JavaScript Promises

Coding Exercise Challenge

🚀 Dive into the World of JavaScript Promises with Our Interactive Coding Exercises!

Are you keen on mastering JavaScript Promises? We've crafted a series of engaging coding exercises to elevate your asynchronous programming skills. These exercises are designed to challenge your understanding and enhance your ability to handle complex operations in JavaScript.

🌐 Engaging Challenges Await:

- Basic Promise Creation: Discover the foundations of crafting promises.

- Promise Consumption: Learn how to effectively handle promise results.
- Chaining Promises: Explore the power and flexibility of promise chaining.
- Error Handling: Master the art of managing errors in asynchronous operations.
- Concurrent Promises: Delve into running multiple promises simultaneously with Promise.all.
- Resolved and Rejected Promises: Experiment with creating pre-resolved and pre-rejected promises.
- Promisifying Callbacks: Transform traditional callback functions into modern promise-based patterns.
- Async/Await Synergy: Uncover the elegance of using async/await with promises for cleaner code.
- The Race to Resolution: Understand how Promise.race can determine the outcome based on the fastest promise.

Whether you're a seasoned developer or just starting out, these exercises offer a great opportunity to deepen your JavaScript knowledge and improve your coding prowess.

Share your insights and experiences as you tackle these exercises. Let's foster a culture of learning and growth in our JavaScript journey!

Question 1: Basic Promise Creation

Q: How do you create a basic promise in JavaScript?

```
const myPromise = new Promise((resolve, reject) => {
  // Your asynchronous code here
  if (/* condition */) {
    resolve('Success');
  } else {
    reject('Error');
  }
});
```

Explanation:

A promise in JavaScript is created using the Promise constructor, which takes a function as an argument. This function has two parameters: resolve and reject, which are used to control the outcome of the promise. If the asynchronous operation is successful, resolve is called; otherwise, reject is called.

Question 2: Consuming a Promise

Q: How do you handle the result of a promise?

```
myPromise.then(
  (value) => { console.log(value); },
  (error) => { console.log(error); }
);
```

Explanation:

The then method is used to handle the result of a promise. It takes two functions as arguments: the first is called if the promise is resolved, and the second is called if the promise is rejected.

Question 3: Chaining Promises

Q: How do you chain multiple promises?

```
firstPromise()
   .then((result) => secondPromise(result))
   .then((newResult) => console.log(newResult))
   .catch((error) => console.log(error));
```

Explanation:

Promises can be chained using the then method. The result of the first promise is passed to the next promise in the chain. If any promise in the chain is rejected, the catch block will handle the error.

Question 4: Error Handling in Promises

Q: How do you handle errors in promises?

```
myPromise
   .then((value) => { console.log(value); })
   .catch((error) => { console.log(error); });
```

Explanation:

The catch method is used for error handling in promises. It is invoked when a promise is rejected or when an error is thrown in the then method.

Question 5: Promise.all Usage

Q: How do you execute multiple promises concurrently and wait for all of them to complete?

```
Promise.all([promise1, promise2, promise3])
  .then((results) => {
    console.log(results); // An array of results
  })
  .catch((error) => {
    console.log(error);
  });
```

Explanation:

Promise.all takes an array of promises and returns a single promise that resolves when all of the promises in the array have been resolved or rejects if any promise in the array rejects.

Question 6: Creating a Resolved Promise

Q: How do you create a promise that is already resolved?

```
const resolvedPromise = Promise.resolve('Resolved value');
```

Explanation:

Promise.resolve() creates a promise that is resolved with the given value. It is useful for converting values to promises.

Question 7: Creating a Rejected Promise

Q: How do you create a promise that is already rejected?

```
const rejectedPromise = Promise.reject('Error message');
```

Explanation:

Promise.reject() creates a promise that is already rejected with the provided reason. It’s used for testing and error handling purposes.

Question 8: Promisify a Callback Function

Q: How do you convert a callback-based asynchronous function to a promise-based one?

```
function promisifyAsyncFunction(asyncFunction) {
  return new Promise((resolve, reject) => {
    asyncFunction((err, result) => {
      if (err) {
        reject(err);
      } else {
        resolve(result);
      }
    });
  });
}
```

Explanation:

To promisify a callback function, wrap it in a new promise. The callback function is invoked inside the promise executor, and the resolve or reject functions are called based on the callback’s outcome.

Question 9: Async/Await with Promises

Q: How do you use async/await with a promise?

```
async function asyncFunction() {
   try {
      const result = await myPromise;
      console.log(result);
   } catch (error) {
      console.log(error);
   }
}
```

Explanation:

async/await syntax provides a more readable way to work with promises. The await keyword is used to wait for the promise to settle, and try/catch is used for error handling.

Question 10: Promise.race Usage

Q: How do you use Promise.race?

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

```
  .then((value) => console.log(value))
  .catch((error) => console.log(error));
```

Explanation:

Promise.race takes an array of promises and returns a promise that resolves or rejects as soon as one of the promises in the array resolves or rejects, with the value or reason from that promise.