1.**Apply and call** - Apply takes an array of arguments, while Call takes zero or more individual parameters

**call** attaches this into function and executes the function immediately:

**bind** attaches this into function and it needs to be invoked separately like this: - call back function

**A**pply uses **A**rrays and **A**lways takes one or two Arguments. When you use **C**all you have to **C**ount the number of arguments.

**Call is slightly faster in performance than Apply**.

Bind is a bit different. It returns a **new function**. Call and Apply execute the current function immediately.

Use .bind() when you want that function **to later be called with a certain context**, useful in **events**. Use **.call() or .apply() when you want to invoke the function immediately**, and modify the context.

Call/apply call the function immediately, whereas bind returns a function that, when later executed, will have the correct context set for calling the original function. This way you can maintain context in async callbacks and events.

Unlike Call and Apply, **Bind doesn't itself call the function**, it can only be used to bind the value of context and other arguments before calling the function.

**.apply(this, [...])**

**.call(this, param1, param2, param3, param4...)**

var teamIndia={

name:'Dhoni',

message:function(name){

alert(this.name+' we lost the worldcup 2015');

}

}

var teamPakistan={

name:'Misabh'

}

teamIndia.message.call(teamPakistan,'Misabh')

**Undefined value:** A value that is not defined and has no keyword is known as undefined value. For example:

1. int number;//Here, number has undefined value.

There are just six **types** in **JavaScript**: Object, Number, String, Boolean, Null, and Undefined

**2. Bubbling and capturing -**The order is called a bubbling order, because an event bubbles from the innermost element up through parents, like a bubble of air in the water.

According to this model, the event:

1. Captures down - through 1 -> 2 -> 3.
2. Bubbles up - through 3 -> 2 -> 1.

All methods of event handling ignore the caputiring phase. **Using addEventListener with last argument true is only the way to catch the event at capturing**.

|  |
| --- |
| elem.addEventListener( type, handler, phase ) |

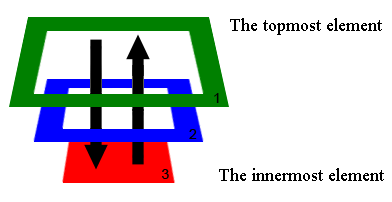
phase = true

The handler is set on the capturing phase.

phase = false

The handler is set on the bubbling phase.

Click in a div below to see capturing in action (no IE<9):

 **capturing**

**diff between null & undefined - undefined** (means a variable has been declared **but has not yet been assigned** a value) **null (** is an assignment value. It can be assigned to a variable as a representation of no value) **event handling in JS**Event bubbling(trigger inner most element first and then it triggers on the parent elements in nested order.) Event Capturing is the opposite of bubbling where the event is triggered on the outer most element and then it triggers the inner children in nested order. **diff prevent default -** method will prevent the link above from following the URL. **stop probhigation** will prevent the event from bubbling up and return false will do both.

**3. Closure** - inner functions referring to local variables of its outer function create closures. In other words, the inner function has access to the outer function’s variables and parameters.

function makeFunc() {

var name = 'abd';

function displayName() {

alert(name);

}

return displayName;

}

var myFunc = makeFunc();

myFunc();

4. **AJAX** **- var xhttp = new XMLHttpRequest();**

xhttp.open("GET", "ajax\_info.txt", true); open(*method, url, async(true)*)  
xhttp.send();

xhttp.onreadystatechange = function() {

**xhttp.readyState** - Holds the status of the XMLHttpRequest. Changes from 0 to 4:   
**0**: request not initialized , **1**: server connection established, **2**: request received , 3: processing request   
**4**: request finished and response is ready

**status -** 200: "OK" , 404: Page not found

$.ajax({type: "GET", mode: "queue",url: xmlFile, dataType: "xml",beforeSend: function () {},

success: function (xml) {});

5. Type of pattern( [The Module Pattern](http://carldanley.com/js-module-pattern/) - [The Singleton Pattern](http://carldanley.com/js-singleton-pattern/) -[The Observer Pattern](http://carldanley.com/js-observer-pattern/) - [The Prototype Pattern](http://carldanley.com/js-prototype-pattern/) -[The Facade Pattern](http://carldanley.com/js-facade-pattern/) -[The Factory Pattern](http://carldanley.com/js-factory-pattern/)

**What is curring:** **Curring is partial invocation of a function**.

Currying means first few arguments of a function is pre-processed and a function is returned. The returning function can add more arguments to the curried function. It's like if you have given one or two spice to the curry and cooked little bit, still you can add further spice to it. A simple example will look like-

function addBase(base){ var addTen = addBase(10);

return function(num){ addTen(5); //15 addTen(80); //90 addTen(-5); //5

return base + num;

}

}

**Scope** - is the accessibility of variables, functions, and objects in some particular part of your code during runtime – global and local scope

**Hoisting -** **Hoisting** is the mechanism of moving the variables and functions declaration to the top of the function scope

**Function Declaration Overrides Variable Declaration When Hoisted**

// **Both the variable and the function are named myName**

var myName;

function myName () {

console.log ("Rich");

}

// The function declaration overrides the variable name

console.log(typeof myName); // function

// But in this example, the variable assignment overrides the function declaration

var myName = "Richard"; // This is the variable assignment (initialization) **that overrides the function declaration.**

function myName () {

console.log ("Rich");

}

console.log(typeof myName); // string

**What is a Function Expression?**

Functions defined via Functions Expressions can be named or anonymous.

/anonymous function expression

var a = function() {

    return 3;

}

//named function expression

var a = function bar() {

    return 3;

}

 console.log(a());

//self invoking function expression

(function sayHello() {

    alert("hello!");

})();

### **Convert string to javascript array**

May 30, 2009 06:38 PM|[LINK](https://forums.asp.net/post/3199072.aspx)

An easier way than splitting or iterating (IMO):

<input type="hidden" id="hiddenArray" value="1,2,3,4" />

<script type="text/javascript">

var values = document.getElementById('hiddenArray').value;

var arr = eval('[' + values + ']');

</script>