this Keyword in JavaScript

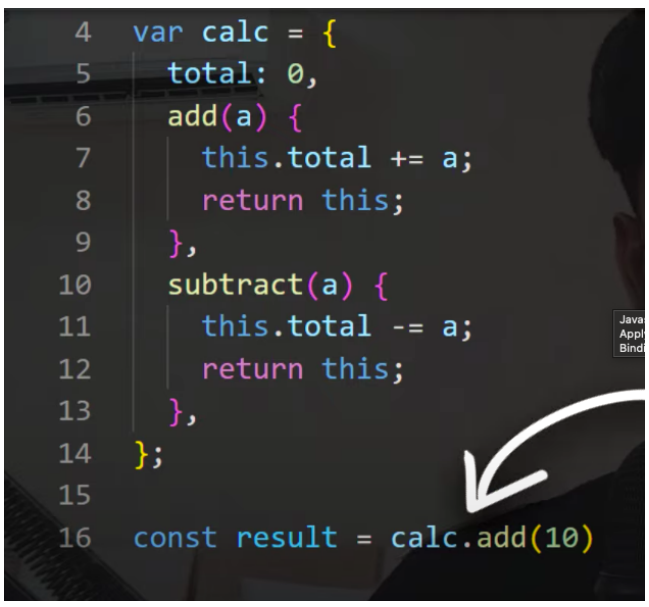
Friday, 1 September 2023 11:02 PM

→ there are 2 types when it comes to object binding in JS :

1) Implicit binding 2) Explicit binding

↳ this is applied when you invoke a fⁿ inside an object using (:) dot operator.

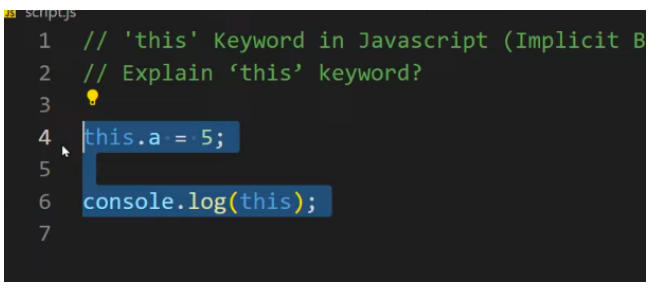
`this` keyword in this scenario will point to the object using which it was invoked i.e object on the left side of the (:) dot.



→ Explicit binding can be applied call(), bind() & apply().

→ 'this' keyword : In Javascript, "this" is used to reference something.

→

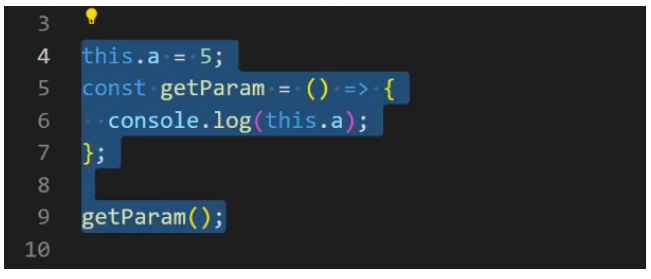


→ `this` keyword in global scope points to the window object.

→

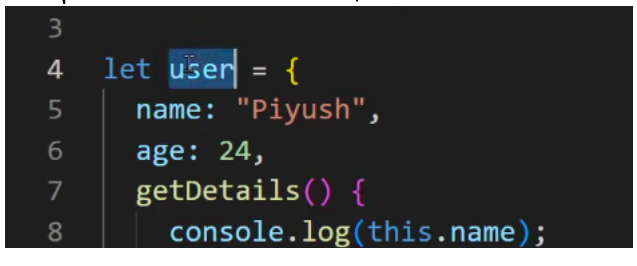
→ `this` keyword in fⁿ scope will point to the global object (ie its parent object). If this fⁿ had been inside any object then `this` would have pointed to that object.

→



→ `this` keyword in arrow fⁿ will point to the global object. `this` in case of arrow fⁿ will point to the outer normal fⁿ if there is one.

→ If there is a normal fⁿ inside an object. Then `this` keyword inside the normal fⁿ points to its parent object.



```
 9    },
10  };
11
12  user.getDetails();
13
```

(o/p is Piyush)

```
 9    },
10  };
11
12  user.getDetails();
13
```

o/p :

script.js:8

```
{name: 'Piyush', age: 24, getDetails: ƒ}
  age: 24
  getDetails: ƒ getDetails()
  name: "Piyush"
  [[Prototype]]: Object
```

→

```
let user = {
  name: "Piyush",
  age: 24,
  childObj: {
    newName: "Roadside Coder",
    getDetails() {
      console.log(this.newName, "and", this.name);
    },
  },
};

user.childObj.getDetails();
```

```
Roadside Coder and undefined    script.js:10
>
```

o/p

→ If the fn is deeply nested then 'this' keyword willonly point to its parent object. This is the case for a normal fn.

→

```
let user = {
  name: "Piyush",
  age: 24,
  getDetails: () => {
    console.log(this.name);
  },
};

user.getDetails();
```

→ But if its an arrow fn, then it prints nothing.

→ If we just console 'this' in line 8, then we get window object.

→ 'this' in case of arrow fn's points to the parent fn which should be a normal fn.

→ eg:

```
let user = {
  name: "Piyush",
  age: 24,
  getDetails() {
    const nestedArrow = () => console.log(this.name);
    nestedArrow();
  },
};

user.getDetails();
```

→ Here nestedArrow() fn takes the value of 'this' from the parent normal fn.

Since, value of 'this' for parent normal fn getDetails() is "user" object, thus, 'this' in case of nestedArrow() fn points to 'user' object & prints "Piyush".

→ If arrow fn will not have a parent normal fn then it will point to global object.

→ 'this' with class :

```
class user {
  constructor(n) {
    this.name = n;
  }

  getName() {
    console.log(this.name);
  }
}

const User = new user("Piyush");

console.log(User);
```

```
user {name: 'Piyush'}    script.js:16
  name: "Piyush"
  [[Prototype]]: Object
>
```

o/p

→ Here when we create a new object from 'user' class in line 14, we basically pass the argument to the constructor fn which creates an object with name property. Thus, we get the above o/p.

→ In getName(), 'this' will refer to everything that is inside the constructor.

→ So if we do User.getName() then o/p will be "Piyush".

→ Output:

```
4  const user = {
5    firstName: "Piyush!",
6    getName() {
7      const firstName = "Piyush Agarwal!";
8      return this.firstName;
9    },
10 };
11
12 console.log(user.getName()); // What is logged?
13
```

→ o/p will be Piyush as "this" keyword in a normal fn refers to its parent object.

→ Output:

```
4  function makeUser() {
5    return {
6      name: "John",
7      ref: this,
8    };
9  }
10
11 let user = makeUser();
12
13 console.log(user.ref.name); // What's the result?
14
```

→ this will print nothing as when we are calling makeUser(), its parent object is the global object which does not have 'name'.

```
4  function makeUser() {
5    return {
6      name: "John",
7      ref: this,
8    };
9  }
10
11 let user = makeUser();
12
13 console.log(user); // What's the result?
14
```

```
script.js:13
▼ {name: 'John', ref: Window} i
    name: "John"
  ▶ ref: Window {window: Window, self: Window,
  ▶ [[Prototype]]: Object
>
```

o/p of above code, thus 'ref' points to window object.

→ How can we fix this, so that ref points to the "name" above it?

→ We can simply make ref a normal fn so, that 'this' inside it points to the parent object & can access "name".

```
4  function makeUser() {
5    return {
6      name: "John",
7      ref() {
8        return this;
9      },
10   };
11 }
12
13 let user = makeUser();
14
15 console.log(user.ref().name); // What's the result?
16
```

→ Output:

```
4  const user = {
5    name: "Piyush Agarwal!",
6    logMessage() {
7      console.log(this.name); // What is logged?
8    },
9  };
10 setTimeout(user.logMessage, 1000);
```

→ Here, the 'user.logMessage' is being used a callback rather than an object's method, so it does not have access to the user object as it is

```
11
```

being run independently, hence, it prints nothing. It points to the window object.

→ To solve this, we can wrap the 'user.logMessage' inside a callback, so that 'user.logMessage' behaves as an object's method, so now it will have access to 'user' object as it is being invoked as method of 'user' object.

```
4  const user = {
5    name: "Piyush Agarwal!",
6    logMessage() {
7      console.log(this.name); // What is logged?
8    },
9  };
10
11 setTimeout(function () {
12   user.logMessage();
13 }, 1000);
14
```

→ Output:

```
4  const user = {
5    name: "Piyush",
6    greet() {
7      return `Hello, ${this.name}!`;
8    },
9    farewell: () => {
10     return `Goodbye, ${this.name}!`;
11   },
12 };
13
14 console.log(user.greet()); // What is logged?
15 console.log(user.farewell()); // What is logged?
```

→ So, for normal fn's, 'this' will point to the object in which it is present.
→ But in case of arrow fn's, it points to the outer normal fn scope, since we don't have any parent normal fn in which farewell() is present

therefore 'this' points to the global object.

→ Create an object calculator:

```
4  let calculator = {
5    read() {
6      this.a = +prompt("a = ", 0);
7      this.b = +prompt("b = ", 0);
8    },
9
10   sum() {
11     return this.a + this.b;
12   },
13
14   mul() {
15     return this.a * this.b;
16   },
17 };
18
19 calculator.read();
20 console.log(calculator.sum());
21 console.log(calculator.mul());
22
```

→ prompt() is just like alert() but it accepts user input.
→ read() behaves like a constructor & takes 2 values
→ '+' besides prompt() converts accepted string to a number.
→ sum() & mul() add & multiply values respectively, & 'this' keyword points to the 'a' & 'b' created inside the object.

→ Output:

```
3
4  var length = 4;
5  function callback() {
6    console.log(this.length); // What is logged?
7  }
8  const object = {
9    length: 5,
10   method(fn) {
11     fn();
12   },
13 };
14 object.method(callback);
15
```

→ o/p is 4
→ since callback() is called inside the method() using regular fn invokation, and during regular fn invokation, 'this' points to the global object, thus, o/p is 4.

Note:

```
javascript                                Copy code

const obj = {
```

→ When there is a fn inside a method of an

```
  outerMethod: function () {
    console.log(this); // 'this' refers to 'obj' in this context

    function innerFunction() {
      console.log(this); // 'this' refers to the global object (e.g., 'windo
    }

    innerFunction();
  }
};

obj.outerMethod();
```

object, then `this` keyword inside method of objectChere, outerMethod() refers to the object.

→ While `this` inside innerFunction() i.e fn inside the method of an object points to the global or window object.

To solve this or have access to `obj` in `this` keyword inside innerFunction() :

1) Use an arrow fn, as arrow f's capture the value of `this` from their enclosing fn.

1. **Use an Arrow Function**: Arrow functions capture the `this` value of their enclosing function, so you can use them to maintain the same `this` context.

javascript — Copy code

```
const obj = {
  outerMethod: function () {
    console.log(this); // 'this' refers to 'obj' in this context

    const innerFunction = () => {
      console.log(this); // 'this' still refers to 'obj'
    };

    innerFunction();
  }
};

obj.outerMethod();
```

2) Store `this` in a variable, since innerFunction() forms closure with outerMethod(), thus, it will have access to `this` context of outerMethod from `self` variable.

1. **Store `this` in a Variable**: You can store the value of `this` in a variable and use that variable inside the inner function.

javascript — Copy code

```
const obj = {
  outerMethod: function () {
    const self = this; // Store 'this' in a variable

    console.log(this); // 'this' refers to 'obj' in this context

    function innerFunction() {
      console.log(self); // Use 'self' variable to access 'obj'
    }

    innerFunction();
  }
};

obj.outerMethod();
```

→ Output :

```
var length = 4;

function callback() {
  console.log(this.length); // What is logged?
}

const object = {
  length: 5,
  method() {
    // arguments = [callback, 2, 3]
    arguments[0]();
  },
};

object.method(callback, 2, 3);
```

→ Normal fn's in JS have arguments which is an array like object.

→ Now, "arguments[0]()" will point to its parent object which in this case will be "[callback, 2, 3]" this array and its length is 3 so o/p will be `3`.

```
script.js:13
Arguments(3) [f, 2, 3, callee: f, Symbol(Symbol.iterator): f]
  0: f callback()
  1: 2
  2: 3
  callee: f method()
  length: 3
  Symbol(Symbol.iterator): f values()
  [[Prototype]]: Object
3                                   script.js:7
>
```

→ If we console arguments we get this o/p.

→ Here we can see that this is an array-like object which has a property "length : 3".

→ Question: Create calc such that we can call following methods & chain them.

```
const result = calc.add(10).multiply(5).subtract(30).add(10);
console.log(result.total);
```

→ ans: What we can do is basically to create a variable, and create methods in which this refers to the object `calc`, now after performing the operation, we return `this` which is basically the object `calc`, through this we can chain methods as after each method call we are returning an object which can call another method.

```
const calc = {
  total: 0,
  add(a) {
    this.total += a;
    return this;
  },
  multiply(a) {
    this.total *= a;
    return this;
  },
  subtract(a) {
    this.total -= a;
    return this;
  },
};
```