## 1- Why do we call JavaScript as dynamic language?

JavaScript is dynamic language means data types of variables can change during the runtime.

## 2- How does JavaScript determine data types?

JavaScript determine data types depending to the value assigned.

## 3- What is typeof functions? And How to check data type in JavaScript?

We can get datatype by using "typeOf" function.

## 4- What are different datatypes in Javascript ?

There are 12 types of datatypes in javascript and we can classify these 12 datatypes into 2 categories(primitive and non-primitive categories).

1- Primitive:
   String – number-bigInt –boolean –null- symbol() – undefined ➔ let x; console.log(x); // undefined
2- non-primitive:
   object – array- function – date – regExp

## 5- Explain Undefined Data types ?

Undefined means the variable has been declared but no value is assigned to it. (initialization)

**Usage Context:**

- When a variable is declared without a value.
- When a function does not explicitly(بشكل صريح) return a value.
- When accessing an object property or array element that does not exist.
  let x; console.log(x); // Output: undefined
  function foo() {} console.log(foo()); // Output: undefined
  let obj = {}; console.log(obj.prop); // Output: undefined

## 6- What is Null ?

Null indicates intentional absence (الغياب المتعمد) of data. Null indicates its not ZERO , Its not empty its just absence of data.

Use Cases:You want to explicitly clear the value of a variable or an object property.

**Usage Context:**

- When you want to explicitly indicate that a variable should have no value.

console.log(undefined == null); // Output: true ➔loosely equal
console.log(undefined === null); // Output: false ➔ strictly equal
Let a =4
Let b= null ➔ بيحول الى صفر فى العمليات الحسابيه
Let c ➔ بيحول الىnan فى العمليات الحسابية
Console.log(a*b) // 0 لأن اى حاجة فى صفر بصفر زى ماعارفين
Console.log(a*c) // Nan لأن بضرب عدد فى شئ مش رقم

## 7- Explain Hoisting ?

Hoisting is a JavaScript mechanism where variable and function declarations are moved to the top of their containing scope during the compile phase. This means that you can use functions and variables before they are actually declared in the code.

**Variable Hoisting:**

When it comes to variables, only the declaration is hoisted, not the initialization. This means that the variable declaration is moved to the top, but any assignment remains in its original place.

```
console.log(x); // Output: undefined
var x = 5;
console.log(x); // Output: 5
```

**let and const Hoisting**

Variables declared with let and const are also hoisted, but they are not initialized. Accessing them before the declaration results in a ReferenceError. This is known as the "temporal(المؤقتة) dead zone."

```
 console.log(y); // ReferenceError: Cannot access 'y' before initialization
   let y = 10;
   console.log(y); // Output: 10
```

```
  <script>
   console.log(y); // ReferenceError: y is not defined
    </script>
```

**Function Hoisting:**

Function declarations are hoisted entirely, meaning both the declaration and the definition are moved to the top of their scope.

```
  hoistedFunction(); // Output: "Hello, world!"

function hoistedFunction() {
  console.log("Hello, world!");
}
```

**Function Expressions and Arrow Functions:**

Function expressions and arrow functions are not hoisted in the same way. Only the variable declaration is hoisted, not the assignment.

```
 nonHoistedFunction(); // TypeError: nonHoistedFunction is not a function

var nonHoistedFunction = function() {
  console.log("This won't work!");
}
```

## 8- What are global variables ?

Global variables in JavaScript are variables that are declared in the global scope, meaning they are accessible from anywhere in the code. In a browser environment, the global scope is the window object. Global variables can be accessed and modified from any function or block of code.

1. **Using var:** When var is used outside of any function or block, it declares a global variable.

```
var globalVar = "I'm a global variable";
console.log(window.globalVar); // Output: I'm a global variable (in browser)
```

2. Using let or const: When let or const is used outside of any function or block, it also declares a global variable, but it does not attach to the window object.

```
let globalLet = "I'm a global variable too";
console.log(globalLet); // Output: I'm a global variable too
console.log(window.globalLet); // Output: undefined (in browser)
```

3. Without var, let, or const: If you assign a value to a variable without declaring it using var, let, or const, it automatically becomes a global variable. However, this practice is not recommended as it can lead to unintentional global variables.

```
 globalVar2 = "I'm also a global variable";
console.log(window.globalVar2); // Output: I'm also a global variable (in
browser)
```

## 9- What happens when you declare variable without VAR or let or const ?

When you declare a variable without using var, let, or const in JavaScript, the variable becomes a global variable, regardless(بغض النظر) of where it is declared. This can lead to several issues:

1. **Global Namespace Pollution(تلوث)**: The variable will be added to the global object (window ), which can cause conflicts(تعارضات) with other variables and functions in the global scope.
2. **Potential(احتمال) for Bugs**: Since the variable is not scoped to the function or block where it is declared, it can lead to unexpected behavior and make the code harder to debug and maintain.

```
    function myFunction() {
    // Without var, let, or const, x becomes a global variable
    x = 10;
}
myFunction();
console.log(x); // Outputs: 10
// If we declare another global variable x elsewhere, it will overwrite the
previous value
var x = 20;
console.log(x); // Outputs: 20
```

## 10- What is Use Strict ?

"use strict" is a directive introduced in ECMAScript 5 (ES5) that enables strict mode in JavaScript which helps catch common coding errors and "unsafe" actions (such as gaining access to the global object). When a script or a function is executed in strict mode, it helps enforce (فرض)stricter(اكثر صرامه) parsing (تفسير)and error handling of JavaScript code.

```
"use strict"
 x=10 //x is not defined
```

Prevents this from Referring to the Global Object: In non-strict mode, this within functions that are not called as object methods refers to the global object. In strict mode, this remains undefined in such cases.

```
    "use strict";
function foo() {
    console.log(this); // Outputs: undefined
}
foo();
```

## 11- How can we avoid Global Variables ?

1. **Encapsulation**: Encapsulate variables within functions, modules, or objects. This keeps them private and inaccessible from the global scope.

Example (Using Function)

```
function myFunction() {
    let localVar = 'I am local';
    console.log(localVar);
}
myFunction();
console.log(localVar); // Error: localVar is not defined
```

Example (Using IIFE - Immediately Invoked Function Expression):

```
(function() {
    let localVar = 'I am local';
    console.log(localVar);
})();
console.log(localVar); // Error: localVar is not defined
```

2. **Module Pattern:** In JavaScript, use the module pattern to create private and public members. ES6 modules (import and export) help encapsulate code.

```
// myModule.js
const privateVar = 'I am private';
export const publicVar = 'I am public';

export function publicFunction() {
    console.log(privateVar);
}
// main.js
import { publicVar, publicFunction } from './myModule.js';
console.log(publicVar); // 'I am public'
publicFunction(); // 'I am private'
```

3. **Closures:** Use closures to create private variables. A closure is a function that retains access to its lexical scope even when executed outside that scope.

```
function createCounter() {
    let count = 0;
    return function() {
        count++;
        return count;
    };
}
const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

4. **Namespace Pattern:** Create a single global object to contain your variables and functions, reducing the number of global variables.

```
const MyApp = {
    config: {
        setting1: true,
        setting2: 'default'
    },
    util: {
        method1: function() {
            console.log('method1');
        },
        method2: function() {
            console.log('method2');
        }
    }
};
MyApp.util.method1(); // 'method1'
console.log(MyApp.config.setting2) // "default"
```

5. **Use Let and Const Instead of Var.**

**12- What are Closure?** A closure is a feature in JavaScript (and in many other programming languages) where an inner function has access to variables from its outer (enclosing) function even after the outer function has finished executing.( Closures capture variables by reference, not by value.)

```
function outerFunction() {
    let outerVariable = 'I am an outer variable';

    function innerFunction() {
        console.log(outerVariable); // Accesses the outer function's variable
    }

    return innerFunction;
}

const myClosure = outerFunction();
myClosure(); // Outputs: 'I am an outer variable

/* ************************************************** */
function createCounter() {
    let count = 0; // Private variable

    return function() {
        count++;
        return count;
    };
}

const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
console.log(counter()); // 3
```

## 13- Why do we need Closures ?

1. **Data Encapsulation and Privacy:** Closures allow for the creation of private variables that are not accessible from the outside scope.

```
function createCounter() {
    let count = 0; // Private variable
    return {
        increment: function() {
            count++;
            return count;
        },
        decrement: function() {
            count--;
            return count;
        },
        getCount: function() {
            return count;
        }};}
const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.getCount());  // 2
```

2. **Stateful Functions:** Closures allow functions to maintain state across multiple invocations. This can be useful in scenarios where you need to track some information over time.

```
function createAdder(x) {
    return function(y) {
        return x + y;
    };
}

const add5 = createAdder(5);
console.log(add5(2));  // 7
console.log(add5(10)); // 15

const add10 = createAdder(10);
console.log(add10(2));  // 12
console.log(add10(10)); // 20
```

3. **Function Factories:** Closures enable the creation of function factories—functions that return other functions with specific, pre-configured behavior.

```
function multiplier(factor) {
    return function(number) {
        return number * factor;
    };
}

const doubler = multiplier(2);
console.log(doubler(5));  // 10
console.log(doubler(10)); // 20

const tripler = multiplier(3);
console.log(tripler(5));  // 15
console.log(tripler(10)); // 30
```


## 14- Explain IIFE (Immediately Invoked Function Expression)?

An IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined. It is a design pattern that creates a function expression and immediately invokes it. This pattern is useful for creating a new scope and avoiding polluting the global namespace.

```
(function() {
    // Code to be executed immediately
})();
```

1. **Function Expression:** The function is defined as an expression rather than a declaration. This is achieved by wrapping the function definition inside parentheses ().

2. **Immediate Invocation (استدعى):** The function is then immediately invoked by adding another pair of parentheses () at the end.
3. **Avoiding Global Variables:** IIFEs create a new scope, which helps in avoiding the pollution of the global namespace with variables that are only needed temporarily.

```
(function() {
    var tempVariable = 'I am local';
    console.log(tempVariable); // 'I am local'
})();

console.log(tempVariable); // Error: tempVariable is not defined
```

## 15- What is name collision(تضارب فى الاسماء) in global scope ?

Name collision in the global scope refers to a situation where two or more identifiers (such as variable names, function names, or class names) in a program have the same name and are defined in the same global namespace. This can cause conflicts and unpredictable behavior because the program might not be able to distinguish between the different entities that share the same name.

1. **Variable Collision**: If two global variables have the same name, one will overwrite the other, leading to unexpected results.
2. **Function Collision**: If two global functions have the same name, the latter definition will overwrite the former, leading to loss of functionality.

### Avoiding Name Collisions:

1. **Namespace Pattern**: Wrap(لف) the code in an object or module to create a unique namespace.

```
var MyApp = MyApp || {};
MyApp.myVariable = "Hello";
MyApp.myFunction = function() {
    console.log("MyApp function");
};
```

2. **Immediately Invoked Function Expressions (IIFE)**: Use IIFEs to create a local scope and avoid polluting the global namespace (and IIFE does not have name , So there is no you can get name collission).

## 16- What is different between normal function VS IIFE?

| items | Normal Function | IIFE |
|---|---|---|
| **Definition and Invocation** | A normal function is defined first and then called explicitly elsewhere in the code | An IIFE is defined and invoked immediately. It runs as soon as it is defined. |
| **Scope** | The variables and functions defined inside a normal function are scoped to that function. They are not accessible outside unless returned or otherwise exposed. | An IIFE creates a new scope immediately, which is particularly useful for avoiding global scope pollution. Variables defined inside an IIFE are not accessible outside it. |
| **Reusability** | Normal functions are reusable and can be called multiple times throughout the code. | IIFEs are often used to create private scopes and avoid name collisions in the global scope. They are not meant to be reusable since they execute only once. |

## 17- What are the various ways to create Javascript objects ?

1. **Object Literals:** This is the most common and simplest way to create objects in JavaScript.

```
let person = {
    name: "John",
    greet: function() {
        console.log("Hello, " + this.name);
    }
};
```

2. **Object Constructor:** You can create an object instance using the Object constructor.

```
let person = new Object();
person.name = "John";
person.greet = function() {
    console.log("Hello, " + this.name);
};
```

3. **Constructor Functions:** Constructor functions are used to create multiple instances of an object type

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
        console.log("Hello, " + this.name);
    };
}
let person1 = new Person("John", 30);
let person2 = new Person("Jane", 25);
```

4. **ES6 Classes:** Classes provide a more familiar syntax for creating constructor functions and dealing with inheritance.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    greet() {
        console.log("Hello, " + this.name);
    }
}
let person1 = new Person("John", 30);
let person2 = new Person("Jane", 25);
```

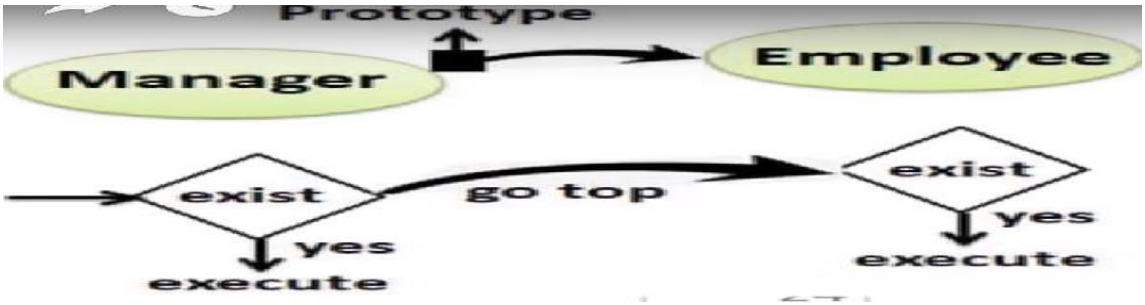
5. **Object.create():**This method creates a new object with the specified prototype object and properties.

```
let personPrototype = {
    greet: function() {
        console.log("Hello, " + this.name);
    }
};
let person = Object.create(personPrototype);
person.name = "John";
person.age = 30;
```

**18- How can we do inheritance in Javascript ?** In JavaScript, inheritance can be achieved in several ways, mainly using prototype-based inheritance and class-based inheritance (introduced in ES6).

1) **Prototype-Based Inheritance:** JavaScript uses prototypes for inheritance. Every JavaScript object has a prototype from which it can inherit properties and methods.

```
function Employee(){
    this.Name ="",
    this .DoWork= function(){
    alert("Basic work");
    }
    this. Attendance= function(){
    alert("Attendance needed");
}
}
function Manager(){
    this.Cäbq ="",
    this. DoWork= function(){
    alert("Manages team");
  }
}
var emp= new Employee();
Manager.prototype =emp;
var man=new Manager()
man.Attendance() //Attendance needed
man.DoWork() //manages team ➔ overriding
```




```
function Person(name) {
    this.name = name;
}
Person.prototype.greet = function() {
    console.log("Hello, " + this.name);
};
function Employee(name, jobTitle) {
    Person.call(this, name); // Call the Person constructor
    this.jobTitle = jobTitle;
}
Employee.prototype = Object.create(Person.prototype);
/* Employee.prototype.constructor = Employee; */
Employee.prototype.work = function() {
    console.log(this.name + " is working as a " + this.jobTitle);
};
let john = new Employee("John", "Developer");
john.greet(); // Output: Hello, John
john.work(); // Output: John is working as a Developer
```

2) **Class-Based Inheritance (ES6):**

```
class Person {
    constructor(name) {
        this.name = name;
    }

    greet() {
        console.log("Hello, " + this.name);
    }
}

class Employee extends Person {
    constructor(name, jobTitle) {
        super(name); // Call the parent class constructor
        this.jobTitle = jobTitle;
    }

    work() {
        console.log(this.name + " is working as a " + this.jobTitle);
    }
}

let john = new Employee("John", "Developer");
john.greet(); // Output: Hello, John
john.work(); // Output: John is working as a Developer
```

## 19- Explain Prototype chaining ?

Prototype chaining is a process where the property/ methods are first checked in the current object , if not found then it checks in the prototype object , if does not find in that it try checking prototype object , until he get the prototype object null.

```
▼ Employee {name: 'John', jobTitle: 'Developer'} i
    jobTitle: "Developer"
    name: "John"
  ▶ [[Prototype]]: Person
```

## 20- What is Let Keyword ?

"Let" helps to create immediate block level local scope.

```
function test() {
        let x=10
        if (x==10) {
            let x=2
            console.log(x) // 2
        }
        console.log(x) //10
      }
      test()
```

فى function دى فى two scope واحد بتاع function نفسه وواحد بتاع if === block بيقصد بيها الكود اللى بين {}

## 21- Explain Temporal Dead Zone ?

TDZ it's a period or it's a state of a variable where variables are named in memory but they are not initialized with any value.

```
console.log(x) // Cannot access 'x' before initialization
   let x=10
```

**22-** Let Vs Var ?

Let : Scoped to the immediate enclosing block.

Var : Scoped the Immediate function body.

```
 if (1==1) {
     console.log(x)  //undefained
     console.log(y) //can not accsess y before initiolation
     var x=10
     let y=5
  }
  console.log(x) // 10
  console.log(y) // y is not defined
```