**Execution Context/Execution Stack**

All javascript code is run on an environment and these environments are called execution context. You can imagine execution context like a box i.e it is a container in which variables are stored and our piece of code is executed.

The default execution context is the global execution context. In default execution context the code that is not inside any function. You can also consider execution context as a Object therefore global execution object is associated with the global objects.

Global Object is nothing but ‘window’ Object.

Hence declaring lastname and window.lastname is the same thing. So, the variable that is declared outside the functions will get automatically attached to the window but the variable that are inside the functions will have there brand new execution context.

We can associate execution context with execution context Object.This Object has three properties:

1. Variable Object: contains function, variable declaration as well as function declaration.
2. Scope chain: contains current variable objects as well as variables of parent variable objects.
3. This variable

When a fn is called a new excution context is created on the top of the execution stack. And this happens in two phases:

1. Creation phase: creation of variable object , scope chain, detemine value of This variable.
2. Exection phase : Code if the fn runs line by line

hello();

function hello() {

    console.log("Hello World!");

}

Output: Hello World!

Above, function hello is called before its even made and still we could see output in console because, Even before the execution the function get stores into Variable object.

Some syntax

var hello = function test(some) {

    console.log("Hello World! " + some);

}

hello("Tanuj");

Hello World! Tanuj

hello("Tanuj");

var hello = function(some) {

    console.log("Hello World! " + some);

}

test.js:8 Uncaught TypeError: hello is not a function

at test.js:8

Here, we did not get output as function is stored in variable and before execution the variables are *undefined*.

**Dynamically change css classes in DOM Object.**

  .a {

       color: brown;

     }

 document.querySelector("#i").classList.add("a");

 document.querySelector("#i").classList.remove("a");

document.querySelector("#i").classList.toggle("a");

**add:** Will only add no matter how many times you use it.

**Remove:** Will only remove no matter how many times you use it.

**Toggle:** Will add if not present and will remove if present.

**Object and functions**

Almost everything in javascript is an Object.

Primitives:

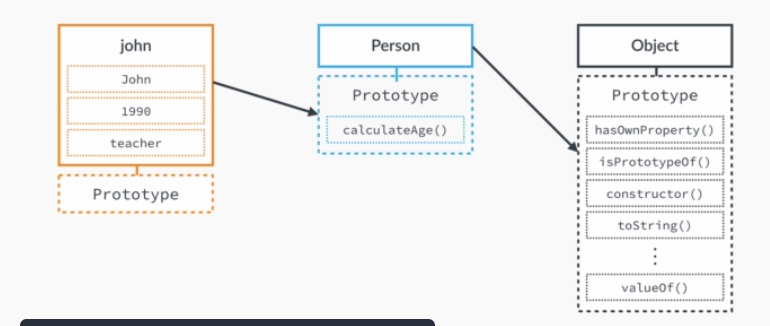
1. Numbers
2. Strings
3. Booleans
4. Undefined
5. Null

Everything else is an Object.

Eg: Arrays, Functions, Objects, Dates etc

Javascript is a prototype based language. Inheritance in javascript works by using the concept of Prototypes. Each and evrey object in

JS has a Prototype property which makes inheritance possible.



Null is the only Object in javascript which has No prototype.

In a nutshell:

* Every JS Object has a Prototype property, which makes inheritance possible in Javascript.
* The Prototype property of an Object is where we put methods and properties that we want other objects to inherit.
* The Constructor’s prototype property is NOT the prototype of the constructor itself, it’s the prototype of ALL instances that are created through it.

**CASE 1: calling a function**

var john = {

    name: "john"

}

var name;

var Person = function (name) {

    this.name = name;

    console.log(this.name)

}

// calling funtion

Person('Surya');

console.log("Name is :: " + this.name);

console.log("Name is :: " + name);

console.log("Name is :: " + john.name);

***OUTPUT***

Surya

Name is :: Surya

Name is :: Surya

Name is :: john

In the above this.name point to the global variable name. If we are calling function this will refer to global variable.

**CASE 2: creating an Object**

var john = {

    name: "john"

}

var name = "global Name";

var person = function (name) {

    this.name = name;

    console.log(this.name)

}

// Creating object

john = new person('Surya');

console.log("Name is :: " + this.name);

console.log("Name is :: " + name);

console.log("Name is :: " + john.name);

***OUTPUT***

Surya

Name is :: global Name

Name is :: global Name

Name is :: Surya

In the above, we created an object, hence in this case ‘this’ refers to Object itself and not the global Object.

**Inheritance**

var person = function () {

    this.name = name;

    this.someMethod = function(){

        console.log('Function')

    }

}

var tanuj = new person();

tanuj.someMethod();

var surya = new person();

surya.someMethod();

OR

var person = function () {

    this.name = name;

}

person.prototype.someMethod = function(){

    console.log('Function');

}

var tanuj = new person();

tanuj.someMethod();

var surya = new person();

surya.someMethod();

Diff btw let/const/var

**let**: It is a datatype which is mutable

**#Important: Its scope is block.**

function testLet(number) {

    let index = 1111;

    if (number < 5) {

        let index = 2222; // Ideally there should be an error

        console.log(index); // 2222

    }

    console.log(index); // 1111

}

testLet(1);

In the above have defined ‘let index’ 2 time but we have not got error because let has block scope.

function testLet(number) {

    let index = 1111;

    if (number < 5) {

        index = 2222; // Now it changes both

        console.log(index); // 2222

    }

    console.log(index); // 2222

}

testLet(1);

**Const : can not be changed, it’s kind of final in java.**

function testConst(number) {

    const index = 1111;

    if (number < 5) {

        const index = 2222; // No error

        console.log(index); // 2222

    }

    console.log(index); // 1111

}

testConst(1);

function testConst(number) {

    const index = 1111;

    if (number < 5) {

         index = 2222; // Error as const

        console.log(index);

    }

    console.log(index);

}

testConst(1);

**var: var is mutable as well as have global scope**

function testConst(number) {

    var index = 1111;

    if (number < 5) {

     var index = 2222;

        console.log(index); // 2222

    }

    console.log(index); // 2222

}

testConst(1);

Function vs arrow functions

Role of this keyword.

var name = "tanuj";

var box = {

    name: 'surya',

    clickMe: function () {

        var c = function () {

            console.log(name)  // tanuj

            console.log(this.name); // tanuj

        }

        return c;

    }

}

box.clickMe()();

In this the output `name` will be derived from global output. ES5 function does not derive its contexts from surrounding i.e, always from global.

var name = "tanuj";

box = {

    name: 'surya',

    clickMe: function () {

        var c = () => {

            console.log(name)  // tanuj

            console.log(this.name); //surya

        }

        return c;

    }

}

box.clickMe()();

ES6 Arrow fn derive its context from surroundings. In the above this => derives `name` from its surroundings.

var name = "tanuj";

var box = {

    name: 'surya',

    clickMe: () => {

        var c = () => {

            console.log(name)  // tanuj

            console.log(this.name); // tanuj

        }

        return c;

    }

}

box.clickMe()();

Again the inner => derives it surrounding from outer => that inturn derives its surroundings from global.

**Arrays**

<div class="box red">RED</div>

<div class="box green">GREEN</div>

<div class="box blue">BLUE</div>

div.solid.red {

    width: 100px;

    border-style: solid;

    background: red;

}

div.solid.green {

    width: 100px;

    border-style: solid;

    background: green;

}

div.solid.blue {

    width: 100px;

    border-style: solid;

    background: blue;

}



**//ES5**

//Select all the div having .box class

var boxes = document.querySelectorAll(".box");

//Take out arrays from the above selection in Array Object

var boxesArray = Array.prototype.slice.call(boxes);

boxesArray.forEach(function(element) {

    element.style.backgroundColor = 'yellow';

});

**//ES6: Array.from(boxes);**

//Select all the div having .box class

var boxes = document.querySelectorAll(".box");

//Take out arrays from the above selection in Array Object

var boxesArray = Array.from(boxes);

boxesArray.forEach(element => {

    element.style.backgroundColor = 'yellow';

});



NOTE: Problem with foreach loop is that we cannot break from it.

**ES5:**

for (let i = 0; i < boxesArray.length; i++) {

    const element = boxesArray[i];

    if (element.className === 'box red') {

        continue; // break;

    }

    element.textContent = 'Changed colour to pink'

    element.style.backgroundColor = 'red';

}

In ES6 we have forof loop which is a combination of foreach and for loop

**ES6:**

for (const element of boxesArray) {

    if (element.className === 'box red') {

        break;

    }

    element.textContent = 'Changed colour to pink'

    element.style.backgroundColor = 'red';

}

Better version: .includes

for (const element of boxesArray) {

    if (element.className.includes('red')) {

        continue;

    }

    element.textContent = 'Changed colour to pink'

    element.style.backgroundColor = 'red';

}

// ES5

// ES5

var ages = [1, 2, 3, 22, 4, 5];

var gt18 = ages.map(function (age) {

    return age >= 18;

})

console.log(gt18); // [false, false, false, true, false, false]

console.log(gt18.indexOf(true)); // 3

console.log(ages[gt18.indexOf(true)]); //22

//ES6

var index = ages.findIndex(element=> element>=18);

console.log(index) //3

console.log(ages[index]); //22

var agesGt18 = ages.find(age => age > 18) // find the first matched

console.log(agesGt18); //22

**Spread Operator**

function addFourAges(a, b, c, d) {

    return a + b + c + d;

}

var sum = addFourAges(1, 2, 3, 4);

console.log(sum); // 10

// ES5

var ages = [1, 2, 3, 4];

var sum1 = addFourAges.apply(null, ages);

console.log(sum1); // 10

// ES6

const sum2 = addFourAges(...ages);

console.log(sum2); // 10

const familyMemberOne = ['Tanuj', 'Pallawee'];

const familyMemberTwo = ['Meera', 'Lalit'];

const totalFamilyMembers = [**...**familyMemberOne, **...**familyMemberTwo];

console.log(totalFamilyMembers); // ["Tanuj", "Pallawee", "Meera", "Lalit"]

**Rest Parameters**

In ES6, This looks same as spread but functions differently and passed in a parameter.

// ES5

function isFullAge() {

   console.log(arguments); // this is a special variable like this keyword and will have [1990, 1991]

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

    args.forEach(element => {

        console.log(element)

    });}

isFullAge(1990, 1991);

//ES6

function isFullAge(...args) {

    args.forEach(element => {

        console.log(element)

    });}

isFullAge(1990, 1991);

// ES5

function restParametersES5(limit) {

    console.log(arguments);

    // if we don't slice it by 1, 21 will also be considered as value of an array

    var args = Array.prototype.slice.call(arguments, 1);

    args.forEach(element => {

        if (element > limit) {

            console.log(`Age is greater than ${limit}`);

        }

    });}

restParametersES5(21, 19, 22, 24); // limit should be 21 and rest other should be array

//ES6

function restParametersES6(limit, ...args) {

    args.forEach(element => {

        if (element > limit) {

            console.log(`Age is greater than ${limit}`);

        }

    });}

restParametersES6(21, 19, 22, 24);

Default Parameters

// ES5

function defaultParametersES5(name, lastname, city) {

    console.log(name, lastname, city); // Tanuj undefined undefined

    name === undefined ? name = 'Tanuj' : name = name;

    lastname === undefined ? lastname = 'Tripathi' : lastname = lastname;

    city === undefined ? city = 'Bareilly' : city = city;

    console.log(name, lastname, city) // Tanuj Tripathi Bareilly

}

defaultParametersES5('Tanuj')

// ES6

function defaultParametersES6(name='Tanuj', lastname='tripathi', city='Bareilly'){

    console.log(name, lastname, city) ;// Tanuj Tripathi Bareilly

}

defaultParametersES6()

Maps

In simple objects we are restricted to String keys for eg:

var john = {

    name: "john"

}

But in ES6 Maps we have can anthing as a key be it Object, Number, Boolean, String, functions.

const question = new Map();

question.set('question', 'What is latest Javascript version?')

question.set(1, 'ES5');

question.set(2, 'ES6');

question.set(3, 'ES7');

question.set(4, 'ES2015');

question.set('answer', 3)

question.set(true, 'Correct Answer');

question.set(false, 'InCorrect Answer');

console.log(question.get('question')); //What is latest Javascript version?

console.log(question.size); // 8

// delete

if (question.has(4)) {

    question.delete(4);

}

// delete everything

question.clear();

// foreach

question.forEach((value, key) => console.log(`key is ${key} and value is ${value}`))

// question.entries() returns array of key, value pair

console.log(question.entries())

// forof loop

for (const [key, value] of question.entries()) {

    if (typeof (key) === 'number') {

        console.log(`Answer is key ${key} , value ${value}`)

    }

}

var answer = parseInt(prompt('What is your answer?'));

var isCorrect = question.get(answer === question.get('answer'));

console.log(isCorrect); // Correct Answer or InCorrect Answer

**Classes**

Classes makes inheritance easy in JS.

// ES5

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

    this.name = name;

    this.yearOfBirth = yearOfBirth;

    this.job = job;

}

Persons.prototype.calculateAge = function () {

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

    console.log(age);

}

var john = new Persons('john', '1993', 'Software Engineer');

john.calculateAge();

Classes: Under the hood they are converted into above written code, Some say its just for more readbility.

//ES6

class Persons {

    constructor(name, yearOfBirth, job) {

        this.name = name;

        this.yearOfBirth = yearOfBirth;

        this.job = job;

    }

    calculateAge() {

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

        console.log(age);

    }

}

const persons = new Persons('john', '1993', 'Software Engineer');

persons.calculateAge();

static : static are directly attached to the class rather than the instance of a class

//ES6

class Persons {

    static hello() {

        console.log("Hello!");

    }

}

Persons.hello();

Inheritance

//ES5

var Person = function (name, age) {

    this.name = name;

    this.age = age;

}

Person.prototype.calculateAge = function () {

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

    console.log(age);

}

var Athlete = function (name, age, height, weight) {

    Person.call(this, name, age);

    this.height = height;

    this.weight = weight;

}

// Athlete is inheriting from Person

Athlete.prototype = Object.create(Person.prototype);

Athlete.prototype.medal = function () {

    console.log("Won Medal!")

}

var john = new Athlete('john', 1993, 'swimmer', 75);

john.calculateAge();

john.medal();

//ES6

With ES6 its much simpler and Object oriented just and syntax similar to java

class Person {

    constructor(name, age) {

        this.name = name;

        this.age = age;

    }

    calculateAge() {

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

        console.log(age);

    }

}

class Athlete extends Person {

    constructor(name, age, height, weight) {

        super(name, age);

        this.height = height;

        this.weight = weight;

    }

    medal() {

        console.log("Won Medal!");

    }

}

var athlete = new Athlete('john', 1993, 'swimmer', 75);

athlete.calculateAge();

athlete.medal();

**Asynchronous Javascript**

It is a fancy term in which some code runs in a background and remaining code still executes. Typically, we use this when we have to fetch some data from some APIS.

const second = () => {

    console.log('In Second');

    setTimeout(() => {

        console.log('Async call')

    }, 5000)

}

const first = () => {

    console.log('In First')

    second();

    console.log('The end')

}

first();

In First

In Second

The end

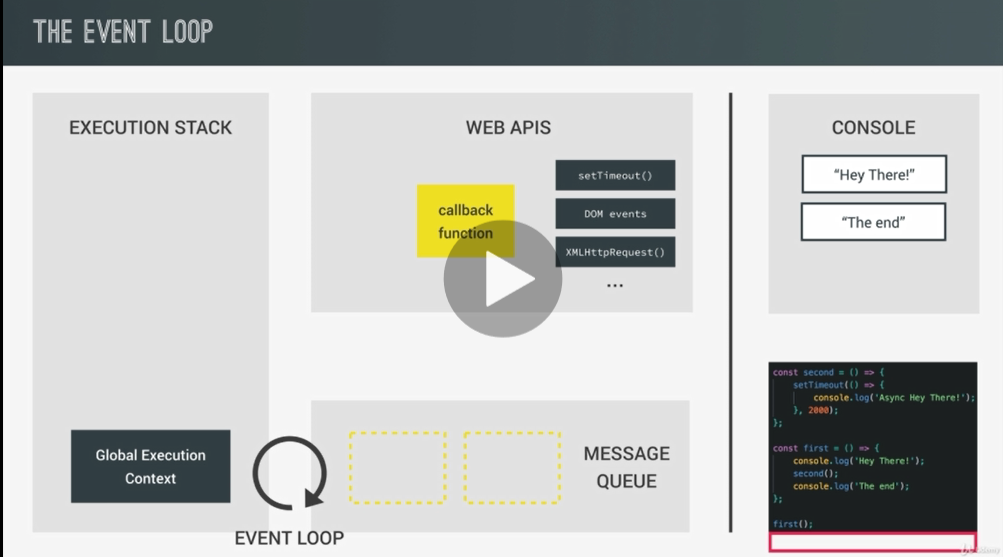
Async call

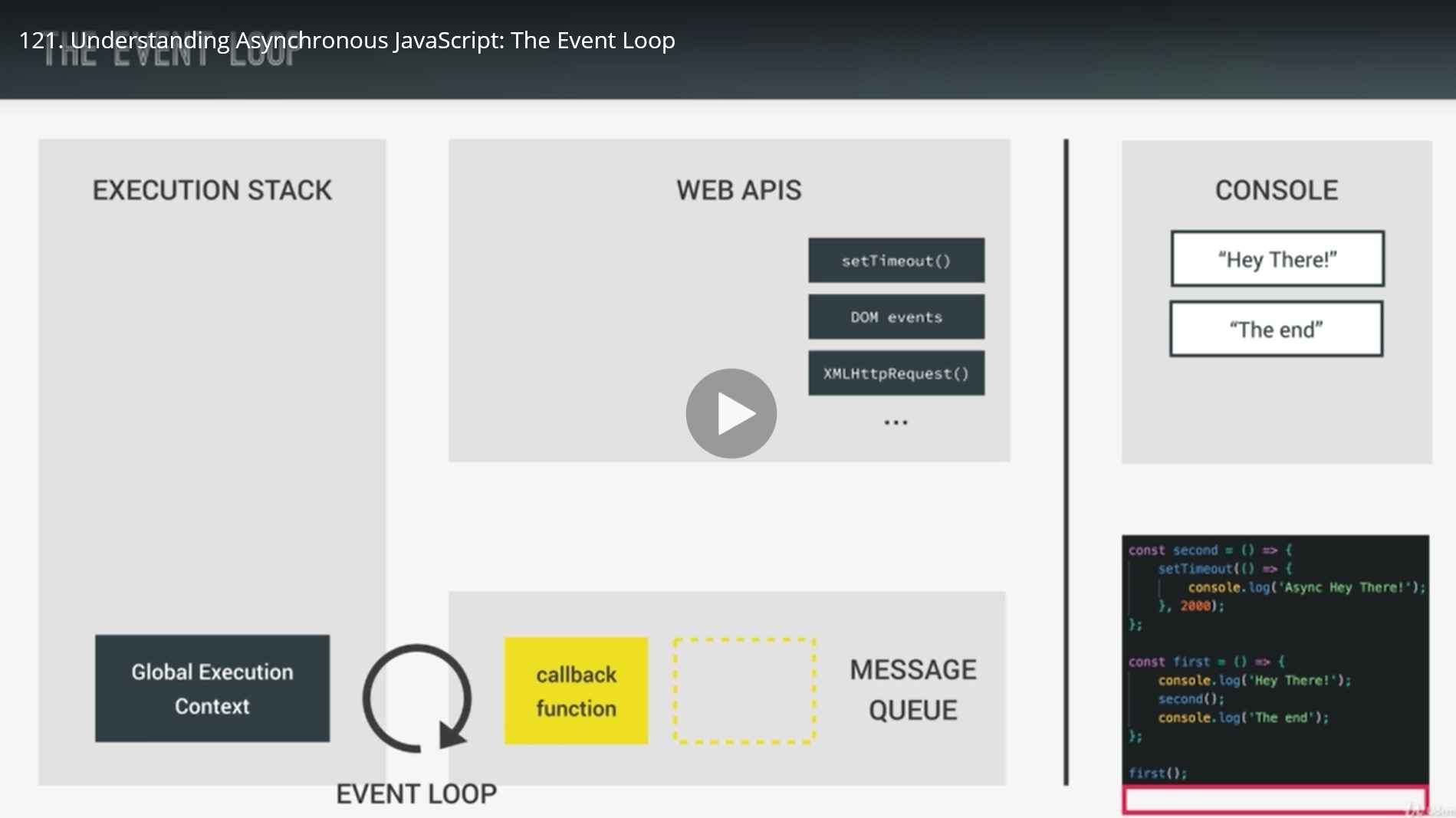
Most of the code we have processed was synchronous code in which JS Engine executes the code line by line in a single thread.

We have WEB APIS that sits outside the JS Engine. We can use its functions because they are also in JS runtime. Eg:

1. setTimeOut()
2. DOM events
3. XMLHttpRequest()

**Ques: How Async actually works under the hood???**





Async works with the combination of Exection stack, WEB APIS, Message Queue.

When we use async fn like setTimeOut() for 5 seconds, it sits on WEB API as a callback function thus not halting the current JS code to get executed. When 5 seconds completes, the callback funtion is now moved to the Message Queue where Event loop monitors it and pushes back it to the Execution stack as soon as the Execution stack is empty.

const second = () => {

    console.log('In Second');

    setTimeout(() => {

        console.log('Async call')

    }, 5000)

}

const first = () => {

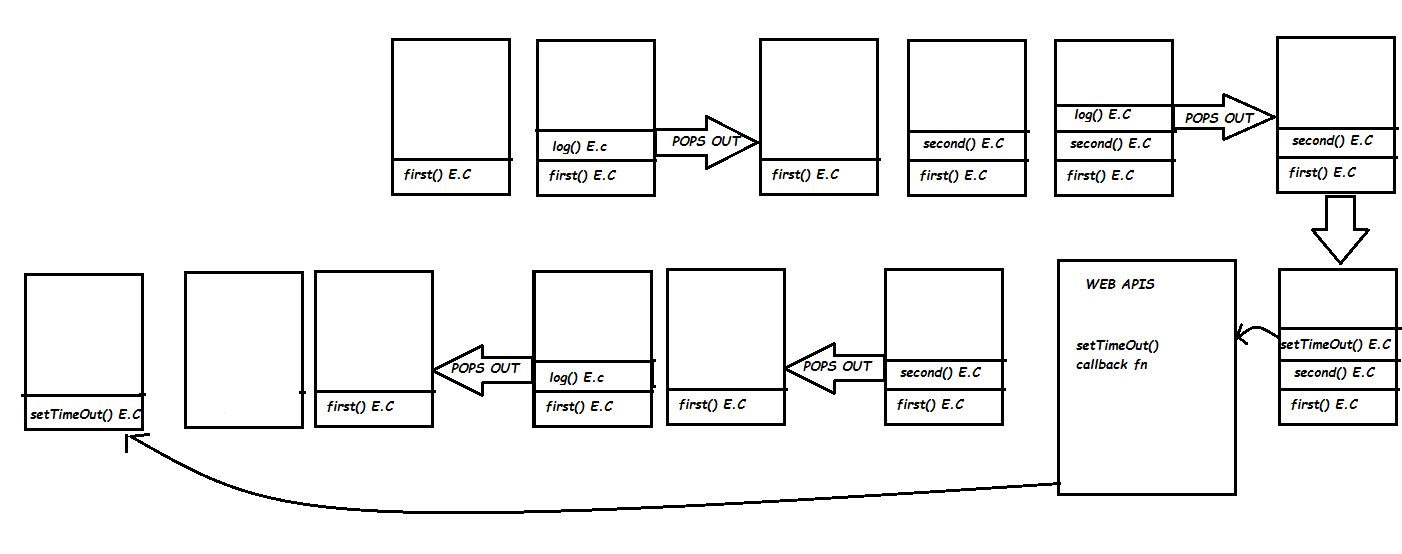
    console.log('In First')

    second();

    console.log('The end')

}

first();



1. first() Execution Context is created on the top of Global Execution Context
2. log() Execution Context is created on the top of first() E.C
3. log() E.C pops out.
4. second() E.C is created on the top of first() E.C
5. log(‘In Second’) E.C is created on the top of second() E.C
6. log() E.C pops out
7. setTimeOut() E.C is created on the top of second() E.C and moved to WEB APIS where it waits as a callback function.
8. second() E.C pops out
9. log(‘The end’) E.C is created on the top of first() E.C
10. log() pops out
11. first() pops out, leaving E.C empty
12. Now if callback method is already finished is move to message queue and Event loop puts the callback fn (setTimeOut()) in E.C and is executed.
13. setTimeOut() pops out.

Normally in ES5

function callbackHell() {

    setTimeout(() => {

        console.log('First timeout');

        setTimeout(() => {

            console.log('Second timeout');

            setTimeout(() => {

                console.log('Third timeout');

            }, 2000);

        }, 2000);

    }, 2000);

}

callbackHell();

It is caled as callback hell as the above is very hard to manage, if callbacks are increased.

To tackle this in ES6, Promise came. Where is becomes very easy to manage callbacks in much better readable form.

**Promise**

Promise basically works on an idea that while processing JS code it promises to get data while running in the background and we can deal with that data in the future.

Promise can be in different states:

1. Pending
2. Settled/Resolved: Fulfilled or Rejected

Practically we can produce and consume Promises. We produce promises i.e, we send our promise to collect the data and then with the help of the callback function we can consume it(for fullfillment or rejections).

const executorFunction = function (resolve, reject) {

    // FAKE ajax call

    setTimeout(function () {

        // When this call is sucessfull it returns this data(array)

        resolve([100, 101, 102, 103]);

    }, 2000);

}

// Promise take an executorFunction which takes two params resolve and reject

const getIDs = new Promise(executorFunction);

getIDs.then(function (responseData) {

    console.log(responseData); // [100, 101, 102, 103]

    // forof loop

    for (const element of responseData) {

        console.log(element);

    }

})

A promise takes executor function that take two arguments resolve and reject.

**ES6**

new Promise((resolve, reject) => {

     resolve([100, 101, 102, 103]);

})

.then(responseData => {

    console.log(responseData); // [100, 101, 102, 103]

    // forof loop

    for (const element of responseData) {

        console.log(element);

    }

})

new Promise((resolve, reject) => {

**reject([100, 101, 102, 103]);**

})

.then(responseData => {

    console.log("SUCCESS");

}).

catch(error => {

    console.log('ERROR OCCURED', error);

})

**Promise chaining with then()**

/\*\*

 \* getIds is a promise that gets some list of IDs from backend(Assume)

 \* Suppose to get data from backend it takes 2 seconds

 \*/

const getIDs = new Promise((resolve, reject) => {

    setTimeout(() => {

        console.log('Getting IDs from backend.....')

        resolve([100, 101, 102, 103]);

    }, 2000)

});

console.log("Hello Buddy!!, You will see the Recipe shortly");

const getRecipe = chefID => {

    return new Promise((resolve, reject) => {

        // ID is chefID

        setTimeout(ID => {

            const recipe = {

                title: 'Pizza',

                chef: 'tanuj'

            }

resolve(`Chef's ID ${ID} Recipe is ${recipe.title} and chef is ${recipe.chef}`)

// We can pass parameter value in setTimeOut() like this

        }, 3000, chefID)

    })

}

getIDs

    .then(response => {

        console.log(`IDs of all chefs are ${response}`);

        getRecipe(response[2])

**.then(recipe => {**

            console.log(`Recipe is::: ${recipe}`)

        })

    })

    .catch(error => {

        console.log(`Error ::: ${error}`)

    })

Instead of above we can use Promise chaining. Instead of using calling method under then that return a Promise we can instead return the method and use then outside. This is called Promise chaining.

/\*\*

 \* we can do chaining if we return Promise in one then

 \*/

getIDs

    .then(response => {

        console.log(`IDs of all chefs are ${response}`);

        return getRecipe(response[2])

    })

**.then(recipe => {**

        console.log(`Recipe is::: ${recipe}`)

    })

    .catch(error => {

        console.log(`Error ::: ${error}`)

    })