call, apply and bind methods

Thursday, 24 August 2023 12:33 AM

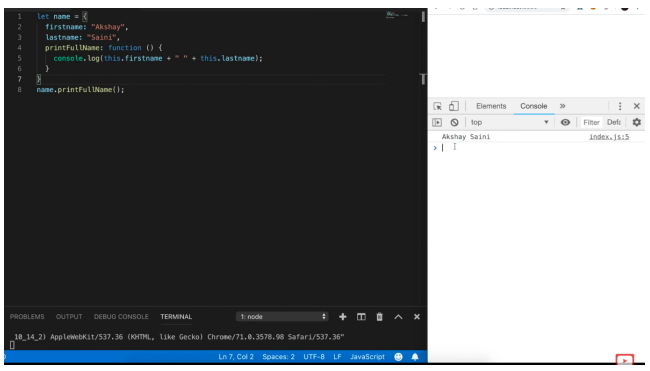
→ Every fⁿ in JS has access to this keyword.

```
let name = {
  firstname: "Akshay",
  lastname: "Saini",
  printFullName: function () {
    console.log(this.firstname + " " + this.lastname);
  }
}
```

→ So this fⁿ has access to this keyword which is here pointing to 'name' object.

→ So o/p of above will be Akshay Saini.

→ How to invoke the above fⁿ?



→ call():

→

```
let name = {
  firstname: "Akshay",
  lastname: "Saini",
  printFullName: function () {
    console.log(this.firstname + " " + this.lastname);
  }
}
name.printFullName();

let name2 = {
  firstname: "Sachin",
  lastname: "Tendulkar",
}
```

→ Now if we have another object and want to print firstName & lastName again, what we can do is copy & paste printFullName() fⁿ.

→ But instead we do function borrowing. So, we can borrow fⁿs from other objects.

→ We do function borrowing using call() method.

→ Each & every fⁿ has access to call() method.

```
// function borrowing
name.printFullName.call(name2);
```

Sachin Tendulkar

→ Syntax for using call() method.

→ Here the 1st argument passed into call is the object which 'name' object's this will be pointing to.

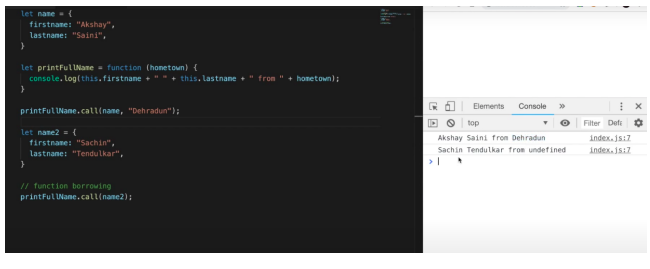
→ So here 'this' will point to 'name2's key: value pairs.

Note: Normally, we don't write fⁿs inside an object & instead declare them outside.

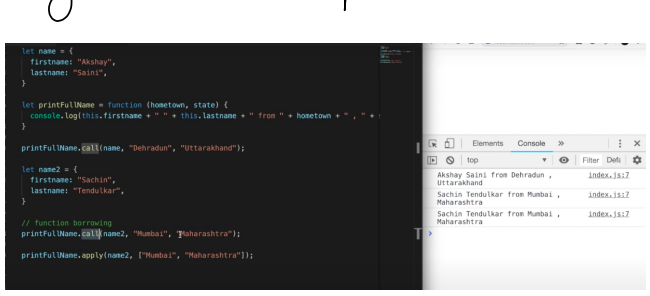
```
let name = {
  firstname: "Akshay",
  lastname: "Saini",
}

let printFullName = function () {
  console.log(this.firstname + " " + this.lastname);
}

printFullName.call(name);
```

→ So, for this case, we will directly call the call() method on the fⁿ & pass the object reference as argument, so that 'this' has access to the attributes of that object.

Note: If we need to pass more arguments to the fⁿ expression, then we can do so by passing these arguments as 2nd or 3rd or so on arguments to the call() method as first argument will always be the object reference.



→ apply(): The only difference between call() & apply() methods are the way we pass arguments. Instead of individually passing the arguments, we pass them as a list in apply(). Rest everything is same.



→ bind(): It looks just like the call method, but instead of directly calling the method, it binds the method to the object & returns a copy of the fⁿ to be invoked later directly.

```
// bind method
let printMyName = printFullName.bind(name2, "Mumbai", "Maharashtra");
console.log(printMyName);
```

o/p:

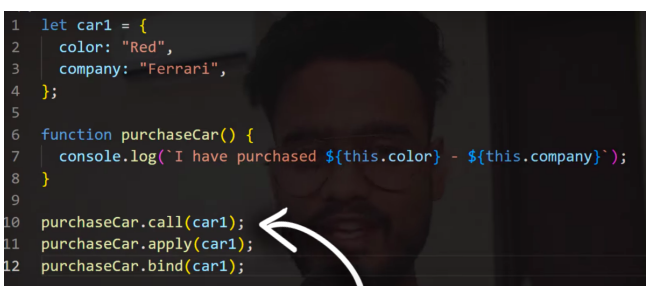
```
f (hometown, state) {                    index.js:24
  console.log(this.firstname + " " +
this.lastname + " from " + hometown + " , " +
state);
}
>
```

```
// bind method
let printMyName = printFullName.bind(name2, "Mumbai", "Maharashtra");
console.log(printMyName);
printMyName();
```

o/p:

```
f (hometown, state) {                    index.js:24
  console.log(this.firstname + " " +
this.lastname + " from " + hometown + " , " +
state);
}
Sachin Tendulkar from Mumbai ,           index.js:7
Maharashtra
```

→ call(), bind() & apply() are used to tie a fⁿ to an object & call the fⁿ if it was a method of the object it was tied to. So, 'this' keyword points that object. These are also used for explicit binding.



→ call():

```
var obj = { name: "Piyush" };
```

→ Now, in normal fⁿ invocation, 'this' points to the

```
function sayHello() {
  return "Hello " + this.name;
}

console.log(sayHello());
```

global object so this will print nothing.

→ So, in order to run sayHello() as method of 'obj' object such that 'this' has access to obj, we will use explicit binding or call() method.

```
var obj = { name: "Piyush" };

function sayHello() {
  return "Hello " + this.name;
}

console.log(sayHello.call(obj));
```

→ If sayHello() had some parameters, then we can send them as args through call() method.

```
var obj = { name: "Piyush" };

function sayHello(age) {
  return "Hello " + this.name + " is " + age;
}

console.log(sayHello.call(obj, 24));
```

→ apply() : It is same as call() ; it just takes arguments as an array.

```
var obj = { name: "Piyush" };

function sayHello(age, profession) {
  return "Hello " + this.name + " is " + age + " and is an " + profession;
}

console.log(sayHello.call(obj, 24, "Software Engineer"));
```

=

```
var obj = { name: "Piyush" };

function sayHello(age, profession) {
  return "Hello " + this.name + " is " + age + " and is an " + profession;
}

console.log(sayHello.apply(obj, [24, "Software Engineer"]));
```

→ bind(): bind() returns us with a fn that can be used later.

```
var obj = { name: "Piyush" };

function sayHello(age, profession) {
  return "Hello " + this.name + " is " + age + " and is an " + profession;
}

const bindFunc = sayHello.bind(obj);

console.log(bindFunc);
```

o/p

```
ƒ sayHello(age, profession) {
  return "Hello " + this.name + " is " + age + " and is an " + profession;
}
```
script.js:12

→ fn borrowing example :

```
const age = 10;

var person = {
  name: "Piyush",
  age: 20,
  getAge: function () {
    return this.age;
  },
};

var person2 = { age: 24 };
console.log(person.getAge.call(person2)); // 24
```

→ So we have explicitly bound getAge() to 'person2'. 'this' inside getAge() method points to 'person2' obj. So getAge() behaves as a method of 'person2'. This also called fn borrowing.

→ Output:

```
var status = "😎";

setTimeout(() => {
  const status = "😍";
```

→ In like 16, 'this' keyword in getStates() points to its parent object (here 'data'). Thus 'undefined' is

```
const data = {
  status: "🦎",
  getStatus() {
    return this.status;
  },
};

console.log(data.getStatus()); // 🦎
console.log(data.getStatus.call(this)); // "😎"
}, 0);
```

the global object.

→ In line 17, 'this' points to the global scope, a inside an arrow fn points to the context of its pa but here we don't have a parent fn thus, it po

→ Call printAnimals() such that it prints all the names & species of objects array:

```
const animals = [
  { species: "Lion", name: "King" },
  { species: "Whale", name: "Queen" },
];

function printAnimals(i) {
  this.print = function () {
    console.log("#" + i + " " + this.species + ": " + this.name);
  };
  this.print();
}
```

```
const animals = [
  { species: "Lion", name: "King" },
  { species: "Whale", name: "Queen" },
];

function printAnimals(i) {
  this.print = function () {
    console.log("#" + i + " " + this.species + ": " + this.name);
  };
  this.print();
}

for (let i = 0; i < animals.length; i++) {
  printAnimals.call(animals[i], i);
}
```

→ We use for loop to iterate over 'animals' ar pass each object inside it using .call() method w takes in the object & another argument as index.
→ So, 'this' keyword in fn printAnimals() ref the object passed as now printAnimals() beh a method of the object that is passed.

→ Append an array to another array:

We can use for loop or .concat() method, but concat() method returns a completely new array. We can also simply do const newArr = [...array, ...e or array.push(...elements)

Another easier way to do this is by using .apply() method.

```
const array = ["a", "b"];
const elements = [0, 1, 2];

array.push.apply(array, elements);

console.log(array);
```

→ So, now the push fn will work on 'array'. In append all elements, we pass 'elements' which is c to 2nd argument of apply() method. So, n push() fn instead of passing whole 'elements' in a single index, pass 'elements's individual elements.

→ Enhancing built-in functions using apply():

Math.max() takes in series of numbers and outputs the maximum i If we pass an array in Math.max() then it returns NaN.

Now, there are other ways to find max. in an array. But w use apply() here.

```js
// Find min/max number in an array
const numbers = [5, 6, 2, 3, 7];

// console.log(Math.max(numbers));

// Loop based algorithm
max = -Infinity, min = +Infinity;

for (let i = 0; i < numbers.length; i++) {
  if (numbers[i] > max) {
    max = numbers[i];
  }
  if (numbers[i] < min) {
    min = numbers[i];
  }
}

```

(using for loop to find max)

```js
// Find min/max number in an array
const numbers = [5, 6, 2, 3, 7];

console.log(Math.max.apply(null,numbers));

```

(using apply() to pass elements of 'numbers' array one by one)
(since we don't need any context, & keep first argument as null)
(object)

→ similary we can do for finding minimum.

→ output:

```js
function f() {
  console.log(this); // ?
}

let user = {
  g: f.bind(null),
};

user.g();

```

→ Now, using "f.bind()" we return a fn & we given the context as null so we have hard fix 'this' keyword's value to the window object
→ when we use .bind() we explicitly bind the f object which can't be changed later. Here we null as context, thus, 'this' keyword will point to w object only (this will be the case even if f() was inside object, still it would have printed window object)

→ If we want 'this' keyword to have access to user object, we m wrap 'f.bind()' inside a fn & give context as 'obj'
→ therefore, 'f.bind(user)' wrapped inside g(){} will make 'this' point to 'user'
→ If you pass 'user' object in line 4 as it is then it will give reference error:

→ Output:

```js
function f() {
  console.log(this.name);
}

f = f.bind({ name: "John" }).bind({ name: "Ann" });

f();

```

→ This will print "John". This is because, a fn once bound to some object can't be bound again. So chaining doesn't actually exist.

→ Impt: line 22 so that 'this' points to the parent object:

```js
function checkPassword(success, failed) {
  let password = prompt("Password?", "");
  if (password == "Roadside Coder") success();
  else failed();
}

```

```js
function checkPassword(success, failed) {
  let password = prompt("Password?", "");
  if (password == "Roadside Coder") success();
  else failed();
}

let user = {
```

```
let user = {
  name: "Piyush Agarwal",

  loginSuccessful() {
    console.log(`${this.name} logged in`);
  },

  loginFailed() {
    console.log(`${this.name} failed to log in`);
  },
};

checkPassword(user.loginSuccessful, user.loginFailed);
```

→ Here 'this' won't point to 'user' object, it instead points to the global object.

```
  name: "Piyush Agarwal",

  loginSuccessful() {
    console.log(`${this.name} logged in`);
  },

  loginFailed() {
    console.log(`${this.name} failed to log in`);
  },
};

checkPassword(user.loginSuccessful.bind(user), user.loginFailed.
```

→ So we basically bind the fn to t object using bind(), so that 'this' poi 'user' object.

→ this happens because context of 'this' depends on where the fn it called. Also, fn is passed as callback it loses its context of 'this' because fn is being called

→ We can make 'this' keyword remember its context using (when fn is pc callback)

1) bind() method as mentioned above

2) Using arrow fn's: As 'this' keyword in this case points to the parent's context.

```
const obj = {
  data: "Hello",
  logData: function () {
    setTimeout(() => {
      console.log(this.data);
    }, 1000);
  },
};

obj.logData(); // Will log "Hello" after 1 second
```

3) By saving this in some variable & then passing the variable. As this forms closure.

```
const obj = {
  data: "Hello",
  logData: function () {
    const self = this;
    setTimeout(function () {
      console.log(self.data);
    }, 1000);
  },
};

obj.logData(); // Will log "Hello" after 1 second
```

→ Partial application for login fn:

→

```
function checkPassword(ok, fail) {
  let password = prompt("Password?", "");
  if (password == "Roadside Coder") ok();
  else fail();
}

let user = {
  name: "Piyush Agarwal",

  login(result) {
    console.log(this.name + (result ? " login successful" : " login
  },
};

// checkPassword(?, ?);
```

Now there is only 1 fn login() but we need to p 2 callbacks to checkPassword.

So how to call checkPassword()?

→

```
function checkPassword(ok, fail) {
  let password = prompt("Password?", "");
  if (password == "Roadside Coder") ok();
  else fail();
}

let user = {
  name: "Piyush Agarwal",

  login(result) {
    console.log(this.name + (result ? " login successful" : " login
  },
};

checkPassword(user.login(true), user.login(false));
```

If we do this then these we will be executed t & then only at line 18

→ And o/p will be:

```
Piyush Agarwal login successful        script.js:14
Piyush Agarwal login failed            script.js:14
>
```

(even before typing anything in alert box)
(it gives error if we type in alert box, that says fail() is not defined)

→ Since we are supposed to return a fn in login, to be used as callback we use bind(). And we will also provide the arguments.

```
3
4  function checkPassword(ok, fail) {
5    let password = prompt("Password?", "");
6    if (password == "Roadside Coder") ok();
7    else fail();
8  }
9
10 let user = {
11   name: "Piyush Agarwal",
12
13   login(result) {
14     console.log(this.name + (result ? " login successful" : " login f
15   },
16 };
17
18 checkPassword(user.login.bind(user, true), user.login.bind(use
19
```

→ Explicit binding with arrow fns:

```
3
4  const age = 10;
5
6  var person = {
7    name: "Piyush",
8    age: 20,
9    getAgeArrow: () => console.log(this.age),
10   getAge: function () {
11     console.log(this.age);
12   },
13 };
14
15 var person2 = { age: 24 };
16
17 person.getAgeArrow.call(person2); // ?
18 person.getAge.call(person2); // 24
19
```

→ Line 18 will print '24' as it's a normal fn, so 'this' keyword points to the 'person' object (parent object)

→ Line 17 will give Undefined, as 'this' key in case of arrow fns, point to the context of their parent fn, but since there is no normal parent fn, thus 'this' points to the global object.

→ Therefore, call(), bind() & apply() or explicit binding doesn't work with arrow fns.