Javascript

Promises Q&A

SWIPE

What is a Promise in JavaScript?

A Promise is a JavaScript object representing the eventual completion or failure of an asynchronous operation. It provides a way to handle asynchronous operations more easily and avoid callback hell.

```
// Create a function that returns a Promise
function fetchData() {
  return new Promise((resolve, reject) => {
    // Simulate an asynchronous operation (e.g., fetching data fro
    setTimeout(() => {
      const data = { message: "Data fetched successfully" };
      // Resolve the Promise with the data
      resolve(data);

      // Uncomment the following line to simulate an error:
      // reject("Error: Unable to fetch data");
    }, 2000); // Simulate a 2-second delay
  });
}

// Use the Promise
fetchData()
  .then((result) => {
    console.log(result.message); // Data fetched successfully
  })
  .catch((error) => {
    console.error(error); // Handle errors here
  });
```

How do you create a Promise in JavaScript?

You can create a Promise using the `Promise` constructor. It takes a single argument, a function (executor), which has two parameters: `resolve` and `reject`. You call `resolve` when the asynchronous operation is successful and `reject` when it fails.

```
const myPromise = new Promise((resolve, reject) => {
  // Perform some asynchronous operation
  if (/* operation successful */) {
    resolve(result);
  } else {
    reject(error);
  }
});
```

What is the purpose of `async/await` in JavaScript?

The `async/await` syntax is used to simplify working with Promises. It allows you to write asynchronous code that looks more like synchronous code, making it easier to read and maintain. The `async` keyword defines a function as asynchronous, and `await` is used inside such a function to wait for a Promise to resolve or reject.

How do you define an `async` function in JavaScript?

You can define an `async` function using the `async` keyword before the function declaration. An `async` function always returns a Promise.

```
async function myAsyncFunction() {
  // Asynchronous code using await
  const result = await somePromise;
  return result;
}
```

What is the difference between `Promise.all()` and `Promise.race()`?

`Promise.all()` waits for all Promises in an array to resolve, and it returns an array of their results. In contrast, `Promise.race()` waits for any one of the Promises in an array to resolve or reject, and it returns the result or error of the first Promise that settles.

How do you handle errors in `async/await`?

You can use a `try/catch` block to handle errors in `async/await`. If an error occurs within the `try` block or any awaited Promise rejects, control is passed to the `catch` block, where you can handle the error.

```
async function myAsyncFunction() {
  try {
    const result = await somePromise;
    // Code if successful
  } catch (error) {
    // Handle the error
  }
}
```

What is Promise chaining?

Promise chaining is a technique where you chain multiple asynchronous operations together using the `.then()` method. It allows you to perform a series of asynchronous tasks sequentially.

```
myPromise.then((result1) => {
  // Code to handle result1
  return result2Promise;
}).then((result2) => {
  // Code to handle result2
});
```

How can you achieve parallel execution of Promises?

You can use `Promise.all()` to execute multiple Promises in parallel. Each Promise in the array runs concurrently, and you get the results in the same order as the Promises.

```
const promises = [promise1, promise2, promise3];
Promise.all(promises).then((results) => {
  // Handle results
});
```

How can you cancel a Promise or cleanup resources when it's no longer needed?

You can use an `AbortController` and the `abort` method to cancel a Promise or cleanup resources associated with it. The Promise can catch the `AbortError` and handle it gracefully.

```
const controller = new AbortController();
const signal = controller.signal;

const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation
  // Listen for abort signal
  signal.addEventListener('abort', () => {
    reject(new DOMException('Aborted', 'AbortError'));
    // Cleanup resources
  });
});

// To cancel the Promise
controller.abort();
```

Explain the concept of 'Promise.race()' and provide a use case for it.

`Promise.race()` resolves or rejects as soon as any of the Promises in the array settles. It's useful for implementing timeout logic. For example, you can use it to set a maximum execution time for an asynchronous operation.

```
Promise.race([myPromise, timeoutPromise]).then((result) => {
  // Handle result (myPromise resolved within timeout)
}).catch((error) => {
  // Handle error (myPromise took too long)
});
```

How can you implement retry logic using Promises?

You can implement retry logic by recursively calling a function that returns a Promise. In each iteration, you can retry or reject the Promise based on specific conditions.

```
function retryOperation(maxRetries) {
  return someAsyncOperation()
    .catch((error) => {
      if (maxRetries > 0) {
        console.log('Retrying...');
        return retryOperation(maxRetries - 1);
      }
      throw error; // Max retries reached
    });
}
```

Add your suggestion in the comments