JavaScript under the hood

* ‘*undefined’* is something that happens when you declare or assign a variable but do not assign any value to it but still try to use that variable or access that variable. At this point of time, the value of that variable is undefined.

For example,

Var a;

Console.log(a); // at this point it logs as ‘undefined’

*By default, as soon as a variable is declared, that’s undefined unless or until we assign some value to it.*

* **Primitive Data types:**
* Undefined
* NULL (manually saying the Js engine that the variable is empty and don’t freak out by saying ‘undefined’)
* Boolean
* Number
* String
* Symbol (ES6)
* Remember the operator precedence table from Js MDN <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence>
* Wanna see something which is much more stricter than ‘===’ See this <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness>
* Remember,
* “ “ is 0, false
* Null is 0, false
* Undefined is 0, false
* **Convert Object to JSON:**

JSON.stringify(objectName)

* **Convert JSON to object:**

JSON.parse(‘{“key”: “value”, “key”: “value”}’);

* A way to pass a function as a parameter in a function:

function hello(param) {

param();

}

hello(function() {

console.log(“Hey there!”);

}

* In an object literal, if the *value* to a *key* is a primitive data type, then it is called as an **object property**. If the *value* to a *key* is a function, then it is called as **object method**.
* In JavaScript, everything is an object. For example, you can pass any kind of data type into the same array and the JS engine will take care of the data type. So, you can pass all the primitive data types as well as objects into the same array.

Var box = [

1,

True,

{

Name: “Nikhil”,

House: “Yellow”

},

function(name) {

var greeting =”Hello”;

console.log (greeting + name);

},

“and this is a string type”

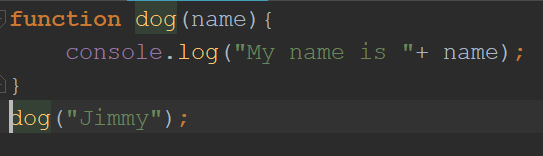
];

Console.log(box); // *displays the whole array structure in the console.*

box[3](box[2].Name); //*this calls the function inside the array with name from object literal in the array.*

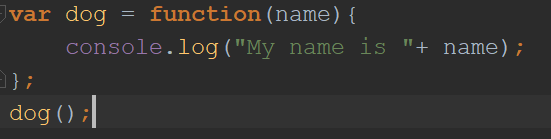
* **Try to Learn More about “Arguments and spreads”.**
* See Lecture 40 to know how to take an alternative to Function overloading in JavaScript.
* Types of **function calls**:

1. Function statement:



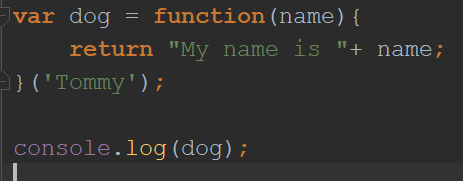
This is normal function statement where the statements in function execute when the function name is called anywhere in the program.

1. Function Expression:



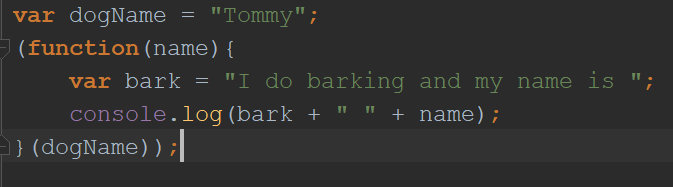
Here, the function is in the form of an expression and stored into a variable called dog. And from there, the variable name can be used to call the function.

1. Immediately invoked function expressions(**IIFE**):



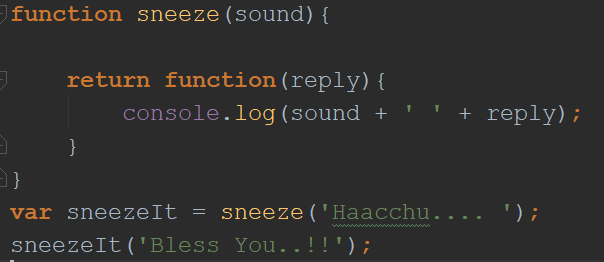
Here the function is called as soon as it is initiated. Once the function is called, it returns the value as a string and stored in the variable dog. Then it can be displayed using the console or whatever you want.

1. Immediately invoked function expression 2 (IIFE):



Here, the function is basically in parenthesis () and executes as soon as JS engine reads it without even any outer statement or expression involving to call it.

* **CLOSURES**:



Here, the output at Console.log is, “Haacchuu… Bless You..!!”

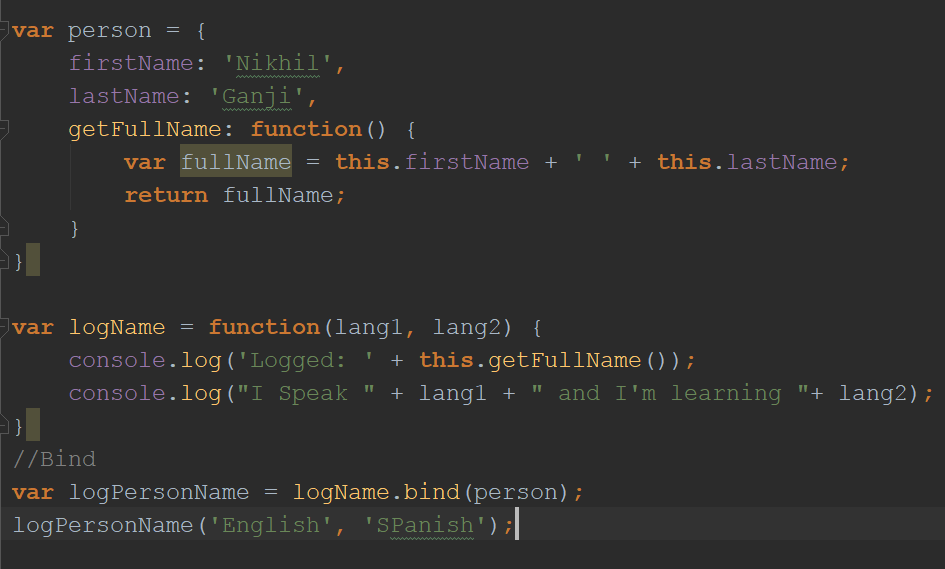
* Closure is basically an inner function that has the access to variables of parent function even after *returning or popping out the parent function from execution context.*
* *Remember the Amitab Bacchan Story that you narrated.*

*Abhishek can listen to the words of Amitab only if the Amitab is alive. But even after Amitab Bacchan passing away, Abhishek can still listen to the words of Amitab’s soul. That’s it. Here this Phenomenon is called* ***closure****.*

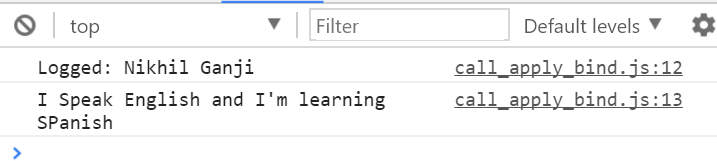
*Abhishek (inner most function) is now a closure who has the access to the words(variables) of Amitab(outer function) who already passed away(popped out from execution stack).*

* **Call, Apply and Bind**

1. **Bind:**



**Output:**



* *Bind* method creates a copy of function (*logName*) which is being called by using DOT(.) operator and whatever object (*person*) is being passed to the bind method is used as a reference by this function (*logName*) having a ‘this’ (*this.getFullName*) keyword in it.
* *Bind* can also permanently change the parameters of a function

function multiply(a,b) {

return a\*b;

}

Var multiplyByTwo = multiply.bind(this, 2);

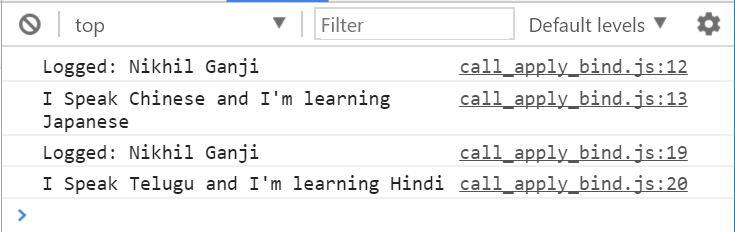
Console.log(multiplyByTwo(3)); //logs 6

In the example above, *bind* has changed the value of parameter ‘a’ to 2.

1. **Call:**



**Output:**



* *Call* is like *bind* in terms of referring the function being called and an object being passed for ‘this’ reference. But unlike *bind*, call do not make a copy of function and store it. *Call* executes the statement right away and calls the function.
* *Apply* is very similar to *call* except that the parameters or arguments in *apply* method are passed in an array like this,

.apply(objectName, [‘param1’ , ‘param2’])

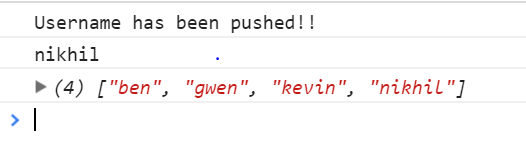
* **Function Currying:**

Creating a copy of function with some preset parameters.

* **Callbacks**



**Output:**



* Callback is a function that must be executed after execution of through which it is being passed as a parameter.
* Say for example *callback* is *function B()* and the other one is *function A().* *B*  is now passed as a parameter in *function A().* And now *function B()* will execute only after the execution of *function A().*  This concept is called as *callback*.
* In the above example, we have given a timeout of 5 secs to *addUser* function and 1 sec to *getUsers* function. Obviously *getUsers* will execute first but there is no meaning of calling *getUsers* without even adding users. So, we have passed *getUsers* as a callback into the *addUser* call and then inside the *addUser* function we have called *callback();* right after finishing the statements in *addUser* function.