**JavaScript**

java script is light weight , cross-platform, Object-oriented computer programming language.

Js is one of the 3 core technologies of web development and it is used as part of webpages

Client side : js was traditionally used only in browser.

**Topic 1: Datatypes**

Variable is like a container which is used to store a value and use over and over again in the project

String,Number,Boolean,Null,Undefined,Symbol

**Types of comments:**

Single line comments: //

Multiple line comments: /\*…\*/

**Variable mutation and type coercion:**

type coercion:

Var job,isMarried;

Job= ‘teacher’;

age = 26;

console.log(job +age);

**Variable mutation:**

Age =’thirty six’

console.log(job +age);

**Operators:**

Js follows operators precedence which decides what should be executed first.

MDN reference can give u much understanding on operator precedence.

**Control Structures:**

About if/else

And also about == and ===(type coercion)

Boolean logic

Switch statement

Var job=’teacher’;

Switch(job){

Case ‘teacher’:

///

Break;

Case ‘driver’

///

Break;

…

Default:

///

}

**Topic 2:Functions:**

Functions are containers which holds a couple of lines which can be used over and over again.

We can pass arguments to it to have some information from outside and it will return some result

To avoid repeating of code we use functions

Function calculateAge(yearOfBirth){

Var age = 2017- yearOfBirth;

Return age;

}

Var ageAnita = calculateAge(1984);

//

Fuctions can also call another functions , and also no need to return some value compulsory.

Function calculateRetirementAge(name, year){

Var age = calculateAge(year);

Var retirmentAge = 65-age;

Console.log(name + retirmentAge);

}

calculateRetirementAge(‘Anita’,33);

**function statements and functions expressions:**

statements:

function f1(par){

///

}

Expressions:

Var someFun = functions(par){

///

}

Difference between statement and expression is :

Exp produces a value in outcome , while a statement performs some action.

**Arrays:**

to course bundle of values into single variable

var names =[‘abc’,’bcv’];

var names = new Array(…);

index based.

Push—add element at the end of the array

Pop-remove the element at the end of the array.

Shift— remove the element at the end of the array.

Unshift- add the element at the start of the array

IndexOf-returns the position of the element.

**Objects and Properties:**

Var obj1 = {

Key: value,

Key1: value1

}

We can retrieve the value by using the key.

var xyz =’key’;

console.log(obj1[xyz]);

obj1.key1 =’value2’;

obj1[key] = ‘value1’;

the variable mutation works same way how it works in the variables.

Var obj1 = new Object();

Object can hold different type of data such as arrays,functions and even other objects;

Var obj1 ={

///

Family:[…],

calcualteAge:function(dob){

//calculate age.

}

Each object have properties and methods.

Loops and Iterations:

For loop and while

Continue- to quit the current iteration of the loop and continue with others

break- to break the loop and quit it

**Topic 3: JS behind the sceens:**

**JS engine**

Js always hoisted on some environment(commonly browser).

Js engine is a program which executed js code

Eg: spider monkey,v8 engine and js core

First our code is read by a parser(which knows the js rules and how it has to be written a valid code).

Parser produces a data structure called abstract syntax tree which is then transelated to machine code and then it runs and produces the result.

**Execution context and execution stack:**

Js code runs on environment , this envi is called execution context.

It is similar to a container

Global execution context is for the code which is not inside any function

The default execution context is global context

Global context is always associated with global object.

In the browser the window object global object.

What about the code inside the function.

So when ever the function is called it gets its own brand new execution context.

Var name =’anita’;

Function first(){

Var a =’hello’;

Second();

Var x = a + name;

}

Function second(){

Var b =’Hi’;

Var z = b+name;

}

First();

Execution context object contains 3 properties

Variable Object(VO)-contains function arguments, variable and function declarations

Scope chain-contains its currents variable objects as well as all its parents objects

This variable-current object

Execution context is actually created in 2 phases

Creation phase:

Creation of vo

Creation of scope

Determine the value of this variable

All the 3 properties are getting created during creation phase

Execution phase:

The code of the function that is generated in the current execution context is ran line by line.

Vo:

Argument object is created, containing all the arguments that were passed into the function.

Code is scanned for function declarations, for each function a property is created in vo

Code is scanned for variable declarations: for each variable, a property is created in vo and set to undefined.

Last 2 points comes under hoisting.

**Hoisting**: in function declartions

function calculateAge(year) {

console.log(2016 - year);

}

calculateAge(1965);(keep this after or before function declaration it will be working, since the function declaration is done in creation context so it will not through the error)

Hoisting: in function expressions:

var retirement = function(year) {

console.log(65 - (2016 - year));

}

retirement(1956);

calling function after function expression it will work as expected

but if we call function before function expression it will through expection this is because expression.

Hoisting works only for declartions but not for expressions.

Same with variables , it will be undefined

c.l(age);🡪undefined

var age =23;

c.l(age);🡪23

function foo() {

console.log(age);🡪undefined🡪function execution context(since variable is not yet assigned)

var age = 65;

console.log(age);🡪65🡪function ec

}

foo();

console.log(age);🡪23🡪global ec

**Scoping chain:**

In Js each function creates its own scope in which the variables it defines are accessible

Lexical scoping:a function that is lexically within another function gets access to the scope of the outer function.

var a = 'Hello!';

first();

function first() {

var b = 'Hi!';

second();

function second() {

var c = 'Hey!';

console.log(a + b + c);

}}

This is called the scope chain

Scope chain works from bottom to top, but it will not work top to bottom

**Execution stack:**

var a = 'Hello!';

first();

function first() {

var b = 'Hi!';

second();

function second() {

var c = 'Hey!';

third()

}

}

function third() {

var d = 'John';

//console.log(c);

console.log(a+d);

}

In execution stack the global scope is at the bottom on top of it the 1st ,2nd and 3rd functions execution context are placed on top of it

Execution stack and scope chain are vise versa.

Execution stack is the order in which functions are called

Scope chain is order in which the functions are written lexically in the code.

**This:**

Regular function call this keyword points to the global object

console.log(this);

Method call this variable points to the object that is calling the method

function calculateAge(year) {

console.log(2016 - year);

console.log(this);

}

This keyword is attached to the execution context.

**DOM manipulation:**

How to manipulate dom

Read from dom

Document object give access to dom

It has couple of methods for eg

Queryselector

var diceDOM = document.querySelector('.dice');

diceDOM.style.display = 'block';

diceDOM.src = 'dice-' + dice + '.png';

events: notifications that are sent to notify the code that something happened on the webpage;

eg: clicking button,resizing a window,scrolling down or pressing a key

we can use event listener to code response to all this events.

Event listener is a function which performs some action for the events.

All the events are put in message queue in js engine .

Any event can be processed only when execution stack is empty.which means all the functions are returned.

After execution stack is empty then one by one event will be processed.

**DRY (don’t repeat your )PRINCIPLE:**

**How to use functions to correctly apply the dry principle**

**State variable:**

**Objects and functions:**

In js everything is object

Arrays,functions,objects,dates,wrappers for numbers,strings,Boolean.

Oop 🡪 heavy use of objects,properties and methods

Objects talk with each other through properties and methods

Oop is used to store data, structure applications into modules and keep code clean.

In java it is called as class but in js we used to call it as constructor or prototype

From constructor(blueprint) can create as many as instances

js is prototype based language.in Js inheritance works based on prototypes.

Each object has prototype property which makes inheritance possible in js.

The prototype property of an object is where we put methods and properties that we want other objects to inherit

The constructors prototype property is not the prototype of the constructor itself.its the prototype of all instances that are created through it.

When a certain method(or property) is called, the search starts in the object itself, and if it cannot be found, the search moves on to the objects prototype this continues until the method is found: prototype chain.

var Person = function(name, yearOfBirth, job) {

this.name = name;

this.yearOfBirth = yearOfBirth;

this.job = job;

this.calculateAge = function(){

c.l(2017-this. yearOfBirth);

}

var john = new Person('John', 1990, 'teacher');

john.calculateAge();

Person.prototype.calculateAge = function() {

console.log(2016 - this.yearOfBirth);

};

Person.prototype.lastName = 'Smith';

var jane = new Person('Jane', 1969, 'designer');

var mark = new Person('Mark', 1948, 'retired');

Object .create:

Another way to create object and get inheritance.

In this 1st need to define a object that will act as prototype and then create new object based on that prototype

var personProto = {

calculateAge: function() {

console.log(2016 - this.yearOfBirth);

}

};

var john = Object.create(personProto);

**Functions:**

**Passing functions as arguments:**

A function is an instance of the object type, it behaves like another object.

var years = [1990, 1965, 1937, 2005, 1998];

function arrayCalc(arr, fn) {

var arrRes = [];

for (var i = 0; i < arr.length; i++) {

arrRes.push(fn(arr[i])); } return arrRes;

}

function calculateAge(el) {

return 2016 - el;

}

var ages = arrayCalc(years, calculateAge);

c.l(ages);

**Functions returning functions:**

function interviewQuestion(job) {

if (job === 'designer') {

return function(name) {

console.log(name + ', can you please explain what UX design is?');

}

} else if (job === 'teacher') {

return function(name) {

console.log('What subject do you teach, ' + name + '?');

}

} else {

return function(name) {

console.log('Hello ' + name + ', what do you do?');

} }}

var teacherQuestion = interviewQuestion('teacher');

var designerQuestion = interviewQuestion('designer');

teacherQuestion('John');

designerQuestion('John');

interviewQuestion('teacher')('Mark');

**Immediate** **Invoked Function Expressions:**

Immediate function without parameters and with parameters.

If we don’t want variables to be accessed outside the functions then invoked function expressions are very useful.

(function () {

var score = Math.random() \* 10;

console.log(score >= 5);

})();

(function (goodLuck) {

var score = Math.random() \* 10;

console.log(score >= 5 - goodLuck);

})(5);

**Closures:**

A inner function has always access to the variables and parameters of its outer function , even after the outer function has returned.

function retirement(retirementAge) {

var a = ' years left until retirement.';

return function(yearOfBirth) {

var age = 2016 - yearOfBirth;

console.log((retirementAge - age) + a);

}

}

retirement(66)(1990);

**Bind,Call and apply methods:**

Functions are special kind of objects so they also get some special methods from function constructor objects.

This methods are allowed to call a function inside the this variable manually.

var john = {

name: 'John',

age: 26,

job: 'teacher',

presentation: function(style, timeOfDay) {

if (style === 'formal') {

console.log('Good ' + timeOfDay + ', Ladies and gentlemen! I\'m ' + this.name + ', I\'m a ' + this.job + ' and I\'m ' + this.age + ' years old.');

} else if (style === 'friendly') {

console.log('Hey! What\'s up? I\'m ' + this.name + ', I\'m a ' + this.job + ' and I\'m ' + this.age + ' years old. Have a nice ' + timeOfDay + '.');

}

}

};

var emily = {

name: 'Emily',

age: 35,

job: 'designer'

};

john.presentation('formal', 'morning');

john.presentation.call(emily, 'friendly', 'afternoon');

call method is used on john variable - the 1st argument is the object which is to be replaced with ‘this’ variable in john object and other variables are the parameters of the function to be called actually.

The call method allows us to set the this variable as a 1st argument

This is called method borrowing .

Emily object borrowed a method from john object.

john.presentation.apply(emily, ['friendly', 'afternoon']);

Call and apply methods are very similar in behavior the only difference is in call method it accepts parameters (from 2nd argument) individually, but in apply method it accepts array(all the parameters passed to function) as a 2nd argument.

If we know the parameters length of the function go for call method, if we don’t know exact number parameters of the function go for apply.

var emilyFormal = john.presentation.bind(emily, 'formal');

emilyFormal('afternoon');

bind is very similar to call, it also allows to set the this variable explicitly.

The difference is bind method will not call the function immediately, but it will generate the copy of function that we can store it somewhere.

That is extremely useful to create a function with preset arguments

**ES6 features:**

ES5 is fully supported in all modern browsers.

New versions are ES6🡪used in modern browsers but not supported by old browers so not be used in production system, ES2016, ES2017🡪 not supported in modern browsers also.

**Features:**

Let and const, blocks and IIFES, strings, arrow functions, destructing, Arrays, The spread operator, Rest and default parameters, Maps, classes and subclasses.

**Let and const:**

Const is used to declared a variable with a value which cannot be modified at any other point

const name6 = 'Jane Smith';

let age6 = 23;

name6 = 'Jane Miller';

console.log(name6);

let provides the block level scope to the variable.

let i = 23;

for (let i = 0; i < 5; i++) {

console.log(i);

}

console.log(i);

**Blocks and IIFS:**

Blocks are not restricted to control stmts , we can also just write a block for simple code.

Syntax: {

//simple code

}

{

const a = 1;

let b = 2;

var c = 3;

}

Console.log(a+b);🡪throws error

Console.log©;🡪works

IIFs in ES5 are

(function(){

///})();

IIFs in ES6:

{

}

**String:** in ES6 use backtick(`) to represent the string along with auto populated values during runtime

let firstName = 'John';

let lastName = 'Smith';

const yearOfBirth = 1990;

function calcAge(year) {

return 2016 - year;

}

// ES5

console.log('This is ' + firstName + ' ' + lastName + '. He was born in ' + yearOfBirth + '. Today, he is ' + calcAge(yearOfBirth) + ' years old.');

// ES6

console.log(`This is ${firstName} ${lastName}. He was born in ${yearOfBirth}. Today, he is ${calcAge(yearOfBirth)} years old.`);

c.l(`${firstName} ${lastName}`);

**Arrow Functions:**

const years = [1990, 1965, 1982, 1937];

map function creates a new array with the results of calling a provided function on every element in this array.

// ES5

var ages5 = years.map(function(el) {

return 2016 - el;

});

console.log(ages5);

// ES6

let ages6 = years.map(el => 2016 - el);

console.log(ages6);

ages6 = years.map((el, index) => `Age element ${index + 1}: ${2016 - el}.`);

console.log(ages6);

**Destructuring:**

It is very convenient way to extract data from a data structure such as object or an array.

// ES5

var john = ['John', 26];

//var name = john[0];

//var age = john[1];

// ES6

const [name, age] = ['John', 26];

console.log(name);

console.log(age);

**Arrays:**

//ES5

var ages = [12, 17, 8, 21, 14, 11];

var full = ages.map(function(cur) {

return cur >= 18;

});

console.log(full);

console.log(full.indexOf(true));

console.log(ages[full.indexOf(true)]);

//ES6

console.log(ages.findIndex(cur => cur >= 18));

console.log(ages.find(cur => cur >= 18));

\*/

**Spread Operator:**

It is very convenient way to expand elements of an array which are like a arguments in function calls.

function addFourAges (a, b, c, d) {

return a + b + c + d;

}

//ES5

var ages = [18, 30, 12, 21];

var sum2 = addFourAges.apply(null, ages);

console.log(sum2);

//ES6

const sum3 = addFourAges(...ages);

console.log(sum3);

**Rest Parameter:**

Rest parameter allows us to pass arbitory number of arguments to a function, induces these arguments in that function.

The notation of spread and rest is very same (…), but the functionality is exact opposite.

Spread takes a array and transforms in to single value.

Rest parameters takes the single values and transforms them into an array when we call a function with multiple parameters.

function isFullAge6(...years) {

years.forEach(cur => console.log( (2016 - cur) >= 18));

}

isFullAge6(1990, 1999, 1965, 2016, 1987);

//ES6

function isFullAge6(limit, ...years) {

years.forEach(cur => console.log( (2016 - cur) >= limit));

} isFullAge6(16, 1990, 1999, 1965, 2016, 1987);

**Default Parameters:**

We use them when 1 or more parameters of a function to be preset.That is to set the default value to the arguments.

function SmithPerson(firstName, yearOfBirth, lastName = 'Smith', nationality = 'american') {

this.firstName = firstName;

this.lastName = lastName;

this.yearOfBirth = yearOfBirth;

this.nationality = nationality;

}

var john = new SmithPerson('John', 1990);

var emily = new SmithPerson('Emily', 1983, 'Diaz', 'spanish');

**Maps:**

This is new data structures in ES6.

Map is a new key value data structure in ES6.

In Map we can use anything (primitive datatypes, objects, functions..etc) as key.

const question = new Map();

question.set(1, 'ES5');

question.set(2, 'ES6');

question.set(3, 'ES2015');

question.set(4, 'ES7');

console.log(question.get('question'));

//question.clear();🡪 to clear entire map object

question.delete(4);🡪 to delete particular value from the map.

To loop a map object we can use for each .

question.forEach((value, key) => console.log(`This is ${key}, and it's set to ${value}`));

**Classes and SubClasses:**

In ES5 blueprints are called function constructors . in ES6 the same are called as classes.

class Person6 {

constructor (name, yearOfBirth, job) {

this.name = name;

this.yearOfBirth = yearOfBirth;

this.job = job;

}

calculateAge() {

var age = new Date().getFullYear - this.yearOfBirth;

console.log(age);

}

static greeting() {

console.log('Hey there!');

}

}

const john6 = new Person6('John', 1990, 'teacher');

Person6.greeting();

**Subclass:**

class Athlete6 extends Person6 {

constructor(name, yearOfBirth, job, olympicGames, medals) {

super(name, yearOfBirth, job);

this.olympicGames = olympicGames;

this.medals = medals;

}

wonMedal() {

this.medals++;

console.log(this.medals); }}

const johnAthlete6 = new Athlete6('John', 1990, 'swimmer', 3, 10);

johnAthlete6.wonMedal();

johnAthlete6.calculateAge();