

Chapter 1: Call, apply & bind method

Let's take some examples to understand call apply & bind method.

Code snippet 1: Creating a name object with some properties

```
let name = {  
  firstName: "Lycan",  
  lastName: "Mishra",  
  getFullName: function () {  
    console.log(this.firstName + " " + this.lastName); // this points to current object name  
  },  
};
```

```
name.getFullName(); // Lycan Mishra
```

Code snippet 2: Creating name2 object with same properties as name.

```
let name2 = {  
  firstName: "Sachin",  
  lastName: "TendulKar",  
  getFullName: function () {  
    console.log(this.firstName + " " + this.lastName); // this points to current object name2  
  },  
};
```

```
name2.getFullName(); // Sachin TendulKar
```

In name2 Object, instead of creating 'getFullName' method, we can borrow it from name object. That's where call, apply and bind methods come in. Using them, we can make function borrowing.

***** Function Borrowing by call method *****

Syntax –

`<Lender>.<FunctionName>.call(<Borrower>,<function argument(s) separated by comma>)`

Lender is the object which lends the function. **FunctionName** is the name of the function to be borrowed. **Borrower** is the object/instance which borrows the function and It points to the current object, more like this keyword, **function argument(s)** are the arguments given to a function call and its optional.

Case 1 - Borrowing function from an object

```
let name1 = {
  firstName: "Lycan",
  lastName: "Mishra",
  getFullName: function () {
    console.log(this.firstName + " " + this.lastName);
  },
};
```

`name1.getFullName()`

```
let name2 = {
  firstName: "Sachin",
  lastName: "Tendulkar"
};
```

Now we need to call **name2 object method getFullName**. At present this method is unavailable inside name2. So, we need to borrow this method from name1 object and then call it explicitly using call (). Below is the code which invokes getFullName method of name2 object.

`name1.getFullName.call(name2)`

Q: How call () is working behind the scene?

Ans: call () borrows getFullName method from name1 object. Makes it available inside name2 object and then calls it via name2 object instance.

Case 2 - Borrowing function from global scope

```
function getFullName() { // global scope
  console.log(this.firstName + " " + this.lastName);
}
```

```
let name1 = {
  firstName: "Lycan",
  lastName: "Mishra",
};
```

```
let name2 = {
  firstName: "Sachin",
  lastName: "Tendulkar",
};
```

```
getFullName.call(name1); // Lycan Mishra
getFullName.call(name2); // Sachin Tendulkar
```

Case 3 - borrowing function from global scope + function taking arguments

```
let name1 = {
  firstName: "Lycan",
  lastName: "Mishra",
};
```

```
function getFullName(hometown, state) {
  console.log(
    this.firstName +
    " " +
    this.lastName +
    " is from " +
    hometown +
    " from state " +
    state
  );
}
```

```
getFullName.call(name1, "Sambalpur", "odisha"); // Lycan Mishra is from Sambalpur from state Odisha
```

***** Function Borrowing by **apply** method *****

Apply method is exactly similar to call method. The only difference is while passing arguments apply method takes them in an array format whereas call method takes them individually separated by comma.

```
getFullName.call(name1, "Sambalpur", "odisha");
```

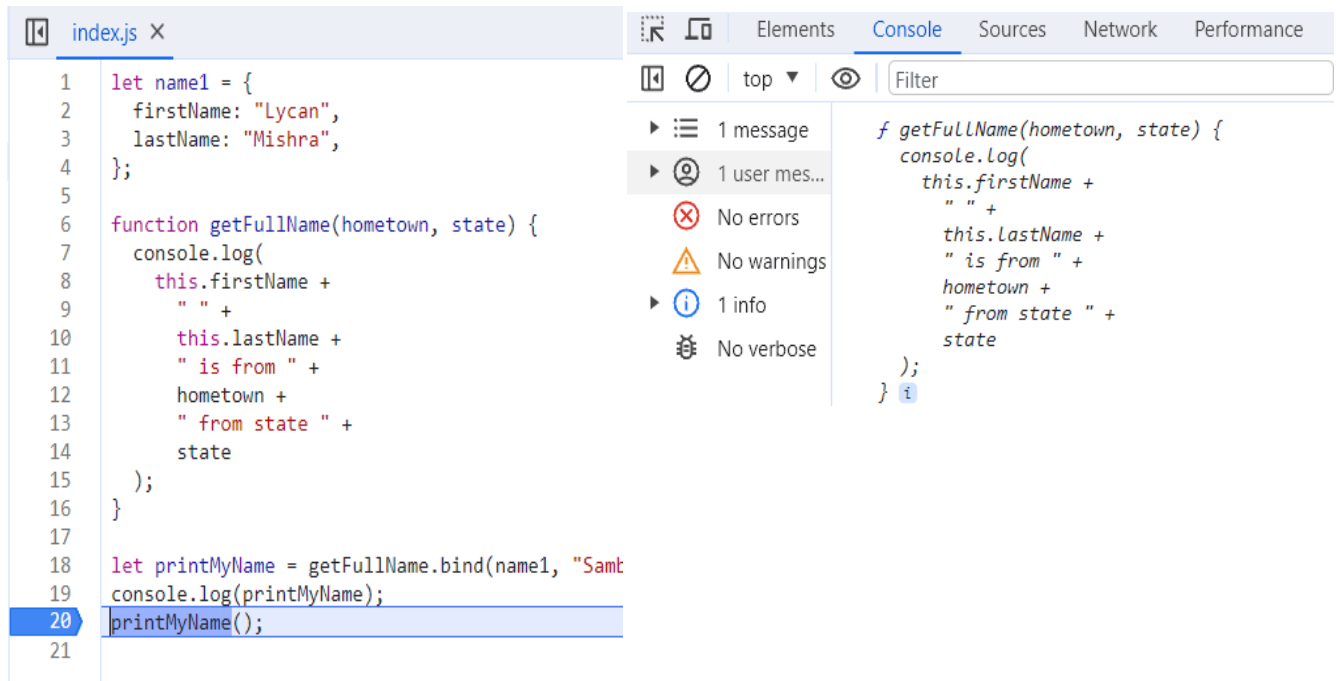
```
getFullName.apply(name1, ["Sambalpur", "odisha"]);
```

***** Function Borrowing by **bind** method *****

bind method is exactly similar to call method. The only difference is **bind** does not invoke the function directly but it returns a function which can be invoked separately, whereas **call** invokes the function directly. **bind keeps a copy of a function which can be invoked later.**

```
getFullName.call(name1, "Sambalpur", "odisha"); // direct function invocation
```

```
let printMyName = getFullName.bind(name1, "Sambalpur", "odisha"); // bind returns a function  
printMyName() // returned function is invoked i.e. indirect function invocation
```



The screenshot shows a web browser's developer console with the 'Console' tab selected. On the left, the 'index.js' file is open, showing the following code:

```
1 let name1 = {  
2   firstName: "Lycan",  
3   lastName: "Mishra",  
4 };  
5  
6 function getFullName(hometown, state) {  
7   console.log(  
8     this.firstName +  
9     " " +  
10    this.lastName +  
11    " is from " +  
12    hometown +  
13    " from state " +  
14    state  
15  );  
16 }  
17  
18 let printMyName = getFullName.bind(name1, "Sambalpur", "odisha");  
19 console.log(printMyName);  
20 printMyName();  
21
```

On the right, the console shows a message log with 1 message, 1 user message, and 1 info message. The log displays the output of the `getFullName` function:

```
f getFullName(hometown, state) {  
  console.log(  
    this.firstName +  
    " " +  
    this.lastName +  
    " is from " +  
    hometown +  
    " from state " +  
    state  
  );  
}
```

Summary -

	call()	apply()	bind()
Execution	At the time of binding	At the time of binding	At the time when we execute the return function
Parameter	any number of arguments one by one.	Array []	array and any number of arguments
Is Return Function	Yes, it returns and calls the same function at that time of binding.	Yes, it returns and calls the same function at that time of binding.	Yes, it returns a new function or copy of the function. Which we can use whenever we want. It's like a loaded gun.

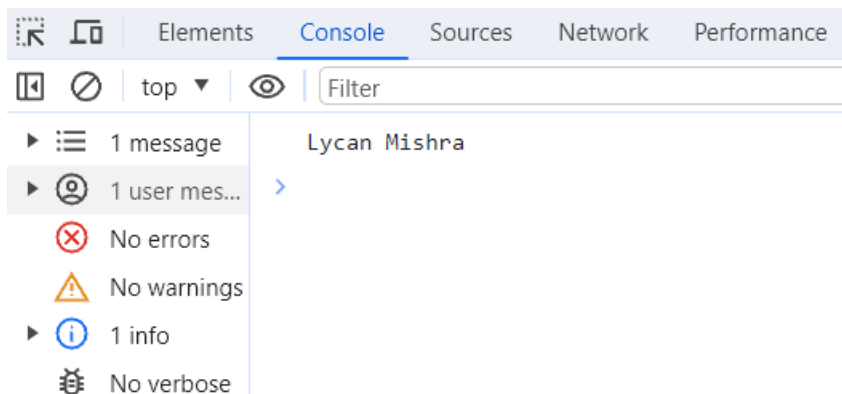
Chapter 2: Polyfill for bind ()

Q: What is a Polyfill?

Ans: Polyfill is a **browser fallback**. What if the browser does not support bind method given by JS. In that case we have to write our own implementation for bind which is known as Polyfill for bind.

JS bind implementation -

```
let name1 = {  
  firstName: "Lycan",  
  lastName: "Mishra",  
};  
  
let printName = function () {  
  console.log(this.firstName + " " + this.lastName);  
};  
  
let printMyName = printName.bind(name1);  
printMyName();
```



Our own bind "**myBind**" implementation -

Case 1: myBind method without taking parameters

```
let name1 = {
  firstName: "Lycan",
  lastName: "Mishra",
};

let printName = function () {
  console.log(this.firstName + " " + this.lastName);
};

Function.prototype.myBind = function (...args) {
  let obj = this; // this points to printName object
  return function () {
    obj.call(args[0]); // args[0] = name1 i.e. received from ...args. Refer previous chapter for how call ()
    // obj.apply(args[0])
  };
};

let printMyName = printName.myBind(name1);
printMyName();
```

Notes:

* Every function has access to bind method. In our own implementation of bind, every function should have access to myBind method. To make this happen we put '**myBind**' method inside `<Function.prototype>`

* As bind method returns a function, myBind should also return a function which is specific to the object on which myBind method is called. In our case the object is printName method

* functions/methods are object in JS, in fact everything is object in JS

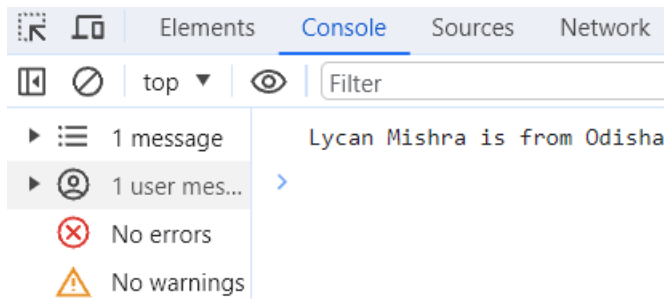
Case 2: myBind method taking parameters

```
let name1 = {  
  firstName: "Lycan",  
  lastName: "Mishra",  
};
```

```
let printName = function (state) {  
  console.log(this.firstName + " " + this.lastName + " is from " + state);  
};
```

```
Function.prototype.myBind = function (...args) {  
  let obj = this;  
  let params = args.slice(1) // params = [args[1], args[2] ... args[lastIndex]]  
  return function () {  
    obj.call(args[0],...params); // args[0] = name2 i.e. received from ...args. Refer previous chapter for how  
    call() works.  
    // obj.apply(args[0],params)  
  };  
};
```

```
let printMyName = printName.myBind(name1,"Odisha");  
printMyName();
```



In below approach we have an issue. `printMyName("India")` statement will show output as below in the console. We get **undefined** because **country** did not receive the value **"India"**

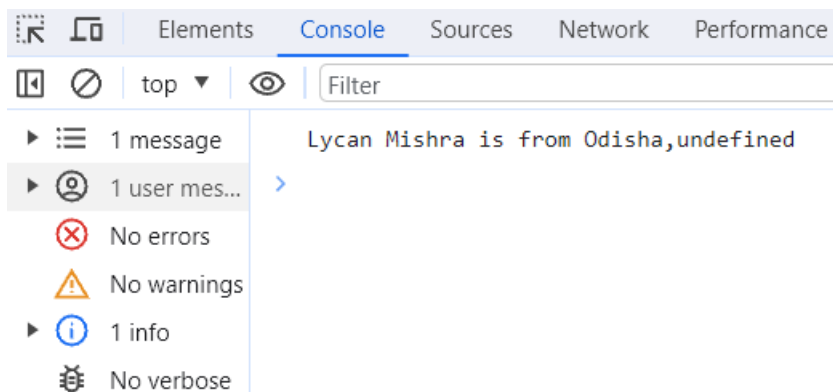
```
let name1 = {
  firstName: "Lycan",
  lastName: "Mishra",
};

let printName = function (state, country) {
  console.log(this.firstName + " " + this.lastName + " is from " + state + ", " + country);
};

Function.prototype.myBind = function (...args) {
  let obj = this;
  let params = args.slice(1)
  return function () {
    obj.call(args[0], ...params);
  };
};

let printMyName = printName.myBind(name1, "Odisha");

printMyName("India");
```



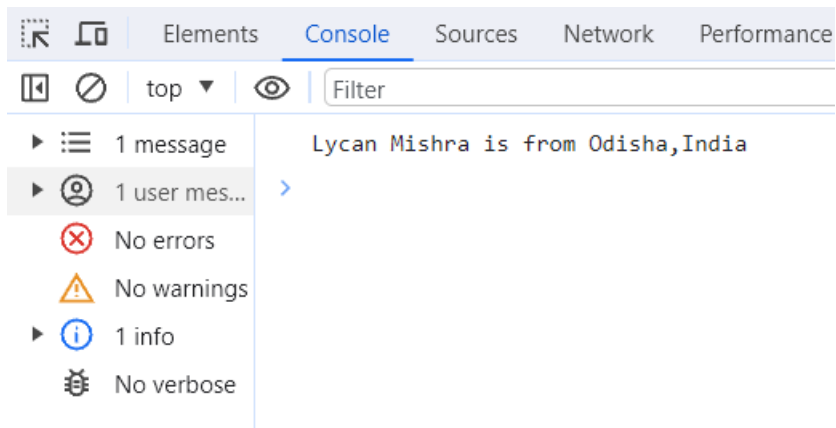
Fixing the issue -

```
let name1 = {
  firstName: "Lycan",
  lastName: "Mishra",
};

let printName = function (state, country) {
  console.log(
    this.firstName + " " + this.lastName + " is from " + state + "," + country
  );
};

Function.prototype.myBind = function (...args) {
  let obj = this;
  let params = args.slice(1);
  return function (...innerFunctionArgs) { //innerFunctionArgs is an array receiving printMyName
function argument values.
    obj.apply(args[0], [...params, ...innerFunctionArgs]); // [array destructuring of params , array
destructuring of innerFunctionArgs]
  };
};

let printMyName = printName.myBind(name1, "Odisha");
printMyName("India");
```



Chapter 3 – Currying in JS

Currying is a JS concept which transforms a function with multiple arguments into a nested series of functions, each taking a single argument.

Advantages of Currying

It helps us creating higher order functions. It is useful in building modular and reusable code.

There are two ways we can curry a function. We can use JS bind to curry a function . Also, we can use closure to curry a function

In below example we are transforming `multiply (2,3)` into `multiply (2)(3)`

Function currying using Closure -

```
let multiply = function (x) {  
  return function (y) {  
    console.log(x * y);  
  };  
};
```

`const Twosmultiple = multiply (2)` // The value of x is 2 and multiply function returns another function which is stored inside Twosmultiple

`Twosmultiple(3)` // The value of y is 3 which is be passed inside the returned function of multiply.

After substituting value of `Twosmultiple` the above two lines of code turns into `multiply(2)(3)` which gives 6 in the console.

Function currying using JS bind method -

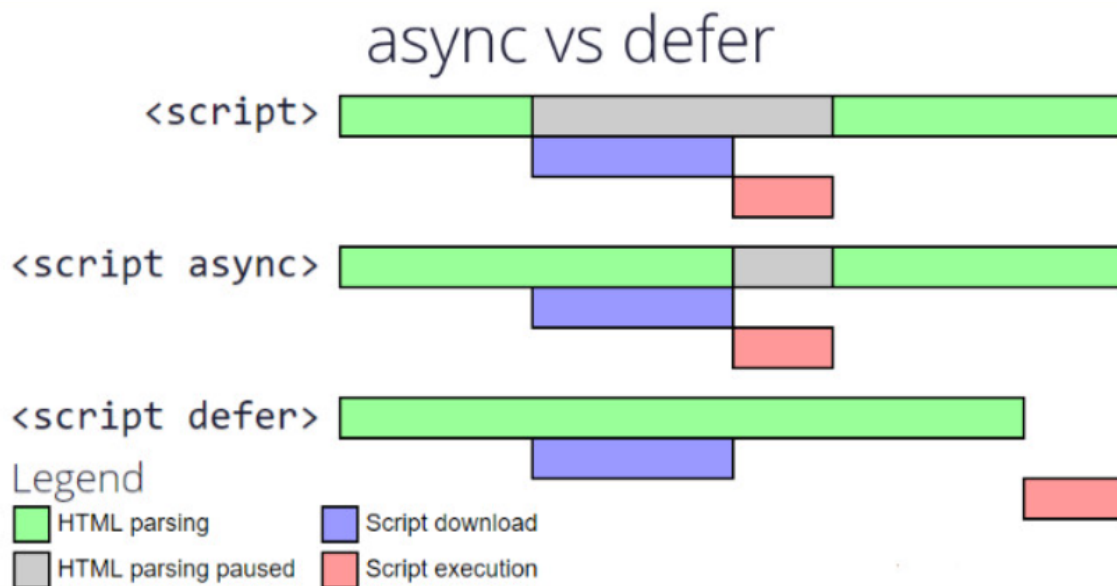
```
let multiply = function (x, y) {  
  console.log(x * y);  
};
```

`let TwosMultiple=multiply.bind(this,2)` // `bind(this,2)` keeps x constant and creates a copy of function `multiply` and assigns the function to `TwosMultiple`.

`TwosMultiple(4)` // value passed inside `TwosMultiple` function is the value of y .This is changeable

This gives 8 as result in the console.

Chapter 4 – Async Vs Defer in JS



`<script src= 'path where script is present'>`

When Html parser encounters `<script>` tag in the html document, parser stops right there until scripts are fetched from the network and get executed. Once script execution is finished Html parser continues.

`<script async src= 'path where script is present'>`

When Html parser encounters `<script async>` tag , it continues until the scripts are fully downloaded from the network. Then the parser stops and script execution started. Once script execution is completed ,Html parser continues parsing forward till end of the document.

`<script defer src= 'path where script is present'>`

When Html parser encounters `<script defer>` tag , it continues until the scripts are downloaded from the network. Then the parser does not stop ,It keeps parsing till end of the document. Once parsing phase is complete, script get executed and completed.

Chapter 5 – Event Bubbling and Event Capturing / Tricking

What is an Event?

An Event is an action that occurs as per the user's instruction as input and gives the output as a response. User's instructions such as clicking on an html element, typing on an html input element trigger specific events such as mouse events and keyboard events whereas output in response indicates what event handler is invoked when the event is triggered.

What is Event Propagation?

When an event occurs in an element inside another element and both elements have registered a handle for that event, the event propagation determines in which order the element receives the event.

What is Event Bubbling?

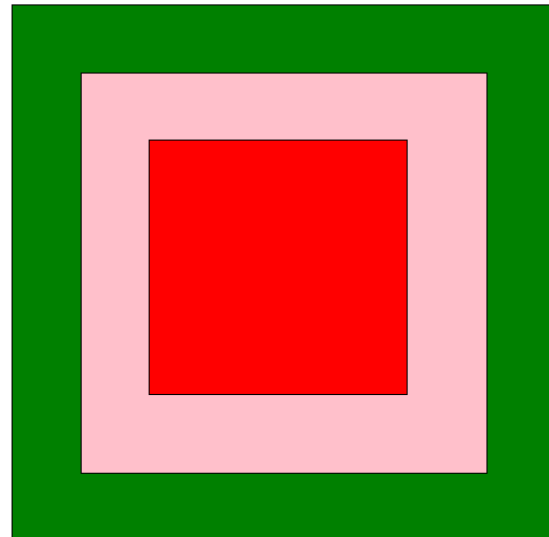
With event bubbling the event is first captured and handled by the innermost element and then propagated to outer elements.

```
index.html x index.css
index.html > html
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8   <link rel="stylesheet" href="./index.css">
9 </head>
10
11 <body>
12
13   <script>
14     const parentID = document.getElementById('parentDivID')
15     const childID = document.getElementById('childDivID')
16     const parentCall = () => {
17       alert("parent div called")
18     }
19     const childCall = () => {
20       alert("child div called")
21       //event.stopPropagation()
22     }
23     parentID.addEventListener('click', parentCall, false)
24     childID.addEventListener('click', childCall, false)
25   </script>
26
27   <div class="parentDiv" id="parentDivID" onclick="parentCall()">
28     <div class="childDiv" id="childDivID" onclick="childCall()"></div>
29   </div>
30
31 </body>
32
33 </html>
```

```

index.html index.css X
index.css > .parentDiv
1  .childDiv {
2      height: 100px;
3      width: 100px;
4      border: 1px solid black;
5      background-color: red;
6      padding: 45px
7  }
8
9  .parentDiv {
10     height: 200px;
11     width: 200px;
12     border: 1px solid black;
13     background-color: pink;
14     padding: 50px
15 }
16
17 .grandParentDiv {
18     height: 300px;
19     width: 300px;
20     border: 1px solid black;
21     background-color: green;
22     padding: 50px
23 }

```



Once we click on **red div** , **child div called** is displayed in first alert box. Then **parent div called** is displayed in second alert box. Meaning we are propagating Mouse click event from bottom to top element of the hierarchical chain . We call this behavior Event bubbling and this is the default behavior of browser.

To stop event bubbling we use `event.stopPropagation()`

```

<script>
    const parentCall = () => alert("parent div called")
    const childCall = () => {
        alert("child div called")
        event.stopPropagation()
    }
</script>

```

Now when we click on **red div** , **child div called** is displayed in first alert box. We wont get **parent div called** in second alert box once we click **ok** on first alert box.

When we click on **pink div** , **parent div called** is displayed in first alert box. Thats it.

What is Event Capturing?

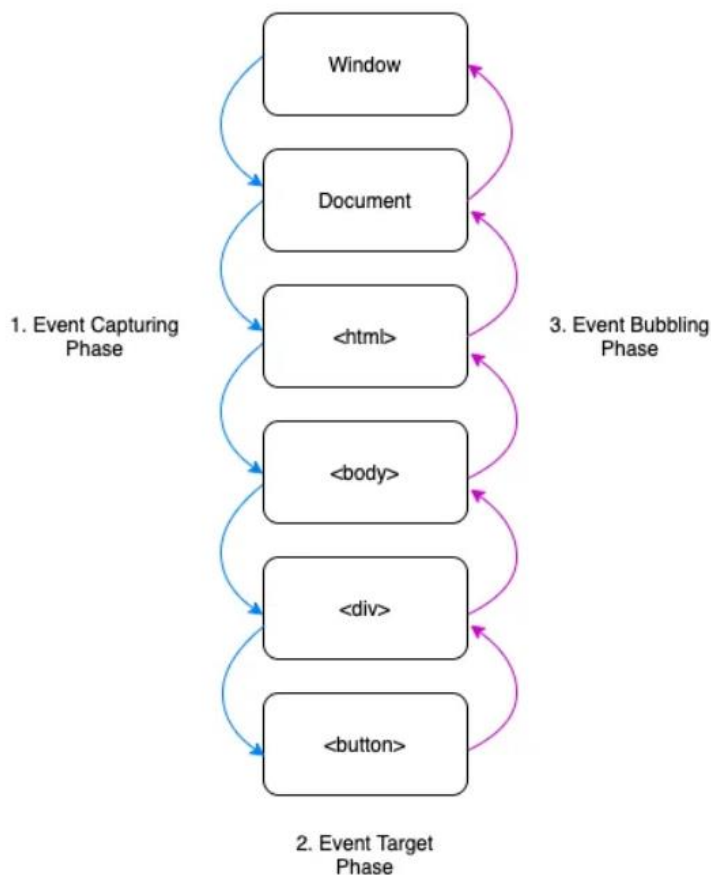
With event capturing the event is first captured by outermost element and propagated to inner elements. Capturing is also called as trickling.

```
,  
parentID.addEventListener('click', parentCall, true)  
childID.addEventListener('click', childCall, true)
```

With these two checks on when we click on child div. **parent div called** is displayed in the alert box first followed by **child div is called** in another alert.

Note – Default value for third argument to `addEventListener ()` is false. Check browser compatibility to use these event listeners

Summary-



Event target phase – element where event is triggered.

Chapter 6 – Explain sum (1)(2)(3)(4)?

sum (1)(2)(3)(4) – sum (1) returns a function which takes a parameter value as 2 which returns another function that takes a parameter value as 3, which again returns a function that takes a parameter value as 4 and finally the innermost function returns the sum as (1+2+3+4). These are higher order functions.

Code implementation

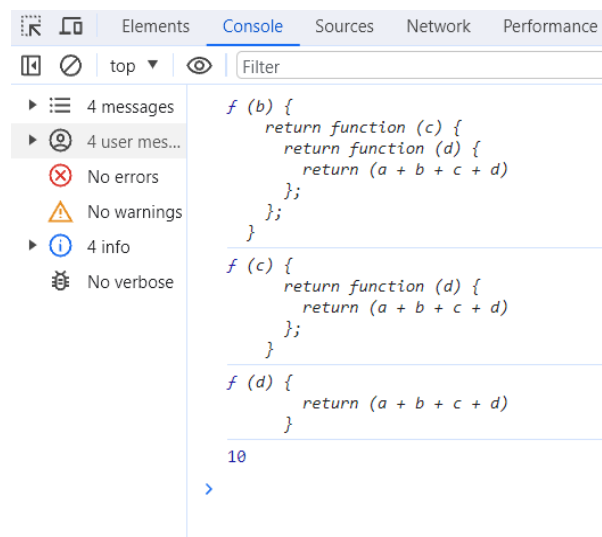
```
function Sum(a) {           // a = 1
  return function (b) {     // b = 2
    return function (c) {   // c = 3
      return function (d) { // d = 4
        return (a + b + c + d)
      };
    };
  };
}
```

```
const sumUptoOne=Sum(1)
console.log(sumUptoOne)
```

```
const sumUptoTwo=sumUptoOne(2)
console.log(sumUptoTwo)
```

```
const sumUptoThree=sumUptoTwo(3)
console.log(sumUptoThree)
```

```
const sumUptoFour=sumUptoThree(4)
console.log(sumUptoFour) // 10
```



sumUptoThree is the innermost function which has access to its lexical env variables a, b, c, d.

Substitution method -

```
const sumUptoOne=Sum(1)
```

```
const sumUptoTwo=sumUptoOne(2) // sumUptoTwo = Sum(1)(2)
```

```
const sumUptoThree=sumUptoTwo(3) // sumUptoThree = Sum(1)(2)(3)
```

```
const sumUptoFour=sumUptoThree(4) // sumUptoFour = Sum(1)(2)(3)(4)
```

```
console.log(sumUptoFour) // or console.log(Sum(1)(2)(3)(4)) which gives 10 in the console.
```


Chapter 7 – Prototypes and Prototype Inheritance

Let's initialise an array and an object.

```
JS index.js X
JS index.js > ...
1 let arr = [1, 2, 3, 4];
2
3 let obj = {
4   name: "Lycan",
5   city: "Sambalpur",
6   getInfo: function () {
7     console.log(this.name + " is from " + this.city);
8   },
9 };
10
```

Now if we click on **arr.** or **obj.** in the console, we got access to several properties and methods, other than the defined ones.

Live reload enabled.

```
> obj.
< 'Sambalpur' city tab
getInfo
name
__defineGetter__
__defineSetter__
__lookupGetter__
__lookupSetter__
__proto__
constructor
hasOwnProperty
isPrototypeOf
propertyIsEnumerable
toLocaleString
toString
valueOf
```

Live reload enabled.

```
> arr.
< 4 length tab
__defineGetter__
__defineSetter__
__lookupGetter__
__lookupSetter__
__proto__
at
concat
constructor
copyWithin
entries
every
fill
filter
find
findIndex
findLast
```

Where do these additional properties and methods come from?

When you create an object in JavaScript, JS Engine put some hidden properties and functions into a special object called **prototype object** and then attaches this object to the original object. This is why we get access to these properties and methods when we type `obj.` in the console.

This is not only for objects but also for functions arrays and other types too. To get access to the prototype object we use `<created object reference>.__proto__`

```
> obj.__proto__
< ▶ {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...}
> arr.__proto__
< ▶ [constructor: f, at: f, concat: f, copyWithin: f, fill: f, ...]
>
```

Why everything in JS is an object?

If you get access to the prototype of each type, down to the prototype chain we will end up getting an object.

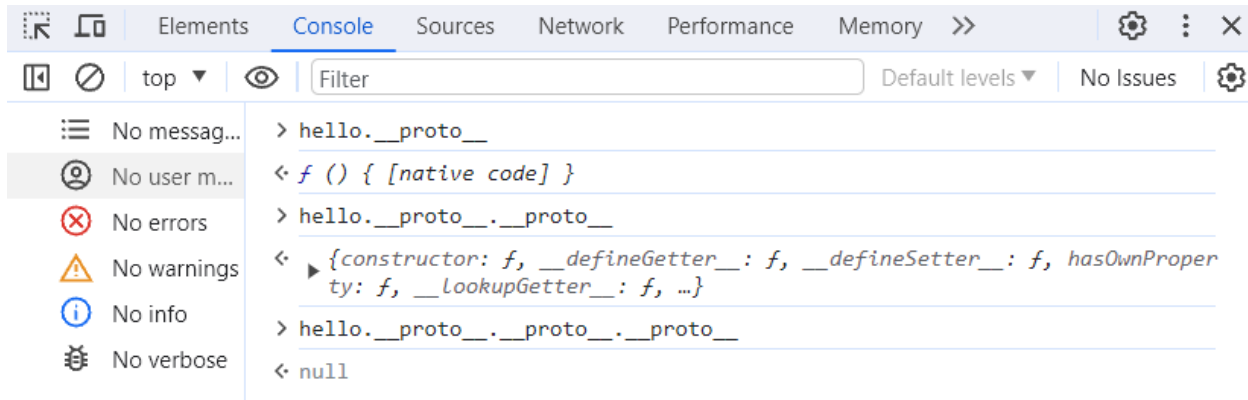
Array prototype chain -

```
> arr.__proto__
< ▶ [constructor: f, at: f, concat: f, copyWithin: f, fill: f, ...]
> arr.__proto__.__proto__
< ▶ {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...}
> arr.__proto__.__proto__.__proto__
< null
>
```

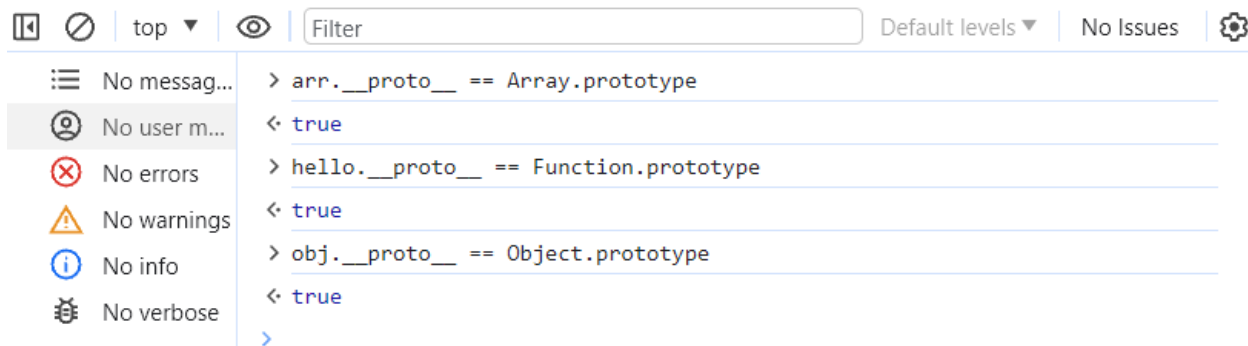
Object prototype chain -

```
> obj.__proto__
< ▶ {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...}
> obj.__proto__.__proto__
< null
>
```

Function prototype chain –



Below statements have same meanings

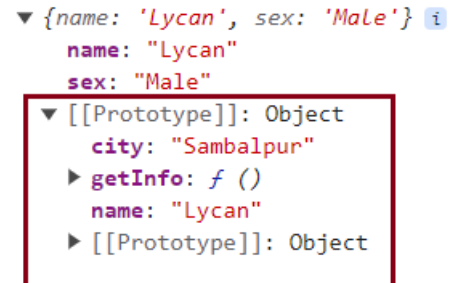


Function Prototype in Action



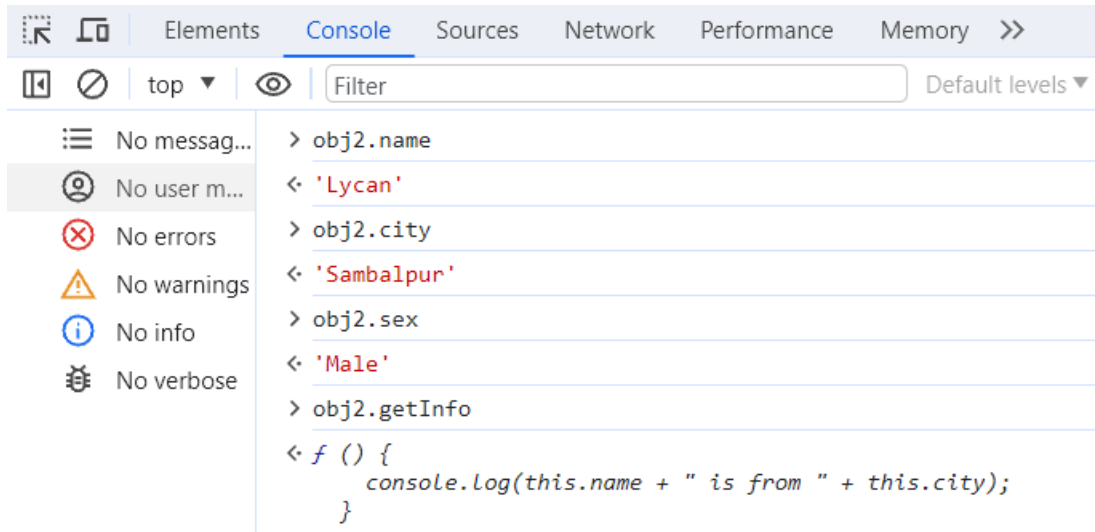
Prototype Inheritance-

Prototype Inheritance is inheriting properties and methods of one object into another object



```
JS index.js  X
JS index.js > ...
1  let arr = [1, 2, 3, 4];
2
3  let obj = {
4    name: "Lycan",
5    city: "Sambalpur",
6    getInfo: function () {
7      console.log(this.name + " is from " + this.city);
8    },
9  };
10
11 let obj2 = {
12   name: "Lycan",
13   sex: "Male",
14 };
15
16
17 obj2.__proto__ = obj;
18
19 console.log(obj2)
20
```

Now obj2 has access to obj properties. This is known as prototype inheritance.



Upcoming concepts

- * Throttling Vs debouncing
- * Event Delegation
- * CORS (Cross Origin Resource Sharing)
- * Thinking recursively