**List the features of ES6**

**Key Features :-**

1. **Let and Const**  
   Let and const are two new ways to declare variables.

Let allows you to declare variables that are limited in scope to the block where they are used.

Const is used to declare constants, which means the value cannot be reassigned once set.

1. **Arrow Functions**  
   Arrow functions provide a shorter syntax for writing functions.  
   They do not have their own 'this' context, which helps avoid some common issues in callbacks.
2. **Template Literals**  
   Template literals allow you to work with strings in a more flexible way.  
   You can use backticks and embed variables or expressions using ${} syntax.
3. **Default Parameters**  
   Functions can now have default values for parameters.  
   If a value is not passed, the default will be used.
4. **Destructuring Assignment**  
   You can extract values from arrays or objects into individual variables.  
   This makes code cleaner and easier to read.
5. Enhanced Object Literals  
   ES6 provides shortcuts when defining objects.  
   You can write property names without repeating variable names, and use shorthand for methods.
6. **Classes**  
   ES6 introduced a simpler syntax for creating objects and dealing with inheritance using the class keyword.  
   This is syntactic sugar over the existing prototype-based inheritance.
7. **Promises**  
   Promises are used for handling asynchronous operations.  
   They allow cleaner code than traditional callback functions.
8. **Modules**  
   JavaScript now supports modular programming using export and import statements.  
   This allows splitting code into reusable files.
9. **Spread and Rest Operators**  
   The spread operator (...) allows elements of an array or object to be expanded.  
   The rest operator collects multiple elements into a single array.
10. For...of Loop  
    A new loop for iterating over iterable objects like arrays, strings, etc.  
    It is easier to use than the traditional for loop.
11. Map and Set  
    New collection types:

* Map is a key-value collection where keys can be any type.
* Set stores unique values of any type.

1. Symbol  
   A new primitive data type that represents unique and immutable identifiers.  
   Useful for creating private object properties.
2. Generators  
   Functions that can be paused and resumed using the yield keyword.  
   Useful for managing asynchronous flows or implementing iterators.

**Explain JavaScript let**

The let keyword in JavaScript is used to declare variables with **block scope**. It was introduced in **ES6 (ECMAScript 2015)** to fix several issues that arose with the older var keyword.

**Key Characteristics of let:**

1. **Block Scoped**  
   A variable declared with let is only available **within the block** (curly braces {}) where it is defined.

{

let x = 10;

console.log(x); // 10

}

console.log(x); // ReferenceError: x is not defined

1. **No Hoisting Like var**  
   let declarations are hoisted to the top of their block, but **not initialized**, meaning you cannot use them before the line they’re declared.

console.log(a); // ReferenceError

let a = 5;

1. **No Re-declaration in the Same Scope**  
   Unlike var, you **cannot declare the same variable twice** in the same scope with let.

let x = 5;

let x = 10; // SyntaxError: Identifier 'x' has already been declared

1. **Can Be Updated**  
   Variables declared with let can be reassigned new values.

let score = 20;

score = 30; // valid

1. **Works Well in Loops**  
   let is often used in for loops to avoid scope-related bugs.

for (let i = 0; i < 3; i++) {

console.log(i); // 0, 1, 2

}

console.log(i); // ReferenceError: i is not defined

**Identify the differences between var and let**

1 Scope  
var is function-scoped, which means the variable is accessible throughout the entire function in which it is declared. let is block-scoped, meaning it is only accessible within the block (enclosed by curly braces) where it is defined. This makes let more suitable for modern programming where limited scope is often preferred.

2 Hoisting  
Both var and let declarations are hoisted to the top of their scope. However, var is initialized with undefined, so it can be accessed before its declaration without throwing an error, though the value will be undefined. let is hoisted but not initialized, so accessing it before declaration results in a ReferenceError.

3 Re-declaration  
Using var, you can re-declare a variable in the same scope without an error. This can lead to bugs in large or complex code. With let, re-declaring a variable in the same scope is not allowed and will throw a SyntaxError, making the code more predictable and safer.

4 Use in loops  
let creates a new variable binding in each iteration of a loop, which works well with closures and asynchronous code. var does not create a new binding per iteration, which can lead to unexpected results when used inside loops with callbacks.

5 Global object property  
If you declare a variable with var in the global scope, it becomes a property of the global object (like window in browsers). A variable declared with let does not become a global object property, which helps avoid conflicts and unintended behavior.

**Explain JavaScript const**

**JavaScript const**

The const keyword in JavaScript is used to declare variables with a constant value. Introduced in **ES6 (**ECMAScript 2015**)**, it ensures that a variable cannot be reassigned after its initial assignment.

**Key Characteristics of const:**

1. **Block Scope**  
   Just like let, const is **block-scoped**. This means the variable is only accessible within the block ({}) in which it is declared.

{

const message = "Hello";

console.log(message); // Hello

}

console.log(message); // ReferenceError

1. **Must Be Initialized**  
   A const variable must be assigned a value at the time of declaration. If not, it will throw an error.

const a; // SyntaxError: Missing initializer

const b = 10; // valid

1. **Cannot Be Reassigned**  
   Once a value is assigned to a const variable, it cannot be reassigned.

const x = 5;

x = 10; // TypeError: Assignment to constant variable

1. **Not Immutable (for Objects and Arrays)**  
   While const prevents reassignment of the variable, it does not make objects or arrays immutable. You can still modify the contents of an object or array declared with const.

const arr = [1, 2, 3];

arr.push(4); // valid

arr[0] = 100; // valid

arr = [5, 6, 7]; // TypeError: Assignment to constant variable

const person = { name: "John" };

person.name = "Jane"; // valid

person = {}; // TypeError

1. **Useful for Constants and Fixed References**  
   Use const when you are sure the reference to a value, object, or array should not change throughout the program.

**Explain ES6 class fundamentals**

ES6 introduced a new and more intuitive way to work with object-oriented programming in JavaScript through the class syntax. This feature provides a cleaner, more familiar structure for developers coming from languages like Java or C++, though under the hood, JavaScript still uses its prototype-based inheritance. Classes in ES6 are primarily used to create objects and manage inheritance in a more readable and maintainable way.

A class is defined using the class keyword followed by a name. Inside the class body, there is typically a constructor() method, which is a special function used to initialize new objects created with the class. Only one constructor is allowed per class. You can also define other methods inside the class without using the function keyword. These methods are added to the class’s prototype and shared by all instances of the class, helping reduce memory usage.

ES6 classes also support inheritance using the extends keyword. This allows one class to inherit the properties and methods of another class. The super() function is used within the child class constructor to call the parent class’s constructor. This inheritance structure makes it easier to reuse code and create more complex object models.

In addition to regular methods, ES6 classes support getters and setters, which allow you to define custom behavior when accessing or updating properties. You can also define static methods, which are called on the class itself rather than on instances of the class. These are useful for utility functions that don’t depend on instance-specific data

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

greet() {

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

}

}

**Explain ES6 class inheritance**

ES6 introduced a simpler and clearer syntax for implementing inheritance in JavaScript using the extends and super keywords. Inheritance allows one class (called the subclass or child class) to inherit properties and methods from another class (called the superclass or parent class). This feature promotes code reuse and helps organize related functionality in a structured way.

To create a subclass, you use the extends keyword followed by the name of the parent class. Inside the child class, the constructor() function can call the parent class’s constructor using the super() function. This is required before you can use this in the subclass constructor. The super() method passes arguments to the parent constructor and ensures the parent’s properties are properly initialized.

The child class can also define its own properties and methods, and it can override methods inherited from the parent class. This allows you to extend or modify the behavior of the base class while keeping shared functionality in one place. Additionally, within overridden methods, you can still call the original method from the parent using super.methodName().

Class inheritance in ES6 provides a cleaner and more familiar way to build complex object hierarchies compared to the older prototype-based syntax. Although the underlying inheritance mechanism is still prototype-based, the new syntax makes it easier to understand, write, and maintain object-oriented code in JavaScript.

Example :-

// Parent class

class Animal {

constructor(name) {

this.name = name;

}

speak() {

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

}

}

// Child class

class Dog extends Animal {

constructor(name, breed) {

super(name); // Call parent class constructor

this.breed = breed;

}

speak() {

console.log(`${this.name} barks. It is a ${this.breed}.`);

}

}

// Creating an object of Dog class

const dog1 = new Dog("Max", "Labrador");

dog1.speak(); // Output: Max barks. It is a Labrador.

**Define ES6 arrow functions**

**1. ES6 Arrow Functions**

Arrow functions are a concise way to write function expressions introduced in ES6.

**Syntax:**

// Traditional function

function add(a, b) {

return a + b;

}

// Arrow function equivalent

const add = (a, b) => a + b;

**Key features:**

* Shorter syntax
* Implicit return when using a single expression (no need for return keyword)
* No own this context — inherits from the surrounding scope

**Examples:**

// No parameters

const greet = () => console.log("Hello!");

// One parameter (parentheses optional)

const square = x => x \* x;

// Multiple statements (requires curly braces and explicit return)

const multiply = (a, b) => {

const result = a \* b;

return result;

};

**Identify set(), map()**

**Set**

* A collection of **unique values** — no duplicates allowed.
* Values can be of any type: primitive or object.
* Order of insertion is preserved.

**Example:**

const numbers = new Set();

numbers.add(1);

numbers.add(2);

numbers.add(2); // Duplicate, will be ignored

console.log(numbers); // Output: Set {1, 2}

console.log(numbers.has(1)); // true

numbers.delete(2);

console.log(numbers.size); // 1

Use **Set** when you want to store unique items.

**Map**

* A collection of **key-value pairs**, similar to objects.
* Keys can be **any type** (not just strings or symbols, like in regular objects).
* Maintains the order of insertion.

**Example:**

const map = new Map();

map.set('name', 'Virat Kohli');

map.set(1, 'One');

map.set(true, 'Boolean key');

console.log(map.get('name')); // 'Virat Kohli'

console.log(map.has(1)); // true

map.delete(true);

console.log(map.size); // 2

Use **Map** when you need keys other than strings, or want guaranteed key insertion order.

**Create a React Application named “cricketapp” with the following components**

**Index.js :-**

import React from 'react';

import ReactDOM from 'react-dom/client';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

  <React.StrictMode>

    <App />

  </React.StrictMode>

);

reportWebVitals();

**App.js :-**

// src/App.js

import React, { useState } from 'react';

import ListofPlayers from './components/ListofPlayers';

import IndianPlayers from './components/IndianPlayers';

function App() {

  const [flag, setFlag] = useState(true); // change true/false to toggle

  return (

    <div className="App" style={{ padding: "20px", fontFamily: "Arial" }}>

      <h1>Cricket App</h1>

      <button onClick={() => setFlag(!flag)}>

        Toggle Flag (Currently: {flag.toString()})

      </button>

      {flag ? <ListofPlayers /> : <IndianPlayers />}

    </div>

  );

}

export default App;

**IndianPlayers.js :-**

// src/components/IndianPlayers.js

import React from 'react';

const IndianPlayers = () => {

  // Array of players, assuming first player is odd index 0, second is even index 1, etc.

  const players = [

    "Virat Kohli",

    "Rohit Sharma",

    "KL Rahul",

    "Jasprit Bumrah",

    "Hardik Pandya",

    "Shikhar Dhawan"

  ];

  // Destructure odd and even players

  // Odd: 0, 2, 4 | Even: 1, 3, 5

  const oddTeamPlayers = players.filter((\_, index) => index % 2 === 0);

  const evenTeamPlayers = players.filter((\_, index) => index % 2 !== 0);

  // Two arrays of players

  const T20players = ["Suryakumar Yadav", "Ishan Kishan"];

  const RanjiTrophyPlayers = ["Cheteshwar Pujara", "Hanuma Vihari"];

  // Merge arrays using spread operator

  const mergedPlayers = [...T20players, ...RanjiTrophyPlayers];

  return (

    <div>

      <h2>Indian Players</h2>

      <h3>Odd Players</h3>

      <ul>

        {oddTeamPlayers.map((player, index) => (

          <li key={index}>{player}</li>

        ))}

      </ul>

      <hr/>

      <h3>Even Players</h3>

      <ul>

        {evenTeamPlayers.map((player, index) => (

          <li key={index}>{player}</li>

        ))}

      </ul>

        <hr/>

      <h3>List of Indian Players Merged</h3>

      <ul>

        {mergedPlayers.map((player, index) => (

          <li key={index}>{player}</li>

        ))}

      </ul>

    </div>

  );

};

export default IndianPlayers;

**ListofPlayers.js :-**

// src/components/ListofPlayers.js

import React from 'react';

const ListofPlayers = () => {

  const players = [

    { name: "Virat Kohli", score: 85 },

    { name: "Rohit Sharma", score: 72 },

    { name: "KL Rahul", score: 65 },

    { name: "Jasprit Bumrah", score: 55 },

    { name: "Hardik Pandya", score: 90 },

    { name: "Shikhar Dhawan", score: 60 },

    { name: "Rishabh Pant", score: 68 },

    { name: "MS Dhoni", score: 75 },

    { name: "Bhuvneshwar Kumar", score: 40 },

    { name: "Yuzvendra Chahal", score: 50 },

    { name: "Ravindra Jadeja", score: 66 }

  ];

  // Filter players with score below 70 using arrow function

  const lowScorers = players.filter(player => player.score < 70);

  return (

    <div>

      <h2>List of Players</h2>

      <ul>

        {players.map(({ name, score }, index) => (

          <li key={index}>{name} - {score}</li>

        ))}

      </ul>

        <hr/>

      <h3>List of Players having score less than 70</h3>

      <ul>

        {lowScorers.map(({ name, score }, index) => (

          <li key={index}>{name} - {score}</li>

        ))}

      </ul>

      <hr/>

    </div>

  );

};

export default ListofPlayers;





