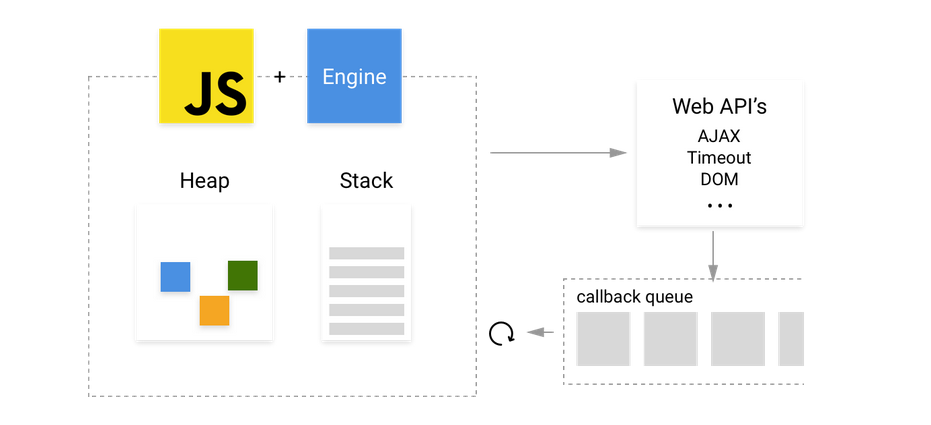
Introduction to JS

How JS works under the hood.



**Stack**

Function calls form a stack of frames.

Function foo (b) {

var a = 10;

return a + b + 11;

}

function bar(x) {

var y = 3;

return foo(x \* y);

}

console.log (bar (7)); //returns 42

When calling bar, a first frame is created containing bar's arguments and local variables. When bar calls foo, a second frame is created and pushed on top of the first one containing foo's arguments and local variables. When foo returns, the top frame element is popped out of the stack (leaving only bar's call frame). When bar returns, the stack is empty.

**Heap**

Objects are allocated in a heap which is just a name to denote a large mostly unstructured region of memory.

**Queue**

A JavaScript runtime uses a message queue, which is a list of messages to be processed. Each message has an associated function which gets called in order to handle the message.

At some point during the event loop, the runtime starts handling the messages on the queue, starting with the oldest one. To do so, the message is removed from the queue and its corresponding function is called with the message as an input parameter. As always, calling a function creates a new stack frame for that function's use.

The processing of functions continues until the stack is once again empty; then the event loop will process the next message in the queue (if there is one).

Example :

(function () {

console.log ('this is the start');

setTimeout (function cb() {

console.log ('this is a msg from call back');

});

console.log ('this is just a message');

setTimeout (function cb1 () {

console.log ('this is a msg from call back1');

}, 0);

console.log ('this is the end');

})();

// "this is the start"

// "this is just a message"

// "this is the end"

// note that function return, which is undefined, happens here

// "this is a msg from call back"

// "this is a msg from call back1"

Hoisting in JS

Hoisting is JavaScript's default behavior of moving declarations to the top.

**JavaScript Declarations are hoisted:**

In JavaScript, a variable can be declared after it has been used.

In other words; a variable can be used before it has been declared.

**Example 1** gives the same result as **Example 2**:

Example 1:

x = 5; // Assign 5 to x  
  
elem = document.getElementById ("demo"); // Find an element   
elem.innerHTML = x;                     // Display x in the element  
  
var x; // Declare x

Example 2:

x = 5; // Assign 5 to x  
  
elem = document.getElementById ("demo"); // Find an element   
elem.innerHTML = x;                 // Display x in the element  
  
var x; // Declare x

## JavaScript Initializations are Not Hoisted

JavaScript only hoists declarations, not initializations.

**Example 1** does **not** give the same result as **Example 2**:

Example 1:

var x = 5; // Initialize x  
var y = 7; // Initialize y  
  
elem = document.getElementById ("demo"); // Find an element   
elem.innerHTML = x + " " + y;            // Display x and y

Example 2:

var x = 5; // Initialize x  
  
elem = document.getElementById ("demo"); // Find an element   
elem.innerHTML = x + " " + y;            // Display x and y  
  
var y = 7; // Initialize y

Does it make sense that y is undefined in the last example?

This is because only the declaration (var y), not the initialization (=7) is hoisted to the top.

Because of hoisting, y has been declared before it is used, but because initializations are not hoisted, the value of y is undefined.

Example 2 is the same as writing:

var x = 5; // Initialize x  
var y;     // Declare y  
  
elem = document.getElementById("demo"); // Find an element   
elem.innerHTML = x + " " + y;            // Display x and y  
  
y = 7;    // Assign 7 to y

Call (), Apply () and Bind ().

You can use call ()/apply () to invoke the function immediately. Bind () returns a bound function that, when executed later, will have the correct context ("this") for calling the original function. So bind () can be used when the function needs to be called later in certain events when it's useful.

**.call ()**

//Demo with JavaScript .call ()

var obj = {name:"Niladri"};

var greeting = function (a, b, c) {

return "welcome "+this.name+" to "+a+" "+b+" in "+c;

};

console.log (greeting.call (obj,"Newtown","KOLKATA","WB"));

// returns output as welcome Niladri to Newtown KOLKATA in WB

The first parameter in call() method sets the **"this"** value, which is the object, on which the function is invoked upon. In this case, it's the **"obj"** object above.

The rest of the parameters are the arguments to the actual function.

**.apply ()**

//Demo with JavaScript .apply()

var obj = {name:"Niladri"};

var greeting = function(a,b,c){

return "welcome "+this.name+" to "+a+" "+b+" in "+c;

};

// array of arguments to the actual function

var args = ["Newtown","KOLKATA","WB"];

console.log ("Output using .apply () below ")

console.log (greeting.apply (obj, args));

/\* The output will be

Output using .apply () below

welcome Niladri to Newtown KOLKATA in WB \*/

Similarly to call () method the first parameter in apply () method sets the **"this"** value which is the object upon which the function is invoked. In this case it's the **"obj"** object above. The only difference of apply () with the call () method is that the second parameter of the apply () method accepts the arguments to the actual function as an array.

**.bind ()**

//Use .bind() javascript

var obj = {name:"Niladri"};

var greeting = function(a,b,c){

return "welcome "+this.name+" to "+a+" "+b+" in "+c;

};

//creates a bound function that has same body and parameters

var bound = greeting.bind(obj);

console.dir(bound); ///returns a function

console.log("Output using .bind() below ");

console.log(bound("Newtown","KOLKATA","WB")); //call the bound function

/\* the output will be

Output using .bind() below

welcome Niladri to Newtown KOLKATA in WB \*/

In the above code sample for bind() we are returning a bound function with the context which will be invoked later