Undefined – When a variable is declared but not assigned, it has the value of undefined and it’s type is also undefined.

var x;

var y = undefined;

Null – It represents a non-existent or a invalid value.

Var z = null;

Json – Java script object notation.

Var obj = {“1” : “Anurag”}

Console.log(obj[“1”])

Payload : Important information in a data block that client send to the server in the body of an HTTP post, put or patch . It contains important information about the request.

Semi-structured : Because it does not require a schema.

Store Array : Yes it can store arrays and it can be type of string, boolean, number, object or null

IIFE – Immediately invoked function expression : It runs as soon as it is defined.

(() => {

let firstVariable;

let secondVariable;

})();

// First and Second variable will be discarded after the function is executed.

Global Variables – They are bad because every js file included in the page runs in the same scope.

Function Scope: Yes it has functional scope: Each function creates a new scope. Variables declared inside the scope are not accessible from outside.

var top = 10

var inner = 50

function foo() {

var inner=20

console.log(inner)

}

foo() -> 20

Destructuring : In Js it makes it possible to unpack values from array, or properties from objects into distinct values.

eg: Let arr = [3, 5]

Let [a,b] = arr

eg:

var one = [“Anurag”, “Yuvraj”];

var first = one[0]

console.log(first); -> Anurag

Spread: It allows an iterable such as array or string to be expanded in places where zero or more arguments are expected or an object expression to be expanded where key-value pairs are expected.

Eg. let arr1 = [3, 5, 8]

let obj1 = { …arr1 }

console.log(obj1); -> {0: 3, 1: 5, 2: 8}

Rest :

It allows a function to accept an indefinite numbers of arguments as an array.

Function myFunc(a, b, …manyMoreArgs) {

console.log(“a”, a);

console.log(“b”, b);

console.log(“manyMoreArgs”, manyMoreArgs);

}

myFunc(1, 2, 3, 4, 5, 6);

// Console Output:

a , 1

b, 2

manyMoreArgs, [“3”, “4”, “5”, “6”]

Arrow Function:

We do not need to use the function keyword

hello = () => {

return “Hello World”;

}

Template Literals :

It uses Back-Ticks (` `) rather than quotes(“”) to define a string.

Eg:

let firstName = “Anurag”;

let lastName = “Dixit”;

let text = `Welcome ${firstName) , ${lastName}!`;

Hoisting :

It is the default behaviour of moving the declarations to the top.

eg:

var x;

x = 5;

console.log(x);

Use Strict :

At the beginning of a script or function “use strict”; –It means strict mode is declared

eg: “use strict”;

x = 3.14; // This will cause an error because x is not declared.

JavaScript Closures:

It is a feature in js that allows inner function to access their outer scope. It is automatically created whenever a function is created.

Eg: function foo() {

var b = 1;

function inner() {

return b;

}

return inner;

}

var get\_func\_inner = foo();

console.log(get\_func\_inner()); -> 1

Callback Method :

In JS , a callback function is a function that is passed as an argument to another function and is executed inside the other function. The function that receives the callback function as an argument is called the “higher-order function”.

When the higher order function is executed, it calls the callback function at a certain point in its execution, passing any necessary arguments to it. The callback function is then executed and its result is used by higher order function to continue its execution .

eg : function greet(name, callback) {

console.log(‘Hello, ‘+ name );

callback();

}

Function sayGoodby() {

Console.log(‘Goodbye’);

}

greet(‘Alice’, sayGoodbye);

When greet is called with the arguments 'Alice' and sayGoodbye, it logs 'Hello, Alice!' to the console and then calls the sayGoodbye function, which logs 'Goodbye!' to the console. Note that the callback function sayGoodbye is defined separately from the higher-order function greet. This allows us to pass different callback functions to greet depending on our needs. Callbacks are commonly used in asynchronous programming, where we want to execute a function once some other operation has completed. By passing a callback function to an asynchronous function, we can ensure that the callback is executed only when the asynchronous operation has completed.

Var : It has function scope, which means that variables declared with `var` are accessible within the function they are declared in, as well as any nested function. If a variable is declared with `var` outside of any function, it becomes a global variable that can be accessed from anywhere in the code. It can be changed at any time.

function example() {

var x = 10;

if (true) {

var x = 20; // This will overwrite the value of x

console.log(x); // Output: 20

}

console.log(x); // Output: 20

}

let : It has block scope, which means that variable declared with `let` are only accessible with block they are declared in. It can be reassigned a new value, but not redeclared within same block.

function example() {

let x = 10;

if (true) {

let x = 20; // This is a different variable than the one outside the block

console.log(x); // Output: 20

}

console.log(x); // Output: 10

}

Const : It also have the block scope. It cannot be reassigned a new value after they are declare.

function example() {

const x = 10;

if (true) {

const x = 20; // This is a different variable than the one outside the block

console.log(x); // Output: 20

}

console.log(x); // Output: 10

}

Promises : They are used to handle asynchronous operations. Promises are objects that represent the eventual completion of an asynchronous operation and provide a way to chain asynchronous operation together.

Overall Promises provide a cleaner and more organized way to handle asynchronous operation in JS.

function fetchUserData(userId) {

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

const xhr = new XMLHttpRequest();

xhr.open('GET', `https://jsonplaceholder.typicode.com/users/${userId}`);

xhr.onload = () => {

if (xhr.status === 200) {

resolve(JSON.parse(xhr.responseText));

} else {

reject(Error(xhr.statusText));

}

};

xhr.onerror = () => {

reject(Error('Network Error'));

};

xhr.send();

});

}

fetchUserData(1)

.then((user) => {

console.log(`User ID ${user.id}: ${user.name}`);

})

.catch((error) => {

console.error(error);

});

In this example, fetchUserData is a function that returns a promise that will eventually resolve with the user data for the given userId. The promise is created with a constructor function that takes a function with two arguments: resolve and reject. The resolve function is called when the asynchronous operation is successful, and the reject function is called when there is an error.

The fetchUserData function uses the XMLHttpRequest object to make an HTTP request to the JSONPlaceholder API to get the user data. When the request completes, the onload event is fired, and the resolve or reject function is called depending on whether the request was successful or not.

The then method is used to register a callback function that will be called when the promise is resolved successfully. In this example, the callback function logs the user data to the console. The catch method is used to register a callback function that will be called if the promise is rejected. In this example, the callback function logs the error to the console.

Proxy :

In JavaScript, a Proxy is an object that wraps another object and intercepts its method calls and property accesses, allowing you to add custom behavior or logic to those operations

Async/Await : It is a feature in Js that allows you to write asynchronous code in a synchronous style. This makes it avoid callback hell.

function fetchUserData(userId) {

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

const xhr = new XMLHttpRequest();

xhr.open('GET', `https://jsonplaceholder.typicode.com/users/${userId}`);

xhr.onload = () => {

if (xhr.status === 200) {

resolve(JSON.parse(xhr.responseText));

} else {

reject(Error(xhr.statusText));

}

};

xhr.onerror = () => {

reject(Error('Network Error'));

};

xhr.send();

});

}

async function getUserData() {

try {

const user = await fetchUserData(1);

console.log(`User ID ${user.id}: ${user.name}`);

} catch (error) {

console.error(error);

}

}

getUserData();

In this example, fetchUserData is a function that returns a promise that will eventually resolve with the user data for the given userId. The promise is created with a constructor function that takes a function with two arguments: resolve and reject. The resolve function is called when the asynchronous operation is successful, and the reject function is called when there is an error.

The getUserData function is declared with the async keyword, which allows us to use the await keyword inside the function to wait for the fetchUserData function to complete before continuing. The await keyword can only be used inside an async function.

Inside the try block, we use the await keyword to wait for the fetchUserData function to complete and return the user data. Once we have the user data, we log it to the console. If there is an error, the catch block is executed, and we log the error to the console.

Asynchronous Callback :

In JavaScript, an asynchronous callback is a function that is passed as an argument to another function, which will execute the callback at a later point in time when some asynchronous operation is completed.

Here is an example of using asynchronous callback in JavaScript:

function doSomethingAsync(callback) {

setTimeout(() => {

console.log("Async operation completed");

callback();

}, 1000);

}

console.log("Before async operation");

doSomethingAsync(() => {

console.log("Callback executed");

});

console.log("After async operation");

In this example, doSomethingAsync is a function that simulates an asynchronous operation by using setTimeout to wait for 1 second before logging a message to the console and then executing the callback function that was passed as an argument.

We call doSomethingAsync and pass a callback function that logs a message to the console. The code following the call to doSomethingAsync continues to execute immediately, without waiting for the asynchronous operation to complete. The output of the code would be:

Before async operation

After async operation

Async operation completed

Callback executed

As you can see, the callback function is executed after the asynchronous operation completes, which happens after the rest of the code has executed.

Using asynchronous callbacks allows us to write code that is more efficient and responsive, because it doesn't block the main thread while waiting for long-running operations to complete. Instead, it allows the main thread to continue executing other code while the asynchronous operation is in progress, and then executes the callback function when the operation is complete.

Map : In JavaScript, map() is a built-in method of arrays that allows you to apply a function to each element of an array and return a new array with the results. The original array is not modified by the map() method.

const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map((num) => {

return num \* 2;

});

console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]

forEach :

In JavaScript, forEach() is a built-in method of arrays that allows you to loop through each element of an array and perform some action for each element. It does not return a new array like map() does, but instead modifies the original array in place.

const numbers = [1, 2, 3, 4, 5];

numbers.forEach((num) => {

console.log(num);

});