JAVASCRIPT ES6 INTERVIEW GUIDE

What is ECMA Script

ECMA stands for **"European Computer Manufacturers Association."** This organisation provides standard javascript for different Browsers.

- ES6 was released june 2015 ad brought many new features, improvements and syntactical enhancements to javaScript:
- ES6 allows you to make the code more modern and raladable .
- By using ES6 features we write less ad do more " **write less, do more** "

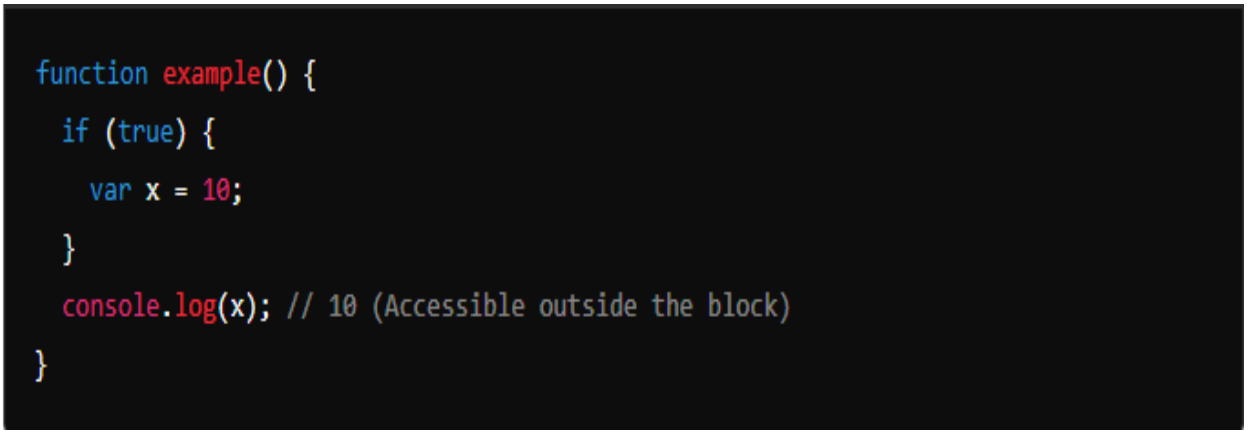
1. Introduced `let` and `const` variables in JavaScript.

The `let` and `const` variables were introduced in JavaScript with the ECMAScript 2015 (ES6) specification and both have Block-Level Scope.

`var`:

- **Function-scoped:** Variables declared with `var` are scoped to the function in which they are defined. If declared outside a function, they are globally scoped.
- **Block-level scope (e.g., inside `{ }`):** Variables declared with `var` do not respect block scope. This means that a `var` variable

declared inside a block (like an `if` statement or loop) can be accessed outside of that block.



```
function example() {
  if (true) {
    var x = 10;
  }
  console.log(x); // 10 (Accessible outside the block)
}
```

`let`:

- **Block Scope:** `let` is block-scoped, meaning it is only accessible within the block (e.g., inside a `{ }`) where it is defined. This is different from `var`, which is function-scoped.
- **Reassignment:** Variables declared with `let` can be reassigned new values multiple times

```
let x = 10;
if (true) {
  let x = 20; // This `x` is different from the `x` outside the block
  console.log(x); // 20
}
console.log(x); // 10
```

`const`:

- **Block Scope:** Like `let`, `const` is also block-scoped.
- **Immutable Binding:** Variables declared with `const` cannot be reassigned once they are initialised. However, if a `const` variable refers to an object or array, the properties or elements of that object or array can still be modified.

```
const y = 30;
y = 40; // Error: Assignment to constant variable.

const obj = { a: 1 };
obj.a = 2; // This is allowed because we're modifying the object, not reassigning the
console.log(obj.a); // 2
```

Key Differences:

	var	let	const
Hoisting	Hosted at top of global scope. Can be used before the declaration.	Hosted at top of some private scope and only available after assigning value. Can not be used before the declaration.	Hosted at top of some private scope and only available after assigning value. Can not be used before the declaration.
Scope	Global scope normally. Start to end of the function inside of the function.	Block scoped always. Start to end of the current scope anywhere.	Block scoped always. Start to end of the current scope anywhere.
Redeclaration	Yes, we can redeclare it in the same scope.	No, we can't redeclare it in the same scope.	No, we can't redeclare or reinitialize it.

2. Hoisting in JavaScript ?

Hoisting in JavaScript means that when you declare variables or functions, JavaScript moves their declarations to the top of the scope before the code actually runs. However, only the declarations are hoisted, not the initializations.

How Hoisting Works:

- **Variable Declarations:** The declarations are hoisted to the top, but their initializations are not. This means that if you try to use a variable before its declaration, it will result in `undefined` if declared with `var`, and it will cause a `ReferenceError` if declared with `let` or `const`.
- **Function Declarations:** Entire function declarations (both the name and the body) are hoisted. This allows functions to be called before they are defined in the code.

```
console.log(a); // Output: undefined
var a = 10;
console.log(a); // Output: 10
```

3. What is the output Both the code ?

```
for (var i = 0; i < 10; i++) {
    setTimeout(() => {
        console.log(i); // This will log 10 for each iteration
    }, 1000);
}
```

Since `var` is not block-scoped, the `i` inside the `setTimeout` callback refers to the same `i` that the `for` loop modifies.
After the loop finishes, `i` is `10`, so every `setTimeout` callback logs `10`.

Output will be 10 times 10 .

```
for (let i = 0; i < 10; i++) {
    setTimeout(() => {
        console.log(i); // This will log 0, 1, 2, ... 9
    }, 1000);
}
```

With `let`, each iteration of the loop has a separate `i` value, so the `setTimeout` callback will log the correct `i` value for each iteration.

Output will be 0,1,2,3,4,5,6,7,8,9

4. Default Parameters in JavaScript ?

Default parameters in JavaScript allow you to set default values for function parameters if no value or `undefined` is provided when the function is called. This feature was introduced in ECMAScript 2015 (ES6).

```
function sum(a = 0, b = 0) {
    return a + b;
}

console.log(sum(5, 10)); // Output: 15
console.log(sum(5));     // Output: 5 (b defaults to 0)
console.log(sum());      // Output: 0 (both a and b default to 0)
```

Passing functions as default parameters can be particularly useful for setting up customizable behaviours in your code.

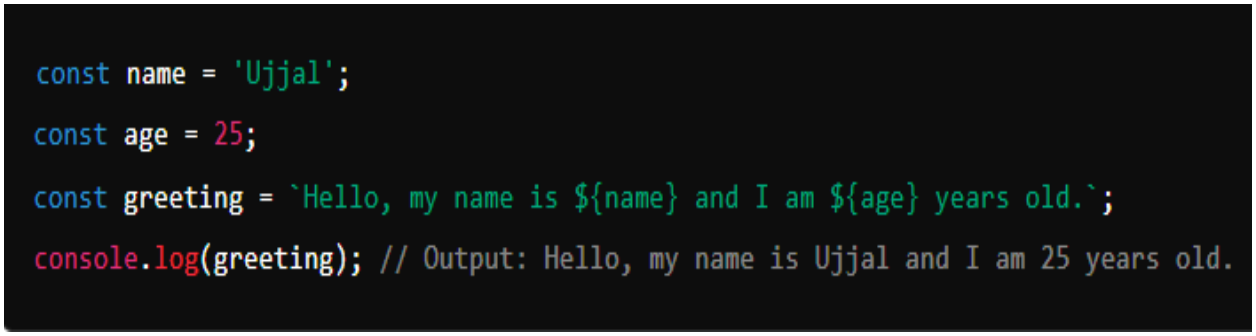
```
function calculate(a, b, operation = (x, y) => x + y) {
    return operation(a, b);
}

console.log(calculate(5, 3));              // Output: 8 (default addition)
console.log(calculate(5, 3, (x, y) => x * y)); // Output: 15 (custom multiplication)
```

5. Template Literals in JavaScript ?

Template literals in JavaScript, introduced with ECMAScript 2015 (ES6), provide a way that you can concat variable with strings .

Template literals use backticks (`) instead of single quotes (') or double quotes (").



```
const name = 'Ujjal';
const age = 25;
const greeting = `Hello, my name is ${name} and I am ${age} years old.`;
console.log(greeting); // Output: Hello, my name is Ujjal and I am 25 years old.
```

5. Find and FindIndex in JavaScript ?

The `find` method returns the **first element** in the array that satisfies and The `findIndex` method returns the **index of the first element** in the array that satisfies , If no elements satisfy the condition, it returns `-1`.

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

const firstEven = numbers.find(num => num % 2 === 0);
console.log(firstEven); // Output: 2 (the first even number in the array)
```

```
const numbers = [10, 20, 30, 40, 50];

const indexOfFirstGreaterThan25 = numbers.findIndex(num => num > 25);
console.log(indexOfFirstGreaterThan25); // Output: 2 (index of the first nu
```

```
let obj = [
    { id: 1, value: 50 },
    { id: 2, value: 55 },
    { id: 3, value: 60 },
    { id: 3, value: 70 }
];

let result = obj.find((item) => item.value > 60);
console.log(result);
```

6. Arrow Function in JavaScript ?

Arrow functions in JavaScript, introduced with ECMAScript 2015 (ES6), provide a **shorter syntax for writing functions**. They also differ from traditional function expressions in terms of how they handle the `this` keyword.

Summary:

- **Arrow functions** provide a concise syntax for writing functions.
- They do not have their own `this` context and inherit `this` from their lexical scope.
- They cannot be used as constructors and do not have an `arguments` object.
- They are particularly useful for writing shorter functions and handling `this` more intuitively in callbacks.

1. Shorter Syntax:

- For a single expression, then no need to `return a statement` .
- If there is only one parameter,then no need to parentheses

```
// Traditional Function
function add(a, b) {
    return a + b;
}

// Arrow Function
const add = (a, b) => a + b;
```

2.`this` Binding:

- Arrow functions do not have their own `this` context.
- This is particularly useful in cases where traditional functions create issues with `this` in callbacks.

```
function Counter() {
    this.num = 0;
    setInterval(function() {
        this.num++; // `this` refers to the global object or undefined in strict
        console.log(this.num);
    }, 1000);
}

// With Arrow Function
function Counter() {
    this.num = 0;
    setInterval(() => {
        this.num++; // `this` refers to the Counter instance
        console.log(this.num);
    }, 1000);
}
```

7. Classes in JavaScript ?

JavaScript classes, introduced with **ECMAScript 2015 (ES6)**, provide a more structured and syntactically clearer way to create and manage objects and inheritance compared to traditional constructor functions.

1. **Class Declaration:**
    - The `class` keyword is used to define a class.
    - A class body contains methods and a special method called `constructor`.
2. **Constructor Method:**
    - The `constructor` method is a special method that is called when an instance of the class is created.

```
class Person {
    name="ujjal"
    constructor(name, age) {
        this.age = age;
    }

    fun()
    {
        this.name="utpal"
        console.log(this.name)
    }
}

const person = new Person('Alice', 30);
console.log(person.name);
console.log(person.fun());
```

3 . **Methods:**

- Methods are defined inside the class body and its user defined function is also called prototype function.

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

    greet() {
        console.log(`Hello, my name is ${this.name}`);
    }
}

const person = new Person('Bob', 25);
person.greet(); // Output: Hello, my name is Bob
```

4 . **Static Methods:**

- Static methods are called on the class itself, not on instances of the class. They are defined using the `static` keyword.

```
class MathUtility {
    static add(x, y) {
        return x + y;
    }
}

console.log(MathUtility.add(5, 3)); // Output: 8
```

5. **Inheritance:**

- Classes can extend other classes using the `extends` keyword. The child class inherits methods and properties from the parent class.

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

    speak() {
        console.log(`${this.name} makes a noise.`);
    }
}

class Dog extends Animal {
    speak() {
        console.log(`${this.name} barks.`);
    }
}

const dog = new Dog('Rex');
dog.speak(); // Output: Rex barks.
```

6. **Inheritance with Super:**

- The `super` keyword is used to call methods and constructors from the parent class.
- We can super() mathed inside the child class constructor.

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

    speak() {
        console.log(`${this.name} makes a noise.`);
    }
}

class Dog extends Animal {
    constructor(name, breed) {
        super(name); // Call the parent class constructor
        this.breed = breed;
    }

    speak() {
        super.speak(); // Call the parent class method
        console.log(`${this.name} barks.`);
    }
}

const dog = new Dog('Rex', 'Labrador');
dog.speak();
```

7. **Getters and Setters:**

- Getters and setters allow you to define methods that get or set the values of properties. They are defined using the `get` and `set` keywords

```
class Rectangle {
    constructor(width, height) {
        this._width = width;
        this._height = height;
    }

    get  area() {
        return this._width * this._height;
    }

    set width(value) {
        this._width = value;
    }

    set height(value) {
        this._height = value;
    }
}

const rect = new Rectangle(5, 10);
console.log(rect.area); // Output: 50
rect.width = 7;
console.log(rect.area); // Output: 70
```

Summary.

- **Constructor** method initializes new instances of a class.
- **Methods** and **Static Methods** allow defining behaviors and functionalities within a class.
- **Inheritance** is supported using the `extends` keyword.
- **Getters** and **Setters** allow for controlled access to properties.
- **`super`** is used to call methods or constructors from a parent class.

4. Lexical Scope in JavaScript ?

Lexical scope in JavaScript refers to the visibility and accessibility of variables based on where they are defined within the code. Iand nested functions have access to variables defined in their outer (enclosing) scopes.

```
function createCounter() {

    let count = 0; // `count` is in the lexical scope of `createCounter`

    return function() {
        count++; // The returned function retains access to `count`
        console.log(count);
    };
}

const counter = createCounter();
counter(); // Output: 1
counter(); // Output: 2
```

In this example, the returned function from `createCounter` forms a closure. It retains access to the `count` variable from its lexical scope, even after `createCounter` has finished executing.

5 . Rest Operator in JavaScript ?

The **rest operator** in JavaScript is a syntax introduced with ECMAScript 2018 (ES8) that allows you to represent an indefinite number of arguments as an array. It is used to collect all remaining elements into a single array parameter in function arguments, destructuring assignments, and more.

The rest operator is denoted by three consecutive dots (`...`) followed by the name of the parameter or variable.

1.Function Parameters:

The rest operator can be used in function parameters to collect multiple arguments into an array.

```
function sum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4)); // Output: 10
console.log(sum(5, 10));      // Output: 15
```

In this example, the `...numbers` parameter collects all the arguments passed to the `sum` function into the `numbers` array.

2. Destructuring Arrays:

- The rest operator can be used to collect the remaining elements of an array into a new array during destructuring.
- It is allows you to unpack values from arrays or properties from objects into distinct variables.

```
const [first, second, ...rest] = [1, 2, 3, 4, 5];

console.log(first);  // Output: 1
console.log(second); // Output: 2
console.log(rest);   // Output: [3, 4, 5]
```

3. Destructuring Objects:

- The rest operator can be used to collect the remaining properties of an object into a new object during destructuring.

```
const { a, b, ...rest } = { a: 1, b: 2, c: 3, d: 4 };

console.log(a);     // Output: 1
console.log(b);     // Output: 2
console.log(rest);  // Output: { c: 3, d: 4 }
```

6 . Spread Operator in JavaScript ?

It allows you to expand or spread elements from an iterable (such as an array or object) into individual elements or properties. It is denoted by three consecutive dots (`...`) followed by the iterable.

```
const fruits = ['apple', 'banana', 'cherry'];
const moreFruits = ['orange', ...fruits, 'grape'];
console.log(moreFruits); // Output: ['orange', 'apple', 'banana', 'cherry', 'grape']
```

In this example, `...fruits` spreads the elements of the `fruits` array into the `moreFruits` array.

```
const person = { name: 'Alice', age: 30 };
const updatedPerson = { ...person, age: 31, city: 'New York' };
console.log(updatedPerson); // Output: { name: 'Alice', age: 31, city: 'New York' }
```

In this example, `updatedPerson` is a new object with the properties from `person` and additional or updated properties.

Shallow Copy: The spread operator creates a shallow copy of an array or object. This means that if the array or object contains nested objects or arrays, those nested structures are not deeply copied but referenced.

7 . Promise in JavaScript ?

A **Promise** in JavaScript is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises are a way to handle asynchronous operations and are part of the ECMAScript 2015 (ES6) specification.

Key Concepts :

1. States of a Promise:

- **Pending:** The initial state of a promise. The promise is neither fulfilled nor rejected.
- **Fulfilled:** The state of a promise when the asynchronous operation is completed successfully.
- **Rejected:** The state of a promise when the asynchronous operation fails.

2.Promise Object:

- The `Promise` object has methods to handle asynchronous results: `then()`, `catch()`, and `finally()`.

```
let  myPromise=new Promise((resolve, reject)=>{

    let data=true;

    if(data){
        resolve("code compaile is successfully")
    }else{
        reject("Compailatio is faiiled ");
    }
})

 myPromise
  .then(result => {
      console.log(result); // Output: Operation was successful!
  })
  .catch(error => {
      console.error(error); // Output: Operation failed!
  });
```

Handling Multiple Promises:

```
const promise1 = Promise.resolve('First');
const promise2 = Promise.resolve('Second');
const promise3 = Promise.resolve('Third');

Promise.all([promise1, promise2, promise3])
    .then(results => {
        console.log(results); // Output: ['First', 'Second', 'Third']
    })
    .catch(error => {
        console.error(error);
    });
```

```
myPromise
    .then(result => {
        console.log(result); // Output: Operation was successful!
    })
    .catch(error => {
        console.error(error);
    })
    .finally(() => {
        console.log('Promise has been settled.'); // Always runs
    });
```

Summary

- **Promise** represents the result of an asynchronous operation and can be in one of three states: pending, fulfilled, or rejected.
- You create a promise using the `Promise` constructor and handle its result with `then()`, `catch()`, and `finally()`.
- Promises can be chained and combined using methods like `Promise.all(){“any is fail then all fail ”}`, `Promise.race(){“fast one return “}`, `Promise.allSettled(){“if any how may pass or how many fail its always give result“}`
- The `finally()` method allows you to execute code regardless of the promise’s outcome.

8 . Async and await in JavaScript ?

async and **await** are modern JavaScript features introduced in ECMAScript 2017 (ES8) that simplify working with asynchronous code. They provide a cleaner syntax for handling asynchronous operations compared to using callbacks or promises directly.

```
async function fetchData() {
    try {
        const response = await fetch('https://jsonplaceholder.typicode.com/todos/1');

        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('Error fetching data:', error);
    }
}

fetchData();
```

9 . Global Function in JavaScript ?

Global functions are functions that are available globally, meaning they can be accessed and invoked from anywhere in your code.

`isFinite()`

- Determines whether a value is a finite number. Returns `true` if the value is a finite number, and `false` otherwise.

```
console.log(isFinite(123));      // Output: true
console.log(isFinite(Infinity)); // Output: false
```

`isNaN()`

- Determines whether a value is NaN (Not-a-Number). Returns `true` if the value is NaN, and `false` otherwise.

```
console.log(isNaN('string')); // Output: true
console.log(isNaN(123));      // Output: false
```

10 . Generators in JavaScript ?

Generators in JavaScript are special functions that can be paused and resumed, allowing them to yield multiple values over time. They provide a way to work with sequences of values and manage asynchronous operations more easily. Generators were introduced in ECMAScript 2015 (ES6).

```
function* myGenerator() {
    yield 1;
    yield 2;
    yield 3;
}

const gen = myGenerator();
console.log(gen.next()); // Output: { value: 1, done: false }
console.log(gen.next()); // Output: { value: 2, done: false }
console.log(gen.next()); // Output: { value: 3, done: false }
console.log(gen.next()); // Output: { value: undefined, done: true }
```

11 . Deep Copy and Shallow Copy JavaScript ?

when we copy one object to another object in this case object data is not copied object reference is copied in this case we can used :

Shallow Copy :

A shallow copy creates a new object, but only copies the top-level properties from the original object. Its work only single level .

```
const original = {  name:"Arun" , place:"kolkata" };
const shallowCopy = Object.assign({}, original);

shallowCopy.name= "utpal";

console.log(original.name); Arun
console.log(shallowCopy.name);  utpal
```

Deep Copy :

A **deep copy** creates a new object and recursively copies all properties and nested objects from the original object.

```
const original = {  name:"Arun" , place:"kolkata" };
const deepCopy = JSON.parse(JSON.stringify(original));

shallowCopy.name= "utpal";

console.log(original.name); Arun
console.log(shallowCopy.name);  utpal
```

Here we have a problem that function is remove here :

```
const _ = require('lodash');

const original = { a: 1, b: { c: 2 } };
const deepCopy = _.cloneDeep(original);

deepCopy.b.c = 3;

console.log(original.b.c); // Output: 2 (original object is not affected)
```


Thank
You