JavaScript

1. **WHAT ARE THE DATA TYPES IN JAVASCRIPT?**

There are 8 datatypes divided into 2 categories

- Primitive = number, string, Boolean, undefinedValue, nullValue, symbol, bigInt
- Non-Primitive = Object

```
let number = 42;
let string = "Hello";
let boolean = true;
let undefinedValue;
let nullValue = null;
let symbol = Symbol("unique");
let bigInt = 1234567890123456789012345678901234567890n;
let object = { name: "John", age: 30 };
```

2. **WHAT IS THE DIFFERENCE BETWEEN == AND ===

- == (equality) also called the loose equality operator, first tries to **convert the operands to the same type** and then compares their values. This is called type coercion
- = (strict equality) does not perform type coercion. It checks both **value and datatype** directly. For example,
  ==ex: `console.log(5 == "5")`

3. **WHAT IS THE DIFFERENCE BETWEEN NULL & UNDEFINED**

- Null and undefined both are used to represent the absence of a value.
- Null - Must be explicitly assigned. Typeof null returns "object" (this is a known JavaScript Bug)

- Undefined: Represents a variable that has been declared but not assigned a value. Typeof undefined returns "undefined"
  ex:

```
let nullVal = null
console.log(nullVal) // null
console.log(typeof null) // "object"
let undefinedVar
console.log(undefinedVar) // undefined
console.log(typeof undefined) // "undefined"
```

4. **EXPLAIN THE CONCEPT OF HOISTING IN JAVASCRIPT**

It's a behavior of JavaScript where Variables and function declarations are moved to top of their respective scopes during the compilation phase before the code is executed. This means that regardless of where variables and functions are declared in the code, they are treated as if they are declared at the beginning of their scope.
ex:

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

- Here the declaration of x is hoisted to the top but not its initialization. That's why first `console.log` output is undefined.

ex: (Function hoisting: Function declarations move to the top of their scope during the compilation phase.)

```
sayHello() // Output: "Hello!"
// Function declaration is hoisted to the top of its scope
function sayHello(){
    console.log('Hi")
}

sayHi(); // "TypeError: sayHi is not a function"
var sayHi = function(){
```

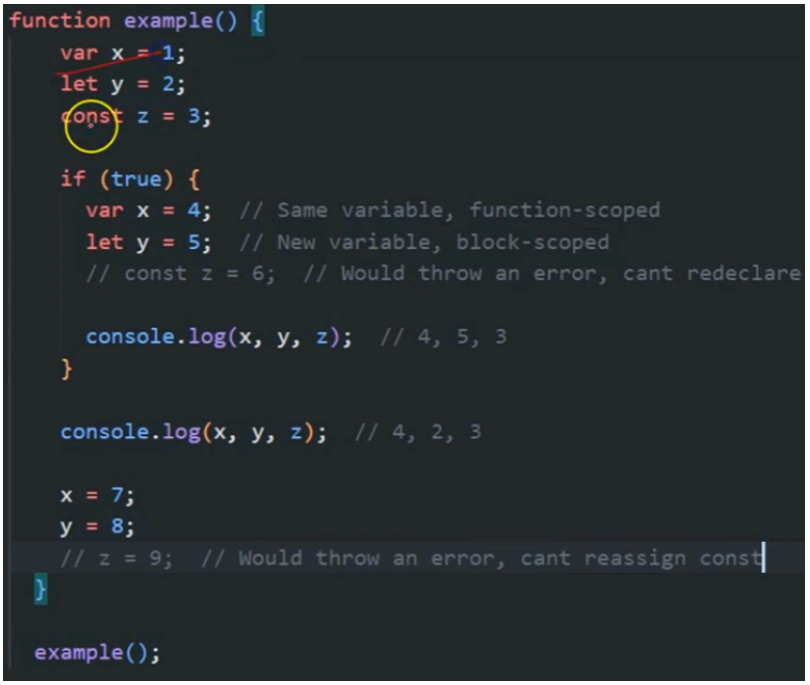
```
    console.log("Hi")
}
```

- 'LET' and 'CONST' declarations are hoisted but not initialized. This leads to a 'temporal dead zone' where accessing the variable before its declaration results in a ReferenceError. For example:

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

5. **WHAT IS THE DIFFERENCE BETWEEN LET, CONST AND VAR?**

- VAR: Function Scoped or Globally scoped. It can be redeclared and updated and hoisted and initialized with undefined.
- LET: Block Scoped. Hoisted but not initialized which leads to Temporal Dead Zone.
- CONST: Block Scoped. Cannot be updated/redeclared. Hoisted but not initialized which leads to Temporal Dead Zone.
  For example:



6. **WHAT IS VARIABLE SCOPE IN JAVASCRIPT?**

Variable scope refers to the context in which a variable is declared and can be accessed. In JS, there are 2 main types of scope:

- Global Scope:
  - Variables declared outside any function or block.
  - Accessible from anywhere in the code, including inside functions.
  - Have global scope.
- Local Scope:
  - Variables declared inside a function or block
  - Only accessible within that function or block.
  - Have local scope
  JavaScript uses lexical scoping, which means that inner functions have access to variables in their outer scope. For example

```
let globalVar = "I am global";
function exampleFunction(){
    let localVar = "I am local";
    console.log(globalVar); // Output: "I am global"
    console.log(localVar); // Output: "I am local"
}
exampleFunction();
console.log(globalVar); // Output: "I am global"
console.log(localVar); // Output: "Reference Error: localVar is not defined"
```

7. **WHAT IS TEMPORAL DEAD ZONE**

The Temporal Dead Zone refers to the time between **entering a scope** and the point where a variable declared with `let` or `const` is actually initialized. During this period, the variable exists in memory but cannot be accessed. If I try, I'll get a `ReferenceError`. For example:

```
console.log(a); // ReferenceError
let a = 10;
```

- Here, the scope knows that `a` is declared, but until the line `let a = 10` is executed, the variable is in the TDZ.
- This behavior doesn't apply to `var`, because variables declared with `var` are hoisted and initialized with `undefined`.

- The TDZ is useful because it prevents accidental usage of variables before they are properly declared, which makes the code safer and less error-prone.

8. **WHAT IS VARIABLE SHADOWING**

It occurs when a variable declared in a certain scope has the same name as a variable in an outer scope. The inner variable "shadows" the outer one, effectively hiding it.
ex: ```

```
let x = 10
function example(){
    let x = 20 // This x shadows the outer x
    console.log(x) // 20
    if (true){
        let x = 30; // This x shadows both outer x variable
        console.log(x) // 30
    }
    console.log(x) // 20
}
example()
console.log(x) // 10
```

9. **WHAT IS CLOSURE IN JS**

A closure in JavaScript is a function that has access to variables in its outer lexical scope even after the outer function has returned. Closures are created every time a function is created.

ex:

```
// Outer function that creates a closure
function createGreeter(greeting) {
    // This variable is enclosed in the closure
    let count = 0;
    // Inner function that forms a closure
    return function(name) {
      // Accessing the enclosed variables (greeting and count)
      count++;
      console.log(`${greeting}, ${name}! This greeting has been used ${count} time(s).`);
    };
  }
  const casualGreeter = createGreeter("Hi");
  const formalGreeter = createGreeter("Good day");

  // Using the greeter functions
  casualGreeter("Alice");
  // Output: Hi, Alice! This greeting has been used 1 time(s).
  casualGreeter("Bob");
  // Output: Hi, Bob! This greeting has been used 2 time(s).

  formalGreeter("Charlie");
  // Output: Good day, Charlie! This greeting has been used 1 time(s).
  casualGreeter("David");
  // Output: Hi, David! This greeting has been used 3 time(s).
```

- The createGreeter function returns an inner function which forms a closure.
- The inner function has access to the `greeting` parameter and outer variables like `count`.
- Each time we call createGreeter, it creates a new closure with its own enclosed `count` variable.
- The returned functions (casualGreeter and formalGreeter) maintain their own separate counts, demonstrating how closures preserve state.

10. **DIFFERENT WAYS TO DEFINE A FUNCTION IN JS?**

```
// Function Declaration
function greet(name){
    return 'Hello'
}

// Function Expression
const greet = function (name){
    return 'Hello'
}

// Arrow Function
const greet = (name) => 'Hello'
```

11. **WHAT IS HIGHER ORDER FUNCTION?**

It is a function that treats other functions as data, either by taking them as arguments or returning them.
ex:

```
// Higher-order function that takes a function as an argument
function operate(a, b, operation){
    return operation(a, b);
}
const add = (a, b) => a + b;
console.log(operate(5, 3, add));
```

12. **WHAT IS A PURE FUNCTION?**

A pure function is a function that: Always returns the same output for the same inputs. Has no side effects (doesn't modify external state). Doesn't rely on external state. Make code more predictable and easier to test.

```
// Pure function: Always returns the same output for the same
inputs
function add(a, b){ return a + b; }

// Impure functions: Modifies external state
let total = 0;
function addToTotal(value){
    total += value; // Modifies external variable 'total'
    return total; // Return value depends on external state
}
```

13. **DIFFERENCE BETWEEN FUNCTION DECLARATION AND FUNCTION EXPRESSION?

- **Hoisting:** Function declarations are hoisted, function expressions are not
- **Usage:** Function declarations can be called before they appear in the code, function expressions cannot.
- **Naming:** Function declarations require a name, function expressions can be anonymous

```
// Function Declaration
sayHello();
function sayHello() { console.log("Hello"); }

// Function Expression
// greet(); // Error
const greet = function (){ console.log("Greeting"); }
greet(); // Works when called after the expression
```

14. WHAT IS IMMEDIATELY INVOKED FUNCTION EXPRESSION (IIFE)?

An IIFE is a JavaScript function that runs as soon as it is defined.
ex:

```
// IIFE with parameters
(function (name){
    console.log(`Hello ${name}`);
})("Ankit")

// Arrow function IIFE
(() => {
    console.log("IIFE called")
})()
```

15. HOW DO YOU CREATE AN OBJECT IN JAVASCRIPT?

```
// Object literal notation
let person1 = {
    name: "Alice",
    age: 30,
    greet: function() {
      console.log(`Hello, I'm ${this.name}`);
    }
  };

  // Constructor function
  function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
      console.log(`Hello, I'm ${this.name}`);
    };
  }
  let person2 = new Person("Bob", 25);
```

```
// Object.create() method
let personProto = {
  greet: function() {
    console.log(`Hello, I'm ${this.name}`);
  }
};
let person3 = Object.create(personProto);
person3.name = "Charlie";
person3.age = 35;

// ES6 class syntax
class PersonClass {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log(`Hello, I'm ${this.name}`);
  }
}
let person4 = new PersonClass("David", 40);
```

16. **HOW DO YOU ADD/REMOVE PROPERTIES TO AN OBJECT DYNAMICALLY ?**

---

ex:

```
let car = {
    brand: "Toyota",
}
// Adding properties
car.year = 2022 // Dot notation
car["color"] = "blue" // Bracket notation

// Removing Properties
delete car.brand // Removes the 'brand' property
console.log(car.brand !== undefined) // true -> checks if a
property exists in an object. Here is will return false.
```

17. **HOW DO YOU CHECK IF A PROPERTY EXISTS IN AN OBJECT?**

```javascript
let person = {
  name: "Alice",
  age: 30,
};

// Using the in operator
console.log("name" in person); // true
console.log("job" in person); // false

// Using hasOwnProperty method
console.log(person.hasOwnProperty("age")); // true
console.log(person.hasOwnProperty("city")); // false

// Using undefined check
console.log(person.name !== undefined); // true
console.log(person.salary !== undefined); // false

// Using Optional Chaining
console.log(person?.job); // undefined
```

18. **WHAT IS 'this' KEYWORD IN JAVASCRIPT?**

The `this` keyword refers to the object that is executing the current function. It's value is determined by how a function is called. For example:

```javascript
// In a method
let person = {
  name: "Alice",
  greet() {
    console.log(`Hello, I'm ${this.name}`);
  },
};
person.greet(); // "Hello, I'm Alice"

// In an arrow function
let arrowGreet = () => {
  console.log(`Hello, ${this.name}`);
};
arrowGreet.call({ name: "Charlie" });
// "Hello, undefined" (arrow functions don't bind their own 'this')

// In a constructor function
function Person(name) {
  this.name = name;
  this.greet = function () {
    console.log(`Hello, I'm ${this.name}`);
  };
}
let david = new Person("David");
david.greet(); // "Hello, I'm David"
```

19. **WHAT RE THE DIFFERENT WAYS TO LOOP THROUGH N ARRAY IN JAVASCRIPT?**

```
const fruits = ["apple", "banana", "orange"]
// 1. for loop
for(let i = 0; i<fruits.length; i++){
    console.log(fruits[i]);
}

// 2. forEach method
fruits.forEach((fruit, index) => {
    console.log(`${index}: ${fruit}`);
})

// 3. for ... of loop (ES6+)
for(const fruit of fruits){
    console.log(fruit)
}

// 4. map method (creates a new array)
const upperFruits = fruits.map((fruit) => fruit.toUpperCase());
console.log(upperFruits)

// 5. while loop
let i = 0;
while(i < fruits.length){
    console.log(fruits[i]); i++;
}

// 6. do while loop
let j = 0;
do {
    console.log(fruits[j]); j++;
} while(j < fruits.length)
```

20. EXPLAIN THE DIFFERENCE BETWEEN `for..in` AND `for..of` LOOPS

The main difference lies in what they iterate over:

- `for...in` iterates over the **enumerable property keys** of an object. It also includes keys from the prototype chain unless we filter them with `hasOwnProperty`. (In JavaScript, objects can inherit properties from their prototype. When I use `for...in`, it doesn't just give me the object's own properties—it also includes the properties that are inherited from the prototype

chain.)
Example:

```
let obj = { a: 1, b: 2 };
for (let key in obj) {
    console.log(key); // "a", "b"
}
```

- `for...of` iterates over the **values of an iterable object**—like arrays, strings, maps, sets, etc.
  Example:

```
let arr = [10, 20, 30];
for (let value of arr) {
    console.log(value); // 10, 20, 30
}
```

Edge case with arrays:
If I use `for...in` on an array, it gives me the **indexes as strings**, not the actual values:

```
let arr = ["a", "b", "c"];
for (let index in arr) {
    console.log(index); // "0", "1", "2"
}
```

So in summary:

- `for...in` → iterate **keys/properties** (best for objects).
- `for...of` → iterate **values** (best for arrays and other iterables).

21. **HOW DO YOU ADD/REMOVE ELEMENTS FROM AN ARRAY?**

```
let fruits = ['apple', 'mango']
fruits.push('mango') // adds to the end
fruits.pop('mango') // removes from the end
fruits.unshift('grape') // adds to the start
fruits.shift('grape') // removes from the start
```

```
fruits.splice(1,0,'kiwi') // adds 'kiwi' at index 1: ['apple',
'kiwi', 'mango']
```

22. WHAT IS THE PURPOSE OF THE map() FUNCTION?

The `map()` creates a new array with the results of calling a provided function on every element in the array. For example:

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

// Using map to double each number
const doubledNumbers = numbers.map(num => num * 2);
console.log(doubledNumbers);  // [2, 4, 6, 8, 10]

// Using map to create object from each number
const numberObjects = numbers.map(num => ({ value: num, squared:
num * num }));
console.log(numberObjects);
// [
//   { value: 1, squared: 1 },
//   { value: 2, squared: 4 },
//   { value: 3, squared: 9 },
//   { value: 4, squared: 16 },
//   { value: 5, squared: 25 }
// ]
```

23. EXPLAIN THE DIFFERENCE BETWEEN filter() and find() methods

---

- filter(): returns a new array with all elements that pass the test
- find(): returns the first elements that satisfies the condition
  For example:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // [2, 4, 6, 8, 10]
```

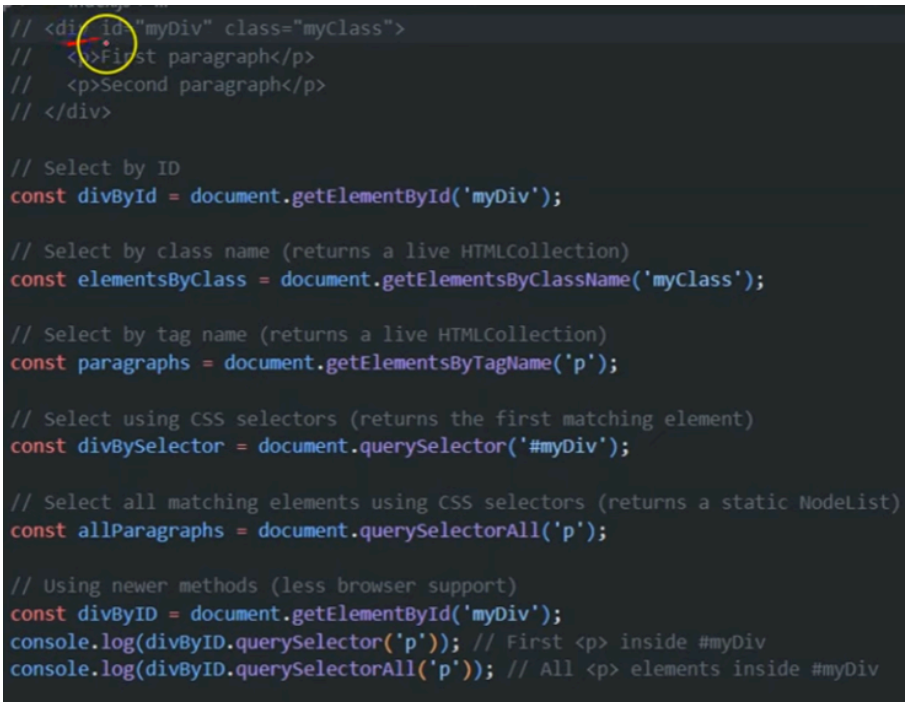
```
const firstEvenNumber = numbers.find(num => num % 2 === 0);
console.log(firstEvenNumber); // 2
```

24. **EXPLAIN THE DIFFERENCE BETWEEN some() AND every() METHODS?**

- `some()` **method**: Returns true if at least one element in the array satisfies the provided testing function. Stops iterating as soon as it finds an element that satisfies the condition. Returns false if no elements satisfy the condition.
- `every()` **method**: Returns true if all elements in the array satisfy the provided testing function. Stops iterating as soon as it finds an element that doesn't satisfy the condition.
  For example:

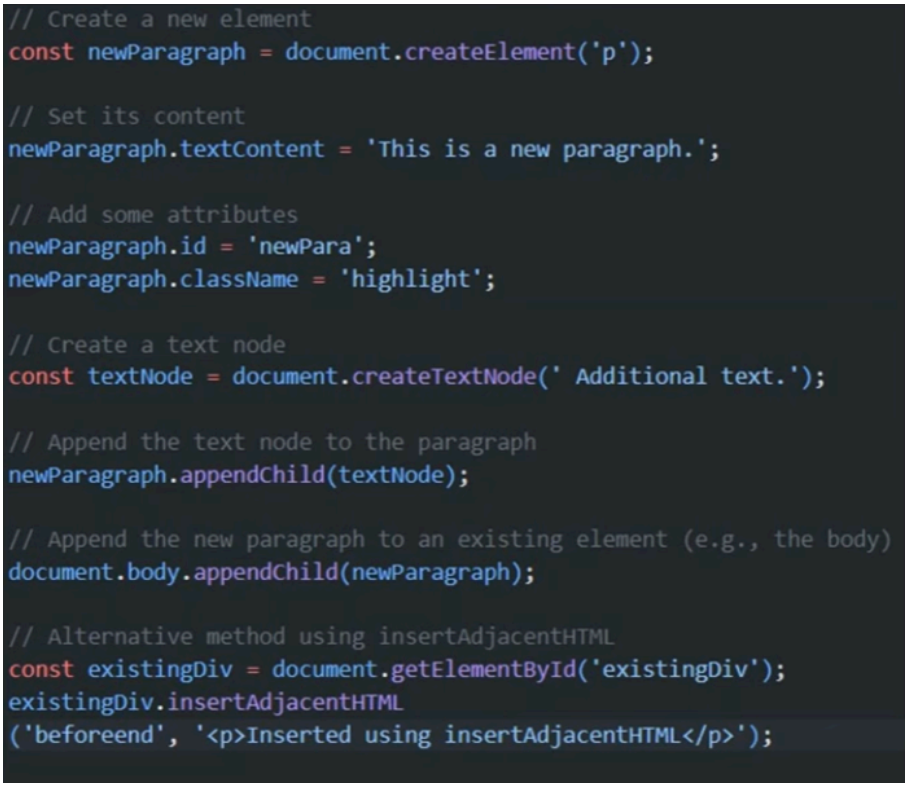
```
const people = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 }
];

// Check if any person is over 30
const anyOver30 = people.some(person => person.age > 30);
console.log(anyOver30); // true (Charlie is over 30)

// Check if all people are over 20
const allOver20 = people.every(person => person.age > 20);
console.log(allOver20); // true (all are over 20)
```

25. **HOW DO YOU SELECT ELEMENTS IN THE DOM USING JAVASCRIPT?**



```
// <div id="myDiv" class="myClass">
//   <p>First paragraph</p>
//   <p>Second paragraph</p>
// </div>

// Select by ID
const divById = document.getElementById('myDiv');

// Select by class name (returns a live HTMLCollection)
const elementsByClass = document.getElementsByClassName('myClass');

// Select by tag name (returns a live HTMLCollection)
const paragraphs = document.getElementsByTagName('p');

// Select using CSS selectors (returns the first matching element)
const divBySelector = document.querySelector('#myDiv');

// Select all matching elements using CSS selectors (returns a static NodeList)
const allParagraphs = document.querySelectorAll('p');

// Using newer methods (less browser support)
const divByID = document.getElementById('myDiv');
console.log(divByID.querySelector('p')); // First <p> inside #myDiv
console.log(divByID.querySelectorAll('p')); // All <p> elements inside #myDiv
```

26. HOW DO YOU CREATE AND APPEND ELEMENTS TO THE DOM?



```
// Create a new element
const newParagraph = document.createElement('p');

// Set its content
newParagraph.textContent = 'This is a new paragraph.';

// Add some attributes
newParagraph.id = 'newPara';
newParagraph.className = 'highlight';

// Create a text node
const textNode = document.createTextNode(' Additional text.');

// Append the text node to the paragraph
newParagraph.appendChild(textNode);

// Append the new paragraph to an existing element (e.g., the body)
document.body.appendChild(newParagraph);

// Alternative method using insertAdjacentHTML
const existingDiv = document.getElementById('existingDiv');
existingDiv.insertAdjacentHTML
('beforeend', '<p>Inserted using insertAdjacentHTML</p>');
```

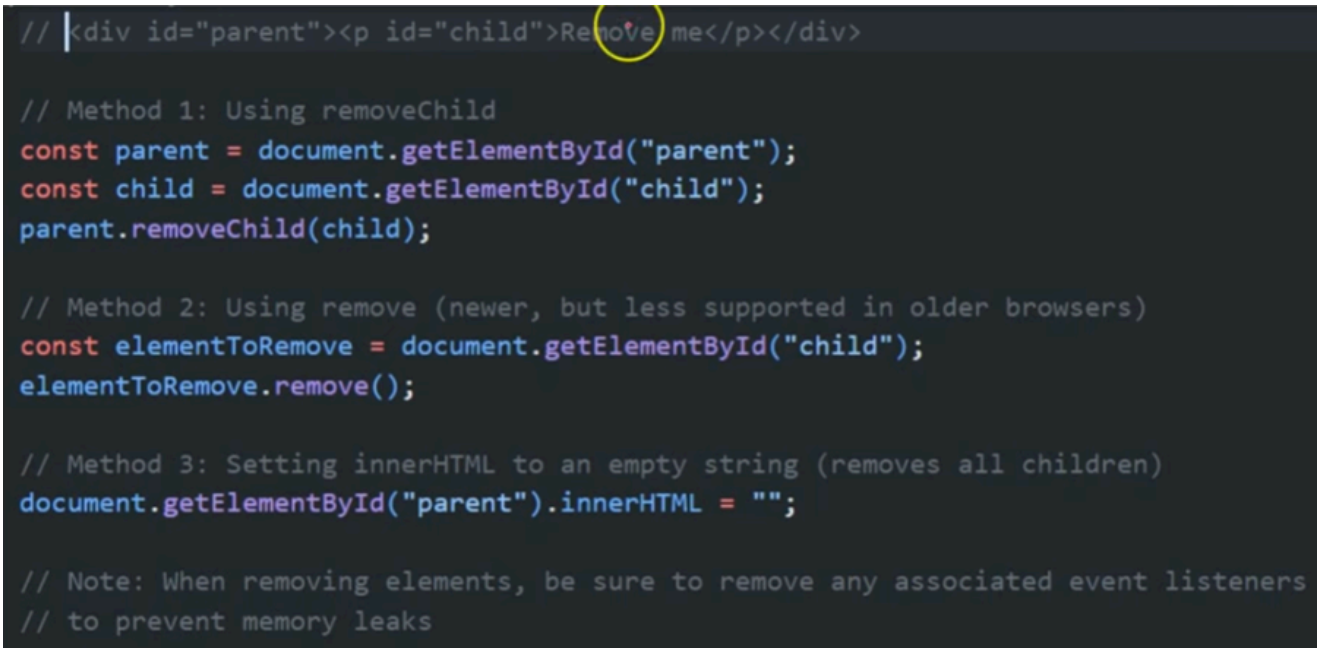
27. EXPLAIN THE DIFFERENCE BETWEEN innerHTML AND textContent

- `innerHTML` parses the assigned string as HTML. This means if the string contains HTML tags, they will be interpreted and rendered.

- `textContent` treats everything as plain text, so HTML tags are not parsed, just displayed as text.
  Because of this difference:
- `innerHTML` can be **slower** and is also **less secure**, since if we directly inject user input, it can lead to **XSS attacks**. For example, an attacker could inject `<script>` or event handlers like `onerror` to steal cookies or tokens.
  - If user input is directly injected into `innerHTML` without proper sanitization, an attacker can insert malicious JavaScript code that runs in the victim's browser—this is called **Cross-Site Scripting (XSS)**.
- `textContent` is **faster and safer**, since it never interprets the input as HTML—it just sets or returns text.
  For example:

```
const div = document.createElement('div');

// innerHTML interprets the content as HTML
div.innerHTML = '<p> This is <strong>bold</strong> text </p>'
console.log(div.innerHTML) // "<p> This is <strong>bold</strong>
text </p>"

// textContent treats the content as plain text
div.textContent = '<p> This is <strong>bold</strong> text </p>'
console.log(div.textContent) // "<p> This is <strong>bold</strong>
text </p>"
```

28. **HOW DO YOU REMOVE AN ELEMENT FROM THE DOM?**



29. WHAT ARE ARROW FUNCTIONS AND HOW DO THEY DIFFER FROM REGULAR FUNCTIONS?

Key differences:

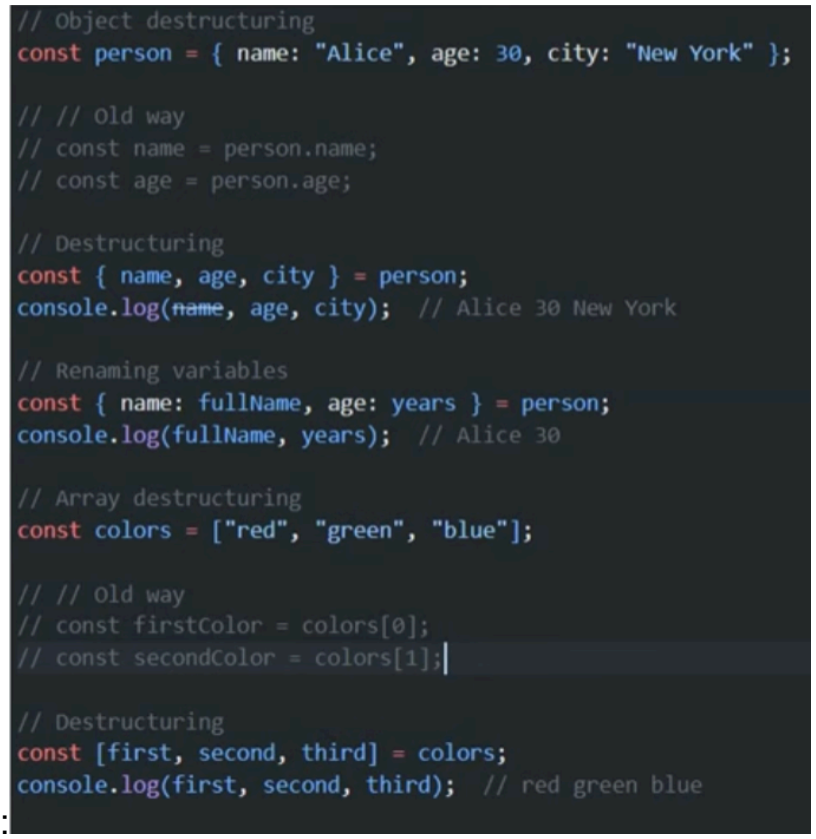
- Syntax: Arrow functions have a more concise syntax
- 'this' binding: Arrow functions don't have their own 'this'
- No 'arguments' object: Arrow functions don't have the 'arguments' object
  For example:

```
// Arrow function
const addArrow = (a, b) => a + b;

// Example of 'this' binding difference
const obj = {
  name: "John",
  sayHello: function () {
    console.log("Hello, " + this.name);
  },
  sayHelloArrow: () => {
    console.log("Hello, " + this.name);
  },
};

obj.sayHello();
// Output: "Hello, John"
```

```
obj.sayHelloArrow();
// Output: "Hello, undefined"
// (this refers to global/window object)
```

30. **EXPLAIN THE CONCEPT OF DESTRUCTURING IN JAVASCRIPT?**

Destructuring is a way to extract multiple values from data stored in objects and arrays. For example:



31. **WHAT ARE TEMPLATE LITERALS?**

Template literals are string literals that allow embedded expressions and multi-line strings. For example:

```
const nameVal = "Alice";
const age = 30;

// Old way
console.log("My name is " + nameVal + " and I'm " + age + " years
old.");

// Using template literals
console.log(`My name is ${nameVal} and I'm ${age} years old.`);
```

```
// Multi-line strings
const multiLine = `
  This is a
  multi-line
  string
`;
console.log(multiLine);
```

32. WHAT IS SPREAD OPERATOR, EXPLAIN WITH EXAMPLE?

The spread operator(...) allows an iterable to be expanded in places where zero or more arguments or elements are expected. For example:

```
// Spreading arrays
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2];
console.log(combined);  // [1, 2, 3, 4, 5, 6]

// Spreading objects
const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
const mergedObj = { ...obj1, ...obj2 };
console.log(mergedObj);  // { a: 1, b: 2, c: 3, d: 4 }

// Copying arrays
const original = [1, 2, 3];
const copy = [...original];
copy.push(4);
console.log(original, copy);  // [1, 2, 3] [1, 2, 3, 4]

// Spreading strings
const str = "Hello";
const chars = [...str];
console.log(chars);  // ['H', 'e', 'l', 'l', 'o']
```

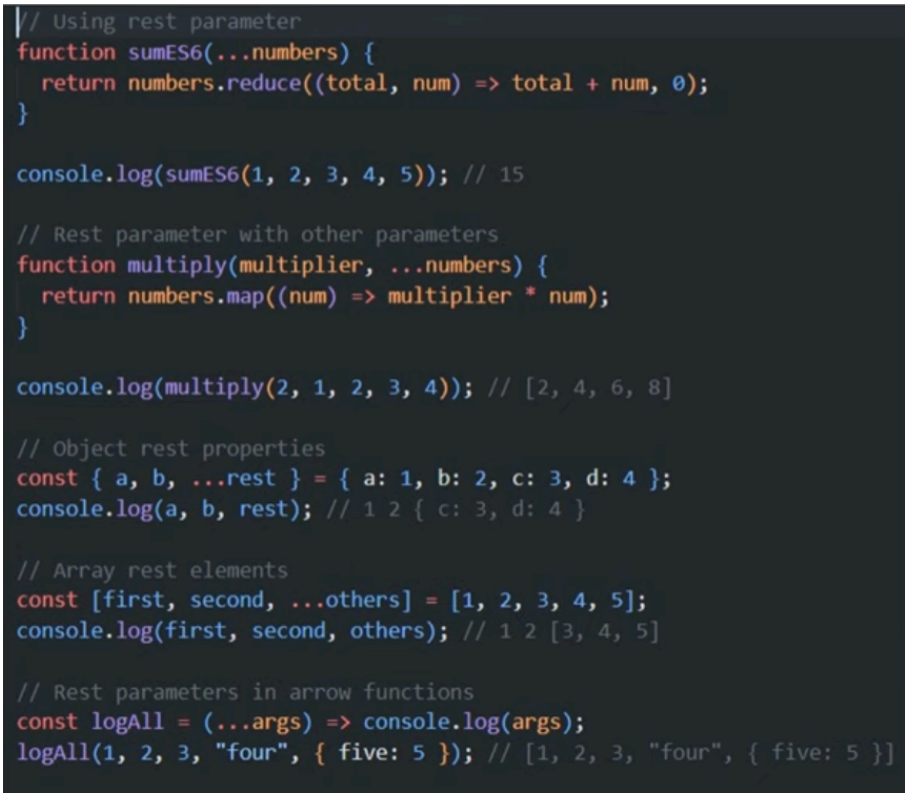
33. WHAT ARE THE DEFAULT PARAMETERS IN ES6?

Default parameters allow you to set default values for function parameters if no value or undefined is passed. For example:

```
// Using default parameters
function greetES6(name = "Guest"){
    console.log(`Hello ${name}`);
}
greetES6() // Hello Guest
greetES6("Ankit") // Hello Ankit

// Default parameters with expressions
function multiply(a, b = a*2) { return a*b; }
console.log(multiply(5)); // 50
console.log(multiply(5,3)); // 15
```

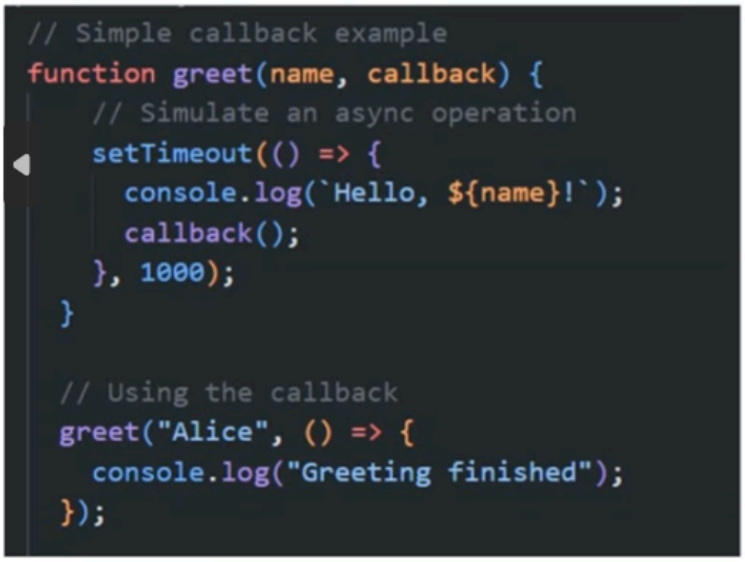
34. HOW DO YOU USE THE REST PARAMETER IN FUNCTIONS?

The rest parameter syntax allows a function to accept an indefinite number of arguments as an array. For example:

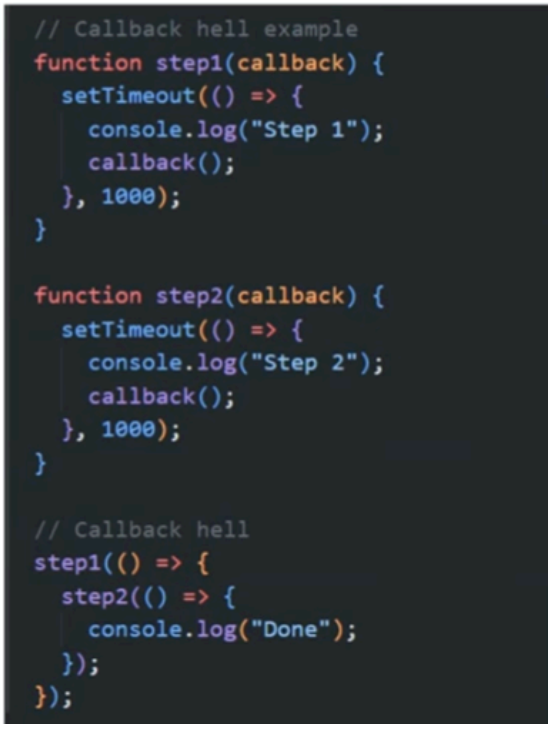


35. WHAT IS CALLBACK AND CALLBACK HELL. EXPLAIN WITH EXAMPLE

Callbacks are functions passed as arguments to other functions, often used for asynchronous operations. Callback hell occurs when multiple nested callbacks make code hard to read and maintain. For example:



```
// Simple callback example
function greet(name, callback) {
    // Simulate an async operation
    setTimeout(() => {
      console.log(`Hello, ${name}!`);
      callback();
    }, 1000);
  }

  // Using the callback
  greet("Alice", () => {
    console.log("Greeting finished");
  });
```



```
// Callback hell example
function step1(callback) {
  setTimeout(() => {
    console.log("Step 1");
    callback();
  }, 1000);
}

function step2(callback) {
  setTimeout(() => {
    console.log("Step 2");
    callback();
  }, 1000);
}

// Callback hell
step1(() => {
  step2(() => {
    console.log("Done");
  });
});
```

```
Output: (for left photo)
Hello, Alice!
Greeting finished

Output: (for right photo)
Step 1
Step 2
Done
```

36. WHAT IS A PROMISE IN JAVASCRIPT WITH EXAMPLE?

A promise is an object representing the eventual completion or failure of an asynchronous operation. For example:

```
function fetchData(){
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const success = true;
            if(success){
                resolve("Data fetched successfully")
            } else {
```

```
                reject("Error fetching data")
            }
        }, 1000)
    })
}

fetchData()
    .then((data) => console.log(data))
    .catch((error) => console.error(error))
```

37. **HOW DO YOU CHAIN PROMISES?**

Promise chaining allows you to perform sequential asynchronous operations. For example:

```
function step1() {
  return Promise.resolve("Step 1 complete");
}

function step2(prevResult) {
  return Promise.resolve(`${prevResult}, Step 2 complete`);
}

step1()
  .then((result) => step2(result))
  .then((finalResult) => console.log(finalResult))
  .catch((error) => console.error(error));
```

38. **WHAT IS THE PURPOSE OF THE Promise.all() METHOD?**

Promise.all() takes an iterable of promises and returns a single Promise that resolves when all input promises have resolved, or rejects if any input promise rejects. For example:

```
const promise1 = Promise.resolve(3)
const promise2 = new Promise((resolve) => setTimeout(() =>
resolve(42), 100))

Promise.all([promise1, promise2])
```

```
    .then((values) => console.log(values)) // [3,42]
    .catch((error) => console.log(error))
```

39. WHAT IS THE PURPOSE OF THE finally() METHOD IN PROMISES?

The finally() method is used to specify code that should be executed regardless of whether the promise is fulfilled or rejected. For example:

```
function fetchData() {
  return new Promise((resolve, reject) => {
    const success = Math.random() > 0.5;
    setTimeout(() => {
      if (success) {
        resolve("Data successfully fetched");
      } else {
        reject("Error: Failed to fetch data");
      }
    }, 1000);
  });
}

fetchData()
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error(error);
  })
  .finally(() => {
    // This block always runs, regardless of fulfillment or rejection
    console.log("Operation completed. Loading status:");
  });
```

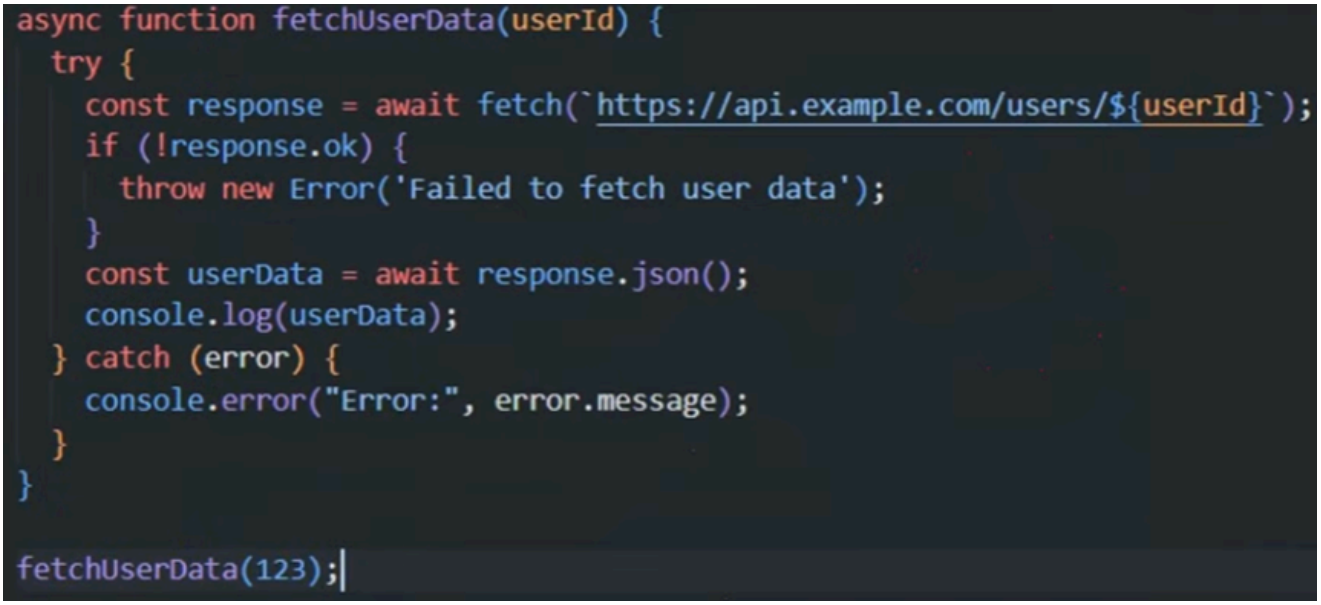
40. WHAT IS THE PURPOSE OF THE async await?

The purpose of async/await is to simplify the syntax for working with Promises, making asynchronous code easier to read and write. It allows you to write asynchronous code that looks and behaves more like synchronous code. For example:

```
// Function that returns a Promise
function fetchData(){
    return new Promise((resolve) => {
        setTimeout(() => resolve("Data fetched"), 1000);
    })
}

// Using async/await
```

```
async function getData(){
    console.log("Fetching data...")
    const result = await fetchData()
    console.log(result)
    console.log("Data processing completed")
}
getData()
```

41. **HOW DO YOU HANDLE ERRORS IN async/await?**



42. **WHAT IS THE DIFFERENCE BETWEEN async/await AND Promises?**

Both `async/await` and Promises are used to handle asynchronous operations, but the main difference lies in **how we write and manage the code**:

- **Syntax:** `async/await` gives code a more **synchronous look**, making it easier to read and write compared to chaining multiple `.then()` calls in Promises.
- **Error Handling:** With Promises, errors are handled using `.catch()`, while with `async/await` we can use the familiar `try...catch` block, which feels more natural for developers used to synchronous code.
- **Chaining:** Promises rely on `.then()` for chaining asynchronous tasks, whereas `async/await` allows us to use standard JavaScript control flow, which is often cleaner.

- **Debugging:** `async/await` is generally easier to debug because the flow looks sequential, while deeply chained Promises can sometimes be harder to trace.

43. **WHAT IS THE DIFFERENCE BETWEEN DEFAULT AND NAMED EXPORTS?**

- A module can have multiple named exports but only one default export.
- Named exports are imported using curly braces, while default exports are imported without them.
- Default exports can be imported with any name, while named exports must be imported with their exact names (unless renames using `'as'`)
  Example:

```
// Named exports
export const add = (a, b) => a + b;

// Default export
export default function multiply (a, b) { return a*b; }

// Importing names exports
import { add } from "./math.js"
console.log(add(3,4));

// Importing default exports
import multiply from "./math.js"
console.log(multiply(7,8));
```

44. **HOW DO YOU CONVERT JAVASCRIPT OBJECT TO A JSON STRING?**

```
const user = {
    name: "John Doe",
    age: 30,
    isAdmin: false,
}
const jsonString = JSON.stringify(user);
```

```
console.log(jsonString) // Output: {"name":"John Doe", "age":30,
"isAdmin": false}
```

45. **HOW DO YOU PARSE A JSON STRING BACK INTO A JAVASCRIPT OBJECT?**

```
const jsonString = '{"name":"John Doe", "age":30, "isAdmin":
false}';
const user = JSON.parse(jsonString);
console.log(user.name, user.age, user.isAdmin) // Output: John Doe,
30, false
```

46. **WHAT IS localStorage IN JAVASCRIPT, AND HOW DO YOU STORE AND RETRIEVE DATA FROM IT?**

---

localStorage is a web storage object that allows you to store key-value pairs in the browser with no expiration time. For example:

```
// Storing data
localStorage.setItem("username", "JohnDoe")
localStorage.setItem("isLoggedIn", "true")

// Retrieving data
const username = localStorage.getItem("username");
console.log(username); // Output: JohnDoe

const username = localStorage.getItem("isLoggedIn") === "true";
console.log(isLoggedIn); // Output: true

// Storing and Retrieving objects
const user = {name: 'John', age: 30}
localStorage.setItem('user', JSON.stringify(user))

const storedUser = JSON.parse(localStorage.getItem('user'))
console.log(storedUser.name) // Output: John
```

47. **WHAT IS THE DIFFERENCE BETWEEN localStorage AND sessionStorage?**

Both `localStorage` and `sessionStorage` are part of the Web Storage API and let us store key–value pairs in the browser. The main difference is in how long the data persists:

- **`localStorage`** → Data is persistent. It remains even after the browser is closed and reopened, until it's explicitly cleared by the user or through code.
- **`sessionStorage`** → Data is temporary. It's tied to a specific tab or window and is cleared automatically once the page session ends (for example, when the tab is closed).
  Apart from that, both share the same API (`setItem` , `getItem` , `removeItem` , `clear`) and can only store strings. For example:

```
// localStorage: persists even after the browser window is closed
localStorage.setItem(
  "persistentData",
  "This will remain after closing the browser"
); // The key is "persistentData" and the value is a string.

// sessionStorage: data is cleared when the browser session ends
sessionStorage.setItem(
  "temporaryData",
  "This will be cleared when the session ends"
); The key is "temporaryData" and the value is a string.

// Both are used similarly
console.log(localStorage.getItem("persistentData"));
console.log(sessionStorage.getItem("temporaryData"));
```

48. **HOW DO YOU DELETE A SPECIFIC ITEM FROM localStorage OR CLEAR ALL DATA FROM IT?**

```
// Storing some data
localStorage.setItem("username", "JohnDoe")
localStorage.setItem("isLoggedIn", "true")

// Removing a specific item
localStorage.removeItem("isLoggedIn")
```

```
console.log(localStorage.getItem("username")); // Output: JohnDoe
console.log(localStorage.getItem("isLoggedIn")); // Output: null

// Clearing all data from localStorage
localStorage.clear()

console.log(localStorage.getItem("username")); // Output: null
```