Explain Promises in detail

explain promise in details and from scratch

Sure Vipul, I'll explain **Promises in JavaScript** from **absolute scratch**, in a very **simple, beginner-friendly, real-life way** so you'll never forget it.
This is long but crystal clear — exactly how you like it.

✅ 1. Why do we need Promises?

JavaScript is **single-threaded**, meaning it can do **only one thing at a time**.

But real programs need time-taking tasks like:

- Calling APIs
- Reading files
- Waiting for timers
- Fetching data from database
- Loading images

These tasks **should not block** the main thread.

So JavaScript handles these tasks **asynchronously**.

Earlier, we used **callbacks**, but they caused:

- callback hell
- pyramid code
- difficult error handling

So we needed something better → **Promises**.

✅ 2. What is a Promise? (Very Simple Definition)

A **Promise** is like a **future value**.

It is a **placeholder** for a value that you will get **later**.

Real-life example:

You order food on Zomato.

- You **place the order** → Promise CREATED

- Order can be:
  - delivered → **resolved**
  - cancelled → **rejected**

You don't know the result immediately; you wait for it → **asynchronous**.

✅ 3. States of a Promise

A Promise has **three states**:

1. **Pending**
   (order placed, waiting)
2. **Fulfilled/Resolved**
   (order delivered)
3. **Rejected**
   (order cancelled)

Once resolved/rejected → it becomes **settled** and cannot change again.

✅ 4. Syntax of a Promise

```js
const promise = new Promise((resolve, reject) => {
  // async code

  if (/* success */) {
    resolve("Success message");
  } else {
    reject("Error message");
  }
});
```

The **executor function** (inside new Promise()) takes two functions:

- **resolve()** → when task is successful
- **reject()** → when task fails

✅ 5. How to use a Promise?

Using `.then()` and `.catch()`:

```js
promise
  .then((result) => {
    console.log("Resolved:", result);
  })
  .catch((error) => {
    console.log("Rejected:", error);
  });
```

✅ 6. Example from scratch

Example: Promise that completes after 2 seconds

```
js

const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Data is ready!");
  }, 2000);
});

myPromise.then(data => {
  console.log(data);
});
```

Output after 2 sec:

```
arduino

Data is ready!
```

✅ 7. Rejection Example

```
js

const myPromise = new Promise((resolve, reject) => {
  const success = false;

  if (success) {
    resolve("Everything OK");
  } else {
    reject("Something went wrong");
  }
});

myPromise
  .then(msg => console.log(msg))
  .catch(err => console.log(err));
```

✅ 8. Promise Chaining

You can chain `.then()` one after another.

```
js

new Promise((resolve) => {
  resolve(10);
})
  .then(num => num * 2) // 20
  .then(num => num + 5) // 25
  .then(result => console.log(result)); // 25
```

✅ 9. Catching Errors in Chaining

```js
new Promise((resolve, reject) => {
  reject("Error in first step");
})
  .then(() => console.log("won’t run"))
  .catch(err => console.log(err));
```

✅ 10. Promise.all()

Runs multiple promises in parallel → waits for all to finish.

```js
Promise.all([
  Promise.resolve(1),
  Promise.resolve(2),
  Promise.resolve(3)
]).then(data => console.log(data));
```

Output:

```csharp
[1, 2, 3]
```

If **any one fails**, all fails.

✅ 11. Promise.race()

Returns first settled promise (resolved or rejected).

```js
const p1 = new Promise(res => setTimeout(() => res(1), 1000));
const p2 = new Promise(res => setTimeout(() => res(2), 500));

Promise.race([p1, p2]).then(console.log); // 2
```

✅ 12. Promise.any()

Returns first **resolved** promise (ignores rejects).

✅ 13. async/await (Promise wrapper)

`async/await` makes promises look like synchronous code.

```js
async function getData() {
  const data = await fetch("/api");
  console.log("data received");
}
```

Behind the scenes → **it's still a Promise**.

⭐ FINAL SUMMARY (Super Simple)

Term	Meaning
Promise	Future value (comes later)
resolve	Success
reject	Failure
pending	Waiting
fulfilled	Success
rejected	Failed
.then()	Handle success
.catch()	Handle failure
async/await	Cleaner syntax for using promises

Want next?

I can also explain with:

✅ Visual diagrams
✅ Real-world examples (Zomato, Swiggy, Ola)
✅ Interview questions on Promises
✅ Practice questions with answers
✅ Promise execution order & event loop concepts

Just tell me **"yes"**.

how this new Promise craetes promise how javascript get to knows that by writing this we are trying to craete promise. Nad from where it's god get accessed of promise by writing this.

Great question Vipul — and this is the **REAL fundamental thing** almost nobody explains.

You're asking:

**"How does JavaScript know that `new Promise()` means create a promise?

Where does **`Promise`** come from?
How does this keyword work internally?"**

Let's break it down **from scratch** and very simply.

✅ 1. Where does `Promise` come from?

JavaScript provides many **built-in objects**, for example:

- **`Array`**
- **`String`**
- **`Date`**
- **`Math`**
- **`JSON`**
- **Promise**

These are called **built-in global objects**.

They are part of the **JavaScript engine** (V8, SpiderMonkey, JavaScriptCore).

When your browser or Node.js starts JavaScript:

✔️ It loads all built-in constructors into the global scope.

So automatically, these are created:

```js
Array
String
Promise
Object
Boolean
```

They are available everywhere without importing anything.

That's why you can write:

```js
new Array()
new Promise()
new Date()
```

JavaScript already **knows** these names because the engine creates them before your code runs.

✅ 2. Is `Promise` a function?

Yes.

`Promise` is actually a **constructor function** built into JavaScript.

You can check:

```js
console.log(typeof Promise); // "function"
```

So when you write:

```js
new Promise(...)
```

You are calling the **Promise constructor**.

Just like:

```js
new Array()
new Date()
new Number()
```

✅ 3. How JavaScript recognizes `new Promise(...)`?

Because the JavaScript engine contains an internal C++ implementation something like:

```javascript
function Promise(executor) {
  // internal code
}
```

You are not allowed to see this code — it is part of the browser / Node.js engine.

When you do:

```js
new Promise((resolve, reject) => {})
```

JavaScript sees:

- “Promise” is a built-in constructor
- “new” means create a new object from that constructor
- Call the internal C++ implementation of Promise

✅ 4. What happens internally when you call `new Promise()`?

1. A new empty Promise object is created
2. JavaScript sets its internal state to **pending**
3. JavaScript calls the executor function you pass:

```js
(resolve, reject) => { ... }
```

4. The engine itself gives you two hidden functions:
    - `resolve()`
    - `reject()`
5. If you call resolve → status becomes **fulfilled**
6. If you call reject → status becomes **rejected**

✅ 5. How does JavaScript connect Promise with .then?

Every promise object has internal methods that come from its **prototype**:

```js
Promise.prototype.then
Promise.prototype.catch
Promise.prototype.finally
```

These are automatically attached because when you do:

```js
new Promise()
```

JavaScript sets:

```ini
promise.__proto__ = Promise.prototype
```

So `.then()` is automatically available.

✅ 6. Who created the Promise object originally?

ECMAScript (JavaScript language standard) defined Promises in **ES6 (2015)**.

Browser vendors (Chrome, Firefox, Safari) and Node.js implemented it inside their engines.

So **Promise** becomes part of the JavaScript specification, like Array or Map.

That's why all engines must support it.

🔥 SUPER SIMPLE SUMMARY

🤔 "How does JavaScript know `new Promise()` creates a promise?"

➡️ Because `Promise` is a **built-in global constructor function** provided by the JavaScript engine.

🤔 "From where is Promise accessed?"

➡️ It is available in the **global scope**, created by the engine even before your code runs.

🤔 "How does new Promise work?"

➡️ It calls the **internally implemented Promise constructor**, which sets up pending → fulfilled/rejected states.

🤔 "How does Promise have .then and .catch?"

➡️ These functions are on `Promise.prototype`, attached automatically to every new promise.

If you want, I can also explain:

✅ Internals of Promise in event loop (Microtask queue)
✅ Diagram of how Promise moves from pending → resolved
✅ How resolve() and reject() are connected internally
✅ 10 interview questions on Promises with answers

Just say *"explain more internals"* or whatever you want.

✅ Internals of Promise in event loop (Microtask queue)
✅ Diagram of how Promise moves from pending → resolved
✅ How resolve() and reject() are connected internally
✅ 10 interview questions on Promises with answers
and aslo from where we get resolve and reject executor frunction from prototype of promise or from some where else

Perfect Vipul — I'll explain **EVERYTHING** in one place:

- ⭐ How Promise works internally with **Event Loop & Microtask Queue**
- ⭐ Diagram of Promise → pending → fulfilled/rejected

- ⭐ How **resolve** and **reject** are created
- ⭐ Whether resolve/reject come from prototype or somewhere else
- ⭐ 10 interview-level questions with answers

All explained in **simple beginner-friendly language**.

🟦 1. How Promises Work Internally in Event Loop (Microtask Queue)

JavaScript uses 3 important queues:

1️⃣ Call Stack → main code runs here

2️⃣ Callback Task Queue → setTimeout, DOM events

3️⃣ Microtask Queue → Promise `.then()`, `catch()`, `finally()`, `queueMicrotask()`, `MutationObserver`

⚡ **Microtask queue always runs before the callback/task queue**.

🔥 Internal Flow of Promise

Example:

```js
console.log("A");

const p = new Promise((resolve, reject) => {
  resolve("done");
});

p.then((v) => {
  console.log("B");
});

console.log("C");
```

Internally:

1️⃣ Call stack runs top to bottom

- prints `"A"`

2️⃣ `new Promise(...)`

- promise created
- executor runs **immediately**
- resolve("done") is called

- Promise state becomes **fulfilled**
- `.then()` callback is pushed into **microtask queue**

3️⃣ prints **`"C"`**

4️⃣ **Call stack empty → Event loop checks microtasks**

- microtask queue has the `.then()` callback
- print **`"B"`**

Output:

```
css

A
C
B
```

🟦 2. How resolve() and reject() Are Created (MOST IMPORTANT CONCEPT)

This is what you asked:

> From where do we get resolve and reject? From prototype of promise or somewhere else?

Answer:

❌ NOT from prototype

❌ NOT from user code

✔️ resolve() and reject() are created internally by the JavaScript engine (C++ code inside V8/SpiderMonkey/JavaScriptCore).

Promise constructor has internal logic like:

```
javascript

function Promise(executor) {
   let resolve = internalCreateResolveFunction(this);
   let reject = internalCreateRejectFunction(this);

   executor(resolve, reject);
}
```

So:

- The engine **creates** **`resolve`** and **`reject`**
- They are passed as arguments to your executor function
- They are NOT stored on **`Promise.prototype`**

- They are special internal functions that can change the promise's internal state

---

🟩 3. Internal Diagram (Pending → Fulfilled / Rejected)

Here is a simplified mental diagram:

```lua
                       +------------------------+
                       |    new Promise()       |
                       |------------------------|
                       |  state: "pending"      |
                       |  value: undefined      |
                       +----------+-------------+
                                  |
                                  | resolve(value)
                                  v
                     +-----------------------------+
                     |   state: "fulfilled"        |
                     |   value: value passed       |
                     +-----------------------------+

                                  |
                                  | reject(error)
                                  v
                     +-----------------------------+
                     |   state: "rejected"         |
                     |   value: error passed       |
                     +-----------------------------+
```

🟩 4. Internal Flow Diagram with Microtask Queue

```pgsql
new Promise(executor)
        |
        +--> creates resolve() and reject() internally
        |
executor(resolve, reject) runs immediately
        |
resolve() or reject()
        |
        +--> Promise state changes (pending → fulfilled/rejected)
        |
.then(callback)
        |
        +--> callback is put into MICROTASK QUEUE
        |
Event Loop
        |
        +--> Executes microtask queue BEFORE task queue
```

🟦 5. Internal Implementation (Simplified)

Here is a simple simulation of how promise works:

```js
function MyPromise(executor) {
  let state = "pending";
  let value;

  function resolve(val) {
    if (state !== "pending") return;
    state = "fulfilled";
    value = val;
    queueMicrotask(() => thenCallback && thenCallback(value));
  }

  function reject(err) {
    if (state !== "pending") return;
    state = "rejected";
    value = err;
    queueMicrotask(() => catchCallback && catchCallback(value));
  }

  let thenCallback;
  let catchCallback;

  this.then = function (cb) {
    thenCallback = cb;
    return this;
  };

  this.catch = function (cb) {
    catchCallback = cb;
    return this;
  };

  executor(resolve, reject);
}
```

🟦 6. 10 Most Important Interview Questions on Promises

1. What is a Promise?

A placeholder for a future value; represents success or failure of an async task.

---

2. States of a Promise?

- pending
- fulfilled
- rejected

---

3. What is Promise chaining?

Using `.then()` one after another where each then returns a new promise.

---

4. What is the difference between microtask queue and callback queue?

- Microtask: Promise callbacks
- Callback: setTimeout, events

- Microtask runs **before** callback queue.

---

5. What is Promise.all?

Runs all promises in parallel → fails if any one fails.

---

6. What is Promise.race?

Returns result of **first settled promise**.

---

7. What is Promise.any?

Returns **first fulfilled** promise; ignores rejects.

---

8. Does executor of Promise run asynchronously?

No. **Executor runs immediately** and synchronously.

Only `.then/catch/finally` run async.

---

9. Why does `.then()` run after synchronous code?

Because `.then()` callbacks go to **microtask queue**.

---

10. Does resolve() immediately change state?

Yes, synchronous.
But `.then()` handlers run **asynchronously** via microtask queue.

🟦 FINAL SUMMARY FOR YOU

✔️ Promise is a built-in global object

✔️ new Promise() creates resolve() and reject() internally

✔️ They are NOT from prototype

✔️ resolve/reject are special internal functions created by JS engine

✔️ .then() goes into microtask queue

✔️ Microtasks run before setTimeout callbacks

✔️ Promise follows pending → fulfilled/rejected lifecycle

If you want, I can also teach:

✅ Difference between callback, promise, async-await
✅ Full event loop diagram
✅ Real API examples using promises
✅ 30 interview questions on promises

Just say **"continue"**.

8. Does executor of Promise run asynchronously?

No. Executor runs immediately and synchronously.

Only .then/catch/finally run async.

unable to understand this point

No problem Vipul — this is a **very important concept** and also the **MOST confusing** part of Promises.

I'll explain it with **super-simple examples**, **execution order**, and **real-life analogy** so it becomes 100% clear.

🟥 First: What does "executor" mean?

Executor is this part:

```js
new Promise((resolve, reject) => {
  // THIS function is the executor
});
```

The executor is the function you pass to Promise.

🟦 Key Rule

❗ The executor function runs immediately, synchronously, and on the main thread.

But...

❗ The `.then()` callback runs asynchronously (microtask queue).

This difference is what many people get confused about.

🟩 Let's prove it with the simplest example

```js
console.log("1");

new Promise((resolve, reject) => {
  console.log("2");  // executor
  resolve();
}).then(() => {
  console.log("3");  // then callback
});

console.log("4");
```

📌 Step-by-step execution:

1 → printed
Then Promise starts
2 → printed immediately
Promise resolved
then() is added to microtask queue
4 → printed
Call stack empty → event loop runs microtasks
3 → printed

Output:

```
1
2
4
3
```

🟧 Why does executor run BEFORE console.log("4")?

Because executor runs **synchronously**.

It blocks the code until it finishes.

`.then()` goes to microtask queue so runs later.

🟦 Why does .then() run AFTER console.log("4")?

Because `.then()` callback runs **asynchronously**, after synchronous code finishes.

🟪 Super Simple Real-Life Analogy

📦 Promise executor = “Restaurant takes your order immediately”

When you go to a restaurant:

- You tell the waiter → *executor runs immediately*

- He writes down your order → synchronous

But…

🍽️ `.then()` = “food served later”

After taking your order, the kitchen prepares food asynchronously.

You receive food **later**, not immediately.

So:

- **Executor = immediate action**
- **Then = later action**

🟨 Better Code Example: Clear Difference

```js
new Promise((resolve) => {
  console.log("Executor starts"); // runs immediately
  resolve();
});

console.log("Synchronous code");

Promise.resolve().then(() => {
  console.log("Then callback"); // runs later
});
```

Output:

```css
Executor starts
Synchronous code
Then callback
```

🟥 Why executor can't be async?

Because executor is meant to:

- Set up the promise
- Start the async operation
- Define how promise should resolve/reject

JavaScript MUST create the promise **before** anything else happens.

So executor runs immediately, ensuring:

- Promise is created
- Executor has a chance to call resolve/reject

`.then()` must wait until executor finishes.

🟦 Internal Flow (Very Important):

```
markdown

Call Stack:
-----------
1. run executor (synchronous)
2. synchronous code continues
3. when everything done → event loop checks microtask queue
4. run .then() callbacks
```

🟩 FINAL SUPER-SIMPLE SUMMARY (NEVER FORGET THIS)

❗ Executor = synchronous

Runs immediately during promise creation.

❗ Then/Catch/Finally = asynchronous

Run later through the **microtask queue**.

❗ That’s why executor runs BEFORE console.log("4")

But `.then()` runs AFTER it.

Want to go even deeper?

I can explain:

- Why microtask queue runs before callback queue
- How Promise job queue works in the V8 engine
- Draw a full diagram of event loop, microtask, call stack, etc
- Explain how async/await uses promises internally

Just say **“yes explain deeper”**.

yes

Perfect Vipul — let’s go **deep into Promise internals, event loop, microtask queue, and async/await**.
I’ll make it **beginner-friendly** with diagrams and step-by-step explanation.

🟦 1. JavaScript Execution Model (Basics)

JavaScript has **single-threaded execution**. This means:

- Only **one thing runs at a time** (main thread)
- Long tasks (network requests, timers) **cannot block UI**
- So JavaScript uses an **event loop**.

Components

Component	Role
Call Stack	Where synchronous code executes
Web APIs / Node APIs	Handle async tasks (setTimeout, fetch, file I/O)
Callback Queue / Task Queue	Holds callbacks from Web APIs for later execution
Microtask Queue / Job Queue	Holds `.then()/.catch()` callbacks, `queueMicrotask`, `MutationObserver`
Event Loop	Picks the next thing to execute

🟦 2. Event Loop + Microtask Queue

1. Run **synchronous code** in call stack
2. When call stack empty, **run all microtasks** in order
3. Then pick one task from **callback queue** (setTimeout, events)
4. Repeat forever

Priority

```
cpp

Microtask queue > Callback queue
```

- Microtasks are executed **before any task queue callback**, even if the task is scheduled earlier.

🟦 3. Promise Internals

When you create a promise:

```
js

const p = new Promise((resolve, reject) => {
  // executor
});
```

Internally:

1. JavaScript **creates a Promise object**
    - state = **`"pending"`**
    - value = undefined
2. Executor function runs **immediately** (synchronously)
3. JS engine creates **internal resolve() and reject() functions**
4. When you call **`resolve(value)`** or **`reject(error)`**:
    - JS **changes promise state** (pending → fulfilled/rejected)
    - Adds **`.then()`** or **`.catch()`** callbacks to **microtask queue**

🟦 4. Microtask vs Task Queue Demo

```js
console.log("Start");

setTimeout(() => console.log("Timeout"), 0);

Promise.resolve().then(() => console.log("Promise 1"));
Promise.resolve().then(() => console.log("Promise 2"));

console.log("End");
```

Step-by-step:

1. **`console.log("Start")`** → prints **Start**
2. **`setTimeout(..., 0)`** → schedules callback in **task queue**
3. **`Promise.resolve().then(...)`** → schedules callbacks in **microtask queue**
4. **`console.log("End")`** → prints **End**
5. Call stack empty → microtasks run first:
    - prints **Promise 1**, then **Promise 2**
6. Task queue runs → prints **Timeout**

Output:

```javascript
Start
End
Promise 1
Promise 2
Timeout
```

✅ Key takeaway: **Promises always run before setTimeout, even with 0ms delay.**

🟦 5. Executor vs Then (Synchronous vs Asynchronous)

```js
console.log("1");

new Promise((resolve) => {
  console.log("2"); // executor: runs immediately
  resolve("Done");
}).then((v) => {
  console.log("3"); // then: runs asynchronously
});

console.log("4");
```

Output:

```
1
2
4
3
```

- Executor = synchronous → prints **2 immediately**
- Then callback → asynchronous → prints **3 later**

🟦 6. Async/Await Internals

`async/await` is **just syntactic sugar for promises**.

```js
async function fetchData() {
  console.log("A");
  const data = await Promise.resolve("Hello");
  console.log("B");
}

console.log("Start");
fetchData();
console.log("End");
```

Step-by-step:

1. `console.log("Start")` → prints **Start**
2. `fetchData()` runs:
   - prints **A**
   - `await Promise.resolve("Hello")` → suspends function
   - `.then()` added to **microtask queue**
3. `console.log("End")` → prints **End**
4. Microtask queue → resumes `fetchData` → prints **B**

Output:

```
css

Start
A
End
B
```

🟦 7. Full Internal Flow Diagram

```
markdown

Call Stack
----------
1. Synchronous code
2. new Promise(executor) → executor runs immediately

Executor calls resolve() / reject()
----------------------------------
- Internal state changes (pending → fulfilled/rejected)
- Callbacks added to Microtask Queue

Event Loop
----------
1. Call Stack empty
2. Run Microtask Queue (Promise then/catch/finally)
3. Run Task Queue (setTimeout, events)
4. Repeat
```

🟦 8. How resolve() / reject() Work Internally

- When you call `resolve(value)`:

```
text

PromiseInternalState.status = "fulfilled"
PromiseInternalState.value = value
MicrotaskQueue.push(thenCallbacks)
```

- When you call `reject(error)`:

```
text

PromiseInternalState.status = "rejected"
PromiseInternalState.reason = error
MicrotaskQueue.push(catchCallbacks)
```

✅ These functions are **created by JS engine internally**, not from prototype.

🟦 9. Summary

Concept	Behavior
Executor	Runs **synchronously**
resolve / reject	Created **internally** by engine, change state
`.then/.catch/.finally`	Runs asynchronously → **microtask queue**
Microtask queue priority	Runs **before task queue** (setTimeout, events)
Async/await	Syntactic sugar → uses Promise internally
Promise states	pending → fulfilled/rejected

Vipul, if you want, I can **draw a full visual diagram showing**:

- Call Stack
- Microtask queue
- Task queue
- Executor → resolve → then
- Async/await → exactly how it moves

It will make **everything crystal clear in one glance**.

Do you want me to make that diagram?