* Semicolon are optional in JS. But it is always recommended to put semicolon in last. Otherwise JS may guess wrong about the end of the statement.
* Semicolon terminates statements not blocks {}. But there is a case where we have to put semicolon in block i.e. A function expression is an expression that ends with a block. If such an expression comes last in a statement (file ended with specific function), it is followed by a semicolon:
* **Identifier:** The first character can be any Unicode letter, or $ sign or \_ sign. Subsequent characters could be any as mentioned and Unicode digit additionally.
* **Values or Datatypes:** In JS we call them values, there are many values present in JS like Booleans, strings, array etc. Every value contains few properties like

|  |  |  |
| --- | --- | --- |
|  | **Primitive** | **Object** |
| **Types** | Boolean, String, Number, undefined, null | Plain Object, Arrays, Regular Expression |
| **Mutability** | You cannot change, add, and remove properties  **Immutable:** *var myname = ‘abc’*  *myname.foo = "aryan"; //undefined*  *myname.length = 1; //no change* | You can freely change, add, and remove properties  **Mutable:** *var myname = {}*  *myname.foo = "aryan";* |
| **Properties** | *var str =’abc’;*  *str.length;*  *str.toUpperCase*  *’abc’.LowerCase* | *var object = {};*  *object.foo = ‘kuchBhi’*  *object.boo = 12345,*  *object.coo = ‘1234.56* |
| **Equality** | *var obj1 = 123;*  *var obj2 = 123;*  *obj1 === obj2; output: true* | Each object has a unique identity and is only (strictly) equal to itself.  *var obj1 = {};*  *var obj2 = {};*  *obj1 === obj2; output: false*  *obj2 =obj1;*  *obj1 === obj2; output: true* |

* **undefined keyword:** it means “no value”.
  1. uninitialized variables are undefined *// var foo; alert(foo); output: undefined*
  2. missing parameters are undefined *// function f(x) {return x}; f (); output: undefined*
  3. if u read non-existence property // *var obj = {}; alert(obj); output: undefined*
* **null keyword:** it means ‘no object”………
* **undefined and null** have no properties, not even standard methods such as toString().
* **typeof :** is mainly used for primitive values to check it type
  1. *typeof ‘true’* **:** Boolean
  2. *typeof ‘abc’* : string
  3. *typeof 123* : number
  4. *typeof {} OR []* : object
  5. ***typeof null returning 'object' is a bug that can’t be fixed, because it would break existing code. It does not mean that null is an object.***
* **instanceof:** it returns true if object has been created from same class
  1. *var bar = new Bar (); bar instanceof Bar; true*
* **Truthy And Falsy:** For any conditional statement like *if,* any of below value will be use to interpret True or False (all below interpreted as False. All other values including Object treated as TRUE
  1. ***undefined, null //Boolean(undefine) ---- false***
  2. *Boolean****: false //Boolean(0)-------------- false***
  3. *Number****:0, NaN // Boolean(NaN)----------false***
  4. *String****: ‘‘ // Boolean(‘’)--------------false***
  5. *Boolean ({})*------------true
* **Functions:**
  1. ***Function Declaration:***

*function add (param1, param2){*

*return param1 + param2;*

*}*

* 1. ***Function Expression:*** *this expression is useful when function expression produces a value which get used directly as outer function’s argument i.e. outerFunction(function (p1,p2) {---});*

*var add = function (param1, param2) {*

*return param1 + param2;*

*}*

* **Hoisted Functions:** It means, we can call any function before defining it. internally JS will move function definition internally in beginning of the current scope.
* ***arguments*: *Passing arbitrary amount of argument in function parameter-*** We can call any function in JS with an any no. of args. it will make available for you with special variable ***arguments.***
* **Convert arguments into array :** arguments is not an array, we just retrive elements like array index. But sometime we need to change few elements based on business requirement. so, we convert arguments into array.

*function convertToarray(){*

*var args ={};*

*args = Array.prototype.slice.call(arguments);*

*--- }*

* **Strict Mode: It** is a way to introduce better error checking in our code. If we use keyword :***use strict”***  in the first line then we cannot use
  + undeclared variable,
  + can’t assign a value in read only property
  + Can’t add a property to an object which is not extensible.
  + For other restricted items: <https://docs.microsoft.com/en-us/scripting/javascript/advanced/strict-mode-javascript#rest>
* **Variables are Function Scoped:** The scope of a variable is always the complete function, not only partial. i.e. Let say, if any variable is declared inside inner block of a function, still the scope of that variable is present in whole outer function block too. (variable wherever it may declare inside nested inner block but it visible through out function scope)
* **Variables are hoisted :** The declared variables is internally moved to first line, but it remains undefined, the assignment will be remain in same place.

|  |  |
| --- | --- |
| ***Actual Code*** | ***How JS Interpret internally*** |
| ***function foo(){***  ***console.log(tmp) //output : undefined  if (true){***  ***var tmp = 3;***  ***}***  ***}*** | ***function foo(){***  ***var tmp;***  ***console.log(tmp) //output : undefined  if (true){***  ***tmp = 3;***  ***}***  ***}*** |

* **Closure:** A closure is an inner function that can access:
  + Its own scope (variables defines inside curly braces.
  + to outer function variables and parameters.
  + to global variables.
  + Each function stays connected to the variables of the functions that surround it, even after it leaves the scope in which it was created

|  |  |
| --- | --- |
| *function incrementCounter(i){*  *function(){*  *i++;*  *return I;*  *}*  *}* | *Output:*  *var incr = incrementCounter(6);*  *>incr();*  *7*  *>incr();*  *8*  *>incr();*  *9* |

* **IIFE (Immediately Invoked Function Expression):** We know that in JS, variables are global, scope and visibility is high. So IIFE is an option through which we can restrict or control the scope of variable within the function. We need to use a pattern for using a function.

**(function () { //start IIFE**

**var tmp = “Abhi” //NOT A GLOBAL VARIABLE**

**} ());**

*Inside the above function, a new concise scope exists, preventing “****tmp”*** *from becoming global.*

* **Single Object:** Set of properties (add as many properties), where each property is a (key, value) pair where key is string and value could be any JS data type

|  |  |
| --- | --- |
| *var myObj = {*  *name: ‘Abhijeet’,*  *describeMe : function(){ “My name is Lakhan”}*  *};* | *Output:*  *>myObj.name*  *Abhijeet*  *>myObj.name = ‘Banerjee’*  *Banerjee*  *>myObj.describeMe()*  *My name is Lakhan* |
| ***“in”*** *operator check whether a property exists* | *>’describeMe in myObj*  *True*  *> foo in myObj*  *false* |
| *The* ***“delete”*** *operator removes property* | *>delete myObj.describeMe*  *True*  *>’describeMe in myObj*  *False* |

* **bind():** When we extract a method, it loses its connection with the object

|  |  |
| --- | --- |
| *'use strict';*  *var jane = {*  *name: 'Jane',*    *describe: function () {*  *return 'Person named '+this.name;*  *}*  *};* | *Output:*  *> var func = jane.describe;*  *> func()*  ***TypeError: Cannot read property 'name' of undefined***  *Solution:*  *> var func2 = jane.describe.****bind****(jane);*  *> func2()*  *'Person named Jane'* |