Class 24 - Javascript OOPS -1

This keyword
This keyword in browser (strict and non strict)
This keyword in Node.js (strict and non strict)
Arrow Functions
This keyword with Arrow Functions

title: This Keyword

description: this keyword in javaScript and its relevance to OOP

1. JavaScript supports object-oriented programming, a paradigm that helps in organizing and structuring code, especially in large applications. The *this* keyword is a fundamental part of this whole picture
2. Let us see an example
3. Encapsulation is a core principle of object-oriented programming. It involves bundling the data (attributes) and the methods (functions) that operate on the data into a single unit or class.
4. code

```
function Car(make, model) {
    this.make = make;
    this.model = model;
    this.displayInfo = function() {
        console.log(`Car: ${this.make} ${this.model}`);
    };
}
```

```
const myCar = new Car("Toyota", "Corolla");
myCar.displayInfo();  // Outputs: Car: Toyota Corolla
```

5. Role of this: In our Car example, this is used to refer to the specific instance of the class. It allows each method within the class to access the attributes specific to that instance.
6. Using this allows you to create multiple instances of Car each with their own set of data. Without this, you'd end up creating separate functions for each car or passing all information every time you need to perform an operation, which is inefficient and error-prone.
7. Another imp aspect of OOPS is abstraction
    a. Abstraction involves hiding the complex implementation details of a system and exposing only the necessary parts of it.
    b. By using this, the internal details like how properties are stored or managed are hidden. The user of the car object doesn't need to understand how displayInfo accesses the car's make and model; they just need to know that calling displayInfo will print the correct information.
8. Lets talk about Event Handling:
    a. In the context of DOM events, this can refer to the HTML element that received the event, making the handler reusable across elements.
    b. code

```
document.getElementById('myButton').addEventListener('click', function()
{
   this.classList.toggle('active');
});
```

9. Function Reuse with Different Contexts:
    a. This allows the same function to be used in different contexts, meaning the function can operate on different data depending on where and how it's called.
    b. code

```
function greet() {
   console.log(`Hello, I am ${this.name}`);
}

const user = { name: 'Alice' };
const admin = { name: 'Bob' };

// call greet with user context -> Outputs: Hello, I am Alice
// call greet with admin context -> Outputs: Hello, I am Bob
```

    c. We will learn more about this approach later
10. Use with arrow functions
11. Protypal inheritance
12. Will talk about #5 and #6 later

Function Context

1. Let us take a closer look at value of *this* inside functions
2. within a function, the value of this can change based on how the function is invoked.
3. There are different ways in which a function can be invoked

a. **Regular function invocation**: this usually refers to the global object (in non-strict mode) or undefined (in strict mode).
b. **Method invocation**: this refers to the object that the method is called on.
c. **Constructor invocation**: this refers to the instance being created by the constructor function.(future class)
d. call() and apply() methods: Explicitly set this for a function call. (this we will underdtand in future classes)
e. Arrow functions: this retains the value of the enclosing lexical context.(this we will understand today on what are arrow functions

this keyword in node js non strict mode

let's break down the behavior of the this keyword in Node.js in non-strict mode for each of the scenarios.

```
// Scenario 1: Console.log(this)
console.log("Scenario 1:");
console.log(this); // Output: {}

// Scenario 2: Console.log(this) -> fn = global object
console.log("Scenario 2:");
function fnGlobal() {
 console.log(this);
}
fnGlobal();
```

```
// Scenario 3: this -> obj -> fn = object itself
console.log("Scenario 3:");
var obj = {
 fn: function () {
   console.log(this);
 }
};
obj.fn();

// Scenario 4: this -> obj -> fn -> fn = global object
console.log("Scenario 4:");
var obj2 = {
 fn: function () {
   console.log(this);
   var nestedFn = function () {
     console.log(this);
   };
   nestedFn();
 }
};
obj2.fn();
```

Scenario	Code	Output	Explanation
1	`console.log(this);`	`{}`	In global context, `this` refers to the empty object.
2	`function fnGlobal() {...}` `fnGlobal();`	global object (inside the function)	In a regular function, `this` refers to the global object.
3	`obj.fn = function() {...}` `obj.fn();`	object Itself (inside the method)	Inside an object method, `this` refers to the object itself.
4	`obj2.fn = function() {...}` `obj2.fn();`	global object (inside the method) `true` (inside nested function)	Inside a nested function, `this` reverts to the global object.

Understanding these behaviors helps in writing clean and predictable code, especially when dealing with methods and nested functions within objects.

title: This in Browser non strict mode

Scenario 1: console.log(this)

console.log(this); // Window Object

Scenario 2: console.log(this) inside a function

```
function exampleFunction() {
   console.log(this);
}
 exampleFunction(); // Window Object
```

In this case, when you call exampleFunction(), it's being invoked as a regular function. The this inside the function still refers to the (Window in the browser).

Scenario 3: this inside an object method

```
var obj = {
  prop: 'I am a property',
  method: function() {
    console.log(this.prop);
  }
};
 obj.method(); // "I am a property"
```

In this scenario, obj is an object containing a method named method. When you call obj.method(), the this inside the method refers to the obj itself. Therefore, this.prop accesses the prop property of the obj object.

Scenario 4: this inside nested functions

```
var obj = {
  prop: 'I am a property',
  method: function() {
    var nestedFunction = function() {
      console.log(this.prop);
    };
    nestedFunction();
  }
};
 obj.method(); // undefined
```

Here, within the nestedFunction, this refers to the global object (Window in the browser). This is because the function nestedFunction

is not a method of obj. As a result, this.prop will be undefined since the global object doesn't have a prop property.

Scenario	`this` Value	Explanation
`console.log(this)`	Window Object	Global context, `this` refers to the (`Window Object` in browser).
fn({ `console.log(this)` }	Window Object	Inside a regular function, `this` still refers to the window object.
`this` in object method	`obj` (object itself)	Inside a method, `this` refers to the object on which the method is invoked.
`this` in nested function	Window Object	Inside a nested function, `this` refers to the window Object.

Understanding these scenarios is important for grasping how this behaves in different contexts within a browser / node environment.

title:This keyword in node strict mode

1. JavaScript strict mode is a feature that was introduced in ECMAScript 5 (ES5). When you use strict mode in JavaScript, it enforces a stricter set of rules and provides better error handling, which helps developers write cleaner, more secure code and catch common programming mistakes.
2. Some of the key characteristics and benefits of strict mode include:

a. Prevents the use of undeclared variables: In non-strict mode, if you assign a value to a variable without declaring it first with var, let, or const, JavaScript creates a global variable. In strict mode, this behavior is not allowed, and attempting to assign a value to an undeclared variable will result in an error.
b. Disallows duplicate parameter names: In non-strict mode, defining multiple function parameters with the same name is allowed. Strict mode disallows this, preventing potential bugs and improving code readability.

3. To enable strict mode in JavaScript, you can add the string "use strict"; at the beginning of a script, When "use strict"; is present at the beginning of a script file or a function body, strict mode applies to all code within that file or function.

let's break down the behavior of the this keyword in Node.js in strict mode for each of the scenarios

1. console.log(this) (Scenario 1):

```
"use strict";
console.log(this); // Outputs an empty object ({})
```

2. console.log(this) inside a Function (Scenario 2):

```
"use strict";
function myFunction() {
 console.log(this);
}
myFunction(); // Outputs undefined
```

In strict mode, when you call a function without specifying its context (this), it's set to **undefined**. This is different from non-strict mode, where it would point to the global object.

This change prevents functions from unintentionally modifying the global object, which can lead to less predictable code and potential security risks.

3. this Inside an Object Method (Scenario 3):

```
"use strict";
var obj = {
 prop: "I'm a property",
 method: function () {
   console.log(this.prop);
 },
};
obj.method(); // Outputs "I'm a property"
```

Same as before

4. this Inside Nested function Methods (Scenario 4):

```
'use strict';
var obj = {
   prop: 'I am a property',
   method: function() {
     var nestedFunction = function() {
       console.log(this); // undefined
     };
     nestedFunction();
   }
 };
  obj.method(); // undefined
```

title: This keyword in browser strict mode

```
"use strict";

// Scenario 1
console.log("Scenario 1:");
console.log(this); // Output: window

function test(){
   console.log(this); // Output: undefined

}
console.log("scenario 2");
test()

// Scenario 3
var obj = {
 fn: function () {
   console.log(this); // Output: obj
 },
};

console.log("Scenario 3:");
obj.fn(); // Output: obj

// interesting Scenario
const fn = obj.fn;

// Scenario 3 variation
console.log("Scenario 4:");
fn(); // Output: undefined
```

title: Arrow Functions with This Keyword

1. Arrow functions, introduced in ECMAScript 6 (ES6), offer a more concise syntax for writing functions and come with some unique features concerning how they handle this.

```
// Traditional Function Expression
const add1 = function (x, y) {
 const a = 10;
 const val = a + x + y;
 console.log(val);
 return x + y;
};

const add2 = (x, y) => {
 const a = 10;
 const val = a + x + y;
 console.log(val);
 return x + y;
};
```

```
const double = function(x) {
   return x * 2;
}

const doubl2q = x => x * 2;
```

2. In these example,s both functions do the same thing: they take arguments and return the result. The arrow function just looks a bit different and is shorter. You can use arrow functions especially when you have short functions or when you want to keep your code concise

3. But not only to make your code concise arrow function is there , there are some differences as well to note between a regular function and an arrow function on how they handle the this keyword
4. In an arrow function, the value of this is lexically scoped, meaning it's determined by the surrounding scope where the arrow function is defined, rather than how it is called. As a result, arrow functions do not have their own this context; they inherit the this value from the enclosing lexical context.
5. In simple words, check if there is a parent function
    a. If a parent function exists, value of this for arrow function will be derived from the parent function
    b. If no parent function then the value of this is global this

```
const joey = {
 nickName: "Joey",
 arrow: () => console.log(`arrow: ${this.nickName}`),
 regular: function () {
   console.log(`regular: ${this.nickName}`); // Joey
 },
};

// window.nickName = "Ayushee";
joe.arrow(); // undefined
joe.regular(); // Joey
```

6. Let us see another example where arrow functions are useful

```
const joey = {
 nickName: "Joey",
   eventuallySayName: function() {
        function actuallySayName() {
            console.log(`first: ${this.nickName}`);
          }

      // Create an arrow function
      const actuallySayNameWithArrow = () => {
        console.log(`second: ${this.nickName}`); // Joe
      };

      // Call that in 1 second
      setTimeout(actuallySayName, 1000); // undefined
      setTimeout(actuallySayNameWithArrow, 1000); // Joey
    },
 arrow: () => console.log(`arrow: ${this.name}`),
 regular: function () {
   console.log(`regular: ${this.name}`); // Joey
 },
};

// Method invocation forces this === joe here
// window.name = "Ayushee";
joey.arrow();
joey.regular();
joey.eventuallySayName();
```