1.
(a)

A **promise** in Node.js is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. Here's a breakdown of your questions:

Functions of `resolve` and `reject` in a Promise
- **`resolve`**: This function is called when the promise is fulfilled successfully. It passes the resolved value as an argument.
- **`reject`**: This function is called when the promise is rejected due to an error. It passes the error as an argument.

These two functions are provided by the JavaScript engine when you create a new Promise. Here's an example of how they work:

```
let myPromise = new Promise((resolve, reject) => {
    let condition = true; // or false based on some operation
    if (condition) {
        resolve("Promise fulfilled successfully!");
    } else {
        reject("Promise rejected with an error.");
    }
});
```

Initial State and Result of a Promise
- **Initial State**: When a promise is created, it is in the `pending` state.
- **Result**: The initial result of a promise is `undefined`.

State Changes
- **Fulfilled**: When the `resolve` function is called, the promise moves from `pending` to `fulfilled`, and the result is the value passed to `resolve`.
- **Rejected**: When the `reject` function is called, the promise moves from `pending` to `rejected`, and the result is the error passed to `reject`.

Registering/Subscribing Consuming Functions
To handle the results of a promise, you use the `.then()` and `.catch()` methods:
- **`then(onFulfilled, onRejected)`**: Registers callbacks for the promise's fulfilled and rejected states.
- **`catch(onRejected)`**: Registers a callback for only the promise's rejected state.

Example:

```
myPromise
    .then((value) => {
```

```
        console.log(value); // logs "Promise fulfilled successfully!"
    })
    .catch((error) => {
        console.log(error); // logs "Promise rejected with an error."
    });
```

This way, you can handle asynchronous operations and their results in a cleaner and more manageable manner. Let me know if you have any further questions or if you need examples in a different context!

(b)

The `async/await` syntax is used for asynchronous programming in many modern languages, including JavaScript, Python, and C#. Here are some key features:

1. **Asynchronous Functions**: By using the `async` keyword, you can define functions that handle asynchronous operations. When called, these functions return a promise.

2. **Await Keyword**: Inside an `async` function, the `await` keyword can be used to pause the execution of the function until the awaited promise resolves. This makes the code look synchronous and easier to read.

3. **Sequential Execution**: Even though the operations are asynchronous, the code execution within an `async` function appears to be sequential, improving readability and maintainability.

4. **Error Handling**: Errors in async/await can be caught using `try...catch` blocks, which is similar to synchronous code.

Example in JavaScript:

```
async function fetchData(url) {
    try {
        let response = await fetch(url);
        if (!response.ok) throw new Error('Network response was not
ok');
        let data = await response.json();
        return data;
    } catch (error) {
        console.error('There was a problem with the fetch operation:',
error);
    }
}
```

In this example, the `fetchData` function uses `async` to define it as asynchronous. The `await` keyword is used to pause the function until the `fetch` promise resolves. Any errors that occur during the fetch operation are caught by the `catch` block.

2.
(a)
Spread Operator (`...`)
The **spread operator** is used to expand elements of an iterable (like an array) or properties of an object. It can be useful for combining arrays or objects.

Example of Spread Operator:

```
let numbers = [1, 2, 3];
let moreNumbers = [...numbers, 4, 5, 6];
console.log(moreNumbers); // Output: [1, 2, 3, 4, 5, 6]

let user = { name: "Alice", age: 25 };
let userWithId = { ...user, id: "U001" };
console.log(userWithId); // Output: { name: 'Alice', age: 25, id: 'U001'
}
```

Rest Operator (`...`)
The **rest operator** is used to collect multiple elements into an array or to collect the remaining properties of an object. It's particularly useful for handling function arguments or extracting properties.

Example of Rest Operator:

```
function sum(...numbers) {
    return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3, 4)); // Output: 10

let { id, ...restProps } = { id: "U001", name: "Alice", age: 25 };
console.log(restProps); // Output: { name: 'Alice', age: 25 }
```

Destructuring in JavaScript ES6
Destructuring allows you to extract values from arrays or properties from objects and assign them to variables in a more concise manner.

Nested Destructuring
Given the object `task`, we can perform nested destructuring to extract the `dates` and `tags` properties.

Example of Nested Destructuring:

```
const task = {
    id: "TSK001",
    name: "Tiling and Flooring",
    dates: {
        startDate: 1644551200000,
        endDate: 1644969599000
    },
    status: "Completed",
    tags: ["Tower A", "First Floor", ["Tiling", "Flooring"]]
};

const { dates: { startDate, endDate }, tags: [ location1, location2, [
tag1, tag2 ]] } = task;

console.log(startDate); // Output: 1644551200000
console.log(endDate);   // Output: 1644969599000
console.log(location1); // Output: Tower A
console.log(location2); // Output: First Floor
console.log(tag1);      // Output: Tiling
console.log(tag2);      // Output: Flooring
```

In this example:
- The `dates` property is destructured to get `startDate` and `endDate`.
- The `tags` array is destructured to get `location1`, `location2`, and an inner array containing `tag1` and `tag2`.

Destructuring can be a powerful feature in simplifying your code and making it more readable.