**ECMAScript**

ECMAScript (or ES) is a scripting-language specification. It was created to standardize JavaScript, to foster multiple independent implementations.

**Arrow Functions**

In ES5, a function would be defined as such:

|  |
| --- |
| var readWikiArticle = function(content) {  // Read it!  }; |

Whereas in ES6, using the new concise arrow function syntax:

|  |
| --- |
| var readWikiArticle = (content) => {  //Read article!  }; |

* Arrow functions also improve variable binding between functions.
* You don't need the function keyword, the return keyword, and the curly brackets. But You can only omit the return keyword and the curly brackets if the function is a single statement.
* The handling of ***'this'*** is also different in arrow functions compared to regular functions. In short, with arrow functions there are no binding of this. In regular functions the ***'this'*** keyword represented the object that called the function, which could be the window, the document, a button or whatever. With arrow functions the ***'this'*** keyword always represents the object that defined the arrow function.

**let and const**

ES2015 introduced two important new JavaScript keywords: let and const. These two keywords provide Block Scope variables (and constants) in JavaScript. Global variables defined with the let keyword do not belong to the window object. Using a let variable before it is declared will result in a ReferenceError.

* Variables defined with var are hoisted to the top and belong to window object.
* Global variables defined with the let keyword do not belong to the window object and let is not hoisted.

**Function Scope**

Variables declared Locally (inside a function) have Function Scope.

**Block Scope**

Before ES2015 JavaScript did not have Block Scope. Variables declared with the let keyword can have Block Scope. Variables declared inside a block {} cannot be accessed from outside the block:

|  |
| --- |
| {  let x = 2;  }  // x can NOT be used here |

**JSON**

JSON stands for JavaScript Object Notation.

**Polyfills**

Polyfills allow **using functionalities from newer ECMA versions in older environments** that lack them. In web development, a polyfill is code that implements a feature on web browsers that do not support the feature.

**Transpiling**

Transpilation is a source to source compilation in which the newer versions of JavaScript are used in the user's source code and the **transpiler rewrites them so that they are compliant with the current specification. Usually, transpilers transpile down to ES3** to maintain compatibility with all versions of browsers. Transpiling is sometimes done to avoid needing polyfills.

**Hoisting**

Hoisting is JavaScript's default behaviour of **moving all declarations to the top of the current scope** (to the top of the current script or the current function).

**Classes**

ES6 introduced classes. A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties is assigned inside a constructor() method. Use the keyword class to create a class, and **always add a constructor method**. The constructor method is called each time the class object is initialized. A simple class definition for a class named "Car":

|  |
| --- |
| class Car {  constructor(brand) {  this.carname = brand;  }  methodName() {  return;  }  static staticMethodName() {  return;  }  present() {  return 'I have a ' + this.carname;  }  get cnam() {  return this.carname;  }  set cnam(x) {  this.carname = x;  }  }  class Model extends Car {  constructor(brand, mod) {  super(brand);  this.model = mod;  }  show() {  return this.present() + ', it is a ' + this.model;  }  } |

Unlike functions, and other JavaScript declarations, **class declarations are not hoisted**.

**ES6 added the following properties to the Number object:**

* EPSILON (Number.EPSILON)
* MIN\_SAFE\_INTEGER (Number.MIN\_SAFE\_INTEGER)
* MAX\_SAFE\_INTEGER (Number.MAX\_SAFE\_INTEGER)

**Default Parameter Values**

ES6 allows function parameters to have default values.

Example

|  |
| --- |
| function myFunction(x, y = 10) {  // y is 10 if not passed or undefined  return x + y;  }  myFunction(5); // will return 15 |

**Array.findIndex()**

The findIndex() method returns the index of the first array element that passes a test function. Below example finds the index of the first element that is larger than 18:

|  |
| --- |
| var numbers = [4, 9, 16, 25, 29];  var first = numbers.findIndex(myFunction);  function myFunction(value, index, array) {  return value > 18;  } |

**Array.find()**

The find() method returns the value of the first array element that passes a test function. This example finds (returns the value of) the first element that is larger than 18:

|  |
| --- |
| var numbers = [4, 9, 16, 25, 29];  var first = numbers.find(myFunction);  function myFunction(value, index, array) {  return value > 18;  } |

**x \*\* y produces the same result as Math.pow(x,y)**

**The "use strict" Directive (ES5)**

"use strict" defines that the JavaScript code should be executed in "strict mode". With strict mode you can, for example, not use undeclared variables.

|  |
| --- |
| x = 3.14; // This will not cause an error.  myFunction();  function myFunction() {  "use strict";  y = 3.14; // This will cause an error  } |

**The "use strict" directive is only recognized at the beginning of a script or a function.**

**Getter & Setter**

This example creates a setter and a getter for the language property:

|  |
| --- |
| var person = {  firstName: "John",  lastName : "Doe",  language : "NO",  get lang() {  return this.language;  },  set lang(value) {  this.language = value;  }  };  // Set an object property using a setter:  person.lang = "en";  // Display data from the object using a getter:  document.getElementById("demo").innerHTML = person.lang; |

**Object.defineProperty()**

Object.defineProperty() is a new Object method in ES5.

It lets you define an object property and/or change a property's value and/or metadata.

|  |
| --- |
| // Create an Object:  var person = {  firstName: "John",  lastName : "Doe",  language : "NO",  };  // Change a Property:  Object.defineProperty(person, "language", {  value: "EN",  writable : true,  enumerable : true,  configurable : true  }); |

**ECMAScript 5 allows string literals over multiple lines if escaped with a backslash:**

|  |
| --- |
| Example  var multiline = "Hello \  Dolly!"; |

**ECMAScript 4 was Never released.**

|  |
| --- |
| ***Bad:***  var i;  for (i = 0; i < arr.length; i++) {  ***Better Code:***  var i;  var l = arr.length;  for (i = 0; i < l; i++) { |

**Defer Keyword**

An alternative is to use defer="true" in the script tag. The defer attribute specifies that the script should be executed after the page has finished parsing, but it **only works for external scripts**.

**with Keyword**

|  |
| --- |
| with (Math) {  a = PI \* r \* r;  x = r \* cos(PI);  y = r \* sin(PI / 2);  } |

**Misunderstanding Floats**

All numbers in JavaScript are stored as 64-bits Floating point numbers (Floats). All programming languages, including JavaScript, have difficulties with precise floating-point values:

|  |
| --- |
| var x = 0.1;  var y = 0.2;  var z = x + y // the result in z will not be 0.3 (0.30000000000000004) |

**Avoid Using eval() as it presents security risk**

**The debugger Keyword**

The debugger keyword stops the execution of JavaScript, and calls (if available) the debugging function.

**Constants**

You cannot reassign them (You can only change their properties)

**The call() and apply() methods**

These are predefined JavaScript methods. They can both be used to call an object method with another object as argument.

The difference is:

* The call() method takes arguments separately.
* The apply() method takes arguments as an array.

|  |
| --- |
| var person = {  fullName: function() {  return this.firstName + " " + this.lastName;  }  }  var person1 = {  firstName: "Mary",  lastName: "Doe"  }  person.fullName.apply(person1);  person.fullName.call(person1);  person.fullName.apply(person1, ["Oslo", "Norway"]);  person.fullName.call(person1, "Oslo", "Norway"); |

**Try Catch Finally**

|  |
| --- |
| try {  Block of code to try  }  catch(err) {  Block of code to handle errors  }  finally {  Block of code to be executed regardless of the try / catch result  } |

**Regex**

|  |  |
| --- | --- |
| i | Perform case-insensitive matching |
| g | Perform a global match (find all matches rather than stopping after the first match) |
| m | Perform multiline matching |
| [abc] | Find any of the characters between the brackets |
| [0-9] | Find any of the digits between the brackets |
| (x|y) | Find any of the alternatives separated with | |
| \d | Find a digit |
| \s | Find a whitespace character |
| \b | Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b |
| \uxxxx | Find the Unicode character specified by the hexadecimal number xxxx |
| n+ | Matches any string that contains at least one n |
| n\* | Matches any string that contains zero or more occurrences of n |
| n? | Matches any string that contains zero or one occurrences of n |

|  |
| --- |
| var patt = /e/;  patt.test("The best things in life are free!"); => true  var obj = /e/.exec("The best things in life are free!");  document.getElementById("demo").innerHTML =  "Found " + obj[0] + " in position " + obj.index + " in the text: " + obj.input; |

**Bit Wise Operations**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Result | Same as | Result |
| 5 & 1 | 1 | 0101 & 0001 = | 0001 |
| 5 | 1 | 5 | 0101 | 0001 = | 0101 |
| ~ 5 | 10 | ~0101 = | 1010 |
| 5 ^ 1 | 4 | 0101 ^ 0001 = | 0100 |
| << | Zero fill left shift | Shifts left by pushing zeros in from the right and let the leftmost bits fall off |  |
| 5 << 1 | 10 | 0101 << 1 = | 1010 |
| >> | Signed right shift | Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off |  |
| 5 >> 1 | 2 | 0101 >> 1 = | 0010 |
| >>> | Zero fill right shift | Shifts right by pushing zeros in from the left, and let the rightmost bits fall off |  |
| 5 >>> 1 | 2 | 0101 >>> 1 = | 0010 |

**Convert Binary to Decimal => parseInt("1011", 2)**

**Convert Decimal to Binary => (dec >>> 0).toString(2);**

**You can check the constructor property to find out if an object is an Array or Date (contains the word "Array" or "Date"):**

|  |
| --- |
| function isArray(myArray) {  return myArray.constructor.toString().indexOf("Array") > -1;  }  function isArray(myArray) {  return myArray.constructor === Array;  }  function isDate(myDate) {  return myDate.constructor === Date;  } |

|  |  |
| --- | --- |
| toExponential() | Returns a string, with a number rounded and written using exponential notation. |
| toFixed() | Returns a string, with a number rounded and written with a specified number of decimals. |
| toPrecision() | Returns a string, with a number written with a specified length |

Example

|  |
| --- |
| var aka = 123456789.123456789;  aka.toExponential()  "1.2345678912345679e+8"  aka.toExponential(5)  "1.23457e+8"  aka.toFixed()  "123456789"  aka.toFixed(5)  "123456789.12346"  aka.toPrecision()  "123456789.12345679"  aka.toPrecision(5)  "1.2346e+8" |

**Infinity in JS => var x = Infinity;**

**Break & Continue**

The break statement "jumps out" of a loop.

The continue statement "jumps over" one iteration in the loop.

**for/in**

The JavaScript for/in statement loops through the properties of an object

|  |
| --- |
| var person = {fname:"John", lname:"Doe", age:25};  var text = "";  var x;  for (x in person) {  text += person[x];  }  Here x takes the value like - fname, lname and age |

**The For/Of Loop**

The JavaScript for/of statement loops through the values of an iterable objects, for/of lets you loop over data structures that are iterable such as Arrays, Strings, Maps, NodeLists, and more.

|  |
| --- |
| var cars = ['BMW', 'Volvo', 'Mini'];  var x;  for (x of cars) {  document.write(x + "<br >");  } |

**Switch cases use strict comparison (===).**

**every() some()**

checks for condition function passed as parameters, then return the data as true/false

|  |
| --- |
| var ages = [32, 33, 16, 40];  function checkAdult(age) {  return age >= 18;  }  //gives false  function myFunction() {  document.getElementById("demo").innerHTML = ages.every(checkAdult);  }  //gives true  function myFunction() {  document.getElementById("demo").innerHTML = ages.some(checkAdult);  } |

**By default, the sort() function sorts values as strings.**

arrayorobject.sort(compareFunction) The compare function should return a negative, zero, or positive value, depending on the arguments

**Highest Number in an Array = Math.max.apply(null, arr);**

* The pop() method removes the last element from an array
* The pop() method returns the value that was "popped out"
* The push(obj) method adds a new element to an array (at the end)
* The push(obj) method returns the new array length
* The shift() method removes the first array element and "shifts" all other elements to a lower index.
* The shift() method returns the string that was "shifted out"
* The unshift(obj) method adds a new element to an array (at the beginning), and "unshifts" older elements
* The unshift(obj) method returns the new array length.

**splice()**

Syntax : array.splice(index, howmany, item1, ....., itemX)

With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array

|  |
| --- |
| var removed = fruits.splice(2, 2, "Lemon", "Kiwi")  **Original Array:**  Banana,Orange,Apple,Mango  **New Array:**  Banana,Orange,Lemon,Kiwi  **Removed Items:**  Apple,Mango |

**slice()**

The slice() method slices out a piece of an array into a new array.

|  |
| --- |
| Banana,Orange,Lemon,Apple,Mango  var citrus = fruits.slice(1,3);  Orange,Lemon |

**Events**

|  |  |
| --- | --- |
| onchange | An HTML element has been changed |
| onclick | The user clicks an HTML element |
| onmouseover | The user moves the mouse over an HTML element |
| onmouseout | The user moves the mouse away from an HTML element |
| onkeydown | The user pushes a keyboard key |
| onload | The browser has finished loading the page |

**Naming Convention => Hyphens are not allowed in JavaScript. They are reserved for subtractions.**

**Include HTML snippets in HTML. => https://www.w3schools.com/howto/howto\_html\_include.asp**

**https://www.beyondjava.net/html-includes**

**JavaScript defines 5 types of primitive data types**

* string
* number
* boolean
* null
* undefined

**The named values, in JavaScript objects, are called properties.**

**Defining Objects**

* Using an Object Literal:

|  |
| --- |
| var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}; |

* Using the JavaScript Keyword new:

|  |
| --- |
| var person = new Object();  person.firstName = "John";  person.lastName = "Doe";  person.age = 50;  person.eyeColor = "blue"; |

**Objects are mutable: They are addressed by reference, not by value.**

|  |
| --- |
| var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}  var x = person;  x.age = 10; // This will change both x.age and person.age  The object x is not a copy of person. It is person. Both x and person are the same object. |

|  |
| --- |
| var person = {  firstName: "John",  lastName : "Doe",  language : "en",  getlang : function() {  return this.language;  },  get lang() {  return this.language;  },  set lang(lang) {  this.language = lang;  }  };  Object.defineProperty()  The Object.defineProperty() method can also be used to add Getters and Setters:  var obj = {counter : 0};  // Define setters  Object.defineProperty(obj, "reset", {  get : function () {this.counter = 0;}  });  Object.defineProperty(obj, "subtract", {  set : function (value) {this.counter -= value;}  }); |

|  |
| --- |
| var x6 = /()/ // new regexp object  var x7 = function(){}; // new function object |

**You cannot add a new property to an existing object constructor**

|  |
| --- |
| Person.nationality = "English"; //wont't reflect on objects created with Person |

**The JavaScript prototype property allows you to add new properties to object constructors:**

|  |
| --- |
| Person.prototype.nationality = "English"; //will reflect on objects created with Person |

**The JavaScript prototype property also allows you to add new methods to objects constructors**

**ECMAScript 5 added a lot of new Object Methods to JavaScript.**

|  |
| --- |
| **Adding or changing an object property**  Object.defineProperty(object, property, descriptor)  Ex. Object.defineProperty(person, "language", {value : "NO"}); |
| **Adding or changing many object properties**  Object.defineProperties(object, descriptors) |
| **Accessing Properties**  Object.getOwnPropertyDescriptor(object, property) |
| **Returns all properties as an array**  Object.getOwnPropertyNames(object) // Returns an array of properties |
| **Returns enumerable properties as an array**  Object.keys(object) // Returns an array of enumerable properties (if enumerable:false then it is rejected) |
| **Accessing the prototype**  Object.getPrototypeOf(object) |
| **Prevents adding properties to an object**  Object.preventExtensions(object) |
| **Returns true if properties can be added to an object**  Object.isExtensible(object) |
| **Prevents changes of object properties (not values)**  Object.seal(object) |
| **Returns true if object is sealed**  Object.isSealed(object) |
| **Prevents any changes to an object**  Object.freeze(object) |
| **Returns true if object is frozen**  Object.isFrozen(object) |

**Changing Meta Data**

ES5 allows the following property meta data to be changed:

writable : true // Property value can be changed

enumerable : true // Property can be enumerated

configurable : true // Property can be reconfigured

writable : false // Property value can not be changed

enumerable : false // Property can be not enumerated

configurable : false // Property can be not reconfigured

**ES5 allows getters and setters to be changed:**

*Defining a getter*

get: function() { return language }

*Defining a setter*

set: function(value) { language = value }

|  |
| --- |
| **This example makes language read-only**  Object.defineProperty(person, "language", {writable: false}); |
| **This example makes language not enumerable**  Object.defineProperty(person, "language", {enumerable: false}); |

**Functions defined using an expression are not hoisted.**

|  |
| --- |
| // Outputs: "Definition hoisted!"  definitionHoisted();  // TypeError: undefined is not a function  definitionNotHoisted();  function definitionHoisted() {  console.log("Definition hoisted!");  }  var definitionNotHoisted = function () {  console.log("Definition not hoisted!");  };  // ReferenceError: funcName is not defined  funcName();  // TypeError: undefined is not a function  varName();  var varName = function funcName() {  console.log("Definition not hoisted!");  }; |

**Function parameters are the names listed in the function definition.**

**Function arguments are the real values passed to (and received by) the function.**

**The Arguments Object**

JavaScript functions have a built-in object called the arguments object. (You can use "arguments" as an object to play around with passed arguments)

* Arguments are Passed by Value
* Changes to arguments are not visible (reflected) outside the function.
* Objects are Passed by Reference
* Changes to object properties are visible (reflected) outside the function.

**Spread Operator (...)**

Spread operator can be used in many cases, like when we want to expand, copy, concat, with math object.

let arr = [1,2,3];

let arr2 = [4,5];

arr = [...arr,...arr2];

**Note: Though we can achieve the same result, but it is not recommended to use the spread in this particular case, as for a large data set it will work slower as when compared to the native concat() method.**

|  |
| --- |
| \*\*\*\*\*\*\*\*\*\*\*References  // changed the original array  let arr = ['a','b','c'];  let arr2 = arr;    arr2.push('d');    console.log(arr2);  console.log(arr); // even affected the original array(arr) |

**The JavaScript call() Method**

It can be used to invoke (call) a method with an owner object as an argument (parameter).

With call(), an object can use a method belonging to another object.

This example calls the fullName method of person, using it on person1:

|  |
| --- |
| var person = {  fullName: function() {  return this.firstName + " " + this.lastName;  }  }  var person1 = {  firstName:"John",  lastName: "Doe"  }  person.fullName.call(person1); // Will return "John Doe" |

|  |
| --- |
| var akash = 100;  function add(){  console.log(akash); //Displays undefined  var akash = 10;  akash = akash + akash;  console.log(akash);  } |

**Function Hoisting is causing the issue in above case**

**Counter Dielema Solution (Private Variables)**

|  |
| --- |
| var add = (function () {  var counter = 0;  return function () {counter += 1; return counter}  })();  add();  add();  add();  // the counter is now 3 |

**This is called a JavaScript closure. It makes it possible for a function to have "private" variables.**

**A closure is a function having access to the parent scope, even after the parent function has closed.**

**DOM**

The HTML DOM (Document Object Model)

document.querySelectorAll("p.intro");

**Navigating Between Nodes**

* parentNode
* childNodes[nodenumber]
* firstChild
* lastChild
* nextSibling
* previousSibling

**The Browser Object Model (BOM) allows JavaScript to "talk to" the browser.**

prompt("someText","DefaultText")//returns the entered value

confirm("someText")//returns true/false, alert("someText")

**Timeout**

|  |
| --- |
| setTimeout(function, milliseconds) |

Executes a function, after waiting a specified number of milliseconds.

The **clearTimeout()** method stops the execution of the function specified in setTimeout().

|  |
| --- |
| setInterval(function, milliseconds) |

Same as setTimeout(), but repeats the execution of the function continuously.

The setTimeout() and setInterval() are both methods of the HTML DOM Window object.

The **clearInterval()** method stops the executions of the function specified in the setInterval() method.

**Cookie**

|  |
| --- |
| document.cookie = " name=value; expires=Thu, 18 Dec 2013 12:00:00 UTC"; |

For deleting a cookie just set expires as some old-time value

**AJAX Asynchronous JavaScript And XML**

AJAX is not a programming language.

AJAX just uses a combination of:

* A browser built-in XMLHttpRequest object (to request data from a web server)
* JavaScript and HTML DOM (to display or use the data)

|  |
| --- |
| var xhttp = new XMLHttpRequest();  xhttp.open("GET", "ajax\_info.txt", true);  xhttp.send();  //xhttp.send(); //post  xhttp.open("GET", "demo\_get.asp", true);  xhttp.send();  In the example above, you may get a cached result. To avoid this, add a unique ID to the URL:  xhttp.open("GET", "demo\_get.asp?t=" + Math.random(), true);  xhttp.send(); |

By sending asynchronously, the JavaScript does not have to wait for the server response, but can instead:

execute other scripts while waiting for server response

deal with the response after the response is ready

**readyState Holds the status of the XMLHttpRequest.**

0: request not initialized

1: server connection established

2: request received

3: processing request

4: request finished and response is ready

With the XMLHttpRequest object you can define a function to be executed when the request receives an answer. The function is defined in the onreadystatechange property of the XMLHttpResponse object:

|  |
| --- |
| xhttp.onreadystatechange = function() {  if (this.readyState == 4 && this.status == 200) {  document.getElementById("demo").innerHTML = this.responseText;  }  }; |

The onreadystatechange event is triggered four times (1-4), one time for each change in the readyState.

Synchronous XMLHttpRequest (async = false) is not recommended because the JavaScript will stop executing until the server response is ready. If the server is busy or slow, the application will hang or stop.

**Functions in JavaScript**

|  |
| --- |
| function functionName(){ /\*function body\*/ }  function functionName(x,y){ return x+y; } |

**Immediately Invoked Function Expression (aka IIFE)**

|  |
| --- |
| (function(n){/\*function body\*/})(n); |

**What happens if function name is repeated?**

Last function of similar name is executed.

**Method Chaining**

|  |
| --- |
| Object.methodOne().methodTwo().methodThree(); |

Method Chaining works in above case only if methodOne() and methodTwo() returns something to call next function upon.

**Check if JS is disabled**

|  |
| --- |
| <script>/\*some javascript\*/</script>  <noscript>/\*for when javascript is diabled\*/</noscript> |

**Direct Instance of an Object**

|  |
| --- |
| var personObj1 = new Object();  personObj1.id = 8;  personObj1.name = "Akash";  personObj1.getData = function(){  return this.id + " " + this.name;  } |

**Empty Object**

|  |
| --- |
| var personObj2 = {}; |

**Object Literal**

|  |
| --- |
| var personObj3 = {  id:1,  name:"Aditya",  getData:function(){  return this.id + " " + this.name;  }  } |

**Using Template**

|  |
| --- |
| function personObj(id,name){  this.id=id;  this.name=name;  this.getData = function(){  return this.id + " " + this.name;  }  }  personObj4 = new personObj(1,"Aryan"); |

**Deleting Property**

|  |
| --- |
| delete personObj4.name; |

**Define Property**

Object.defineProperty() - method defines a new property directly on an object or modifies an existing property on an object and return the object.

|  |
| --- |
| Object.defineProperty(man, 'species', {  value: 'Human Being',  writable: false,  configurable: false,  enumerable: true  }); |

**Constructor Level Function**

|  |
| --- |
| function Employee(r){  this.role =r;  this.display = function()  {  alert('CTOR Level '+this.role);  }  } |

**Prototype Level Function**

|  |
| --- |
| Employee.prototype.display = function(){  alert("Prototype Level "+this.role);  } |

**Object Level Function**

|  |
| --- |
| emp1.display = function(){  alert('Object Level '+this.role);  } |

**Abstraction and Encapsulation**

|  |
| --- |
| function Person(){  this.Id = 100;  this.hobbies = ['Music','Cricket'];  // private data member  var name="Default";  // private method  var privateMethod = function(){  console.log("Calling Private Method!");  }  // Priviledged Method  this.getInfo = function(){  privateMethod();  return this.Id+' '+name;  }  Person.population++;  } |

**Static Property**

|  |
| --- |
| Person.population = 0; |

**Prototype Property**

Only Constructor has prototype property and not the instances. Prototype property can also be used by its object unlike static which can only be used by Class

|  |
| --- |
| Person.prototype.hasBrain = true; |

**Inheritance using Prototype Chaining**

|  |
| --- |
| Child.prototype = new Parent('Mother');  Child.prototype.constructor = Child; |

**Limitations**

1. Arguments can't be passed to the super/base class data member while creating sub/derived class object.

2. Inherited Instance Properties of super /base class become

**Inheritance using classical Inheritance**

|  |
| --- |
| function Child(rel){  Parent.call(this,rel);// Inheritance  } |

**Limitations**

1. No Function reuse - inmemory no reuse

2. Child can't access methods defined at parent's prototype level

3.Pseudoclassical Inheritance - Combination of prototype chaining and classical inheritance

Human - base/super class

Student - derived/sub class

|  |
| --- |
| function Human(name){  this.name = name;  }  Human.prototype.introduction = function(){  alert('Hi, I am '+this.name);  }  // Prototype Chaining  Student.prototype = new Human();  Student.prototype.constructor = Student;  function Student(nm,college,courses){  Human.call(this,nm);// Classical Inheritance  this.college = college;  this.courses = courses;  }  Student.prototype.introduction = function(){  alert('Hi, I am '+this.name+  ', I am a student of '+this.college+  ', I study '+this.courses);  }  Student.prototype.takeExams = function(){  alert('This is Exam time!');  } |

**Polymorphism**

|  |
| --- |
| function letMeIntroduce(obj){  obj.introduction();// polymorphism  } |

**With closure**

|  |
| --- |
| var getHits = (function () {  var hits = 0;// initialized to 0  function addHits() { // lexical scope  hits++;  return hits;  }  return addHits;  })(); |

closure can retain the environment of the outer function.

**Module Pattern**

|  |
| --- |
| var empModule = (function(){  var empList = [];  return {  add:function(e){  empList.push(e);  },  show:function(){  for(var index in empList){  console.log(empList[index]);  }  }  }  })();// IIFE |

**Old Way and New Way of Creating Class**

**Old Way**

|  |
| --- |
| function personObj(id,name){  this.id=id;  this.name=name;  this.getData = function(){  return this.id + " " + this.name;  }  } |

**New Way**

|  |
| --- |
| class Car {  constructor(brand) {  this.carname = brand;  }  methodName() {  return;  }  static staticMethodName() {  return;  }  present() {  return 'I have a ' + this.carname;  }  get cnam() {  return this.carname;  }  set cnam(x) {  this.carname = x;  }  } |