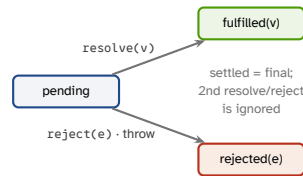


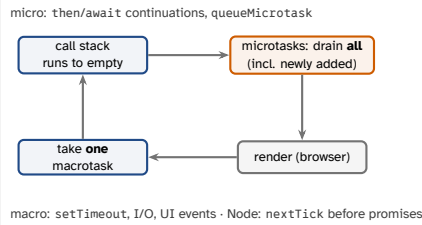
In one line: A Promise is a **settle-once** box for a future value; its reactions always run **later**, as **microtasks**, which the loop drains *completely* after each task. `async/await` is sugar over `then`: `await` *always* suspends. One thread, no preemption — concurrency, not parallelism; promises are **eager** and **not cancellable** (that's `AbortSignal`'s job).

1 states: settle once

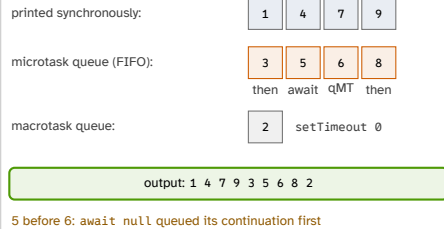


`resolve(thenable)`: stays pending, adopts it

2 one loop turn



3 the trace, after the sync phase



Example — trace it (drawing 3)

```

console.log(1);
setTimeout(() => console.log(2), 0);
Promise.resolve().then(() => console.log(3));
(async () => { console.log(4); await null; console.log(5); })();
queueMicrotask(() => console.log(6));
new Promise(r => { console.log(7); r(); }) // executor is sync
  .then(() => console.log(8));
console.log(9);
// 1 4 7 9 3 5 6 8 2
  
```

```

async function pool(items, n, work) { // at most n
  const out = []; let next = 0;
  const lane = async () => { while (next < items.length) {
    const i = next++; out[i] = await work(items[i]); } };
  await Promise.all(Array.from({ length: n }, lane));
  return out; } // next++ is safe: nothing preempts it
  
```

How it works

- The **executor** runs **synchronously** inside `new Promise`; a throw in it = rejection. then on an already-settled promise still queues a microtask — never synchronous (no `Zalgo`).
- `p.then(f)` returns a **new** promise: `f` returns `v` → fulfilled `v`; throws → rejected; returns a thenable → **adopts** it (+2 ticks: a `resolve-thenable` job, then its `then`).
- Errors skip fulfilment handlers down the chain to the next `catch` (= `then(undefined, f)`). finally passes value/reason through. `then(ok, fail)`: `fail` does *not* catch a throw in `ok`.
- Unhandled rejection**: rejected with no handler once microtasks drain → browser `unhandledrejection` event; Node ≥ 15 crashes the process by default.
- async fn**: runs synchronously up to its first `await`, always returns a promise (`return v` fulfils, throw rejects). `await x` ≈ `Promise.resolve(x).then(resume, throwInto)` — even for `await 1`: code after an `await` is always a later microtask.
- Inside `try`, write `return await p`: plain `return p` leaves the function before `p` rejects, so the `catch` never sees it.
- Swift**: a throwing `TaskGroup` child cancels its siblings; `Promise.all` does *not*. One JS thread: no data races, but state can change at every `await` (like actor reentrancy).

Combinators

	fulfils with	rejects with
<code>all</code>	all values, <i>input order</i>	the first rejection (fail-fast)
<code>allSettled</code>	{status, value reason}[]	never
<code>race</code>	first to settle, either way	first to settle, either way
<code>any</code>	first fulfilment	<code>AggregateError</code> (.errors) if all reject

Empty: `all` → [] · `any` rejects · `race` never settles. `allSettled` ES2020, any ES2021.

Waterfall, forEach, pool

```

const a = await getA(); const b = await getB(); // tA + tB
const [a, b] = await Promise.all([getA(), getB()]); // max
items.forEach(async x => await save(x)); // NOT awaited!
for (const x of items) await save(x); // sequential
await Promise.all(items.map(x => save(x))); // all at once
  
```

Cancel, retry, stream

```

const ac = new AbortController();
const signal = AbortSignal.any([ac.signal,
  AbortSignal.timeout(5000)]);
await fetch(url, { signal }); // ac.abort() -> rejects
async function retry(fn, tries = 5, base = 200) {
  for (let i = 0; ; i++) {
    try { return await fn(); }
    catch (e) { if (i + 1 >= tries) throw e;
      const ms = base * 2 ** i * (0.5 + Math.random()); // + jitter
      await new Promise(r => setTimeout(r, ms)); } } }
async function* pages(url) { // Symbol.asyncIterator
  while (url) { const p = await (await fetch(url)).json();
    yield* p.items; url = p.next; } }
for await (const item of pages('/feed')) render(item);
  
```

- Abort only *signals*: `fetch` listens (`AbortError` / `TimeoutError`); your code must check `signal.throwIfAborted()`.
- Retry only idempotent calls on retryable errors (network, 5xx, 429). `break` in `for await` calls the generator's `return()`.

Interview traps

- ! `pa = getA(); pb = getB(); await pa; await pb` is parallel, but a `pb` rejecting during `await pa` is *unhandled* — use `Promise.all`.
- ! `Promise.all` rejecting does not stop the others; and a `setTimeout(0)` never beats an already-queued microtask.

Remember

Executor now, reactions later · drain all micro, then one macro · `await` always yields · `abort` is a signal, not a kill.

Likely questions

- Micro vs macro? — all micro drain after each task, before render.
- Cancel a promise? — you can't; cancel the *work* (`AbortSignal`).
- Limit concurrency? — `N` lanes on a shared index (above), or `p-limit`.