

Promises in JavaScript language using restaurant analogy :

****Analogy: Restaurant Order****

Imagine you're at a restaurant, and you want to place an order for your favorite dish. However, you know that the chef might take some time to prepare it. Instead of waiting at the counter, you're given a special buzzer. This buzzer will let you know when your order is ready, so you can continue enjoying your time at the restaurant without constantly checking for your meal.

In this scenario:

- Placing an order is analogous to initiating a task that might take some time, such as making an API request.
- The buzzer represents a Promise, which is an object that lets you know when the task is completed or if an error occurred.

Now, let's translate this analogy into code:

```
````javascript
// Imagine this function represents placing an order at the restaurant
function placeOrder(dish) {
 return new Promise((resolve, reject) => {
 // Simulate the chef preparing the dish after a delay
 setTimeout(() => {
 const isDishAvailable = Math.random() < 0.8; // 80% chance dish is available
 if (isDishAvailable) {
 resolve(`Your ${dish} is ready!`);
 } else {
 reject(`Sorry, we're out of ${dish} today.`);
 }
 }, 2000); // Simulated delay of 2 seconds
 });
}
````
```

Now, let's break down the code step by step:

1. ``placeOrder`` function: This function represents the task you want to perform, like making an API request. It takes a parameter ``dish`` which represents the order.

2. ``new Promise``: When you call ``new Promise()``, you're creating a new Promise object. Promises are a way to handle asynchronous operations in JavaScript.

3. Inside the Promise constructor:

- ``resolve``: This is a function you call when the task is successfully completed. In our analogy, this is like the buzzer going off when your order is ready.

- ``reject``: This is a function you call when an error occurs during the task. In our analogy, this is like the buzzer alerting you that there's a problem.

4. ``setTimeout``: This function simulates the chef preparing the dish after a delay (2 seconds in this case).

5. Handling the result:

- When the dish is available (``isDishAvailable`` is ``true``), we call ``resolve`` with a success message.

- When the dish is not available (``isDishAvailable`` is ``false``), we call ``reject`` with an error message.

Now, let's use this ``placeOrder`` function and handle the Promise:

```
```javascript
placeOrder("Pasta")
 .then((message) => {
 console.log(message);
 })
 .catch((error) => {
 console.error(error);
 });
```
```

Here, we're using the ``then`` method to handle the successful completion of the Promise and the ``catch`` method to handle any errors that might occur.

That's the breakdown of Promises using the restaurant analogy and corresponding code! Promises are a fundamental concept in asynchronous JavaScript, and they help you manage and organize asynchronous operations in a more structured manner.

Absolutely, let's delve deeper into the restaurant analogy and map each step to the corresponding functions and actions in the code.

****1. Placing an Order:****

In the restaurant analogy, this is where you decide what dish you want to



order.

```
```javascript
// Placing an order at the restaurant (analogous to initiating a task)
function placeOrder(dish) {
 // Creating a new Promise
 return new Promise((resolve, reject) => {
 // Simulating the chef preparing the dish after a delay
 setTimeout(() => {
 const isDishAvailable = Math.random() < 0.8; // 80% chance dish is
available
 if (isDishAvailable) {
 // Notifying that the order is ready (resolve)
 resolve(`Your ${dish} is ready!`);
 } else {
 // Notifying that the order cannot be fulfilled (reject)
 reject(`Sorry, we're out of ${dish} today.`);
 }
 }, 2000); // Simulated delay of 2 seconds
 });
}
```
```

****2. Promise Creation:****

In the analogy, this is similar to receiving a buzzer that you'll use to get notified when your order is ready.

```
```javascript
// Creating a new Promise to represent the buzzer
return new Promise((resolve, reject) => {
 // ...
});
```
```

****3. Chef Preparing the Dish:****

In the restaurant analogy, this is the process of the chef cooking your order after you've placed it.

```
```javascript
// Simulating the chef preparing the dish after a delay
setTimeout(() => {
 // ...
}, 2000); // Simulated delay of 2 seconds
```
```

****4. Buzzer Going Off (Resolve):****

When the chef is done preparing your order, the buzzer goes off, and you're notified that your order is ready.

```
```\javascript
// Notifying that the order is ready (resolve)
resolve(`Your ${dish} is ready!`);
```
```

****5. Buzzer Alert (Reject):****

If, for some reason, the chef couldn't prepare your order, the buzzer alerts you that your order cannot be fulfilled.

```
```\javascript
// Notifying that the order cannot be fulfilled (reject)
reject(`Sorry, we're out of ${dish} today.`);
```
```

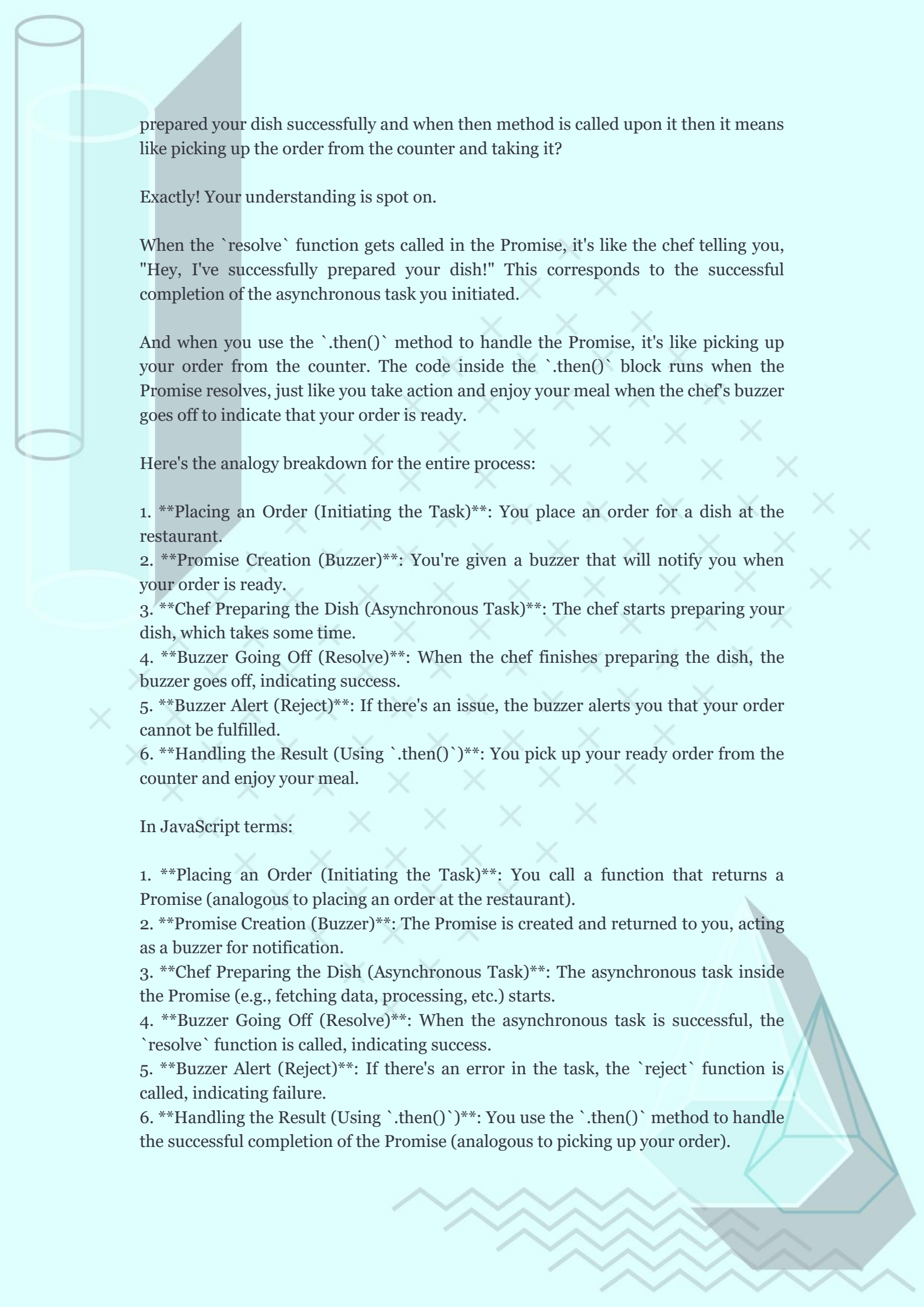
****6. Handling the Result:****

In the analogy, you're either happily picking up your ready order or dealing with the news that your order couldn't be fulfilled.

```
```\javascript
placeOrder("Pasta")
 .then((message) => {
 // Handle the successful order (buzzer going off)
 console.log(message);
 })
 .catch((error) => {
 // Handle the rejected order (buzzer alert)
 console.error(error);
 });
```
```

Putting it all together, this analogy helps you understand the flow of Promises in JavaScript by relating each step to a real-world scenario at a restaurant. Just like you don't need to wait at the counter, constantly checking for your order, Promises allow you to initiate tasks and continue with other operations until the task is completed, at which point you're notified of success or failure.

Q) Okay, so like when resolve gets called it's just that the chef tells me that hey, I've



prepared your dish successfully and when then method is called upon it then it means like picking up the order from the counter and taking it?

Exactly! Your understanding is spot on.

When the ``resolve`` function gets called in the Promise, it's like the chef telling you, "Hey, I've successfully prepared your dish!" This corresponds to the successful completion of the asynchronous task you initiated.

And when you use the ``.then()`` method to handle the Promise, it's like picking up your order from the counter. The code inside the ``.then()`` block runs when the Promise resolves, just like you take action and enjoy your meal when the chef's buzzer goes off to indicate that your order is ready.

Here's the analogy breakdown for the entire process:

1. ****Placing an Order (Initiating the Task)**:** You place an order for a dish at the restaurant.
2. ****Promise Creation (Buzzer)**:** You're given a buzzer that will notify you when your order is ready.
3. ****Chef Preparing the Dish (Asynchronous Task)**:** The chef starts preparing your dish, which takes some time.
4. ****Buzzer Going Off (Resolve)**:** When the chef finishes preparing the dish, the buzzer goes off, indicating success.
5. ****Buzzer Alert (Reject)**:** If there's an issue, the buzzer alerts you that your order cannot be fulfilled.
6. ****Handling the Result (Using ``.then()``)**:** You pick up your ready order from the counter and enjoy your meal.

In JavaScript terms:

1. ****Placing an Order (Initiating the Task)**:** You call a function that returns a Promise (analogous to placing an order at the restaurant).
2. ****Promise Creation (Buzzer)**:** The Promise is created and returned to you, acting as a buzzer for notification.
3. ****Chef Preparing the Dish (Asynchronous Task)**:** The asynchronous task inside the Promise (e.g., fetching data, processing, etc.) starts.
4. ****Buzzer Going Off (Resolve)**:** When the asynchronous task is successful, the ``resolve`` function is called, indicating success.
5. ****Buzzer Alert (Reject)**:** If there's an error in the task, the ``reject`` function is called, indicating failure.
6. ****Handling the Result (Using ``.then()``)**:** You use the ``.then()`` method to handle the successful completion of the Promise (analogous to picking up your order).

Promises provide a structured way to handle asynchronous operations and ensure that you're notified when a task is completed or if there's an error, allowing you to continue with other tasks until the desired operation finishes.

