Shailesh Diwanji

JS

1. this in Global Scope

```
console.log(this);
//=> window (browser, non-strict)
"use strict";
console.log(this);
//=> undefined
```

Explanation :

1. Non-strict mode (browser) → this = window
2. Strict mode → this = undefined
3. Node.js → this = {} (empty object in modules, not global)

Shailesh Diwanji

JS

2. this in Object Methods

```
const user = {
  name: "Shailesh",
  getName() {
    return this.name;
  }
};
console.log(user.getName()); //=> Shailesh
//=>But if you assign the method to a variable → this is lost.
const fn = user.getName;
console.log(fn());
//=> undefined (or window.name if set)
```

Explanation :

1. When a method is called on an object, this refers to that object.

Shailesh Diwanji

JS

3. this in Functions

```
function show() {
  console.log(this);
}
show();
 //=> window (non-strict), undefined (strict)

const obj = { fn: show };
obj.fn(); //=> obj
```

Explanation :

1. In a normal function, this depends on the call site (how it's called).

Shailesh Diwanji

JS

4. this in Arrow Functions

```js
const obj = {
  name: "Shailesh",
  greet: () => console.log(this.name)
};
obj.greet(); //=> undefined (because arrow inherits from global scope)

const obj2 = {
  name: "Shailesh",
  greet() {
    const arrow = () => console.log(this.name);
    arrow();
  }
};
obj2.greet(); //=> "Shailesh"
```

Explanation :

1. Arrow functions don’t have their own this.
2. They capture this from the surrounding lexical scope.

Shailesh Diwanji

JS

5. this in Classes

```
class User {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hello, ${this.name}`);
  }
}
const u = new User("Shailesh");
u.greet(); //=> Hello, Shailesh

//If you pass a method around, you might lose this

const greetFn = u.greet;
greetFn(); //=> undefined

//solution
const greetBound = u.greet.bind(u);
greetBound(); //=> Hello, Shailesh
```

Explanation :

1. Classes in JS are just syntactic sugar over constructor functions.

Shailesh Diwanji

JS

6. this in setTimeout / setInterval

```
setTimeout(function() {
  console.log(this);
//=>  window
}, 1000);

setTimeout(() => {
  console.log(this);
//=> inherits lexical scope
}, 1000);
```

Explanation :

1. In browser timers, the callback runs in the global context.

Shailesh Diwanji

JS

7. this with call, apply, bind

```
function greet(msg) {
  console.log(`${msg}, ${this.name}`);
}
const user = { name: "Shailesh" };

greet.call(user, "Hi");
//=> Hi, Shailesh
greet.apply(user, ["Hey"]);
//=> Hey, Shailesh
const bound = greet.bind(user, "Hello");
bound();
//=> Hello, Shailesh
```

Explanation :

1. call(thisArg, ...args) → invoke immediately with this.
2. apply(thisArg, [args]) → same, but arguments as array.
3. bind(thisArg, ...args) → doesn't call, returns new function with bound this.

Shailesh Diwanji

JS

8. this in Event Handlers

```
document.querySelector("button").addEventListener("click", function() {
  console.log(this);
//=> <button> element
});

document.querySelector("button").addEventListener("click", () => {
  console.log(this);
//=> lexical scope (likely window or parent context)
});
```

Explanation :

1. In DOM event handlers, this = element that fired the event.