JavaScript is a ***popular Scripting programming language of the Web.***

JavaScript defined **behaviors** of web pages.

JavaScript was **invented** by **Brendan Eich in 1995** and became an **ECMA standard in 1997.**

**ECMA-262** is the **official name of the standard**.

**ECMAScript** is the **official name of the language**.

**European Computer Manufacturers Association. (ECMA)**

ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5(2009), and ES6(2015).

Since 2016 new versions are named by year (ECMAScript 2016 / 2017 / 2018).

|  |
| --- |
| [Multi-paradigm](about:blank): [event-driven](about:blank), [functional](about:blank), [imperative](about:blank), declarative, [OOP](about:blank) |

Features

High-level | Just-in-time compile language | Dynamic typing | prototype-based object-orientation

How JavaScript run inside JS Engine

1. The Script gets loaded as a UTF-16-byte stream from either the network/ server, cache, or a service worker and passed to a byte stream decoder.
2. The byte stream decoder decodes the bytes into tokens. The tokens are sent to the parser.
3. The parser generates nodes based on the tokens and create an AST (Abstract Syntax Tree).
4. The interpreter walks through the AST and generate byte code.
5. The byte code and type feedback are sent to the optimizing compiler, which generates highly optimized machine code.

***Reference*** [How JavaScript Works ? | Javascript - Lecture 3 | Web Development Course - YouTube](https://www.youtube.com/watch?v=2lRQTdpwhFk)

# Where to run JavaScript

**Node**

**Open-Source** | **Cross platform.**

**Node** is **not a programming language** nor a **framework**. It’s a **runtime environment** to Execute JS code outside browsers.

Graphical user interface, diagram

Description automatically generated

V8

Microsoft Edge: Chakra

C++ and additional files

# JavaScript Programs

A **computer program** is a **list of "*instructions*"** to be ***"executed"*** by a ***computer.***

In a programming language, these programming instructions are called **statements**.

**A JavaScript program is a list of programming statements**.

In HTML, JavaScript programs are executed by the web browser.

# JavaScript Statements

***Building blocks of statement***: ***Values, Operators, Expressions, Keywords, and Comments***.

# JavaScript HTML methods

getElementById() = To search element by Id

*To change value of HTML Element*

document.getElementById("demo").innerHTML = "Hello JavaScript";

*To change styling of HTML Element*

document.getElementById("demo").style.fontSize = "35px";

document.getElementById("demo").style.display = "none";

*To change image source URL*

document.getElementById("demo").src='pic\_bulboff.gif’;

You can place an external script reference in <head> or <body> as you like.

Placing scripts at the bottom of the <body> element improves the display speed, because script interpretation slows down the display.

# External linking JavaScript

<script src="myScript.js"></script>

The script will behave as if it was located exactly where the <script> tag is located.

## **External JavaScript Advantages**

* It **separates** HTML and JS code
* It makes HTML and JavaScript **easier** to **read and maintain**
* Cached JavaScript files can speed up page loads
* External scripts are practical when the same code is used in many different web pages.

JavaScript accepts both double and single quotes

***JavaScript is case sensitive.***

# Output

* Writing into an **HTML element**, using innerHTML.
* Writing into the **HTML output** using document.write().

Using document.write() after an HTML document is loaded, will **delete all existing HTML**

The document.write() method should only be used for testing.

* Writing into an **alert box**, using window.alert() | alert().

You can skip the window keyword.

***In JavaScript, the window object is the global scope object, that means that variables, properties, and methods by default belong to the window object.***

This also means that specifying the window keyword is optional

* Writing into the **browser console**, using console.log().

Used for debugging purpose

# Input

String = prompt() | Number = Number(prompt()) | Integer = parseInt(prompt()) |

Float = parseFloat(prompt())

# Variable Mutation

**Variable mutation** to mutate the value of a variable. In other words, we can change the value as well as the type of a variable at any time. This process is known as **Variable Mutation**.

var name = "John"

name = 123 // ***Explicitly converted***

# Type Coercion

JavaScript automatically converts or changes the type of any variable as needed and this process is known as **Type Coercion**.

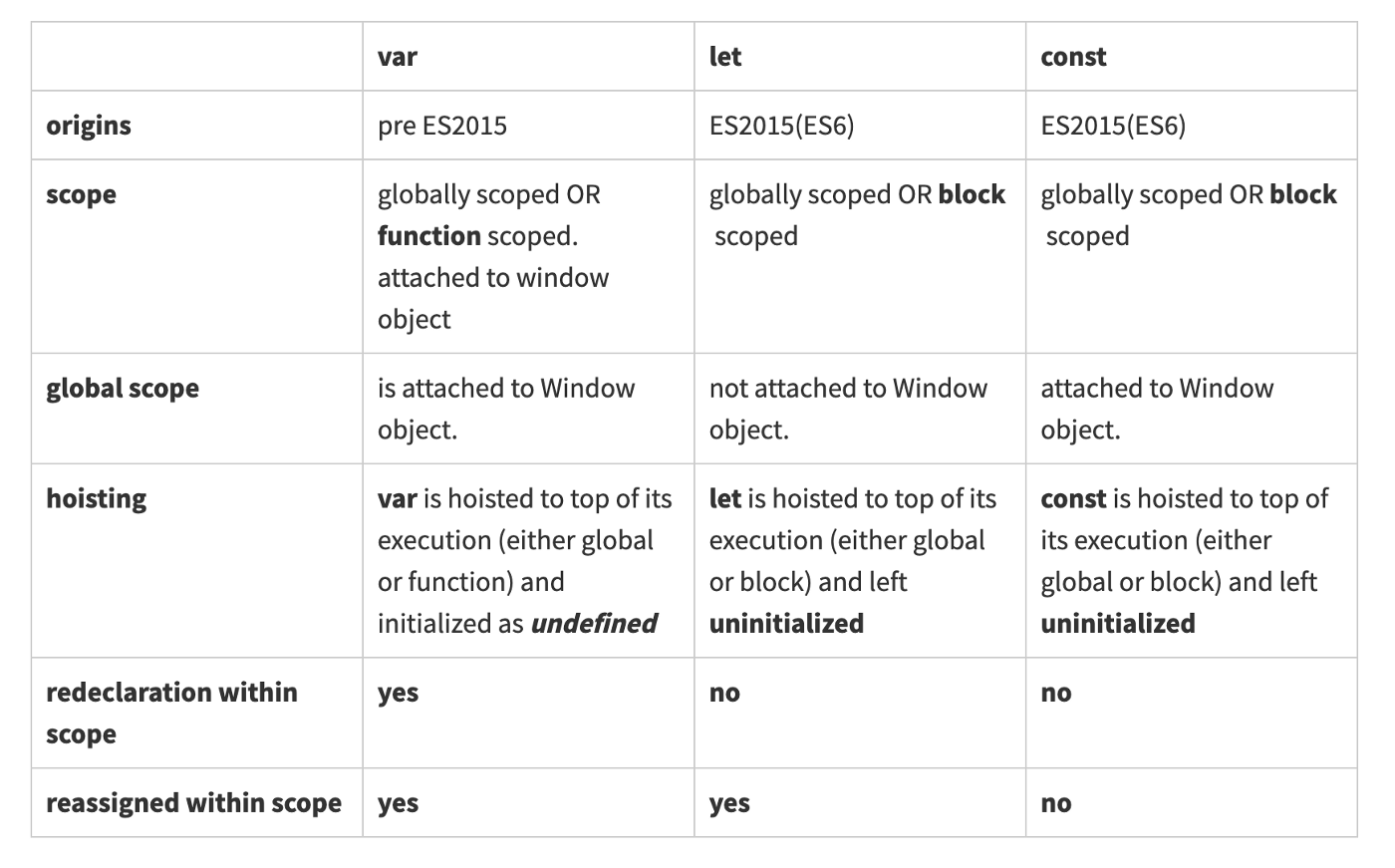
var name = "John"

var age = 24

But when we print these variables, we got the output as a **string**. We did not explicitly change the type of variables, but JavaScript did it. We call this process as **Type Coercion.**

console.log("My name is " + name + "and I am " +age+ "years old”) // ***Implicitly converted***

# Variables



{ … }

Block scope

## If you **re-declare** a JavaScript variable declared with var, it will not lose its value.

var carName = "Volvo";  
var carName; // “Volvo”

You **cannot re-declare** a variable declared with let or const.

This will not work:

let carName = "Volvo";  
let carName; // error // carName = “Honda works” // *redeclaration does not work for let and const but reassigning works only for let and var*

# Hoisting

Variables defined with var are **hoisted** to the top and can be initialized at any time.

Meaning: You can use the variable before it is declared.

carName = "Volvo";  
var carName;

Variables defined with let are also hoisted to the top of the block, but not initialized.

Meaning: Using a let variable before it is declared will result in a Error

# var x = 10; let x = 10; // Here x is 10 // Here x is 10 {// var does not support block scope {//block scope var x = 2; let x = 2; // Here x is 2 // Here x is 2 } } // Here x is 2 // Here x is 10

Use const when you declare:

A new Array | A new Object | A new Function | A new RegExp

The keyword const is a little misleading.

It does not define a constant value. It defines a ***constant reference*** to a value.

Because of this you **can NOT**:

* Reassign a constant value
* Reassign a constant array
* Reassign a constant object

But you **CAN**:

* Change the elements of constant array
* Change the properties of constant object

A variable **var** declared without a value will have the value undefined.

# Values

Fixed values are called **Literals**. (10.2, 3, ‘Saranj’)

Variable values are called **Variables**. (var x= 2, let, const)

# Comments

Single Line comments = //…

Multi Line comments = /\*…\*/

# Operator Precedence & Type Coercion

let x = "5" + 2 + 3; // 523 // ”5” + “2” + “3”  
let x = 2 + 3 + "5"; // 55

# JavaScript Types are Dynamic

JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

let x;           // Now x is undefined  
x = 5;           // Now x is a Number  
x = "John";      // Now x is a String

# DataTypes

|  |  |  |
| --- | --- | --- |
| Data Types | Description | Example |
| String | represents textual data | 'hello', "hello world!" etc |
| Number | an integer or a floating-point number  Even scientific numbers | 3, 3.234, 3e-2 etc. |
| BigInt | an integer with arbitrary precision | 900719925124740999n , 1n etc. |
| Boolean | Any of two values: true or false | true and false |
| undefined | a data type whose variable is not initialized | let a; |
| null | denotes a null value | let a = null; |
| Symbol | data type whose instances are unique and immutable | let value = Symbol('hello'); |
| Object | key-value pairs of collection of data | let student = { }; |

## ***JavaScript Numbers are Always 64-bit Floating Point***

# JavaScript Primitives

A **primitive value** is a value that has no properties or methods.

A **primitive data type** is data that has a primitive value.

JavaScript defines 5 types of primitive data types:

1. string
2. number
3. boolean
4. null
5. undefined

Primitive values are immutable (they are hardcoded and therefore cannot be changed).

if x = 3.14, you can change the value of x. But you cannot change the value of 3.14.

Primitives are copied by their value & Objects are copied by their reference.

# Template Literals |Template Strings |String Templates |Back-Tics Syntax

let text = `Hello ${x} World!`;

**It is used for-**

## Quotes Inside Strings, Multiline Strings, Interpolation, Variable Substitutions, Expression Substitution, Injecting HTML Templates

# Null Vs undefined

**Definition:**

**Null:** It is the intentional absence of the value. It is one of the primitive values of JavaScript.

**Undefined:** It means the value does not exist in the compiler. It is the global object.

**Type:**

|  |  |
| --- | --- |
| Null | Undefined |
| Type - Object | Type - undefined |
| When we define a variable to null then we are trying to convey that the variable is empty. | When we define a variable to undefined then we are trying to convey that the **variable does not exist.** |

null == undefined // (true == compares only value)

null === undefined // false (=== compares value and type, Values are same but not the type )

***It means null is equal to undefined but not identical.***

**Differentiating using isNaN():**

isNaN(2 + null) // false // 2 + null = 2

isNaN(2 + undefined) // true // 2 + undefined =NaN

**Examples:**

let x = null, y = undefined; **Both are False Value**

Undefined: When variable is not assigned a value

var temp;

if(temp === undefined) => True

Accessing values which does not exist

var temp=['a','b','c'];

if(temp[3] === undefined) => True

let x = ''”; // string

var y; // undefined

# This

|  |
| --- |
| In an object method, this refers to the object. |
| Alone, this refers to the global object. |
| In a function, this refers to the global object. |
| In a function, in strict mode, this is undefined. |
| In an event, this refers to the element that received the event. |
| Methods like call(), apply(), and bind() can refer this to any object. |

In regular functions the this keyword represented the object that called the function, which could be the window, the document, a button or whatever.

With arrow functions, the this keyword ***always*** represents the object that defined the arrow function.

var x = 'saranj'; // string

let y = new String('saranj'); // object

var z = 'saranj'; // string

let a = new String('saranj'); // object

console.log(x == y) // true

console.log(x === y) // false

console.log(x == z) // true

console.log(x === z) // true

console.log(a == y) // false

Comparing two JavaScript objects **always** returns **false**.

In JavaScript, **arrays** use **numbered indexes**.

In JavaScript, **objects** use **named indexes**.

# Function declaration/ statement

function functionName(…) {…}

# Function Expression

let functionVar = function(…) {…}

let functionVar = (…) => {…}

invoking functionVar(…)

# typeof

//string

console.log(typeof "saranj")

console.log(typeof '2')

//object

console.log(typeof [1, 2, 3, 4])

console.log(typeof null)**\***

console.log(typeof {name: 'saranj', age:22})

//undefined

console.log(typeof undefined)

//number

console.log(typeof NaN)

console.log(typeof Infinity)

console.log(typeof 12)

console.log(typeof 12.32)

console.log(typeof 2e12)

//function

console.log(typeof function hello(){})

const x = () => {};

console.log(typeof x)

# Arrow Function

const jobs = [

    {id: 1, isActive: true},

    {id: 2, isActive: true},

    {id: 3, isActive: false},

    {id: 4, isActive: true}

];

function fun(d){

    if(d.isActive){

        return d

    }

}

const ans = jobs.filter(fun);

console.log(ans); //[{ id: 1, isActive: true }, { id: 2, isActive: true }, { id: 4, isActive: true }]

const ans = jobs.filter(job\_dic => job\_dic.isActive)

console.log(ans) //[{ id: 1, isActive: true }, { id: 2, isActive: true }, { id: 4, isActive: true }]

# Class

class ClassName {  
  constructor() { ... }  
}

A JavaScript class is **not** an object.

It is a **template** for JavaScript objects.

class Car {  
  constructor(name, year) {  
    this.name = name;  
    this.year = year;  
  }  
  age(x) {  
    return x - this.year;  
  }  
}  
  
let date = new Date();  
let year = date.getFullYear();  
  
let myCar = new Car("Ford", 2014);  
document.getElementById("demo").innerHTML=  
"My car is " + myCar.age(year) + " years old.";

const kid = {

    name: 'saranj',

    eat(){

        console.log(this); // this returning the reference of kid

    }

};

console.log(kid) // { name: 'saranj', eat: [Function: eat] }

kid.eat() // { name: 'saranj', eat: [Function: eat] }

const eat = kid.eat;

console.log(eat); // [Function: eat]

//eat() // error

const eat2 = kid.eat.bind(kid)

console.log(eat)

eat2() // { name: 'saranj', eat: [Function: eat] }

// function in javascript are objects they have properties and methods

# Encapsulation

const car = {

    name: "hero",

    model: "xyz",

    description(){

        console.log("car");

    }

};

car.description(); // car

console.log(car.name); // hero

# Inheritance

class Animal{

    constructor(type){

        this.type = type;

    }

    eat(){

        console.log('Animals eat');

    }

}

class Cat extends Animal{

    constructor(type, legs){

        super(type);

        this.legs = legs;

    }

    food(){

        console.log('cats loves to drink milk')

    }

}

catObj = new Cat('landAnimal', 4);

catObj.food();

catObj.eat();

animalObj = new Animal('animals');

animalObj.eat();

# Factory Function

function Circle(radius){

    return {

        radius, // radius: radius

        draw: () => {

            console.log('draw circle') ;

        }

    };

}

const c1 = Circle(1); const c2 = Circle(2);

console.log(c1.radius); // 1 console.log(c2.radius); // 2

c1.draw(); // draw circle c2.draw(); // draw circle

# Constructor Function

function Circle(r){

    this.radius = r;

    this.draw = () => {

        console.log("Circle is drawing");

    }

}

const c = new Circle(1); // Circle.call({}, 1) // new creates {} empty object implicitly

//and Circle.apply({}, [1, 2, 3])// to pass array of parameters

console.log(c.radius); // 1

c.draw(); // Circle is drawing

new keyword create an empty object & without new keyword this point to windows

# IIFE

An **IIFE** (Immediately Invoked Function Expression) is a [JavaScript](about:blank) [function](about:blank) that runs as soon as it is defined.

| **IIFE** | [**Arrow function**](about:blank)**IIFE** | [**async**](about:blank)**IIFE** |
| --- | --- | --- |
| (function () {  /\* ... \*/  })(); | (() => {  /\* ... \*/  })(); | (async () => {  /\* ... \*/  })(); |

It is a design pattern which is also known as a [Self-Executing Anonymous Function](about:blank) and contains two major parts

Eg - console.log(((x) => x + 5)(3)); // 8

# Async

# Callbacks

A callback is a function passed as an argument to another function.

function print(content){

    console.log(content)

}

function add(n1, n2, callBackFunction){

    sum = n1 + n2

    callBackFunction(sum)

}

add(12, 3, print)

Wrong:  add(12, 3, print())

# Higher-Order Functions

In JavaScript, functions can be assigned to variables in the same way that strings or arrays can. They can be passed into other functions as parameters or returned from them as well.

A “higher-order function” is a function that accepts functions as parameters and/or returns a function.

# Asynchronous

Functions running in parallel with other functions are called asynchronous.

When using the JavaScript function setTimeout(), you can specify a callback function to be executed on time-out

setTimeout(() => {

    console.log("Hello India")

}, 4000); // display content after 4sec in parallel with other code

When using the JavaScript function setInterval(), you can specify a callback function to be executed for each interval:

setInterval(

    () => {

        console.log('Love JS');

    },

    1000 ); // display content after every 1sec in parallel with other code

# Promises

"Producing code" is code that can take some time

"Consuming code" is code that must wait for the result

A Promise is a JavaScript object that links producing code and consuming code

|  |  |
| --- | --- |
| myPromise.state | myPromise.result |
| "pending" | undefined |
| "fulfilled" | a result value |
| "rejected" | an error object |

function f(x){

    console.log('Hello Promise '+x);

}

let p = new Promise((success, failure) => {

    let value = 0;

    if(value == 0){

        success("OK");

    }else{

        failure("Error");

    }

});

p.then(

    (value) => {

        f(value);

    },

    (error) => {

        f(error);

    }

)

**async** makes a function return a Promise

**await** makes a function wait for a Promise

# Spread operator

const first = [1, 2 , 3]

const second = [34,3]

const combine = [...first, ...second]

const clone = [...first] // to clone an object

console.log(clone, typeof clone) // [1, 2 , 3] object

console.log(combine, typeof(combine)) ) // [ 1, 2, 3, 34, 3 ] object

const f = {name: 'hello'}

const s = {class: 3}

const combo = {...f, ...s}

const combo2 = {...f, ...s, fname:'saranj', lname:'bule'}

console.log(combo, typeof combo) // { name: 'hello', class: 3 } object

console.log(combo2) // { name: 'hello', class: 3, fname: 'saranj', lname: 'bule' }

# Destructuring

function calculate(a, b) {

const add = a + b;

const subtract = a - b;

const multiply = a \* b;

const divide = a / b;

return [add, subtract, multiply, divide];

}

const [add, subtract, multiply, divide] = calculate(4, 7);

document.write("<p>Sum: " + add + "</p>");

document.write("<p>Difference " + subtract + "</p>");

document.write("<p>Product: " + multiply + "</p>");

document.write("<p>Quotient " + divide + "</p>");