Asynchronous JavaScript

> 🎓 **CS Perspective:** This section covers JavaScript's **concurrency model**, which uses a single-threaded **event loop** with non-blocking I/O. Unlike multi-threaded languages, JS achieves concurrency through **cooperative multitasking**—long operations yield control back to the event loop. Promises implement the **Promise/Future pattern** from concurrent programming, representing a value that may not yet exist. `async/await` is **syntactic sugar** over Promises, using **coroutines** (functions that can suspend and resume) under the hood. The microtask queue (Promises) vs. macrotask queue (setTimeout) distinction affects execution order. `Promise.all` enables **parallel execution** while `Promise.race` implements a **timeout pattern**. This model avoids race conditions and deadlocks common in multi-threaded programming.

📌 Synchronous vs Asynchronous

Synchronous	Asynchronous
Line by line	Non-blocking
Blocking	Callbacks run later
One task at a time	Multiple tasks "at once"

```
// Sync - blocks
const p = document.querySelector('.p');
p.textContent = 'Hello';
alert('Blocking!');

// Async - non-blocking
setTimeout(() => console.log('Later'), 2000);
console.log('Now'); // Runs first!
```

📌 AJAX & APIs

AJAX: Asynchronous JavaScript And XML
API: Application Programming Interface

```
// Modern: Fetch API
const response = await fetch('https://api.example.com/data');
const data = await response.json();
```

📌 Promises

A container for a future value:

```
const promise = fetch('https://api.example.com');

// States:
// - Pending (initial)
// - Fulfilled (success)
// - Rejected (error)
// Once settled, state cannot change
```

📌 Consuming Promises

.then() chain

```
fetch('https://restcountries.com/v3.1/name/portugal')
  .then(response => response.json())  // Returns promise
  .then(data => {
    console.log(data[0]);
    return fetch(`https://restcountries.com/v3.1/name/${neighbour}`);
  })
  .then(response => response.json())
  .then(data => console.log(data[0]))
  .catch(err => console.error(err))   // Catches ANY error
  .finally(() => console.log('Done')); // Always runs
```

Error Handling

```
fetch('url')
  .then(response => {
    if (!response.ok)
      throw new Error(`Error ${response.status}`);
    return response.json();
  })
  .catch(err => console.error(err.message));
```

📌 Async/Await

Syntactic sugar over promises:

```
const getCountry = async function(country) {
  try {
    // Await pauses async function until promise settles
    const res = await
```

```
  fetch(`https://restcountries.com/v3.1/name/${country}`);

      if (!res.ok) throw new Error('Country not found');

      const data = await res.json();
      console.log(data[0]);
      return data[0]; // Returns promise!

    } catch (err) {
      console.error(err);
      throw err; // Re-throw to propagate
    }
};

// Calling async function
getCountry('portugal');
console.log('FIRST'); // Runs first! (async is non-blocking)
```

Important Rules

1. `await` only works inside `async` functions
2. `async` functions always return a **promise**
3. Use `try/catch` for error handling

📌 Returning Values from Async Functions

```
const getJSON = async function() {
  const res = await fetch(url);
  return await res.json();
};

// Option 1: .then()
getJSON().then(data => console.log(data));

// Option 2: IIFE with async
(async function() {
  const data = await getJSON();
  console.log(data);
})();
```

📌 Running Promises in Parallel

Promise.all()

Runs all promises simultaneously, **short-circuits if any rejects**:

```
const getAll = async function(c1, c2, c3) {
  const data = await Promise.all([
    getJSON(`https://api.com/${c1}`),
    getJSON(`https://api.com/${c2}`),
    getJSON(`https://api.com/${c3}`)
  ]);
  console.log(data); // Array of results
};
```

Promise.race()

Returns first **settled** promise (fulfilled OR rejected):

```
const res = await Promise.race([
  getJSON('https://api.com/italy'),
  getJSON('https://api.com/egypt'),
  getJSON('https://api.com/mexico')
]);
console.log(res); // Fastest response

// Timeout pattern
const timeout = function(sec) {
  return new Promise((_, reject) =>
    setTimeout(() => reject(new Error('Timeout!')), sec * 1000)
  );
};

const res = await Promise.race([
  getJSON('https://api.com/data'),
  timeout(5) // Reject if > 5 seconds
]);
```

Promise.allSettled() (ES2020)

Returns all results, **never short-circuits**:

```
const res = await Promise.allSettled([
  Promise.resolve('Success'),
  Promise.reject('Error'),
  Promise.resolve('Success 2')
]);
// [{status: 'fulfilled', value: 'Success'},
//  {status: 'rejected', reason: 'Error'},
//  {status: 'fulfilled', value: 'Success 2'}]
```

Promise.any() (ES2021)

Returns first **fulfilled** promise, ignores rejections:

```
const res = await Promise.any([
  Promise.reject('Error'),
  Promise.resolve('Success'),
  Promise.resolve('Success 2')
]);
console.log(res); // 'Success'
```

📌 Promise Combinators Summary

Combinator	Returns	Short-circuits
`Promise.all`	All values	On first reject
`Promise.race`	First settled	On first settled
`Promise.allSettled`	All outcomes	Never
`Promise.any`	First fulfilled	On first fulfill

📌 Building Promises

```
const lotteryPromise = new Promise(function(resolve, reject) {
  console.log('Drawing...');

  setTimeout(function() {
    if (Math.random() >= 0.5) {
      resolve('You WIN 💰 ');
    } else {
      reject(new Error('You lost 💩 '));
    }
  }, 2000);
});

lotteryPromise
  .then(res => console.log(res))
  .catch(err => console.error(err));
```

Promisifying

Convert callback-based APIs to promises:

```
// Promisify setTimeout
const wait = function(seconds) {
  return new Promise(resolve =>
    setTimeout(resolve, seconds * 1000)
```

```
  );
};

wait(2).then(() => console.log('2 seconds passed'));

// Promisify geolocation
const getPosition = function() {
  return new Promise(function(resolve, reject) {
    navigator.geolocation.getCurrentPosition(resolve, reject);
  });
};

const pos = await getPosition();
```

📌 The Event Loop

JavaScript is **single-threaded** but handles async via:

1. **Call Stack** - Executes code
2. **Web APIs** - Handle async operations (setTimeout, fetch, DOM)
3. **Callback Queue** - Regular callbacks
4. **Microtask Queue** - Promise callbacks (PRIORITY!)

```
console.log('Start');

setTimeout(() => console.log('Timer'), 0);

Promise.resolve('Promise').then(res => console.log(res));

console.log('End');

// Output:
// Start
// End
// Promise (microtask - priority!)
// Timer (callback queue)
```

Priority

1. Sync code (call stack)
2. Microtasks (promises)
3. Callbacks (setTimeout, events)

📌 Top-Level Await (ES2022)

In **modules only**:

```
// Blocks module execution until resolved
const data = await fetch('https://api.example.com/data');
const json = await data.json();

console.log(json);
export default json;
```

⚠ **Blocking**: Can slow down module imports

📌 Handling Multiple Sequential Requests

```
// Sequential (slower)
const get3Countries = async function() {
  const [c1] = await getJSON(`https://api.com/portugal`);
  const [c2] = await getJSON(`https://api.com/canada`);
  const [c3] = await getJSON(`https://api.com/tanzania`);
  console.log([c1.capital, c2.capital, c3.capital]);
};

// Parallel (faster!)
const get3Countries = async function() {
  const data = await Promise.all([
    getJSON(`https://api.com/portugal`),
    getJSON(`https://api.com/canada`),
    getJSON(`https://api.com/tanzania`)
  ]);
  console.log(data.map(d => d[0].capital));
};
```

🧪 Quick Reference Cheatsheet

```
// Fetch
const res = await fetch(url);
const data = await res.json();

// Promise chain
fetch(url)
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err))
  .finally(() => console.log('Done'));

// Async/Await
async function getData() {
  try {
    const res = await fetch(url);
```

```
    if (!res.ok) throw new Error('Error!');
    return await res.json();
  } catch (err) {
    throw err;
  }
}

// Combinators
Promise.all([p1, p2, p3]);        // All must succeed
Promise.race([p1, p2, p3]);       // First to settle
Promise.allSettled([p1, p2, p3]); // All outcomes
Promise.any([p1, p2, p3]);        // First to succeed

// Create promise
new Promise((resolve, reject) => {
  if (success) resolve(value);
  else reject(new Error('Failed'));
});

// Quick utilities
Promise.resolve(value);
Promise.reject(new Error('Failed'));
```

✅ Exam Tips

1. **Promises are microtasks** - higher priority than callbacks
2. `async` functions **always return promises**
3. `await` only works inside `async` functions (or top-level in modules)
4. `catch` **handles any error** in the promise chain
5. `finally` runs **regardless** of success/failure
6. `Promise.all` **short-circuits** on first rejection
7. `Promise.race` returns first **settled** (not first fulfilled)
8. `Promise.any` ignores rejections, returns first **fulfilled**
9. Use `Promise.all` when requests are **independent**
10. **Microtask queue has priority** over callback queue
11. `response.json()` returns a **promise**
12. Always **re-throw errors** if not handling them completely