JavaScript Function Binding & Closure

Topics Covered:

- Function Binding
- Need for Binding
- this Keyword
- Function Closure
- Closure in Loops

Topics in Detail:

Function Binding

- **Bind()** method is used for **binding a function**.
- **Function Binding** is nothing but **binding a method** from **one object** to **another** object.
- **Bind()** method is used to call a **function** with the **this** value.
- **Bind()** easily sets the **object** to be **bound** with the **this** keyword when the function is **invoked**.

Syntax

```
fn.bind(thisArg[, arg1[, arg2[, ...]]])
```

- **Bind()** will return a **new function** that is the copy of the function **fn.**
- **Binding** that **new function** with that **thisArg** object and arguments **(arg1, arg2, …).**

Need for Binding

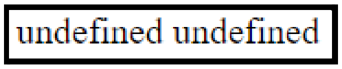
- Whenever **this** keyword is **not bound** to an **object**, we need the **Bind()** method for **function binding.**
- **this** will be **lost** when the function is a **callback function**.

```
const employee = {
  firstName:"Bruce",
  lastName: "Lee",
  display: function() {
    let x = document.getElementById("demo");
    x.innerHTML = this.firstName + " " + this.lastName;
  }
}
setTimeout(employee.display, 3000);
```

In the above example, display() method is called back by setTimeout() method.

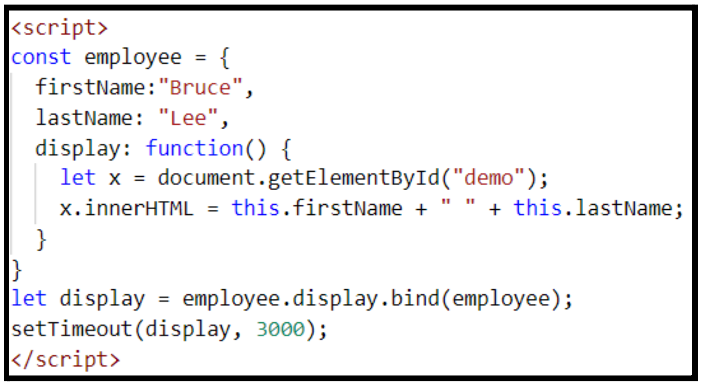
In case of the callback function, this will be lost, and the result will be as below.

Result



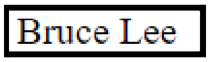
```
undefined undefined
```

This can be resolved by bind() function.



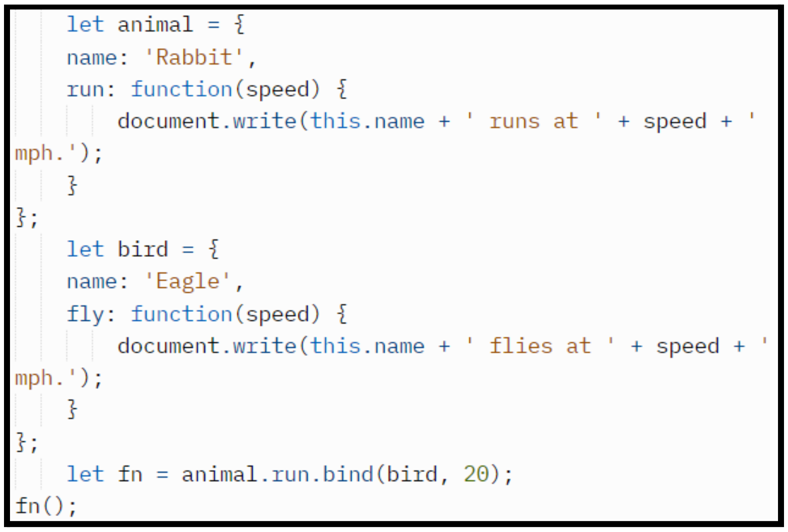
```
<script>
const employee = {
  firstName:"Bruce",
  lastName: "Lee",
  display: function() {
    let x = document.getElementById("demo");
    x.innerHTML = this.firstName + " " + this.lastName;
  }
}
let display = employee.display.bind(employee);
setTimeout(display, 3000);
</script>
```

Result



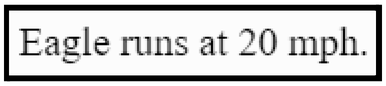
```
Bruce Lee
```

- **Bind()** allows an **object** to **borrow** a **method** from **another object** without making a copy of that method.



```
    let animal = {
    name: 'Rabbit',
    run: function(speed) {
        document.write(this.name + ' runs at ' + speed + '
mph.');
    }
};
    let bird = {
    name: 'Eagle',
    fly: function(speed) {
        document.write(this.name + ' flies at ' + speed + '
mph.');
    }
};
    let fn = animal.run.bind(bird, 20);
fn();
```

Result



```
Eagle runs at 20 mph.
```

Here, the bird object borrows the run method from the animal object.

this Keyword

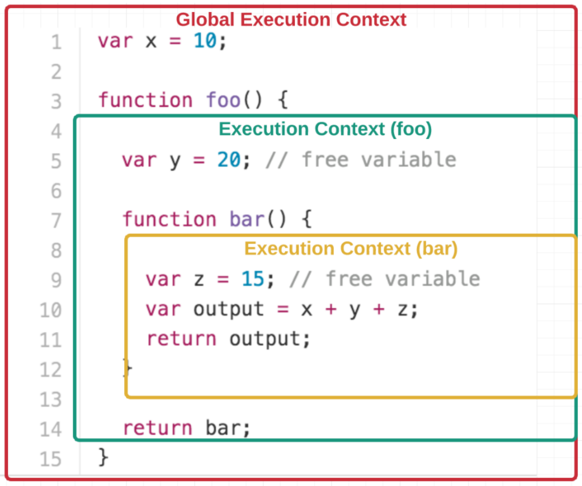
- The **this** keyword refers to an **object**.

The this keyword refers to different objects depending on how it is used:

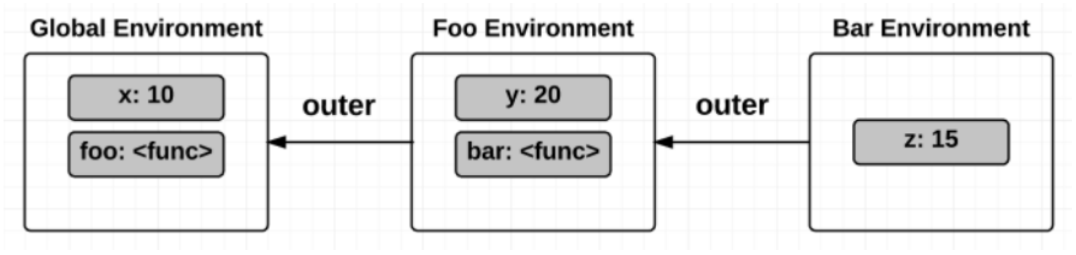
Places used	Reference
object method	this refers to the object
this Keyword	this refers to the global object
function	this refers to the global object
function, in strict mode	this is undefined
Event	this refers to the element that received the event
Methods like call(), apply(), and bind()	this refers to any object

Function Closure

- **Closure** is a feature in which an **inner function** can access the **outer function variable**.
- **Closure** is **created** every time with the **creation of the function**.
- Closure **preserves** the **outer scope** within the **inner scope**.
- Scope chains of Closure:
  - Access to its **own scope**.
  - Access to the **variables** of the **outer function**.
  - Access to the **global variables**.
- **Lexical Scoping** defines the **scope of the variable** depending on the **position** of that variable in source code.



```
var x = 10;

function foo() {

  var y = 20; // free variable

  function bar() {

    var z = 15; // free variable
    var output = x + y + z;
    return output;
  }

  return bar;
}
```



Closure in Loops

In the below example, we will see the difficulties while using closure function in loops

```
for (var index = 1; index <= 3; index++) {
    setTimeout(function () {
        console.log('after ' + index + ' second(s):' + index);
    }, index * 1000);
}
```

Actual Output	Expected Output
after 4 second(s):4 after 4 second(s):4 after 4 second(s):4	after 1 second(s):1 after 2 second(s):2 after 3 second(s):3

- Our intention is to display a message in loop after 1, 2 and 3 seconds at the time of each iteration.
- But We see the **same message after 4 seconds** is that the callback passed to the setTimeout() a closure because the JS engine remembers that last iteration value, i.e 4.
- All **three closures** created by the **for-loop** share the **same global scope** and access the **same value of i**.
- To resolve this issue, we have the below solutions.

Solution 1: immediately invoked function expression

```
for (var index = 1; index <= 3; index++) {
    (function (index) {
        setTimeout(function () {
            console.log('after ' + index + ' second(s):' + index);
        }, index * 1000);
    })(index);
}
```

Solution 2: Using let keyword

```
for (let index = 1; index <= 3; index++) {
    setTimeout(function () {
        console.log('after ' + index + ' second(s):' + index);
    }, index * 1000);
}
```