1. List the Features of ES6

ES6 brought a wealth of improvements, including but not limited to:

- **`let` and `const` declarations:** New ways to declare variables with improved scoping rules.
- **Arrow Functions:** A concise syntax for writing function expressions.
- **Classes:** Syntactic sugar over JavaScript's prototype-based inheritance, making object-oriented programming cleaner.
- **Modules:** Native support for organizing code into separate files and managing dependencies (import/export).
- **Template Literals:** Enhanced string literals allowing for easier string interpolation and multi-line strings.
- **Destructuring Assignment:** A convenient way to extract values from arrays or properties from objects into distinct variables.
- **Default Parameters:** Ability to set default values for function parameters.
- **Rest and Spread Operators:** Powerful operators for handling array and object elements.
- **Promises:** A more robust way to handle asynchronous operations.
- **Iterators and For...of loops:** New protocols for iterating over data structures.
- **Generators:** Functions that can be paused and resumed.
- **`Map` and `Set` Data Structures:** New collections for storing key-value pairs and unique values, respectively.
- **Symbols:** A new primitive data type.

---

2. Explain JavaScript `let`

`let` is a keyword introduced in ES6 for declaring variables. Unlike `var`, `let` provides **block-scoped** variable declarations. This means a variable declared with `let` is only accessible within the block (e.g., inside `if` statements, `for` loops, or any `{}` block) where it is defined, and not outside of it.

Example:

```
function exampleLet() {
  let x = 10;
  if (true) {
    let y = 20;
    console.log(x); // Output: 10 (x is accessible)
    console.log(y); // Output: 20 (y is accessible within this block)
  }
  console.log(x); // Output: 10 (x is accessible)
  // console.log(y); // Error: y is not defined (y is block-scoped)
}

exampleLet();
```

3. Identify the Differences between `var` and `let`

The primary differences between `var` and `let` revolve around **scoping**, **hoisting**, and **re-declaration**.

Feature	`var`	`let`
Scoping	**Function-scoped** or global-scoped.	**Block-scoped**.
Hoisting	Variables are hoisted to the top of their function or global scope and initialized with `undefined`.	Variables are hoisted to the top of their block, but they are *not* initialized. Accessing them before declaration results in a `ReferenceError` (Temporal Dead Zone).
Re-declaration	Can be re-declared within the same scope without error.	Cannot be re-declared within the same block scope.
Global Object	In global scope, `var` declarations become properties of the `window` object (in browsers).	In global scope, `let` declarations do *not* become properties of the `window` object.

Export to Sheets

Example (Hoisting & Re-declaration):

```
// var example

console.log(a); // Output: undefined (hoisted and initialized)

var a = 10;

var a = 20; // No error, re-declared

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


// let example

// console.log(b); // Error: Cannot access 'b' before initialization (Temporal Dead Zone)

let b = 10;

// let b = 20; // Error: Identifier 'b' has already been declared

console.log(b); // Output: 10
```

4. Explain JavaScript `const`

`const` is another variable declaration keyword introduced in ES6, similar to `let` in terms of **block-scoping** and **Temporal Dead Zone**. The key difference is that `const` stands for "constant."

- **Immutable Binding:** A variable declared with `const` must be initialized at the time of declaration and **cannot be reassigned** to a new value later.
- **Object Mutability:** While the *binding* itself is constant (you can't assign a different object/array to the `const` variable), if the value is an object or an array, its *properties* or *elements* **can still be modified**.

Example:

```
// Primitive value
const PI = 3.14159;
// PI = 3.14; // Error: Assignment to constant variable.


// Object
const person = {
  name: "Alice",
  age: 30
};
person.age = 31; // No error, object property can be modified
console.log(person.age); // Output: 31
// person = { name: "Bob", age: 25 }; // Error: Assignment to constant variable.


// Array
const numbers = [1, 2, 3];
numbers.push(4); // No error, array content can be modified
console.log(numbers); // Output: [1, 2, 3, 4]
// numbers = [5, 6]; // Error: Assignment to constant variable.
```

5. Explain ES6 Class Fundamentals

ES6 `classes` are a syntactic sugar over JavaScript's existing prototype-based inheritance. They provide a cleaner, more object-oriented syntax for creating objects and dealing with inheritance, making JavaScript code look more like traditional class-based languages (e.g., Java, C++).

Key Fundamentals:

- **`class` keyword:** Used to declare a class.
- **`constructor` method:** A special method for creating and initializing an object created with a class. It's automatically called when you create a new instance using the `new` keyword.
- **Methods:** Functions defined inside the class that operate on the instance's data.
- **`this` keyword:** Refers to the instance of the class.

Example:

```
class Animal {
  // Constructor method to initialize properties
  constructor(name, sound) {
    this.name = name;
    this.sound = sound;
  }

  // Method defined within the class
  makeSound() {
    console.log(`${this.name} says ${this.sound}`);
  }

  // Another method
  greet() {
    console.log(`Hello, I am ${this.name}.`);
  }
}

// Creating an instance of the class
const dog = new Animal("Buddy", "Woof");
dog.makeSound(); // Output: Buddy says Woof
dog.greet();     // Output: Hello, I am Buddy.

const cat = new Animal("Whiskers", "Meow");
```

```
cat.makeSound(); // Output: Whiskers says Meow
```

6. Explain ES6 Class Inheritance

Class inheritance in ES6 allows one class (the **child** or **derived** class) to extend another class (the **parent** or **base** class), inheriting its properties and methods. This promotes code reusability and forms "is-a" relationships (e.g., a `Dog` IS-A `Animal`).

Key Keywords:

- **`extends` keyword:** Used by the child class to inherit from the parent class.
- **`super()` keyword:** Used within the child class's constructor to call the parent class's constructor. This *must* be called before using `this` in the child constructor. It can also be used to call methods on the parent object.

Example:

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

  makeSound() {
    console.log(`${this.name} says ${this.sound}`);
  }

  eat() {
    console.log(`${this.name} is eating.`);
  }
}

// Dog is a child class that extends Animal
class Dog extends Animal {
  constructor(name, sound, breed) {
    // Call the parent class's constructor
```

```
        super(name, sound);
        this.breed = breed;
    }

    // New method specific to Dog
    fetch() {
        console.log(`${this.name} is fetching!`);
    }

    // Override parent method
    makeSound() {
        console.log(`${this.name} barks: ${this.sound}!`);
    }
}

const myDog = new Dog("Max", "Woof", "Golden Retriever");
myDog.makeSound(); // Output: Max barks: Woof! (Overridden method)
myDog.eat();       // Output: Max is eating. (Inherited method)
myDog.fetch();     // Output: Max is fetching! (Dog-specific method)
```

7. Define ES6 Arrow Functions

Arrow functions provide a more concise syntax for writing function expressions in ES6. They are often preferred for their brevity and how they handle the `this` keyword.

Syntax:

```
// Basic syntax
(parameters) => expression

// With a single parameter (parentheses optional)
parameter => expression
```

```
// With no parameters
() => expression


// With a block body (requires explicit return)
(parameters) => {
  // statements
  return value;
}
```

Key Characteristics:

- **Concise Syntax:** Especially for short, single-expression functions.
- **No `this` binding:** Arrow functions do not have their own `this` context. Instead, `this` inside an arrow function refers to the `this` of the *enclosing lexical context* (where the arrow function is defined). This solves common `this` binding issues in traditional function expressions.
- **No `arguments` object:** They do not have their own `arguments` object.
- **Not suitable for constructors:** They cannot be used as constructors with the `new` keyword.

Example:

```
// Traditional function expression
const addOld = function(a, b) {
  return a + b;
};
console.log(addOld(2, 3)); // Output: 5


// Arrow function (concise)
const add = (a, b) => a + b;
console.log(add(2, 3)); // Output: 5


// Arrow function with no parameters
const greet = () => "Hello!";
console.log(greet()); // Output: Hello!


// Arrow function with single parameter
```

```
const square = x => x * x;
console.log(square(4)); // Output: 16

// Arrow function with block body and explicit return
const multiply = (a, b) => {
  const result = a * b;
  return result;
};
console.log(multiply(4, 5)); // Output: 20

// 'this' binding example
class MyComponent {
  constructor() {
    this.value = 10;
  }

  // Arrow function: 'this' is lexically bound to MyComponent instance
  handleClick() {
    setTimeout(() => {
      console.log(this.value); // 'this' refers to MyComponent instance
    }, 100);
  }
}
const comp = new MyComponent();
comp.handleClick(); // Output: 10 (after 100ms)
```

8. Identify `Set()`, `Map()`

ES6 introduced two new built-in data structures: `Set` and `Map`, offering more efficient and structured ways to manage collections of data compared to plain JavaScript objects or arrays for certain use cases.

a. `Set()`

A `Set` is a collection of **unique values**. It lets you store any type of value, whether primitive values or object references.

Key Features:

- **Uniqueness:** A value can only occur once in a `Set`. Adding a duplicate value has no effect.
- **No Order:** Elements in a Set do not have an index or maintain insertion order in a guaranteed way (though implementations often do).
- **Methods:**
  - `new Set()`: Creates a new Set.
  - `add(value)`: Adds a new element to the Set.
  - `delete(value)`: Removes an element from the Set.
  - `has(value)`: Checks if a value exists in the Set (returns boolean).
  - `clear()`: Removes all elements from the Set.
  - `size`: Returns the number of elements in the Set.
  - `forEach()`, `for...of`: Can be iterated over.

Example:

```
const uniqueNumbers = new Set();
uniqueNumbers.add(1);
uniqueNumbers.add(2);
uniqueNumbers.add(1); // Duplicate, will not be added again
uniqueNumbers.add(3);

console.log(uniqueNumbers); // Output: Set { 1, 2, 3 }
console.log(uniqueNumbers.size); // Output: 3
console.log(uniqueNumbers.has(2)); // Output: true
console.log(uniqueNumbers.has(4)); // Output: false

uniqueNumbers.delete(2);
console.log(uniqueNumbers); // Output: Set { 1, 3 }

// Converting array to Set to remove duplicates
const numbersArray = [1, 2, 2, 3, 4, 4, 5];
const uniqueArray = [...new Set(numbersArray)]; // Using spread operator
console.log(uniqueArray); // Output: [1, 2, 3, 4, 5]
```

b. `Map()`

A `Map` is a collection of **key-value pairs**. It is similar to an object, but with some crucial differences:

Key Features:

- **Any Data Type as Key:** Unlike plain objects where keys are implicitly converted to strings, `Map` allows keys of any data type (including objects, functions, or any primitive value).
- **Maintains Insertion Order:** `Map` objects iterate their elements in insertion order.
- **Methods:**
  - `new Map()`: Creates a new Map.
  - `set(key, value)`: Adds or updates a key-value pair.
  - `get(key)`: Retrieves the value associated with a key.
  - `delete(key)`: Removes a key-value pair.
  - `has(key)`: Checks if a key exists (returns boolean).
  - `clear()`: Removes all key-value pairs.
  - `size`: Returns the number of key-value pairs.
  - `forEach()`, `for...of`: Can be iterated over.

Example:

```
const userRoles = new Map();


// Setting key-value pairs
userRoles.set("admin", "John Doe");
userRoles.set("editor", "Jane Smith");
userRoles.set(123, "Guest User"); // Using a number as a key


const userObject = { id: 1, name: "Alice" };
userRoles.set(userObject, "Registered User"); // Using an object as a key


console.log(userRoles); // Output: Map { 'admin' => 'John Doe', 'editor' => 'Jane Smith', 123 =>
'Guest User', { id: 1, name: 'Alice' } => 'Registered User' }


// Getting values
console.log(userRoles.get("admin")); // Output: John Doe
console.log(userRoles.get(123));   // Output: Guest User
console.log(userRoles.get(userObject)); // Output: Registered User
```

```
// Checking existence
console.log(userRoles.has("editor")); // Output: true
console.log(userRoles.has("viewer")); // Output: false

console.log(userRoles.size); // Output: 4

userRoles.delete("editor");
console.log(userRoles.size); // Output: 3

// Iterating over a Map
for (const [key, value] of userRoles) {
   console.log(`${key}: ${value}`);
}
// Output:
// admin: John Doe
// 123: Guest User
// [object Object]: Registered User (if userObject is logged directly)
```