Call Apply & Bind :-

At its core, call, Apply & Bind solve the fundamental JS quirk of "this" - namely, how to control what this refers to when a function Runs.

In JS all function are objects, This means they can have their own properties and methods.

These techniques helps you to manage the Execution context of function.

The problem :- Losing the Right "this" context.

① methods detached from their object :-

```
const user = {
    name = "Sahil"
    greet () {
        console.log(`I'am ${this.name}`)
    }
};
```

we took the function Reference out of 'user'

```
const greetFn = user.greet
greetFn()  //-> I'am undefined.
```

- Here when you call greetFn(), "this" is no longer bound to "user". It defaults (in non-strict mode) to the global object or becomes "undefined"
- So basically you lost the link between the method and its original object

Fix

greet.call(user)

↓

Attaching "this" or the context of object

② Callback functions and Event Handlers :-

A callback function is a function which you pass as an argument to another function, to be "called back" late

Ex :-

```
function greet(name){
    console.log("Hello" + name);
}
setTimeout(greet, 1000, "world")
// After 1 second, "Hello world" is printed.
```

Event Handler :- A specific kind of callback attached to DOM Element (e.g. button click). when event fires the handler is invoked.

Ex :-

```
const btn = document.querySelector("button")
btn.addEventListener("click", function(e){
    console.log("Button clicked");
});
```

This anonymous function is Event Handler.

Ex
```
const calculator = {
    number: 10,
    multiply(x) {
        console.log(this.number * x);
    }
};
setTimeout(calculator.multiply, 1000, 5);
// After 1s: prints NaN
```

- In JS, the value of "this" inside a function depends on how that function is called. It is not tied just to where the function is defined, but rather to who calls it or how it is called.

Or when we do
```
myObj.doSomething();
```
The JS engine binds "this" inside doSomething to myObj. But if you ~~call~~ Extract the method and call it "naked", then "this" typically falls and fall back to the global or undefined.

- In above Example. "SetTimeout" Eventually Invokes the callback, it Essentially does Something like.

```
// After 1s
calculator.multiply(5)
// Actually NO - This is not how JS Handle it
```

Instead it just does
multiply(5)
// call the function "naked" without a base object

Fix :-

① setTimeout(calculator.multiply.bind(calculator), 1000, 5);

This classic traditional way is to create a new function where "this" is permanently bound to specific object

② setTimeout(() => calculator.multiply(5), 1000);

Arrow functions don't have their own "this" they capture "this" from the outer lexical scope. If you wrap in a simple arrow, you can directly call the method on object

③ Real Life Event Handler Problem :-

HTML -> <button id="btn">Click me</button>

TS ->

```
const myobj = {
    count = 0,
    handleClick(event) {
        this.count++;
        console.log(this.count);
    }
}

const btn = document.getElementById("btn");
btn.addEventListener("click", myobj.handleClick);
```

- when the button is clicked, browsers can call your handler with "this" set to HTMLElement i.e <button id="btn">. So Inside handleClick, this.count is trying to access buttonElement.count which is undefinded.

Fix

```
btn.addEventListener("click", myobj.
    handleClick.bind(myobj));
```

(4) Problem - Borrowing from one object to use on another.

```
const alice = { name: "Alice" };
const bob = { name: "Bob" };

function sayName() {
    console.log(this.name);
}

alice.speak = sayName;
alice.speak();
bob.speak = sayName;
bob.speak();
```

"Or"

```
const alice = {
    name: "Alice";
};
const bob = { name: "Bob" };

sayname.call(alice)
sayname.apply(bob)
```

(5) Partial Application :-

with Bind(), you can create a partially applied function, where some arguments are preset

```
function multiply (a, b) {
        return a * b;
}
const double = multiply.bind (2);
console.log (double (5));
//o/p : 10
```

① Call

The call method calls a function with a given "this" value and arguement provided Individually by comma seperated value.

fn.call (this, arg1, arg2 ...)
(> optional

Immediately Invoked

② Apply

The Apply method calls a function with a given "this", the arguments provided as an an array or an array-like object

fn.apply (this, [arg1, arg2, args...])
(> optional

Immediately Invoked

③ Bind : The Bind method creates a new function that when called has its "this" keyword set to it

Additional you can prepend arg to a function for Partial Application (8)

```
const boundGreet = user.greet.bind(user);
```

Modern Solutions Instead of call, Apply, Bind

1. Arrow function and lexical this
2. React -> use of useState, useCallback and function Components
   - use callback for memoized callbacks
3. class fields with Arrow methods

```
increment () {
  this.setState(({count}) => ({count: count+1}))
}
```

4. spread syntax instead of "apply"

~~Polyfills~~

1. ~~call~~

```
Function.prototype.myCall = function(context, ...args) {
  context = context || (typeof window !== 'undefined' ? window : global);
```

Polyfills

① call

```
Function.Prototype.mycall = function (context, ...args){
  ① context = context || (typeof window !== "undefined" ? window || global);
  ② const uniqueKey = Symbol('uniqueKey')
  ③ context[uniqueKey] = this;
  ④ const result = context[uniqueKey](...args)
  ⑤ delete context[uniqueKey];
  ⑥ return result;
};
```

① If no context is provided, default to window in browser and global in node.

② Create a unique Symbol key to avoid property name collisions

```
Symbol('uniqueKey') === Symbol('uniqueKey')
// o/p false
```

③ In this line "this" will be the function definition. So we will attach function definition / method as a property in the context object

④ call function with provided context and arg

⑤ delete temp property

⑥ return Result

② apply

Same as call :-

Just one line addition to check if arg is array otherwise assign empty array

```
argsArray = Array.isArray(...args) ? ...args : [];
const result = context[uniqueKey](...argsArray);
```

③ Bind

```
Function.prototype.myBind = function(context, ...initialArgs) {
    const originalFunction = this;
    return function(...laterArgs) {
        const uniqueProp = Symbol('uniqueKey');
        context = context || typeof window !== "undefined" ?
                  window : global;
        context[uniqueProp] = originalFunction;
        const result = context[uniqueProp](...initialArgs, ...laterArgs)
        delete context[uniqueProp]
        return result;
    };
};
```