Synchronous & Asynchronous In javascript:

Synchronous javascript as the name suggests synchronous means to be in a sequence that every statements of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed.

```js
const prompt = require("prompt-sync")();

console.log("One Print Statement");
console.log("Two Print Statement");
console.log("Three Print Statement");
```

```
PS E:\HTML_CSS_JS> node demo.js
One Print Statement
Two Print Statement
Three Print Statement
PS E:\HTML_CSS_JS>
```

Asynchronous code allows the program to be executed immediately where the synchronous code will block further execution of the remaining code until it finishes the current one. This may not like a big problem but when you see it in a bigger picture you realize that it may lead to delaying the user interface.

```js
const prompt = require("prompt-sync")();

console.log("One Print Statement");
setTimeout(() => {
    console.log("Two Print Statement");
}, 2000);
console.log("Three Print Statement");
```

```
PS E:\HTML_CSS_JS> node demo.js
One Print Statement
Three Print Statement
Two Print Statement
PS E:\HTML_CSS_JS>
```

Asynchronous javascript(callback function):

```js
const prompt = require("prompt-sync")();

function func1(){
    console.log("func1 function");
}
function func2(){
    console.log("func2 function");
}

function func3(num1, num2, callback){
    console.log(num1 * num2);
    callback();
}
func3(10, 20, func1);
func3(50, 60, func2);
func3(60, 80, function(){
    console.log("func3 function");
})
```

```
PS E:\HTML_CSS_JS> node demo.js
200
func1 function
3000
func2 function
4800
func3 function
PS E:\HTML_CSS_JS>
```

Asynchronous javascript(settiemout):

```
const prompt = require("prompt-sync")();

function func0(){
    console.log("func0 function");
}
setTimeout(func1, 2000);
function func1(){
    console.log("function will executed after 2000 millisecond");
}

func0();
```

```
PS E:\HTML_CSS_JS> node demo.js
func0 function
function will executed after 2000 millisecond
PS E:\HTML_CSS_JS>
```

Asynchronous javascript(setinterval):

```
const prompt = require("prompt-sync")();

setInterval(func1, 2000);
function func1(){
    console.log("func1 function");
}
```

```
PS E:\HTML_CSS_JS> node demo.js
func1 function
func1 function
func1 function
```

Promises In javascript:

A javascript Promise object contains both the producing code and calls to the consuming code. When the producing code obtains the result, it should call one of the two callbacks. For Success myresolve(result value), For Error myreject(error object). A javascript promise object can be “pending”, “fulfilled”, “rejected”. The Promise object supports two properties: state and result.

While a Promise object is "pending" (working), the result is undefined. When a Promise object is "fulfilled", the result is a value. When a Promise object is "rejected", the result is an error object.

```
const prompt = require("prompt-sync")();
function func1(some1){
    console.log("value : ",some1,"\n");
}
let promise = new Promise(function(myResolve, myReject){
    let x = parseInt(prompt("Enter Value : "));

    if(x >= 20){
        myResolve("Ok");
    }else{
        myReject("Error");
    }
});
promise.then(
    function(value){func1(value);},
    function(error){func1(error);}
);
```

```
PS E:\HTML_CSS_JS> node demo.js
Enter Value : 15
value :  Error

PS E:\HTML_CSS_JS> node demo.js
Enter Value : 32
value :  Ok

PS E:\HTML_CSS_JS>
```

```
demo.html   demo.js
demo.js > ...
const prompt = require("prompt-sync")();

//setTimeout and Promises
let promise = new Promise(function(myResolve, myReject){
    setTimeout(function(){
        myResolve("JavaScript Program");
    }, 2000);
});
promise.then(function(value){
    console.log("value : ",value,"\n");
});
```

```
PS E:\HTML_CSS_JS> node demo.js
value :  JavaScript Program

PS E:\HTML_CSS_JS>
```

.then() & .catch() in javascript:

The then() and catch() methods are used to handle promises in javascript. Promises are a way of representing an asynchronous operation, such as a network request or a database lookup. When a promise is created, it is in a "pending" state. It will eventually resolve to a value or reject with an error.

The then() method takes a function as an argument. This function will be called when the promise resolves. The function will receive the resolved value as an argument.

The catch() method takes a function as an argument. This function will be called when the promise rejects. The function will receive the rejection reason as an argument.

If the network request fails, the reject() function is called with the rejection reason. The catch() function is then called with the rejection reason as an argument. The catch() function logs the reason for the failure to the console.

Then() and catch() are powerful tools for handling asynchronous operations in javascript. By using these methods, you can ensure that your code runs smoothly even if there are errors.

```
demo.html   demo.js
demo.js > ...
const prompt = require("prompt-sync")();

//.then() .catch() in promise
function func1(value){
    console.log(value,"\n");
}
let promise = new Promise((resolve, reject)=>{
    let num = parseInt(prompt("Enter Value : "));
    if(num >= 50){
        resolve("Num Is Greater Than 50");
    }else{
        reject("Num Is Smaller Than 50");
    }
});
promise.then(value =>{
    func1(value);
}).catch(error =>{
    func1(error);
});
```

```
PS E:\HTML_CSS_JS> node demo.js
Enter Value : 55
Num Is Greater Than 50

PS E:\HTML_CSS_JS> node demo.js
Enter Value : 11
Num Is Smaller Than 50

PS E:\HTML_CSS_JS>
```

demo.html | demo.js

demo.js > catch() callback

```javascript
const prompt = require("prompt-sync")();

let promise = new Promise(function (resolve, reject){
    let x = parseInt(prompt("Enter Value : "));
    if(x >= 100){
        resolve();
    }else{
        reject();
    }
});
promise.
        then(function(){
            console.log("Greater Than 100\n");
        }).
        catch(function(){
            console.log("Smaller Than 100\n");
        });
```

```
PS E:\HTML_CSS_JS> node demo.js
Enter Value : 200
Greater Than 100

PS E:\HTML_CSS_JS> node demo.js
Enter Value : 80
Smaller Than 100

PS E:\HTML_CSS_JS>
```

demo.js

demo.js > ...

```javascript
const prompt = require("prompt-sync")();

let promise = new Promise(function (resolve, reject){
    resolve("This Is The Resolve Promise");
});
promise.
    then(function(success){
        console.log(success);
    }, function(error){
        console.log(error);
    });
```

```
PS E:\HTML_CSS_JS> node demo.js
This Is The Resolve Promise
PS E:\HTML_CSS_JS>
```

demo.js

demo.js > promise

```javascript
const prompt = require("prompt-sync")();

let promise = new Promise(function (resolve, reject){
    reject("This Is The Reject Promise");
});
promise.
    then(function(success){
        console.log(success);
    }, function(error){
        console.log(error);
    });
```

```
PS E:\HTML_CSS_JS> node demo.js
This Is The Reject Promise
PS E:\HTML_CSS_JS>
```

Promise Chaining:

demo.js

demo.js > promise.then() callback

```javascript
const prompt = require("prompt-sync")();

let promise = new Promise((resolve, reject)=>{
    resolve("Hello JavaScript");
});
promise
    .then(function(result1){
        console.log(result1);
        return new Promise((resolve, reject)=>{
            resolve("JavaScript Is Awsome");
            //reject("Rejected Message");
        })
    })
    .then((result2)=>{
        console.log(result2);
    })
    .catch((result3)=>{
        console.log(result3);
    })
```

```
PS E:\HTML_CSS_JS> node demo.js
Hello JavaScript
JavaScript Is Awsome
PS E:\HTML_CSS_JS>
```

demo.js

demo.js > promise.then() callback

```javascript
const prompt = require("prompt-sync")();

let promise = new Promise((resolve, reject)=>{
    resolve("Hello JavaScript");
});
promise
    .then(function(result1){
        console.log(result1);
        return new Promise((resolve, reject)=>{
        //    resolve("JavaScript Is Awsome");
            reject("Rejected Message");
        })
    })
    .then((result2)=>{
        console.log(result2);
    })
    .catch((result3)=>{
        console.log(result3);
    })
```

```
PS E:\HTML_CSS_JS> node demo.js
Hello JavaScript
Rejected Message
PS E:\HTML_CSS_JS>
```

demo.js

demo.js > promise

```javascript
const prompt = require("prompt-sync")();

let promise = new Promise((resolve, reject)=>{
    // resolve("Hello JavaScript");
    reject("The Main Rejected Message");
});
promise
    .then(function(result1){
        console.log(result1);
        return new Promise((resolve, reject)=>{
        //    resolve("JavaScript Is Awsome");
            reject("Rejected Message");
        })
    })
    .then((result2)=>{
        console.log(result2);
    })
    .catch((result3)=>{
        console.log(result3);
    })
```

```
PS E:\HTML_CSS_JS> node demo.js
The Main Rejected Message
PS E:\HTML_CSS_JS>
```

Error Handling & Exception:

An exception signifies the presence of an abnormal condition which requires special operable techniques. In programming terms, an exception is the anomalous code that breaks the normal flow of the code. Such exceptions require specialized programming constructs for its execution.

In programming, exception handling is a process or method used for handling the abnormal statements in the code and executing them. It also enables to handle the flow control of the code/program. For handling the code, various handlers are used that process the exception and execute the code. For example, the Division of a non-zero value with zero will result into infinity always, and it is an exception. Thus, with the help of exception handling, it can be executed and handled.

A throw statement is used to raise an exception. It means when an abnormal condition occurs, an exception is thrown using throw. The thrown exception is handled by wrapping the code into the try…catch block. If an error is present, the catch block will execute, else only the try block statements will get executed. Thus, in a programming language, there can be different types of errors which may disturb the proper execution of the program.

Types Of Error:

1. Synatx Error:

When a user makes a mistake in the pre-defined syntax of a programming language, a syntax error may appear.

2. Runtime Error:

When an error occurs during the execution of the program, such an error is known as Runtime error. The codes which create runtime errors are known as Exceptions. Thus, exception handlers are used for handling runtime errors.

3. Logical Error:

An error which occurs when there is any logical mistake in the program that may not produce the desired output, and may terminate abnormally. Such an error is known as Logical error.

Error Object:

When a runtime error occurs, it creates and throws an Error object. Such an object can be used as a base for the user-defined exceptions too. An error object has two properties: Name & Message

Name is an object property that sets or returns an error name. Message is the property returns an error message is the String from.

Although Error is a generic constructor, there are following standard built-in error types or error constructors beside it

```js
const prompt = require("prompt-sync")();

try{
    addalert("Welcome To Try Section");
}catch(error){
    console.log(error.message);
}
```

```
PS E:\HTML_CSS_JS> node demo.js
addalert is not defined
PS E:\HTML_CSS_JS>
```

```js
const prompt = require("prompt-sync")();

let val1 = prompt("Enter Value : ");
try{
    if(val1.trim() == "") throw "Empty String";
    if(isNaN(val1)) throw "Not A Number";
    if(val1 > 10) throw "Higher Than 10";
    if(val1 < 10) throw "Lower Than 10";
}
catch(error){
    console.log(error);
}
```

```
PS E:\HTML_CSS_JS> node demo.js
Enter Value : ABCD
Not A Number
PS E:\HTML_CSS_JS> node demo.js
Enter Value : 20
Higher Than 10
PS E:\HTML_CSS_JS> node demo.js
Enter Value : 8
Lower Than 10
PS E:\HTML_CSS_JS> node demo.js
Enter Value :
Empty String
PS E:\HTML_CSS_JS>
```

```js
const prompt = require("prompt-sync")();

let val1 = prompt("Enter Value : ");
try{
    if(val1.trim() == "") throw "Empty String";
    if(isNaN(val1)) throw "Not A Number";
    if(val1 > 20) throw "Higher Than 20";
    if(val1 < 10) throw "Lower Than 10";
}
catch(error){
    console.log(error);
}
finally{
    console.log("value : ",val1);
}
```

```
PS E:\HTML_CSS_JS> node demo.js
Enter Value :
Empty String
value :
PS E:\HTML_CSS_JS> node demo.js
Enter Value : Atish
Not A Number
value : Atish
PS E:\HTML_CSS_JS> node demo.js
Enter Value : 22
Higher Than 20
value : 22
PS E:\HTML_CSS_JS> node demo.js
Enter Value : 18
value : 18
PS E:\HTML_CSS_JS> node demo.js
Enter Value : 8
Lower Than 10
value : 8
PS E:\HTML_CSS_JS>
```

Evalerror:

It creates an instance for the error that occurred in the eval(), which is a global function used for evaluating the js string code.

Internalerror:

It creates an instance when the js engine throws an internal error.

Rangeerror:

It creates an instance for the error that occurs when a numeric variable or parameter is out of its valid range. A rangeerror is thrown if you use a number that is outside the range of legal values.

```
demo.js
const prompt = require("prompt-sync")();

let val1 = parseInt(prompt("Enter Value : "));
try{
    val1.toPrecision(500);
}
catch(error){
    console.log(error.name);
}
```

```
PS E:\HTML_CSS_JS> node demo.js
Enter Value : 200
RangeError
PS E:\HTML_CSS_JS> node demo.js
Enter Value : 500
RangeError
PS E:\HTML_CSS_JS> node demo.js
Enter Value : 10
RangeError
PS E:\HTML_CSS_JS>
```

Referenceerror:

It creates an instance for the error that occurs when an invalid reference is de-referenced.

```
demo.js > ...
const prompt = require("promp

let val1 = 5;
try{
    val1 = val2 + 10;
}
catch(error){
    console.log(error.name);
}
```

```
PS E:\HTML_CSS_JS> node demo.js
ReferenceError
PS E:\HTML_CSS_JS>
```

Syntaxerror:

An instance is created for the syntax error that may occur while parsing the eval().

```
demo.js > ...
const prompt = require("promp

try{
    eval("alert('hello)");
}
catch(error){
    console.log(error.name);
}
```

```
PS E:\HTML_CSS_JS> node demo.js
SyntaxError
PS E:\HTML_CSS_JS>
```

Typeerror:

When a variable is not a valid type, an instance is created for such an error.

```
JS demo.js > ...
const prompt = require("promp
let num = 10;
try{
num.toUpperCase();
}
catch(error){
    console.log(error.name);
}
```

```
PS E:\HTML_CSS_JS> node demo.js
TypeError
PS E:\HTML_CSS_JS>
```

URL Error:

An instance is created for the error that occurs when invalid parameters are passed in encodeuri() or decodeuri().

```
JS demo.js > ...
const prompt = require("promp
let num = 10;
try{
decodeURI("%%%");
}
catch(error){
    console.log(error.name);
}
```

```
PS E:\HTML_CSS_JS> node demo.js
TypeError
PS E:\HTML_CSS_JS>
```

Await keyword:

```
JS demo.js
JS demo.js > func1
const prompt = require("prompt-sync")();

function display(some){
    console.log("Value : ",some);
}
async function func1(){
    return "Atish";
}
func1().then(
    function(value) {display(value);},
    function(error) {display(error);}
);
```

```
PS E:\HTML_CSS_JS> node demo.js
Value :  Atish
PS E:\HTML_CSS_JS>
```

demo.js

demo.js > func1

```js
const prompt = require("prompt-sync")();

function display(some){
    console.log("Value : ",some);
}
async function func1(){
    return "Atish";
}
func1().then(
    function(value) {display(value);}
);
```

```
PS E:\HTML_CSS_JS> node demo.js
Value :  Atish
PS E:\HTML_CSS_JS>
```

demo.js

demo.js > display1 > promise1

```js
const prompt = require("prompt-sync")();

async function display(){
    let promise = new Promise(function(resolve, reject){
        resolve("I Love JavaScript");
    });
    console.log(await promise);
}
display();

async function display1(){
    let promise1 = new Promise(function(resolve){
        resolve("I Love Java");
    });
    console.log(await promise1);
}
display1();
```

```
PS E:\HTML_CSS_JS> node demo.js
I Love JavaScript
I Love Java
PS E:\HTML_CSS_JS>
```

demo.js

demo.js > dispaly

```js
const prompt = require("prompt-sync")();

async function dispaly(){
    let promise = new Promise(function(resolve){
        setTimeout(function(){
            resolve("I Love JavaScript");
        },3000);
    });
    console.log(await promise);
}
dispaly();
```

```
PS E:\HTML_CSS_JS> node demo.js
I Love JavaScript
PS E:\HTML_CSS_JS>
```

Event loop:

Javascript is a single threaded non blocking asynchronous concurrent language. It means that the main thread where javascript code is run, runs in one line at a time manner and there is no possibility of running code in parallel.

It has a call stack, an event loop and a callback queue, along with that other apis. V8 is the javascript runtime which has a call stack and a heap. The heap is used for memory allocation and the stack holds the execution context. DOM, settimeout, XML, httprequest doesn't exist in V8 source code.

```
JS demo.js
JS demo.js > ...
console.log("Before Delay");
function delaybyseconds(sec){
  let start = now = Date.now();
  while(now - start < (sec * 1000)){
    now = Date.now();
  }
}
delaybyseconds(5);
console.log("After Delay");
```

```
PS E:\HTML_CSS_JS> node demo.js
Before Delay
After Delay
PS E:\HTML_CSS_JS>
```

Heap Memory:

In this memory the data stored randomly and memory allocated.

Stack Memory:

This memory allocated in the form of stacks. Mainly used for functions.

Function Call Stack:

The function stack is a function that keeps track of all other functions executed in run time. Ever seen a stack trace being printed when you ran into an error in javascript? That is nothing but a snapshot of the function stack at that point when the error occurred.

```
JS demo.js
JS demo.js > ...
function leveltwo(){
  console.log("Inside Level Two");
}
function levelone(){
  console.log("Inside Level One");
  leveltwo();
}
function main(){
  console.log("Inside Main Function");
  levelone();
}
main();
```

```
PS E:\HTML_CSS_JS> node demo.js
Inside Main Function
Inside Level One
Inside Level Two
PS E:\HTML_CSS_JS>
```

Asynchronous Callbacks:

Sometimes the javascript code can take a lot of time and this can block the page re render. Javascript has asynchronous callbacks for non blocking behaviour. Javascript runtime can do only one thing at a time. Browser gives us t=other things which work along with the runtime like web apis. In node js these are available as C++ apis.

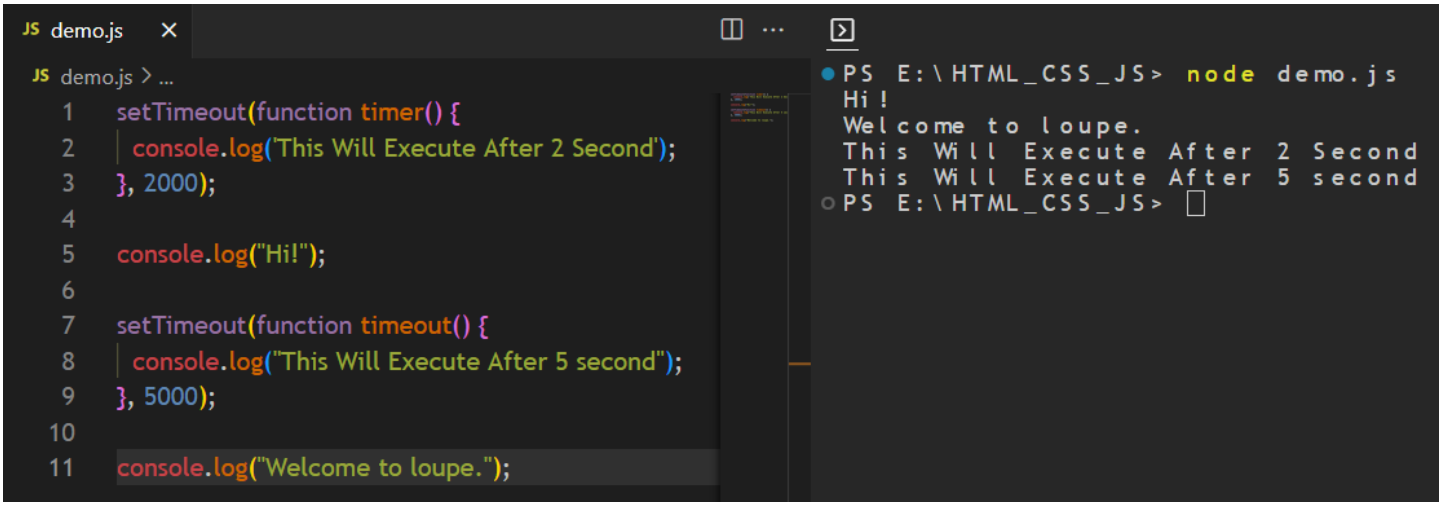
Task Queue:

Javascript can do only one thing at a time. The rest are queued to the task queue waiting to be executed. When we run settimeout, webapis will run a timer and push the function provided to settimeout to the task queue once the timer ends. These tasks will be pushed to the stack where they can executed.

Event Loop:

Javascript has a runtime model based on an event loop, which is responsible for executing the code collecting and processing events and executing qeued sub-tasks. The event loop pushed the task from the task queue to the call stack. Settimeout can be used to defer a function until all the pending tasks have been executed. We can see how these things work in action by visiting.

An event loop is something that pulls stuff out of the queue and places it onto the function execution stack whenever the function stack becomes empty. The event loop is the secret by which javascript gives us an illusion of being multithreaded even though it is single-threaded.



Synchronous Execution:

In a synchronous world, you would take one order, wait for it to be prepared, and then move on to the next order. This means the restaurant would serve one customer at a time, which is not efficient. In javascript terms, this would be like running code in a single thread without asynchronous operations.

```javascript
function takeOrder(order){
  console.log("Taking Order : "+order);
  preparedFood(order);
  console.log("Order served : "+order);
}
function preparedFood(order){
  console.log("preparing food for order : "+order);
}
takeOrder("User1");
takeOrder("User2");
```

```
PS E:\HTML_CSS_JS> node demo.js
Taking Order : User1
preparing food for order : User1
Order served : User1
Taking Order : User2
preparing food for order : User2
Order served : User2
PS E:\HTML_CSS_JS>
```

Asynchronous Execution:

```javascript
function takeOrder(order){
  console.log("Taking Order : ",order);
  setTimeout(function(){
    prepareFood(order);
  }, 1000);
}
function prepareFood(order){
  console.log("Preparing Food For Order : ",order);
  console.log("Order Served : ",order);
}
takeOrder("User 1");
takeOrder("User 2");
```

```
PS E:\HTML_CSS_JS> node demo.js
Taking Order :  User 1
Taking Order :  User 2
Preparing Food For Order :  User 1
Order Served :  User 1
Preparing Food For Order :  User 2
Order Served :  User 2
PS E:\HTML_CSS_JS>
```

Call stack in javascript:

The call stack is a fundamental component of javascript's runtime environment, and it's crucial for understanding how javascript manages function calls and maintains the execution context. The call stack is a data structure that keeps track of function calls in a program. When a function is called, a new frame is pushed onto the stack, and when a function returns, its frame is popped off the stack. This stack-like behavior ensures that javascript can manage the flow of function calls and execute code in a structured manner.

```javascript
function foo(){
  console.log("Inside Foo");
}
function bar(){
  console.log("Inside Bar");
  foo();
}
bar();
```

```
PS E:\HTML_CSS_JS> node demo.js
Inside Bar
Inside Foo
PS E:\HTML_CSS_JS>
```

Recursion:

Recursive functions also use the call stack. Each recursive call creates a new execution context, and when the base case is reached, the calls are popped off the stack in reverse order.

```
JS demo.js > ...
function countdown(n){
  if(n <= 0){
    return;
  }
  console.log(n);
  countdown(n - 1);
}
countdown(10);
```

```
PS E:\HTML_CSS_JS> node demo.js
10
9
8
7
6
5
4
3
2
1
PS E:\HTML_CSS_JS>
```

Closure In javascript:

In javascript, a closure is a fundamental and powerful concept. It occurs when a function "closes over" its surrounding lexical scope, retaining access to the variables, parameters, and functions within that scope even after the outer function has finished executing. Closures are essential for maintaining data encapsulation, creating private variables, and enabling more flexible and modular code.

```
JS demo.js > ...
function func(){
  let outer = "Outer Function";
  function func1(){
    console.log(outer);
  }
  return func1;
}
let op = func();
op();
```

```
PS E:\HTML_CSS_JS> node demo.js
Outer Function
PS E:\HTML_CSS_JS>
```

```
JS demo.js
JS demo.js > ...
function func(msg){
  return function(name){
    console.log(`${msg},${name}`);
  };
}
let hello = func('Hello');
let bye = func("Good Bye");
hello('Atish');
bye('Lipun');
```

```
PS E:\HTML_CSS_JS> node demo.js
Hello,Atish
Good Bye,Lipun
PS E:\HTML_CSS_JS>
```

```javascript
function createCounter() {
  let count = 0;

  return {
    increment: function () {
      count++;
    },
    decrement: function () {
      count--;
    },
    getCount: function () {
      return count;
    },
  };
}

const counter = createCounter();
counter.increment();
counter.increment();
console.log(counter.getCount()); // Outputs: 2
```

```
PS E:\HTML_CSS_JS> node demo.js
2
PS E:\HTML_CSS_JS>
```

Higher Order Function In javascript:

In javascript, a higher-order function is a function that either takes one or more functions as arguments (callbacks) or returns a function as its result. Higher-order functions are a powerful and essential concept in functional programming and are commonly used in modern javascript development. They allow for more modular and reusable code by promoting the separation of concerns and enabling functions to be treated as first-class citizens.

Map:

```javascript
const numbers = [1, 2, 3, 4, 5];
const squaredNumbers = numbers.map(function (num) {
  return num * num;
});
console.log(squaredNumbers);
```

```
PS E:\HTML_CSS_JS> node demo.js
[ 1, 4, 9, 16, 25 ]
PS E:\HTML_CSS_JS>
```

Filter:

```javascript
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(function (num) {
  return num % 2 === 0;
});
console.log(evenNumbers);
```

```
PS E:\HTML_CSS_JS> node demo.js
[ 2, 4 ]
PS E:\HTML_CSS_JS>
```

Reduce:

```js
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce(function (accumulator, currentValue) {
  return accumulator + currentValue;
}, 0);
console.log(sum);
```

```
PS E:\HTML_CSS_JS> node demo.js
15
PS E:\HTML_CSS_JS>
```

Foreach():

```js
const colors = ["red", "green", "blue"];
colors.forEach(function (color) {
  console.log(color);
});
```

```
PS E:\HTML_CSS_JS> node demo.js
red
green
blue
PS E:\HTML_CSS_JS>
```

Function Composition:

```js
function add(x) {
  return function (y) {
    return x + y;
  };
}
const add5 = add(5);
console.log(add5(3)); // 8
```

```
PS E:\HTML_CSS_JS> node demo.js
8
PS E:\HTML_CSS_JS>
```

This Keyword in JS:

The this keyword is a reference variable that refers to the current object.

```
JS demo.js > ...
// const prompt = require('prompt-sync')();
let obj= {
    name: "Atish",
    city: "BBSR",
    satte: "Odisha",
    fullAddress: function(){
        return this.name+"--"+this.city+"--"+this.satte;
    }
}
let fetch = obj.fullAddress();
console.log(fetch);
```

```
powershell ×
PS E:\HTML_CSS_JS> node demo.js
Atish--BBSR--Odisha
PS E:\HTML_CSS_JS>
```

Global Context:

In global context variables are declared outside the function. Here this keyword refers to the window object.

```
JS demo.js > web
// const prompt = require('prompt-sync')();
let website = "MAKS";
function web(){
    console.log(website);
}
web();
```

```
powershell ×
PS E:\HTML_CSS_JS> node demo.js
MAKS
PS E:\HTML_CSS_JS>
```

Call() & apply() method:

The call() and apply() method allows us to write a method that can be used on different objects.

```
JS demo.js > ...
// const prompt = require('prompt-sync')();
let emp = {
    address: function(){
        return this.name+"--"+this.city+"--"+this.state;
    }
}
let add = {
    name: "Atish",
    city: "BAM",
    state: "Odisha"
}
console.log(emp.address.call(add));
console.log(emp.address.apply(add));
```

```
powershell ×
PS E:\HTML_CSS_JS> node demo.js
Atish--BAM--Odisha
Atish--BAM--Odisha
PS E:\HTML_CSS_JS>
```

JavaScript Hoisting:

Hoisting is a mechanism in javascript that moves the declaration of variables and functions at the top so in javascript we can use variables and functions before declaring them.

```
JS demo.js > ...
// const prompt = require('prompt-sync')();
x = 10;
console.log(x);
var x;
console.log(sum(10,20));
function sum(a,b){
    return a +b;
}
```

```
powershell ×
PS E:\HTML_CSS_JS> node demo.js
10
30
PS E:\HTML_CSS_JS>
```

JavaScript Strict Mode:

Being a scripting language sometimes the js code displays the correct result even it has some errors to overcome this problem we can use the js strict mode. The js provides “use Strict” expression to enable the strict mode if there is any silent error or mistake in the code it throws an error.

```
JS demo.js > sum
1  // const prompt = require('prompt-sync')();
2  console.log(sum(10,20));
3  function sum(a,a){
4      "use Strict";
5      return a + a;
6  }
```

```
powershell ×
PS E:\HTML_CSS_JS> node demo.js
40
PS E:\HTML_CSS_JS>
```

More On JS Promise:

Promise in real-life express a trust between two or more person and an assurance that a particular thing will surely happen in JavaScript a promise is an Object which ensures to produces a single value in the future promise in js is used for managing and tracking asynchronous operations

Terminology of promises:

Pending: the pending promise is neither rejected nor fulfiled yet.

Fulfilled: the related promise action is fulfilled successfully.

Rejected: the related promise action is failed to be fulfilled.

Settled: either the action is fulfilled or rejected.

Thus a promise represents the completion of an asynchronous operation with its result it can be either successful completion of the promise or its failure, but eventually completed. Promises uses a then() which is executed only after the completion of the promise resolve.

Promises Of Promise:

Unless the current execution of the JS event loop is not completed(success or failure) callbacks will never be called before it. Even if the callbacks with then() are present but they will be called only after the execution of the asynchronous operations completely. When multiple callbacks can be executed in a chain one after other following the sequence in which they were inserted.

Methods in Promise:

Promise.resolve(promise):

This method returns promise only if promis.constructor == promise.

Promise.resolve(thenable):

Makes a new promise from thenable containing then().

Promise.resolve(obj):

Makes a promise resolved for an object.

Promise.reject(obj):

Makes a promise rejection for the object.

Promise.all(array):

If any item in the array is fulilled as soon it resolves the promise of if any item is rejected as soon it rejects the promise.

Constructor in Promise:

New Promise(function(resolve,reject){});

Here resolve(thenable) denotes that the promise will be resolved with then(). Resolve(obj) denotes promise will be fulfilled with the object. reject (obj) denotes promise rejected with the object.

```
JS demo.js > p > <function> > x
// const prompt = require('prompt-sync')();
let p = new Promise(function(resolve, reject){
    let x = 100 + 250;
    if(x == 125)
        resolve("Executed and Resolved Successfully");
    else
        reject("rejected");
});
p.then(function(fromResolve){
    console.log("Promise is"+fromResolve);
}).catch(function(fromReject){
    console.log("Promise is"+fromReject);
});
```

```
powershell ×
PS E:\HTML_CSS_JS> node demo.js
Promise isExecuted and Resolved Successfully
PS E:\HTML_CSS_JS> node demo.js
Promise isrejected
PS E:\HTML_CSS_JS>
```

Resolve: when the promise is executed successfully the resolve argument is invoked which provides the result.

Reject: when the promise is rejected, the reject arguments is invoked which results in an error.

It means either resolve is called or object is called. Here then() has taken one argument which will executed if the promise is resolved otherwise catch() will be called with the rejection of the promise.

A better option to deal with asynchronous operations. Promise provides easy error handling and better code redability.

JavaScript eval() function:

The eval() function in js is used to evaluate the expression. It is js global function which evalutaes the specified String as js code and executed it. The parameter of the eval() function is a string. if the parameter represents the stataments eval() evaluates the statements. If the parameter is an expression eval() evaluates the expression, if the parameter of eval() is not a String the function returns the parameter unchanged.

```
JS demo.js > ...
1   // const prompt = require('prompt-sync')();
2
3   let cal = eval("20 + 50 * 60 - 7 / 45 % 115 + 48");
4   console.log(cal);

powershell ×

PS E:\HTML_CSS_JS> node demo.js
3067.8444444444444
PS E:\HTML_CSS_JS>
```

This vs Window:

`this` is a reference to the current execution context or object within which it is used. The value of `this` can change based on how a function is invoked, and it is often used in the context of functions or methods. When used in different contexts, `this` can refer to different objects, making it dynamic and dependent on the current code execution.

`window` is a global object in a browser's environment and represents the global scope. It contains properties and methods that are accessible globally in a web page. For example, global variables and functions are properties and methods of the `window` object. Unlike `this`, which can change its context depending on where it's used, `window` always refers to the global object, providing access to global variables and functions.

Sharpener Stacks Question:

demo.js

```
const prompt = require("prompt-sync")();
class stack{
    constructor(stack){
        this.stack = stack;
    }
    push(value){
        this.stack.push(value);
    }
    pop(){
        if(this.stack.length === 0){
            return -1;
        }return this.stack.pop();
    }
}
function imp(arr){
    let stack1 = new stack(arr);
    stack1.push(1);
    console.log(stack1.pop());
    console.log(stack1.pop());
    console.log(stack1.pop());
    stack1.push(1);
    console.log(stack1.pop());
    console.log(stack1.pop());
    console.log("====================");
}
let arr = [3,4,5,7,8];
imp(arr);
let arr1 = [3,2];
imp(arr1);
```

```
PS E:\HTML_CSS_JS> node demo.js
1
8
7
1
5
====================
1
2
3
1
-1
====================
PS E:\HTML_CSS_JS>
```

Sharpener Queue Question:

demo.js

```
const prompt = require("prompt-sync")();
class queue{
    constructor(queue){
        this.queue = queue;
        this.minpos = 0;
        this.maxpos = queue.length - 1;
    }
    push(value){
        this.queue.push(value);
        this.maxpos = this.queue.length - 1;
    }
    pop(){
        if(this.minpos > this.maxpos){
            return -1;
        }
        let value = this.queue[this.minpos];
        this.minpos++;
        return value;
    }
}
function imp(arr){
    let queue1 = new queue(arr);
    queue1.push(1);
    console.log(queue1.pop());
    console.log(queue1.pop());
    console.log(queue1.pop());
    queue1.push(1);
    console.log(queue1.pop());
    console.log(queue1.pop());
    console.log("-------------------------");
}
let arr = [3,4,5,7,8];
imp(arr);
let arr1 = [3,2];
imp(arr1);
```

```
PS E:\HTML_CSS_JS> node demo.js
3
4
5
7
8
-------------------------
3
2
1
1
-1
-------------------------
PS E:\HTML_CSS_JS>
```

Sharpener Callbacks Question1:

demo.js

```
const prompt = require("prompt-sync")();
function c3rdpost(callback){
    setTimeout(() => {
        console.log('Post Three');
        if(callback){
            callback();
        }
    }, 2000);
}
function c4thpost(){
    setTimeout(() => {
        console.log('Post Four');
    }, 1000);
}
c3rdpost(function(){
    c4thpost();
})
```

```
PS E:\HTML_CSS_JS> node demo.js
Post Three
Post Four
PS E:\HTML_CSS_JS>
```

Sharpener Callback Question2:

demo.js

```
const prompt = require("prompt-sync")();
function c3rdpost(callback){
    setTimeout(() => {
        console.log('Post Three');
        if(callback){
            callback();
        }
    }, 3000);
}
function c4thpost(callback){
    setTimeout(() => {
        console.log('Post Four');
        if(callback){
            callback();
        }
    }, 2000);
}
function c5thpost(callback){
    setTimeout(() => {
        console.log('Post Five');
        if(callback){
            callback();
        }
    }, 1000);
}
c3rdpost(() => {
    c4thpost(() => {
        c5thpost();
    });
});
```

```
PS E:\HTML_CSS_JS> node demo.js
Post Three
Post Four
Post Five
PS E:\HTML_CSS_JS>
```

Sharpener Promise Question1:

demo.js

```
const prompt = require("prompt-sync")();
let post  = [
    {title : 'Post One'},
    {title  : 'Post Two'}
];
function printpost(){
    post.forEach((post) => {
        console.log(post.title);
    })
}
function c3rdpost(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            post.push({title : 'post three'});
            resolve();
        }, 3000);
    })
}
function c4thpost(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            post.push({title : 'post four'});
            resolve();
        }, 2000);
    })
}
c3rdpost()
.then(() => c4thpost())
.then(() => printpost());
```

```
PS E:\HTML_CSS_JS> node demo.js
Post One
Post Two
post three
post four
PS E:\HTML_CSS_JS>
```

Sharpener Promise Question2:

demo.js

```
const prompt = require("prompt-sync")();
const posts = [{ title: 'POST1' }, { title: 'POST2' }];
function printPost() {
  posts.forEach((post) => {
    console.log(post.title);
  });
}
function create3rdPost() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        posts.push({ title: 'POST3' });
        resolve();
      }, 3000);
    });
}
function create4thPost() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        posts.push({ title: 'POST4' });
        resolve();
      }, 2000);
    });
}
function create5thPost() {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        posts.push({ title: 'POST5' });
        resolve();
      }, 1000);
    });
}
create3rdPost()
    .then(() => create4thPost())
    .then(() => create5thPost())
    .then(() => printPost());
```

```
PS E:\HTML_CSS_JS> node demo.js
POST1
POST2
POST3
POST4
POST5
PS E:\HTML_CSS_JS>
```

Sharpenner Break Promise Question1:

demo.js

```javascript
const prompt = require("prompt-sync")();
const posts = [{ title: 'POST1' }, { title: 'POST2' }];
function printPost() {
  posts.forEach((post) => {
    console.log(post.title);
  });
}
function create3rdPost() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      posts.push({ title: 'POST3' });
      resolve();
    }, 3000);
  });
}
function create4thPost() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      posts.push({ title: 'POST4' });
      resolve();
    }, 2000);
  });
}
function deletePost() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      posts.pop();
      resolve();
    }, 1000);
  });
}
create3rdPost()
  .then(() => deletePost())
  .then(() => create4thPost())
  .then(() => printPost());
```

```
PS E:\HTML_CSS_JS> node demo.js
POST1
POST2
POST4
PS E:\HTML_CSS_JS>
```

Sharpener Break Promise Question2:

demo.js

```javascript
const prompt = require("prompt-sync")();
const posts = [{ title: 'POST1' }];3
function create2ndPost() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      posts.push({ title: 'POST2' });
      resolve();
    }, 3000);
  });
}
function create3rPost() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      posts.push({ title: 'POST3' });
      resolve();
    }, 2000);
  });
}
function deletePost() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (posts.length > 0) {
        const poppedElement = posts.pop();
        resolve(poppedElement);
      } else {
        reject("ERROR: ARRAY IS EMPTY");
      }
    }, 1000);
  });
}
```

```
PS E:\HTML_CSS_JS> node demo.js
POST2
POST3
POST1
ERROR: ARRAY IS EMPTY
PS E:\HTML_CSS_JS>
```

```javascript
create2ndPost()
  .then(() => deletePost())
  .then((deletedPost) => {
    console.log(deletedPost.title);
  })
  .then(() => create3rPost())
  .then(() => deletePost())
  .then((deletedPost) => {
    console.log(deletedPost.title);
  })
  .then(() => deletePost())
  .then((deletedPost) => {
    console.log(deletedPost.title);
  })
  .then(() => deletePost())
  .then((deletedPost) => {
    console.log(deletedPost.title);
  })
  .catch((error) => {
    console.log(error);
  });
```

Make & Break Promises:

```
demo.js
demo.js > create2ndBlog > <function> > setTimeout() callback
const prompt = require("prompt-sync")();
const blogs = [];
function create1stBlog() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      blogs.push({ title: 'BLOG1' });
      resolve();
    }, 3000);
  });
}
function create2ndBlog() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      blogs.push({ title: 'BLOG2' });
      resolve();
    }, 2000);
  });
}
function deleteBlog() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (blogs.length > 0) {
        const deletedBlog = blogs.pop();
        resolve(deletedBlog);
      } else {
        reject('ERROR');
      }
    }, 1000);
  });
}
```

```
PS E:\HTML_CSS_JS> node demo.js
BLOG2
BLOG1
ERROR
PS E:\HTML_CSS_JS>
```

```
create1stBlog()
  .then(() => create2ndBlog())
  .then(() => deleteBlog())
  .then((deletedBlog) => {
    console.log(deletedBlog.title);
  })
  .then(() => deleteBlog())
  .then((deletedBlog) => {
    console.log(deletedBlog.title);
  })
  .then(() => deleteBlog())
  .then((deletedBlog) => {
    console.log(deletedBlog.title);
  })
  .catch((error) => {
    console.log(error);
  });
```

MCQ On Promise:

```
demo.js > ...
const prompt = require("prompt-sync")();
console.log('start')
const promise1 = new Promise((resolve, reject) => {
  console.log(1)
})
console.log('end');
```

```
powershell
PS E:\HTML_CSS_JS> node demo.js
start
1
end
PS E:\HTML_CSS_JS>
```

demo.js > ...

```js
const prompt = require("prompt-sync")();
console.log('start');
const promise1 = new Promise((resolve, reject) => {
  console.log(1)
  resolve(2)
})
promise1.then(res => {
  console.log(res)
})
console.log('end');
```

powershell

```
PS E:\HTML_CSS_JS> node demo.js
start
1
end
2
PS E:\HTML_CSS_JS>
```

demo.js > ...

```js
const prompt = require("prompt-sync")();
console.log('start');
const promise1 = new Promise((resolve, reject) => {
  console.log(1)
  resolve(2)
  console.log(3)
})
promise1.then(res => {
  console.log(res)
})
console.log('end');
```

powershell

```
PS E:\HTML_CSS_JS> node demo.js
start
1
3
end
2
PS E:\HTML_CSS_JS>
```

demo.js > ...

```javascript
const prompt = require("prompt-sync")();
console.log('start');
const promise1 = new Promise((resolve, reject) => {
  console.log(1)
})
promise1.then(res => {
  console.log(2)
})
console.log('end');
```

powershell

```
PS E:\HTML_CSS_JS> node demo.js
start
1
end
PS E:\HTML_CSS_JS>
```

demo.js > ...

```javascript
const prompt = require("prompt-sync")();
console.log('start')
const fn = () => (new Promise((resolve, reject) => {
  console.log(1);
  resolve('success')
}))
console.log('middle')
fn().then(res => {
  console.log(res)
})
console.log('end')
```

powershell

```
PS E:\HTML_CSS_JS> node demo.js
start
middle
1
end
success
PS E:\HTML_CSS_JS>
```

JS demo.js > ...

```js
const prompt = require("prompt-sync")();
console.log('start')
Promise.resolve(1).then((res) => {
  console.log(res)
})
Promise.resolve(2).then((res) => {
  console.log(res)
})
console.log('end');
```

powershell

```
PS E:\HTML_CSS_JS> node demo.js
start
end
1
2
PS E:\HTML_CSS_JS>
```

JS demo.js > ...

```js
const prompt = require("prompt-sync")();
console.log('start')
setTimeout(() => {
  console.log('setTimeout')
})
Promise.resolve().then(() => {
  console.log('resolve')
})
console.log('end')
```

powershell

```
PS E:\HTML_CSS_JS> node demo.js
start
end
resolve
setTimeout
PS E:\HTML_CSS_JS>
```

JS demo.js > ...

```js
const prompt = require("prompt-sync")();
const promise = new Promise((resolve, reject) => {
    console.log(1);
    setTimeout(() => {
      console.log("timerStart");
      resolve("success");
      console.log("timerEnd");
    }, 0);
    console.log(2);
});
promise.then((res) => {
    console.log(res);
});
console.log(4);
```

powershell

```
PS E:\HTML_CSS_JS> node demo.js
1
2
4
timerStart
timerEnd
success
PS E:\HTML_CSS_JS>
```

JS demo.js > timer2

```js
const prompt = require("prompt-sync")();
const timer1 = setTimeout(() => {
console.log('timer1');
const promise1 = Promise.resolve().then(() => {
      console.log('promise1')
    })
}, 0)
const timer2 = setTimeout(() => {
    console.log('timer2')
}, 0)
```

powershell

```
PS E:\HTML_CSS_JS> node demo.js
timer1
promise1
timer2
PS E:\HTML_CSS_JS>
```

```js
const prompt = require("prompt-sync")();
console.log('start');
const promise1 = Promise.resolve().then(() => {
  console.log('promise1');
  const timer2 = setTimeout(() => {
    console.log('timer2')
  }, 0)
});
const timer1 = setTimeout(() => {
  console.log('timer1')
  const promise2 = Promise.resolve().then(() => {
    console.log('promise2')
  })
}, 0)
console.log('end');
```

```
PS E:\HTML_CSS_JS> node demo.js
start
end
promise1
timer1
promise2
timer2
PS E:\HTML_CSS_JS>
```

Too Many Promise In Life:

```js
const prompt = require("prompt-sync")();
function updateLastUserActivityTime() {
  return new Promise((resolve) => {
   setTimeout(() => {
    const updatedLastActivityTime = new Date().toTimeString();
    resolve(updatedLastActivityTime);
   }, 1000);
  });
}
function createPost(post) {
  return new Promise((resolve) => {
   setTimeout(() => {
    const newPost = { content: post };
    resolve(newPost);
   }, 500);
  });
}
function deletePost(post) {
  return new Promise((resolve) => {
   resolve();
  });
}
createPost("New post content")
  .then((newPost) => {
   console.log("New Post Created:", newPost);
   return updateLastUserActivityTime();
  })
  .then((updatedLastActivityTime) => {
   console.log("Last Activity Time Updated:", updatedLastActivityTime);
   return deletePost();
  })
  .then(() => {
   console.log("Last Post Deleted");
  })
  .catch((error) => {
   console.error("An error occurred:", error);
  });
```

```
PS E:\HTML_CSS_JS> node demo.js
New Post Created: { content: 'New post content' }
Last Activity Time Updated: 19:02:17 GMT+0530 (India Standard Time)
Last Post Deleted
PS E:\HTML_CSS_JS>
```

Async & Await:

demo.js

demo.js > getColdDrinks > <function> > setTimeout() callback

```javascript
const prompt = require("prompt-sync")();
function getButter() {
    return new Promise((resolve, reject) => {
     setTimeout(() => {
      console.log("Husband got butter");
      resolve("butter");
     }, 2000);
    });
}
function getColdDrinks() {
    return new Promise((resolve, reject) => {
     setTimeout(() => {
      console.log("Husband got cold drinks");
      resolve("cold drinks");
     }, 1500);
    });
}
   async function exampleWithDrinks() {
    try {
     const butter = await getButter();
     console.log(`Received: ${butter}`);
     const drinks = await getColdDrinks();
     console.log(`Received: ${drinks}`);
    } catch (error) {
     console.error(`Error: ${error}`);
    }
}
const promises = [getButter(), getColdDrinks()];
async function handleMultiplePromises() {
    try {
     const results = await Promise.all(promises);
     results.forEach((result, index) => {
      console.log(`Received: ${result} (Promise ${index + 1})`);
     });
    } catch (error) {
     console.error(`Error: ${error}`);
    }
}
exampleWithDrinks();
handleMultiplePromises();
```

powershell

```
PS E:\HTML_CSS_JS> node demo.js
Husband got cold drinks
Husband got butter
Received: butter (Promise 1)
Received: cold drinks (Promise 2)
Husband got butter
Received: butter
Husband got cold drinks
Received: cold drinks
PS E:\HTML_CSS_JS>
```