1. **Explain spread operator and rest parameters with examples?**

Answer:

* **Spread Operator:**

The spread operator is used to expand elements of an array or object. It allows you to copy the contents of one array into another or include the properties of one object into another.

**Example:**

// Example 1: Copying Arrays

*const* arr1 = [1, 2, 3];

*const* arr2 = [...arr1]; // Copies elements of arr1 into a new array arr2

console.log(arr2); // Output: [1, 2, 3]

// Example 2: Concatenating Arrays

*const* arr3 = [4, 5, 6];

*const* combinedArray = [...arr1, ...arr3]; // Concatenates arr1 and arr3

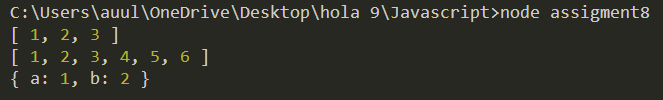
console.log(combinedArray); // Output: [1, 2, 3, 4, 5, 6]

// Example 3: Copying Object Properties

*const* obj1 = { a: 1, b: 2 };

*const* obj2 = { ...obj1 }; // Copies properties of obj1 into a new object obj2

console.log(obj2); // Output: { a: 1, b: 2 }



* **Function Argument Spreading:**

The spread operator can be used to spread the elements of an array as individual arguments to a function.

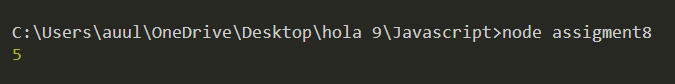
**Example:**

// Example: Math.max with spread operator

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

*const* maxNumber = Math.max(...numbers);

console.log(maxNumber); // Output: 5



* **Rest Parameters:**

Rest parameters allow a function to accept an indefinite number of arguments as an array. It uses the ... syntax followed by the parameter name.

**Example:**

// Example: Sum of numbers using rest parameter

*function* sum(...*numbers*) {

  return *numbers*.reduce((*total*, *num*) *=>* *total* + *num*, 0);

}

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



In the **sum** function, the **...numbers** syntax collects all the arguments passed to the function into an array named **numbers**. This makes it easy to work with a variable number of arguments

* **Combining Spread and Rest:**

You can also combine the spread and rest syntax.

**Example:**

// Example: Combining spread and rest

*function* example(*arg1*, *arg2*, ...*restArgs*) {

    console.log(*arg1*); // Output: 1

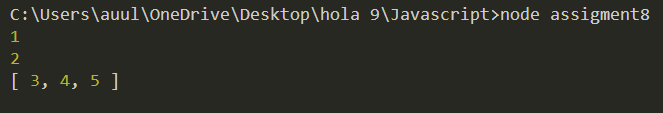
    console.log(*arg2*); // Output: 2

    console.log(*restArgs*); // Output: [3, 4, 5]

  }

*const* arrayToSpread = [3, 4, 5];

  example(1, 2, ...arrayToSpread);



In this example, **arg1** and **arg2** are regular parameters, and **restArgs** collects the remaining arguments using the rest parameter syntax.These features make JavaScript more flexible and expressive when working with arrays and function parameters.

**2.Explain rest operator using object and array destructuring with example.**

**Answer:**

* **Rest Operator in Array Destructuring:**
* **Definition:**

The rest operator in array destructuring allows you to collect the remaining elements of an array into a new array. It is denoted by the ... syntax and is useful when you want to capture multiple elements without explicitly specifying each one.

**Example:**

// Definition

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

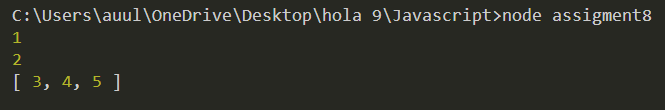
*const* [first, second, ...rest] = numbers;

// Examples

console.log(first);  // Output: 1

console.log(second); // Output: 2

console.log(rest);   // Output: [3, 4, 5]



In this example, the **first** and **second** variables capture the first two elements of the **numbers** array, and the **...rest** collects the remaining elements into a new array called **rest**.

* **Rest Operator in Object Destructuring:**
* Definition:

The rest operator in object destructuring allows you to collect the remaining properties of an object into a new object. It is also denoted by the ... syntax and is useful when you want to extract specific properties and handle the rest separately.

**Example:**

// Definition

*const* person = {

  firstName: "Virat",

  lastName: "Kholi",

  age: 35,

  Country: "India",

};

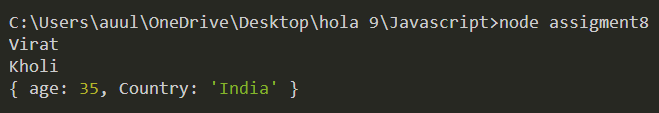
*const* { firstName, lastName, ...otherInfo } = person;

// Examples

console.log(firstName); // Output: Virat

console.log(lastName); // Output: Kholi

console.log(otherInfo); // Output: { age: 35, Country: 'India' }

****

In this example, **firstName** and **lastName** capture specific properties of the **person** object, while the **...otherInfo** collects the remaining properties into a new object called **otherInfo**.

These rest operator examples illustrate how they provide a concise and flexible way to handle remaining elements or properties during array and object destructuring in JavaScript.

**3.Explain localStorage methods like setItem(), getItem(), remove(), clear() with example?**

**Answer:**

* **setItem(key, value):**
* **Definition:**

The setItem method is used to store a key-value pair in the local storage. The key is a string that acts as an identifier, and the value can be any valid JavaScript data type (string, number, boolean, object, etc.).

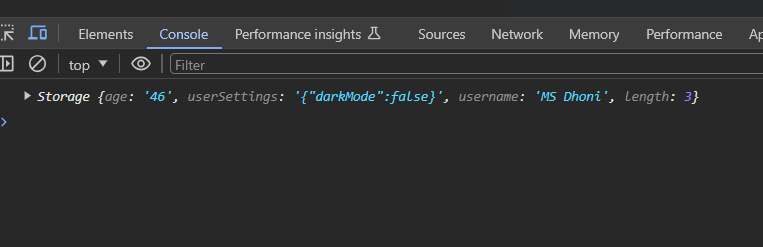
**Example:**

// Storing data in localStorage

localStorage.setItem('username', 'MS Dhoni');

localStorage.setItem('age', 46);

console.log(localStorage);

****

* **getItem(key):**
* **Definition:**

The getItem method retrieves the value associated with a specified key from the local storage.

**Example:**

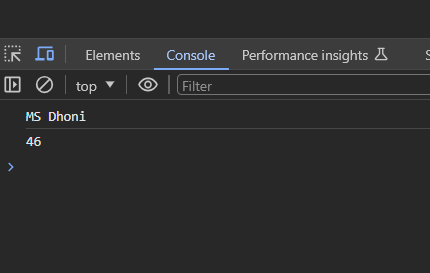
// Retrieving data from localStorage

*const* username = localStorage.getItem("username");

*const* age = localStorage.getItem("age");

console.log(username); // Output: MS Dhoni

console.log(age); // Output: 46

****

* **removeItem(key):**
* **Definition:**

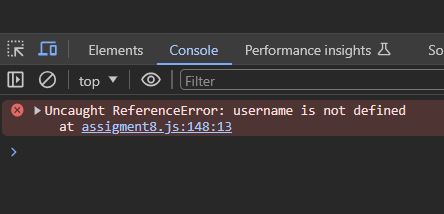
The removeItem method removes the key-value pair associated with the specified key from the local storage.

**Example:**

// Removing data from localStorage

localStorage.removeItem("username");

console.log(username);

****

* **clear():**
* **Definition:**

The clear method is used to remove all key-value pairs from the local storage, effectively clearing the entire storage.

**Example:**

// Clearing all data from localStorage

localStorage.clear();

console.log(username);

**4. Explain setTimeout() and setInerval() with examples**

**Answer:**

* **setTimeout():**
  + **Definition:**

The setTimeout() function is used to execute a function or code snippet after a specified delay, measured in milliseconds.

**Syntax:**

**setTimeout(function, delay, param1, param2, ...);**

* **function:** The function to be executed.
* **delay:** The time (in milliseconds) to wait before executing the function.
* **param1, param2, ...:** Optional parameters to be passed to the function when it is executed.

**Example:**

// Display a message after 2 seconds

setTimeout(*function* () {

  console.log("This message is displayed after 5 seconds.");

}, 5000);

* **setInterval():**
* **Definition:**

The setInterval() function is used to repeatedly execute a function or code snippet at a specified interval.

**Syntax:**

**setInterval(function, interval, param1, param2, ...);**

* **function:** The function to be executed.
* **interval:** The time (in milliseconds) between each execution of the function.
* **param1, param2, ...:** Optional parameters to be passed to the function each time it is executed.

**Example:**

// Display a message every 3 seconds

*let* count = 0;

*const* intervalId = setInterval(*function*() {

  count++;

  console.log(`Message ${count} displayed every 3 seconds.`);

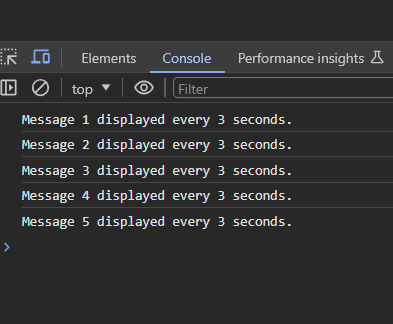
  // Stop after displaying 5 messages

  if (count === 5) {

    clearInterval(intervalId); // Clears the interval

  }

}, 3000);

****

**5. W.A.P of Promise() using .then() and .catch() with the condition (n>3)?**

**Answer:**

// Function that returns a promise

*function* checkNumber(*n*) {

  return new *Promise*((resolve, reject) *=>* {

    if (*n* > 3) {

      resolve(`Number ${*n*} is greater than 3.`);

    } else {

      reject(`Number ${*n*} is not greater than 3.`);

    }

  });

}

// Example usage

*const* inputNumber = 4;

checkNumber(inputNumber)

  .then((*message*) *=>* {

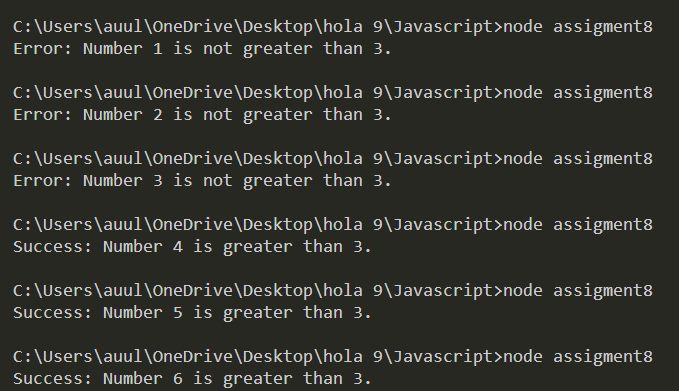
    console.log("Success:", *message*);

  })

  .catch((*message*) *=>* {

    console.error("Error:", *message*);

  });

****

**6.Explain async and await with using fetch(‘https://dummyjson.com/users/’)**

**Answer:**

* async and await are features in JavaScript that simplify asynchronous code, making it look and behave more like synchronous code. They are often used in conjunction with promises, and one common use case is with the fetch API for making asynchronous HTTP requests.
* **Using fetch with async and await:** Let's see an example where we use fetch to make an HTTP request to 'https://dummyjson.com/users/' and then use async and await to handle the response.

// Function to fetch user data asynchronously

async *function* fetchUserData() {

  try {

    // Use the fetch function to make an HTTP GET request

*const* response = await fetch("https://dummyjson.com/users/");

    // Check if the response status is okay (status code 200-299)

    if (!response.ok) {

      throw new *Error*(`HTTP error! Status: ${response.status}`);

    }

    // Parse the response JSON

*const* data = await response.json();

    // Log the user data

    console.log("User Data:", data);

    return data; // You can return the data or perform additional operations

  } catch (error) {

    console.error("Error fetching user data:", error.message);

    throw error; // Rethrow the error or handle it as needed

  }

}

// Call the function and handle the result

fetchUserData()

  .then((*userData*) *=>* {

    // Do something with the user data

    console.log("User data received:", *userData*);

  })

  .catch((*error*) *=>* {

    // Handle errors

    console.error("Error in fetchUserData:", *error*);

  });

async *function* abcd() {

  //async return a promise

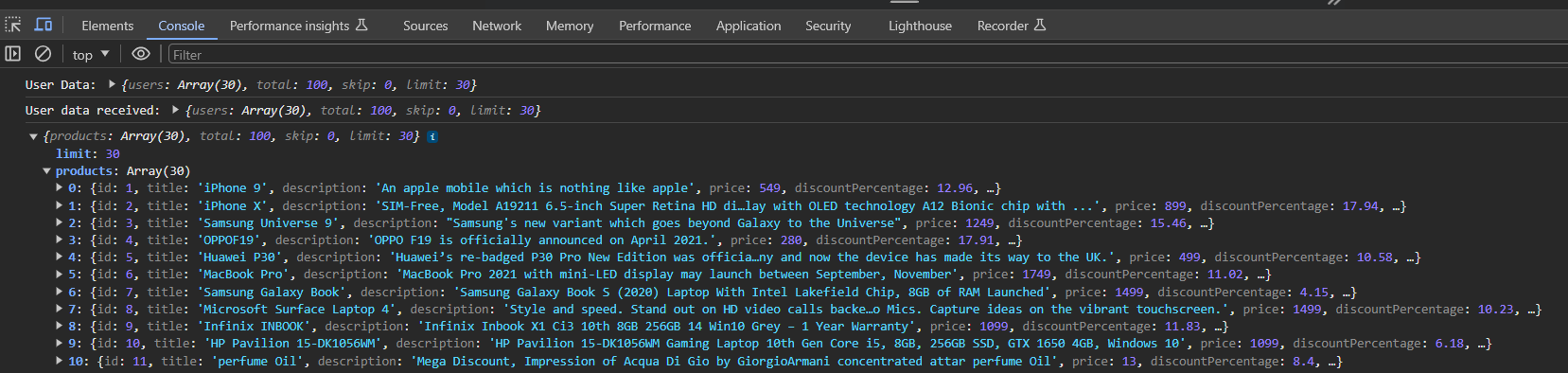
*let* result = await fetch("https://dummyjson.com/products/"); //await waits for a promise

*let* data = await result.json();

  console.log(data);

}

abcd();



**In this example:**

* The fetchUserData function is declared as an async function. This allows the use of the await keyword inside the function.
* Inside the function, await fetch('https://dummyjson.com/users/') is used to make an asynchronous HTTP GET request. The await keyword ensures that the code waits for the request to complete and returns the response.
* We check if the response status is okay (status code 200-299), and then use await response.json() to parse the response as JSON.
* The user data is logged to the console, and the data is returned.
* The function is called using fetchUserData().then() to handle the resolved promise or .catch() to handle any errors.
* **This example demonstrates how async and await make asynchronous code more readable and easier to reason about, especially when working with promises and asynchronous operations like fetching data from an API.**