Promise

Theory Questions

What is a Promise in JavaScript, and why is it used?

A Promise is an object representing the eventual completion or failure of an asynchronous operation. It's used to avoid callback hell and handle asynchronous tasks more cleanly.

What are the three states of a Promise?

- Pending: The initial state, operation not completed yet.
- Fulfilled: Operation completed successfully.
- Rejected: Operation failed.

How does a Promise differ from a callback function?

Promises handle errors more cleanly and allow chaining multiple asynchronous operations. They improve code readability compared to nested callbacks.

What is the purpose of .then(), .catch(), and .finally() in a Promise?

.then(): Handles successful resolution of the Promise.
.catch(): Handles errors or rejection of the Promise.
.finally(): Executes code after the Promise is settled, regardless of success or failure.

Can a Promise be rejected and resolved multiple times? Why or why not?

No, a Promise can only settle (resolve or reject) once. This ensures consistent behavior and avoids unexpected state changes.

What is promise chaining? How is it implemented?

Promise chaining means linking multiple .then() calls. Each .then() gets the resolved value of the previous one:

```
fetchData()
  .then((data) => process(data))
  .then((result) => save(result))
  .catch((error) => console.error(error));
```

How does error handling work in promise chains?

If any .then() in the chain throws an error, the next .catch() block will handle it.

What happens if you return a value inside a .then() block?

The returned value becomes the resolved value of the next .then() block.

What if you return a Promise inside a .then()? How does it affect the chain?

The next .then() waits for the returned Promise to resolve or reject before continuing.

How do you handle errors in Promises?

Use a .catch() block to handle errors:

```
fetchData()
  .then((data) => process(data))
  .catch((error) => console.error(error));
```

What is the difference between .catch() and .then(null, errorHandler)?

.catch() is a shorthand for .then(null, errorHandler) but is preferred for readability and error-handling conventions.

What happens if an error occurs but you don't handle it in a Promise chain?

The error is unhandled and may cause the application to crash or show a warning.

What is Promise.all() and how does it work?

Promise.all() runs multiple Promises in parallel and resolves only when all Promises succeed. If any Promise fails, it rejects with the error.

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

Promise.all(): Fails fast; rejects if any Promise fails.
Promise.allSettled(): Waits for all Promises to settle (resolve or reject) and provides their results.

How does Promise.race() work? Can you give an example?

Resolves or rejects as soon as the first Promise settles:

```
Promise.race([promise1, promise2])
  .then((result) => console.log(result))
  .catch((error) => console.error(error));
```

What is Promise.any()? How is it different from Promise.race()?

Promise.any(): Resolves with the first fulfilled Promise. Rejects only if all Promises fail.
Promise.race(): Resolves or rejects as soon as the first Promise settles.

How would you create a custom Promise?

```
const customPromise = new Promise((resolve, reject) => {
  const success = true;
  if (success) {
    resolve("Operation successful");
  } else {
    reject("Operation failed");
  }
})
```

Write a Promise to simulate an asynchronous task like a timer.

```
const delay = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

delay(1000).then(() => console.log("Executed after 1 second"));
```

How can you convert a callback-based function to a Promise-based one?

Use the Promise constructor:

```
const readFilePromise = (filePath) =>
  new Promise((resolve, reject) => {
    fs.readFile(filePath, 'utf8', (err, data) => {
      if (err) reject(err);
      else resolve(data);
    });
  });
```

What are the benefits of using Promises over traditional callback functions?

Improved readability and maintainability.
Chaining enables sequential execution.
Centralized error handling.
Comparison with Async/Await

How do Promises compare to async/await in JavaScript?

async/await is built on Promises but provides a cleaner, synchronous-looking syntax for handling asynchronous operations.

Can you rewrite a Promise chain using async/await? What are the key differences?

```
async function fetchData() {
  try {
    const data = await getData();
    const processed = await process(data);
    await save(processed);
  } catch (error) {
    console.error(error);
  }
}
Key Difference: async/await reduces nesting and is easier to read.
```

What is a "Promise hell"? How do you avoid it?

Promise hell happens when nested .then() calls make code unreadable. Avoid it by using chaining or async/await.

What happens if you forget to return a Promise in a .then() chain?

The next .then() will receive undefined and may cause unexpected behavior.

How do you handle a situation where multiple Promises need to be resolved, but some can fail without affecting the others?

Use Promise.allSettled() to process all results regardless of success or failure.

What is the difference between try-catch and .catch()?#

try-catch: Used for synchronous code and works with throw statements. It doesn't handle errors in asynchronous code like Promises.
.catch(): Specifically handles errors in Promises or asynchronous operations.
Example:

```
// try-catch for synchronous code
try {
JSON.parse("Invalid JSON"); // Throws error
} catch (error) {
console.error("Error caught:", error.message);
}
```

```
// .catch() for Promises
fetch("invalid-url")
.then((response) => response.json())
.catch((error) => console.error("Promise error caught:", error.message));
```

Can you combine try-catch with Promises?

Answer:
Yes, you can use try-catch with async/await to handle errors:

```
async function fetchData() {
try {
const response = await fetch("https://api.example.com/data");
const data = await response.json();
console.log(data);
} catch (error) {
console.error("Error caught:", error.message);
}
```

What happens if an error is not caught in a Promise?

Answer:
If an error is not caught using .catch(), it becomes an unhandled rejection, which might crash the application or trigger a warning.

Which is better: .catch() or try-catch?

Answer:
Use .catch() for handling Promise errors.
Use try-catch with async/await for better readability in complex asynchronous code.

Event Loop:

A mechanism that handles execution of JavaScript code, managing the call stack, and processing tasks from queues (microtask and macrotask).
It ensures non-blocking, asynchronous operations.

Microtasks:

Include operations like resolving Promises or queueMicrotask().
Executed after the current stack but before any macrotasks.

Macrotasks:

Include operations like setTimeout, setInterval, and DOM events.
Executed after microtasks and may trigger additional microtasks.

What is the Event Loop in JavaScript?

Answer:
The Event Loop ensures JavaScript executes tasks in a non-blocking way. It manages the execution of synchronous code, asynchronous code, microtasks, and macrotasks by moving tasks to the call stack as needed.

What are Microtasks in JavaScript?

Answer:
Microtasks are smaller tasks like Promise callbacks or queueMicrotask(). They are prioritized and executed before macrotasks, after the current synchronous code completes.

```
Promise.resolve().then(() => console.log("Microtask 1"));
queueMicrotask(() => console.log("Microtask 2"));
console.log("Synchronous code");
// Output:
// Synchronous code
// Microtask 1
// Microtask 2
```

What are Macrotasks in JavaScript?

Answer:
Macrotasks include setTimeout, setInterval, and I/O events. They are executed after all microtasks are cleared.

```
setTimeout(() => console.log("Macrotask"), 0);
Promise.resolve().then(() => console.log("Microtask"));
console.log("Synchronous code");
// Output:
// Synchronous code
// Microtask
// Macrotask
```

What is the order of execution for synchronous code, microtasks, and macrotasks?

Answer:
First, execute synchronous code.
Next, process microtasks (Promise callbacks, queueMicrotask).
Finally, process macrotasks (e.g., setTimeout, setInterval).

What happens if a microtask schedules another microtask?

Answer:
Newly scheduled microtasks are added to the queue and executed before macrotasks.

```
Promise.resolve().then(() => {
console.log("Microtask 1");
Promise.resolve().then(() => console.log("Microtask 2"));
});
setTimeout(() => console.log("Macrotask"), 0);
console.log("Synchronous code");
// Output:
// Synchronous code
// Microtask 1
// Microtask 2
// Macrotask
```

What is the difference between microtasks and macrotasks?

Answer:
Microtasks have higher priority and execute before macrotasks.
Examples of Microtasks: Promise callbacks, queueMicrotask.
Examples of Macrotasks: setTimeout, setInterval, DOM events.

What is the purpose of queueMicrotask()?

Answer:
It allows you to schedule a microtask explicitly, ensuring it runs after the current stack but before macrotasks.

```
queueMicrotask(() => console.log("Microtask"));
console.log("Synchronous code");
// Output:
// Synchronous code
// Microtask
```