Class-27 Call, Apply , Bind and their Polyfills

title: Call Method

Before Moving to the Call method , Let's first see a basic Object example , We will create a basic object method and will call it

```
const person1 = {
 name: "Mr X",
 age: 25,

 printNameAndAge: function () {
   console.log(`My name is ${this.name} and I am ${this.age} years old`);
 },
};

person1.printNameAndAge();
```

What will this code do?

this keyword over here in this object is inside a method , So it will point to the object itself and we can access properties of this object by just accessing the key for the value

So the output will be

My name is Mr X and I am 25 years old

What happens if we add below to the code

```
const person2 = {
   name : MRr Y,
   age : 30
}

person1.printNameAndAge()


person2.printNameandAge()
```

What happens here?

The output for this code will be-

My name is Mr X and I am 25 years old
Uncaught TypeError: person2.printNameandAge is not a function

because for the person1 object the method is defined so the this keyword can access the properties of the object,and prinyts the desired result

but for person2 object there is no method defined and by the name printNameandAge

there is another way in which you can call the method printAgeandName for person2 without even defining it for the person2 object!

In situations like this we can use the **call** method

CALL

Now what is the call method and what it does?

1. In JavaScript, the call() method is a powerful function used to invoke (call) a function and explicitly set what *this* refers to within that function.
2. Simply put It allows you to specify the value of this within a function
3. Imagine you have a function that needs to behave as if it belongs to an object, even though it wasn't originally written as part of that object.
4. The call() method allows you to do just that. It lets you run the function as if it is a method of any object you choose by specifying that object as the first argument of call(). it basically sets the context of the this keyword for the Object you are trying to call the method for
5. Let's use it
6. To allow person2 to use the printNameAndAge method from person1, we will utilize the JavaScript call method

```
// Using the call method to allow person2 to use person1's method
person1.printNameAndAge.call(person2);
```

Code explanation

1. How the call Method Works:
2. Identify the Function: First, you identify the function you want to invoke. In this case, it's person1.printNameAndAge.
3. The call Method: You apply .call() to this function. This method allows you to specify the context (this value) in which the function should be executed. The first argument you pass to call becomes the this inside that function.
4. Setting the Context: **By writing person1.printNameAndAge.call(person2);, you are telling JavaScript to execute person1's printNameAndAge method but with person2 as the this context.** That means this.name and this.age within printNameAndAge now refer to the properties of person2.
5. Function Execution: The function runs as if person2 has its own printNameAndAge method, displaying "My name is Mark and I am 30 years old"
6. Not Only this you can also pass arguments with the call method , let's see how passing arguments works
7.

Example with Additional Parameters:

First, let's extend the printNameAndAge method to accept an additional parameter, location, and incorporate that into the code:

```
const person1 = {
   name: 'Mr X',
   age: 25,

   printNameAndAge: function(location) {
       console.log(`My name is ${this.name}, I am ${this.age} years old, and I live in
${location}.`);
   },
};

const person2 = {
   name: 'Mr Y',
   age: 30
};

// Using printNameAndAge with person1's context and an additional argument
person1.printNameAndAge('New York');

// Using call to invoke person1's method on person2 with an additional argument
person1.printNameAndAge.call(person2, 'San Francisco');
```

How Arguments are Passed with the call Method:

1. Function and Context: As before, you start with the function you want to invoke, person1.printNameAndAge, and use the call method.
2. Setting the Context: **The first argument of call sets the this context**. In the example person1.printNameAndAge.call(person2, 'San Francisco');, person2 is used as the context. Thus, inside the function, this.name resolves to 'Mark' and this.age to 30.
3. Passing Additional Arguments: After the context argument, any subsequent arguments you pass to call are passed as arguments to the function being invoked. In the example, the string 'San Francisco' is passed as the location parameter to the

printNameAndAge function. Therefore, the function outputs: My name is Mark, I am 30 years old, and I live in San Francisco.

4. The ability to specify both the context and additional arguments makes call very powerful and flexible for function invocation in JavaScript, especially for methods that need to be shared or reused across different objects with similar structures but different data.

title: Apply Method

1. The apply method in JavaScript is similar to the call method but differs in how it handles arguments passed to the function.
2. Both methods are used to invoke a function with a specified this context, allowing you to control what this refers to within the function.
3. However, **while call requires arguments to be listed explicitly, apply takes arguments as an array**, making it suitable for situations where the number of arguments is not fixed or when you don't know the number of arguments in advance or they are already in an array form..
4. Here's a breakdown of how the apply method works using a similar example:
5. First, let's create an object with a method that we might want to invoke with a different this contex

```
const person1 = {
   name: 'Mr X',
   age: 25,
   describe: function() {
       console.log(`My name is ${this.name} and I am ${this.age} years old`);
   }
};

person1.describe();  // Output: My name is Mrinal and I am 25 years old
```

6. Suppose we have another object, person2, and we want to use the describe method from person1 but with person2 as the this context:

```
const person2 = {
   name: 'Mr Y',
   age: 30
};
```

7. Using the apply Method
8. To use person1's describe method with person2 as the context, we can use apply:
9. person1.describe.apply(person2); // Output: My name is Mr Y and I am 30 years old
10. Here, apply allows person2 to use person1's describe method. The apply method does not need any additional arguments for this case, but if there were any, they would be passed as an array.

Example with additional params

```
const person1 = {
name: "Mr X",
age: 25,
 describe: function (location, hobby) {
   console.log(
     `My name is ${this.name}, I am ${this.age} years old, I live in ${location}, and
my hobby is ${hobby}.`
   );
 },
};

// Using describe with person1's context and additional arguments
person1.describe("New Delhi", "coding");

const person2 = {
 name: "Mr Y",
 age: 30,
};

// Using apply to invoke person1's method on person2 with additional arguments
person1.describe.apply(person2, ["San Francisco", "Travelling
"]);
```

1. Let's see an Example of Why we need apply method and where it can be useful and what do I mean by when I say it can handle dynamic data
2. The apply method is especially useful in scenarios where you need to pass an array of arguments to a function, which might not be known until runtime, or when working with functions that accept a variable number of arguments.

Scenario: Calculating Maximum of Array Elements

Imagine you need to find the maximum number in an array. JavaScript provides the Math.max() function, which returns the largest of zero or more numbers. However, Math.max() does not accept an array directly; it expects numbers as separate arguments.

```javascript
const numbers = [5, 6, 2, 3, 7];

// Using Math.max directly with an array does not work:
console.log(Math.max(numbers)); // NaN - because Math.max expects separate number
arguments, not a single array.

// Using apply to spread the numbers array into individual arguments:
const maxNumber = Math.max.apply(null, numbers);

console.log(maxNumber);  // Output: 7
```

Why apply Is Needed

Dynamic Argument Lists: In this case, the numbers array could change in size dynamically. With apply, you can pass any array of numbers regardless of its size, and Math.max will be called as if you wrote Math.max(5, 6, 2, 3, 7).

Ease of Use: Using apply is simpler and cleaner than manually spreading an array into function arguments or using loops to determine the maximum value.

Comparison with call

Using call, you would have to explicitly know and pass each element of the array as a separate argument, which is impractical for large or dynamically changing arrays:

```
// Hypothetically using call for a known, fixed set of arguments:
const maxNumberUsingCall = Math.max.call(null, 5, 6, 2, 3, 7);  // This works but is not dynamic.

console.log(maxNumberUsingCall);  // Output: 7
```

However, with the introduction of ES6 spread syntax, you can achieve a similar result to apply with cleaner syntax, making the spread operator often a better choice for situations like this:

```
// Using ES6 spread operator:
const maxNumberUsingSpread = Math.max(...numbers);

console.log(maxNumberUsingSpread);  // Output: 7
```

title: Bind Method

1. The bind method in JavaScript is another important function from the family of methods that control the context of this and how functions are called.
2. It's especially useful when you need to ensure that a function's this context is permanently set, no matter where or how the function is called later

3. In other words, The bind method in JavaScript is like giving a function a reminder of who it is talking about. When you have a function that uses this to refer to an object, bind helps make sure that this always refers to the object you want, even if the function is used in different places or in different ways.

```
let user = {
   name: 'Alice',
   age: 25,
   greet: function() {
       console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
   }
};

// We need this function to always use `user` as its context
let boundGreet = user.greet.bind(user);

// Now, no matter how we call `boundGreet`, it always logs the correct message.
boundGreet();  // Outputs: Hello, my name is Alice and I am 25 years old.
```

Alice is an object with properties name and age, and a method called introduce.
The introduce method prints a message about Alice using this.name and this.age.

We use bind to create a new function introduceAlice that "remembers" to talk about Alice, no matter where we use it.

If you were to use introduceAlice() somewhere in your program, it would always print Alice's introduction, ensuring that this inside introduce always refers to Alice.

If we would create another user2 and bind greet to another user2

```
let user = {
   name: 'Alice',
   age: 25,
   greet: function() {
       console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
   }
};

const user2 = {
   name: 'Bob',
   age: 30
};

// We need this function to always use `user` as its context
let boundGreet = user.greet.bind(user);
const boundGreet2 = user.greet.bind(user2);

// Now, no matter how we call `boundGreet`, it always logs the correct message.
boundGreet();  // Outputs: Hello, my name is Alice and I am 25 years old.
boundGreet2(); // Outputs: Hello, my name is Bob and I am 30 years old.
```

Using bind in Event Handlers

This example will demonstrate how to use bind to ensure that a method maintains its context (this) when used as an event handler in a web application.

First, we'll need a simple HTML button that users can click to trigger the event:

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Bind Method Example</title>
</head>
<body>
   <button id="introduceButton">Introduce Alice</button>

   <script src="index.js"></script>
</body>
</html>
```

Now, let's write the JavaScript code that defines an object and uses the bind method for an event handler. This script should be saved in a file named script.js which is referenced in the HTML above:

```
const alice = {
   name: 'Alice',
   age: 30,
   introduce: function() {
       // console.log(this)
       console.log(`My name is ${this.name} and I am ${this.age} years old.`);
   }
};

// Get the button element from the HTML
const button = document.getElementById('introduceButton');

// Add an event listener to the button
// We use bind to ensure 'this' inside introduce refers to 'alice'
// button.addEventListener('click', alice.introduce);
const boundIntroduce = alice.introduce.bind(alice);
button.addEventListener('click', boundIntroduce);
```

title: Simple Polyfill of Call Method

Suppose we do not have support for the call method and now We need to create our own Polyfill for the Call method

Polyfills in programming are pieces of code (usually JavaScript on the web) that provide modern functionality on older browsers that do not natively support it. Essentially, a polyfill is a fallback implementation for features that are expected in modern environments but are missing from the user's environment because they are using an outdated browser or platform.

Like suppose in our modern Browsers, We are able to use Map Filter and Reduce etc. methods, but maybe in some browser this support is not provided. So you will have to write your own implementation of these functions

so let's start with writing polyfills for these three methods , first let's start with Map

```
let car = {
   name: "Mercedes",
   color: "White",
 };
 function buyCar(price) {
  console.log(`I bought a ${this.color} ${this.name} of ${price} `);
}
```

I have this simple code and now I want to call the buyCar function in context of the car object but it should be done from my implementation of the call method , let's try building that

Intuition

1. First we need to define a custom method on function's prototype so that all the functions can call this custom method

   ```
   Function.prototype.myCall = function (context = {}, ...args) {
   ```

   a. This line adds a new method myCall to the Function prototype, making it available to all functions. The myCall method takes an initial argument context which will be used as the *this* context when the function is called. If context is not provided, it defaults to an empty object ({}). The ...args uses rest parameters syntax to collect all remaining arguments passed into an array.

   b. This syntax tells JavaScript to gather the rest of the arguments passed to myCall into an array named args.

   c. This is useful because the original call method can accept an indefinite number of arguments, which are passed to the function after the first argument (which sets the context).

Here, ...args captures all those arguments whatever their number.

2. Then we need to ensure that only functions can call this custom method

```
Function.prototype.myCall = function (context = {}, ...args) {
   if(typeof this !== 'function'){
        throw new Error (this + 'Is not callable')
    }
};
```

   a. This section checks if the method where myCall is being applied is actually a function. If this (referring to the function on which myCall is invoked) is not a function, it throws an error. This ensures that myCall is only used with functions.

3. Assigning the called function

```
Function.prototype.myCall = function (context = {}, ...args) {
   if(typeof this !== 'function'){
       throw new Error (this + 'Is not callable')
   }
    context.myFunction = this; // buyCar
};
```

   a. Here, the function (referred to by this) is assigned to a property myFunction on the context object

4. Executing the function with the context and arguments:

```
Function.prototype.myCall = function (context = {}, ...args) {
   if(typeof this !== 'function'){
       throw new Error (this + 'Is not callable')
   }
   context.myFunction = this // buyCar -> car.myFunction =
buyCar
    context.myFunction(...args) // buyCar(5000000)
    // this(...args)

};
```

a. The function is then called as a method of context using the spread syntax to pass the collected args. This line is where the function that myCall was called on actually executes, using the provided context as this and args as its arguments.

Example usage

```
let car = {
 name: "Mercedes",
 color: "White",
};

function buyCar(price) {
 console.log(`I bought a ${this.color} ${this.name} of ${price} `);
}

Function.prototype.myCall = function (context = {}, ...args) {
 //   console.log(this)
 if (typeof this !== "function") {
   throw new Error(this + "Is not callable");
 }
 // context -> car
 context.myFunction = this; // buyCar
 context.myFunction(...args);
};

buyCar.myCall(car, "5000000");
```

Code breakdown

myCall function is called on buyCar: buyCar.myCall(car, "5000000") is effectively calling myCall with this set to the buyCar function, context set to the car object, and args containing the string "5000000".

Setting the context: Inside myCall, context.myFunction = this; assigns the buyCar function to a new property myFunction on the car object.

Invoking the function: context.myFunction(...args); translates to car.myFunction("5000000");. Here, car.myFunction refers to the buyCar function, and since it is called as a method of car, this inside buyCar refers to the car object.

Function execution: The buyCar function executes with this bound to the car object. Therefore, this.name is "Mercedes" and this.color is "White". The argument "5000000" is passed as price.

Output: The function prints:

I bought a White Mercedes of 5000000

title: Simple Polyfill of Apply Method

To create a polyfill for the apply method, which is similar to the call method but takes an array of arguments rather than a list of arguments, we can adapt the previous approach used for myCall.

The apply method is primarily different in how it handles the arguments passed to the function, accepting an array rather than a comma-separated list.

So an additional check for array is needed

```
Function.prototype.myApply = function (context = {}, argsArray = []) {
 if (typeof this !== "function") {
   throw new Error(this + " is not callable");
 }

 if (!Array.isArray(args)) {
   throw new Error(this + "We need an array for args");
 }
/**
     * car.myFunction = buyCar
     */
   context.myFunction = this
   /**
     * car.myFunction(5000000)
     */
   context.myFunction(...argsArray) // sprad the array and invoking
the function

};
```

Example usage

```
Function.prototype.myApply = function (context = {}, argsArray = []) {
 if (typeof this !== "function") {
   throw new Error(this + " is not callable");
 }

 if (!Array.isArray(argsArray)) {
   throw new Error(this + "We need an array for args");
 }

/**
    * car.myFunction = buyCar
    */
   context.myFunction = this
   /**
    * car.myFunction(5000000)
    */
   context.myFunction(...argsArray) // spread the array and invoking the function


let car = {
   name: "Mercedes",
   color: "White",
};

function buyCar(price) {
   console.log(`I bought a ${this.color} ${this.name} for ${price}`);
}

buyCar.myApply(car, ["3000000"]);
```

title: Simple Polyfill of Bind Method

The bind method in JavaScript is used to create a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called. Here's how you can create a polyfill for the bind method:

1. Function.prototype.myBind definition:

   ```
   Function.prototype.myBind = function (context, ...boundArgs) {
   ```

   This adds a method myBind to the Function prototype, making it available to all functions. The myBind method takes a context object which will become the this value when the new function is called. The ...boundArgs collects any additional arguments passed to myBind, which will be the initial arguments to the target function when the bound function is called.

2. Type check:

   ```
   Function.prototype.myBind = function (context, ...boundArgs) {
     if (typeof this !== 'function') {
         throw new Error(this + ' is not callable');
     }
   }
   ```

   a. This check ensures that myBind is applied only to functions, maintaining consistency with the original bind method by throwing an error if it's used incorrectly.
3. Assigning the function to this

   ```
   const targetFunction = this;  // The function on which myBind
   is called
   ```

4. Now return a function that can be called later

```
Function.prototype.myBind = function (context, ...boundArgs) {
   if (typeof this !== 'function') {
       throw new Error(this + ' is not callable');
   }
           const targetFunction = this;  // The function on which myBind is
called

   return function (...args) {
        return targetFunction.apply(context, [...boundArgs,
...args]);
    };
}
```

This section creates and returns a new function. When this new function is called, it calls targetFunction (the function on which myBind was invoked) using the apply method. The apply method is called with the context set as this and a combined array of boundArgs and any new arguments passed to the bound function (...args).

Example usage

```
Function.prototype.myBind = function (context, ...boundArgs) {
   if (typeof this !== 'function') {
       throw new Error(this + ' is not callable');
   }
   const targetFunction = this;  // The function on which myBind is called

   return function (...args) {
       return targetFunction.apply(context, [...boundArgs, ...args]);
   };
}

let car = {
   name: "Mercedes",
   color: "White",
};

function buyCar(price, year) {
   console.log(`I bought a ${this.color} ${this.name} for ${price}
made in ${year}`);
}

const buyMyCar = buyCar.myBind(car, "3000000");
buyMyCar("2020");
```