be added to this initial value. This ensures that the calculation is correct, even if the array is empty.

Identify the output of the below code.

const testArray = [1, 2, 3, 4, 5];

const res = testArray.reduce((acc, curr) => {

if (curr % 2 === 0) {

return acc + curr;

}

return acc;

}, 0);

console.log(res);

**6 (calculating sum for the even numbers only)**

Implement an autocomplete feature for a search input field. Given an array of words, write a function that suggests words based on the current input. The output of the function will be an array of suggested words that start with the input characters, limiting the number of suggestions (e.g., a maximum of 7 suggestions).

