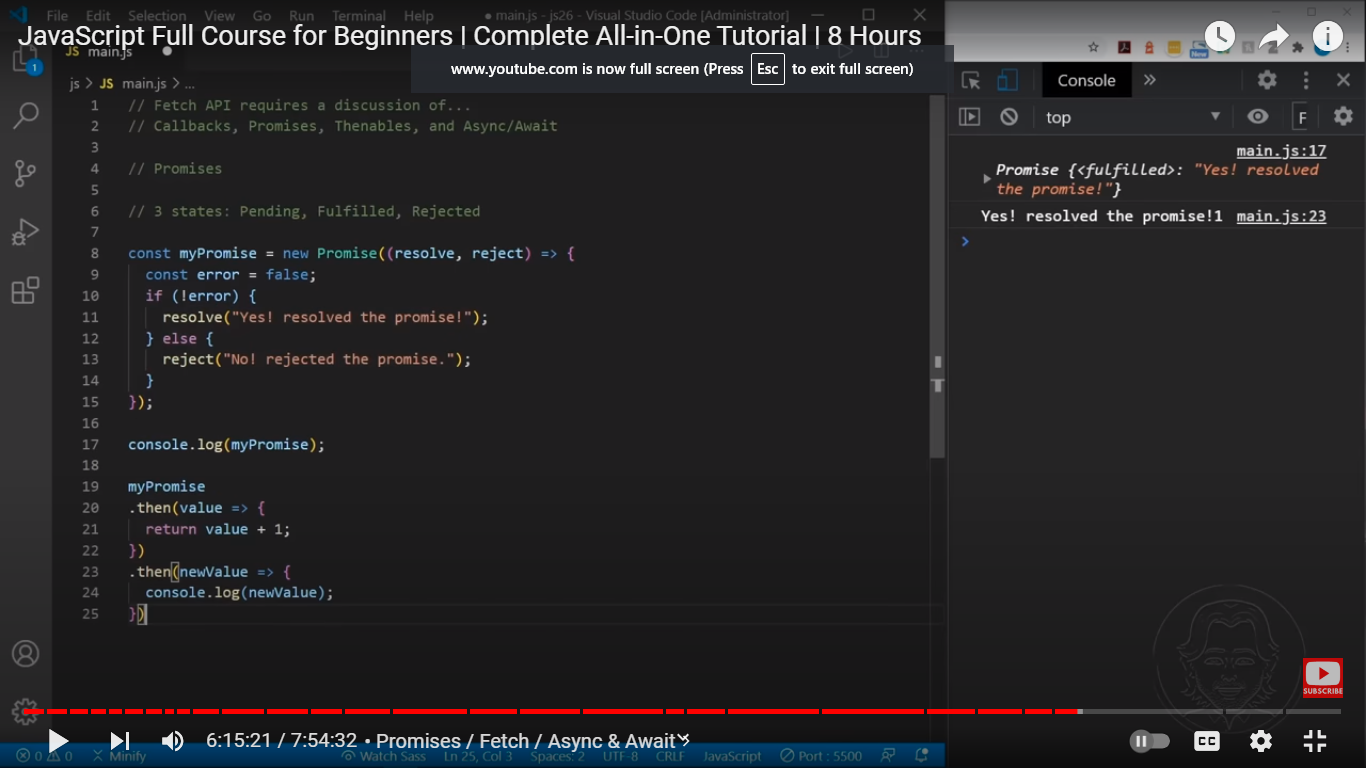
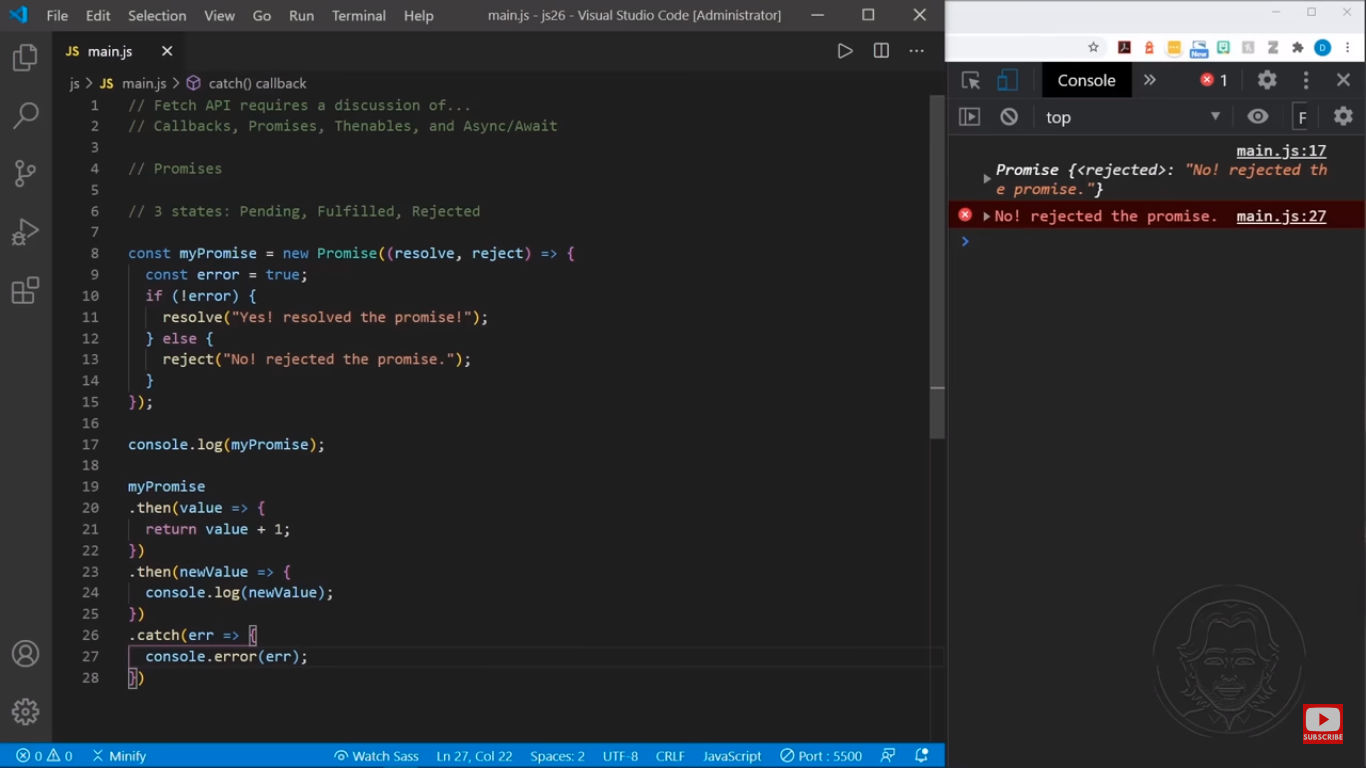
Promises

**Promise**

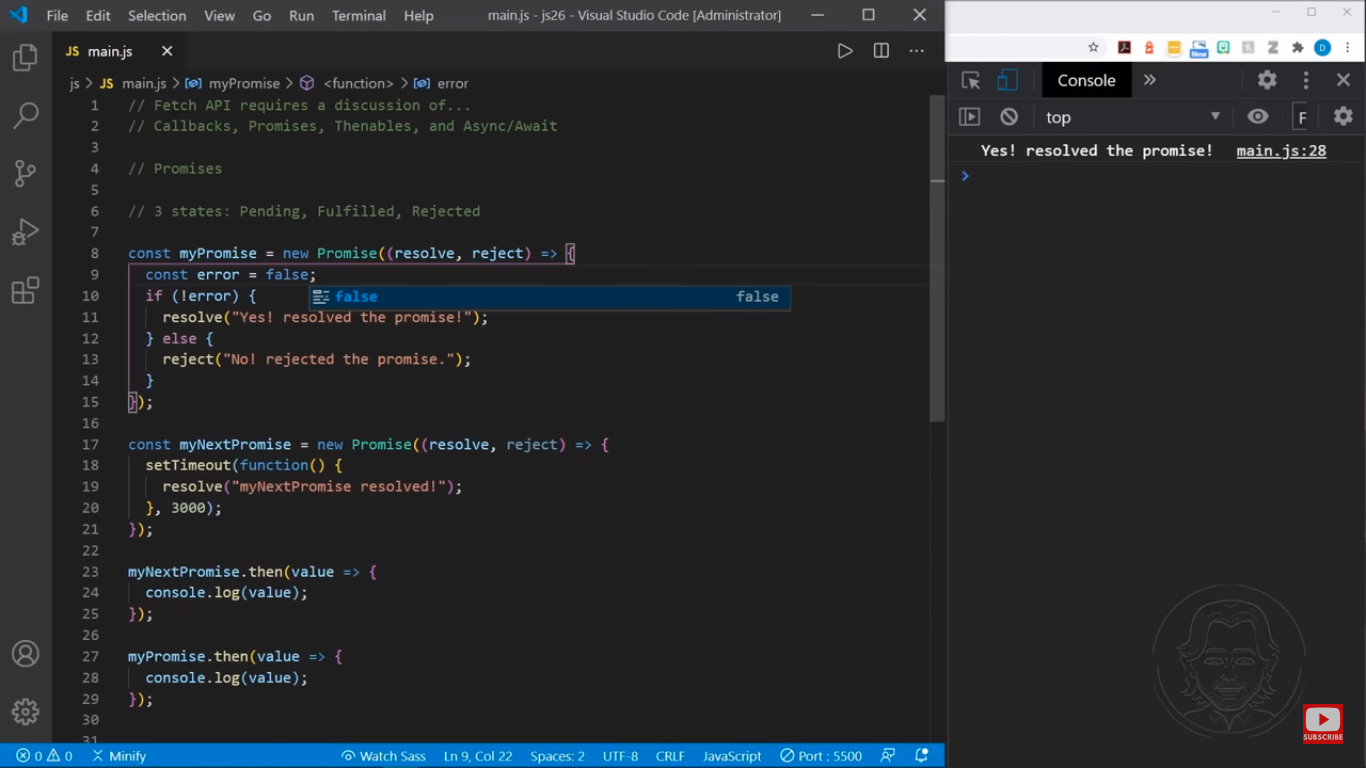
The **Promise** object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

There are three stages of promises: Pending, Fulfilled, Rejected

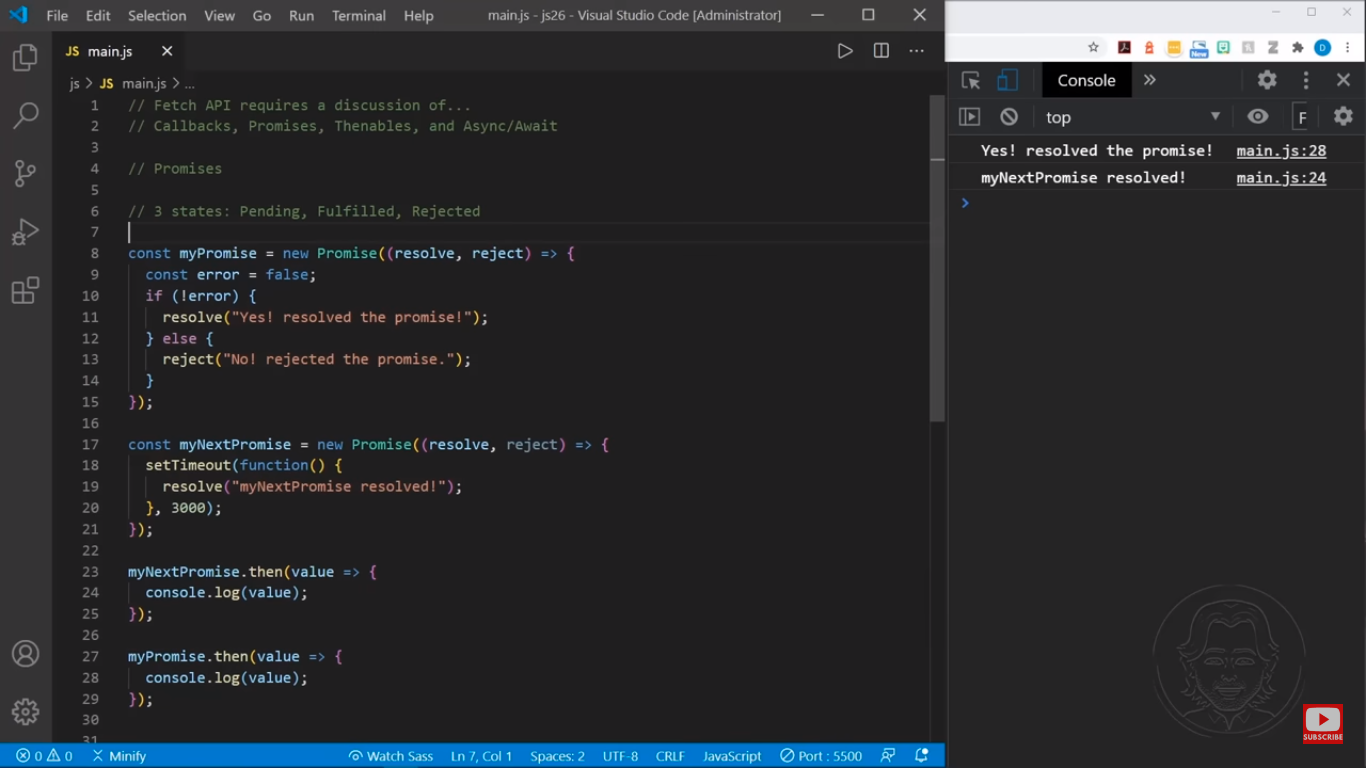




IMP Examples



3 seconds later



The js promises function object is “you go ahead I catch up later”

the myNextPromises object is called first and then, myPromises is called after MyNextPromieses but the myPromises is executed first because MyNextPromises is delayed 3 second

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#description)

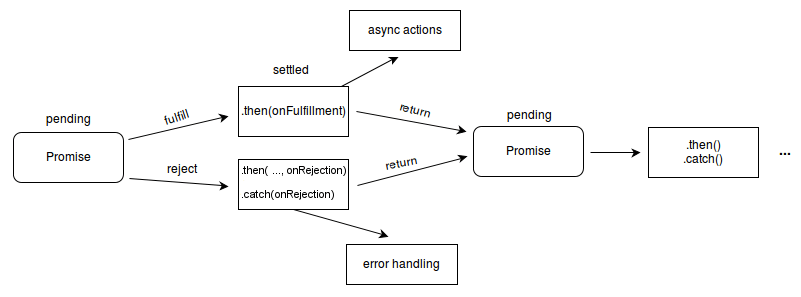
A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a *promise* to supply the value at some point in the future.

A Promise is in one of these states:

* *pending*: initial state, neither fulfilled nor rejected.
* *fulfilled*: meaning that the operation was completed successfully.
* *rejected*: meaning that the operation failed.

The *eventual state* of a pending promise can either be *fulfilled* with a value or *rejected* with a reason (error). When either of these options occur, the associated handlers queued up by a promise's then method are called. If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

A promise is said to be *settled* if it is either fulfilled or rejected, but not pending.



JS

Copy to Clipboard

new Promise((resolveOuter) => {

resolveOuter(

new Promise((resolveInner) => {

setTimeout(resolveInner, 1000);

}),

);

});

This promise is already resolved at the time when it's created (because the resolveOuter is called synchronously), but it is resolved with another promise, and therefore won't be fulfilled until 1 second later, when the inner promise fulfills. In practice, the "resolution" is often done behind the scenes and not observable, and only its fulfillment or rejection are.

[**Chained Promises**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#chained_promises)

The methods [Promise.prototype.then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then), [Promise.prototype.catch()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch), and [Promise.prototype.finally()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/finally) are used to associate further action with a promise that becomes settled. As these methods return promises, they can be chained.

The .then() method takes up to two arguments; the first argument is a callback function for the fulfilled case of the promise, and the second argument is a callback function for the rejected case. Each .then() returns a newly generated promise object, which can optionally be used for chaining; for example:

JS

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

setTimeout(() => {

resolve("foo");

}, 300);

});

myPromise

.then(handleFulfilledA, handleRejectedA)

.then(handleFulfilledB, handleRejectedB)

.then(handleFulfilledC, handleRejectedC);

Processing continues to the next link of the chain even when a .then() lacks a callback function that returns a Promise object. Therefore, a chain can safely omit every *rejection* callback function until the final .catch().

Handling a rejected promise in each .then() has consequences further down the promise chain. Sometimes there is no choice, because an error must be handled immediately. In such cases we must throw an error of some type to maintain error state down the chain. On the other hand, in the absence of an immediate need, it is simpler to leave out error handling until a final .catch() statement. A .catch() is really just a .then() without a slot for a callback function for the case when the promise is fulfilled.

JSCopy to Clipboard

myPromise

.then(handleFulfilledA)

.then(handleFulfilledB)

.then(handleFulfilledC)

.catch(handleRejectedAny);

Using [arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) for the callback functions, implementation of the promise chain might look something like this:

JSCopy to Clipboard

myPromise

.then((value) => `${value} and bar`)

.then((value) => `${value} and bar again`)

.then((value) => `${value} and again`)

.then((value) => `${value} and again`)

.then((value) => {

console.log(value);

})

.catch((err) => {

console.error(err);

});

**Note:** For faster execution, all synchronous actions should preferably be done within one handler, otherwise it would take several ticks to execute all handlers in sequence.

The termination condition of a promise determines the "settled" state of the next promise in the chain. A "fulfilled" state indicates a successful completion of the promise, while a "rejected" state indicates a lack of success. The return value of each fulfilled promise in the chain is passed along to the next .then(), while the reason for rejection is passed along to the next rejection-handler function in the chain.

The promises of a chain are nested in one another, but get popped like the top of a stack. The first promise in the chain is most deeply nested and is the first to pop.

(promise D, (promise C, (promise B, (promise A) ) ) )

When a nextValue is a promise, the effect is a dynamic replacement. The return causes a promise to be popped, but the nextValue promise is pushed into its place. For the nesting shown above, suppose the .then() associated with "promise B" returns a nextValue of "promise X". The resulting nesting would look like this:

(promise D, (promise C, (promise X) ) )

A promise can participate in more than one nesting. For the following code, the transition of promiseA into a "settled" state will cause both instances of .then() to be invoked.

JSCopy to Clipboard

const promiseA = new Promise(myExecutorFunc);

const promiseB = promiseA.then(handleFulfilled1, handleRejected1);

const promiseC = promiseA.then(handleFulfilled2, handleRejected2);

An action can be assigned to an already "settled" promise. In that case, the action (if appropriate) will be performed at the first asynchronous opportunity. Note that promises are guaranteed to be asynchronous. Therefore, an action for an already "settled" promise will occur only after the stack has cleared and a clock-tick has passed. The effect is much like that of setTimeout(action, 0).

JSCopy to Clipboard

const promiseA = new Promise((resolve, reject) => {

resolve(777);

});

// At this point, "promiseA" is already settled.

promiseA.then((val) => console.log("asynchronous logging has val:", val));

console.log("immediate logging");

// produces output in this order:

// immediate logging

// asynchronous logging has val: 777

[**Thenables**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#thenables)

The JavaScript ecosystem had made multiple Promise implementations long before it became part of the language. Despite being represented differently internally, at the minimum, all Promise-like objects implement the *Thenable* interface. A thenable implements the [.then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) method, which is called with two callbacks: one for when the promise is fulfilled, one for when it's rejected. Promises are thenables as well.

To interoperate with the existing Promise implementations, the language allows using thenables in place of promises. For example, [Promise.resolve](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve) will not only resolve promises, but also trace thenables.

JSCopy to Clipboard

const aThenable = {

then(onFulfilled, onRejected) {

onFulfilled({

// The thenable is fulfilled with another thenable

then(onFulfilled, onRejected) {

onFulfilled(42);

},

});

},

};

Promise.resolve(aThenable); // A promise fulfilled with 42

[**Promise concurrency**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#promise_concurrency)

The Promise class offers four static methods to facilitate async task [concurrency](https://en.wikipedia.org/wiki/Concurrent_computing):

[Promise.all()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all)

Fulfills when **all** of the promises fulfill; rejects when **any** of the promises rejects.

[Promise.allSettled()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled)

Fulfills when **all** promises settle.

[Promise.any()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/any)

Fulfills when **any** of the promises fulfills; rejects when **all** of the promises reject.

[Promise.race()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race)

Settles when **any** of the promises settles. In other words, fulfills when any of the promises fulfills; rejects when any of the promises rejects.

All these methods take an [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterable_protocol) of promises ([thenables](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#thenables), to be exact) and return a new promise. They all support subclassing, which means they can be called on subclasses of Promise, and the result will be a promise of the subclass type. To do so, the subclass's constructor must implement the same signature as the [Promise()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/Promise) constructor — accepting a single executor function that can be called with the resolve and reject callbacks as parameters. The subclass must also have a resolve static method that can be called like [Promise.resolve()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve) to resolve values to promises.

Note that JavaScript is [single-threaded](https://developer.mozilla.org/en-US/docs/Glossary/Thread) by nature, so at a given instant, only one task will be executing, although control can shift between different promises, making execution of the promises appear concurrent. [Parallel execution](https://en.wikipedia.org/wiki/Parallel_computing) in JavaScript can only be achieved through [worker threads](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API).

[**Constructor**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#constructor)

[Promise()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/Promise)

Creates a new Promise object. The constructor is primarily used to wrap functions that do not already support promises.

[**Static properties**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#static_properties)

[Promise[@@species]](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/@@species)

Returns the constructor used to construct return values from promise methods.

[**Static methods**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#static_methods)

[Promise.all()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all)

Takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when all of the input's promises fulfill (including when an empty iterable is passed), with an array of the fulfillment values. It rejects when any of the input's promises reject, with this first rejection reason.

[Promise.allSettled()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled)

Takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when all of the input's promises settle (including when an empty iterable is passed), with an array of objects that describe the outcome of each promise.

[Promise.any()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/any)

Takes an iterable of promises as input and returns a single Promise. This returned promise fulfills when any of the input's promises fulfill, with this first fulfillment value. It rejects when all of the input's promises reject (including when an empty iterable is passed), with an [AggregateError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/AggregateError) containing an array of rejection reasons.

[Promise.race()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race)

Takes an iterable of promises as input and returns a single Promise. This returned promise settles with the eventual state of the first promise that settles.

[Promise.reject()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject)

Returns a new Promise object that is rejected with the given reason.

[Promise.resolve()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve)

Returns a Promise object that is resolved with the given value. If the value is a thenable (i.e. has a then method), the returned promise will "follow" that thenable, adopting its eventual state; otherwise, the returned promise will be fulfilled with the value.

Generally, if you don't know if a value is a promise or not, [Promise.resolve(value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve) it instead and work with the return value as a promise.

[**Instance properties**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#instance_properties)

These properties are defined on Promise.prototype and shared by all Promise instances.

[Promise.prototype.constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/constructor)

The constructor function that created the instance object. For Promise instances, the initial value is the [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/Promise) constructor.

Promise.prototype[@@toStringTag]

The initial value of the [@@toStringTag](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/toStringTag) property is the string "Promise". This property is used in [Object.prototype.toString()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/toString).

[**Instance methods**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#instance_methods)

[Promise.prototype.catch()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch)

Appends a rejection handler callback to the promise, and returns a new promise resolving to the return value of the callback if it is called, or to its original fulfillment value if the promise is instead fulfilled.

[Promise.prototype.finally()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/finally)

Appends a handler to the promise, and returns a new promise that is resolved when the original promise is resolved. The handler is called when the promise is settled, whether fulfilled or rejected.

[Promise.prototype.then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then)

Appends fulfillment and rejection handlers to the promise, and returns a new promise resolving to the return value of the called handler, or to its original settled value if the promise was not handled (i.e. if the relevant handler onFulfilled or onRejected is not a function).

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#examples)

[**Basic Example**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#basic_example)

JSCopy to Clipboard

const myFirstPromise = new Promise((resolve, reject) => {

// We call resolve(...) when what we were doing asynchronously was successful, and reject(...) when it failed.

// In this example, we use setTimeout(...) to simulate async code.

// In reality, you will probably be using something like XHR or an HTML API.

setTimeout(() => {

resolve("Success!"); // Yay! Everything went well!

}, 250);

});

myFirstPromise.then((successMessage) => {

// successMessage is whatever we passed in the resolve(...) function above.

// It doesn't have to be a string, but if it is only a succeed message, it probably will be.

console.log(`Yay! ${successMessage}`);

});

[**Example with diverse situations**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#example_with_diverse_situations)

This example shows diverse techniques for using Promise capabilities and diverse situations that can occur. To understand this, start by scrolling to the bottom of the code block, and examine the promise chain. Upon provision of an initial promise, a chain of promises can follow. The chain is composed of .then() calls, and typically (but not necessarily) has a single .catch() at the end, optionally followed by .finally(). In this example, the promise chain is initiated by a custom-written new Promise() construct; but in actual practice, promise chains more typically start with an API function (written by someone else) that returns a promise.

The example function tetheredGetNumber() shows that a promise generator will utilize reject() while setting up an asynchronous call, or within the call-back, or both. The function promiseGetWord() illustrates how an API function might generate and return a promise in a self-contained manner.

Note that the function troubleWithGetNumber() ends with a throw. That is forced because a promise chain goes through all the .then() promises, even after an error, and without the throw, the error would seem "fixed". This is a hassle, and for this reason, it is common to omit onRejected throughout the chain of .then() promises, and just have a single onRejected in the final catch().

This code can be run under NodeJS. Comprehension is enhanced by seeing the errors actually occur. To force more errors, change the threshold values.

JSCopy to Clipboard

// To experiment with error handling, "threshold" values cause errors randomly

const THRESHOLD\_A = 8; // can use zero 0 to guarantee error

function tetheredGetNumber(resolve, reject) {

setTimeout(() => {

const randomInt = Date.now();

const value = randomInt % 10;

if (value < THRESHOLD\_A) {

resolve(value);

} else {

reject(`Too large: ${value}`);

}

}, 500);

}

function determineParity(value) {

const isOdd = value % 2 === 1;

return { value, isOdd };

}

function troubleWithGetNumber(reason) {

const err = new Error("Trouble getting number", { cause: reason });

console.error(err);

throw err;

}

function promiseGetWord(parityInfo) {

return new Promise((resolve, reject) => {

const { value, isOdd } = parityInfo;

if (value >= THRESHOLD\_A - 1) {

reject(`Still too large: ${value}`);

} else {

parityInfo.wordEvenOdd = isOdd ? "odd" : "even";

resolve(parityInfo);

}

});

}

new Promise(tetheredGetNumber)

.then(determineParity, troubleWithGetNumber)

.then(promiseGetWord)

.then((info) => {

console.log(`Got: ${info.value}, ${info.wordEvenOdd}`);

return info;

})

.catch((reason) => {

if (reason.cause) {

console.error("Had previously handled error");

} else {

console.error(`Trouble with promiseGetWord(): ${reason}`);

}

})

.finally((info) => console.log("All done"));

[**Advanced Example**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#advanced_example)

This small example shows the mechanism of a Promise. The testPromise() method is called each time the [<button>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button) is clicked. It creates a promise that will be fulfilled, using [setTimeout()](https://developer.mozilla.org/en-US/docs/Web/API/setTimeout), to the promise count (number starting from 1) every 1-3 seconds, at random. The Promise() constructor is used to create the promise.

The fulfillment of the promise is logged, via a fulfill callback set using [p1.then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then). A few logs show how the synchronous part of the method is decoupled from the asynchronous completion of the promise.

By clicking the button several times in a short amount of time, you'll even see the different promises being fulfilled one after another.

**HTML**

HTMLPlayCopy to Clipboard

<button id="make-promise">Make a promise!</button>

<div id="log"></div>

**JavaScript**

JSPlayCopy to Clipboard

"use strict";

let promiseCount = 0;

function testPromise() {

const thisPromiseCount = ++promiseCount;

const log = document.getElementById("log");

// begin

log.insertAdjacentHTML("beforeend", `${thisPromiseCount}) Started<br>`);

// We make a new promise: we promise a numeric count of this promise,

// starting from 1 (after waiting 3s)

const p1 = new Promise((resolve, reject) => {

// The executor function is called with the ability

// to resolve or reject the promise

log.insertAdjacentHTML(

"beforeend",

`${thisPromiseCount}) Promise constructor<br>`,

);

// This is only an example to create asynchronism

setTimeout(

() => {

// We fulfill the promise

resolve(thisPromiseCount);

},

Math.random() \* 2000 + 1000,

);

});

// We define what to do when the promise is resolved with the then() call,

// and what to do when the promise is rejected with the catch() call

p1.then((val) => {

// Log the fulfillment value

log.insertAdjacentHTML("beforeend", `${val}) Promise fulfilled<br>`);

}).catch((reason) => {

// Log the rejection reason

console.log(`Handle rejected promise (${reason}) here.`);

});

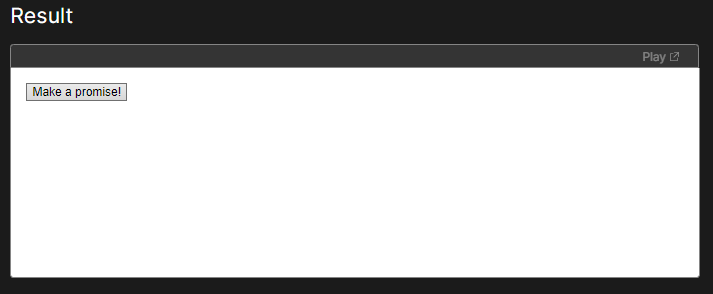
// end

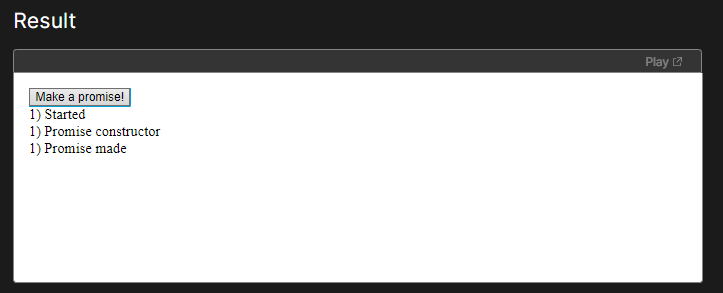
log.insertAdjacentHTML("beforeend", `${thisPromiseCount}) Promise made<br>`);

}

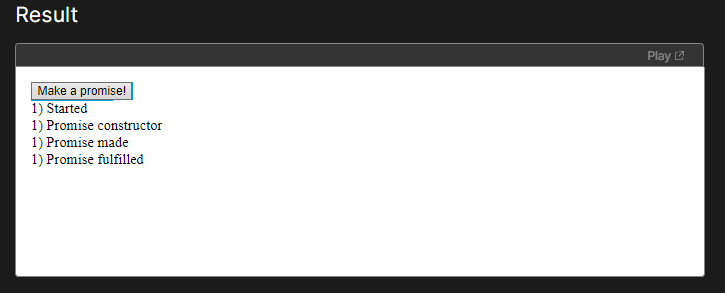
const btn = document.getElementById("make-promise");

btn.addEventListener("click", testPromise);





After sometime



[**Loading an image with XHR**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#loading_an_image_with_xhr)

Another simple example using Promise and [XMLHttpRequest](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest) to load an image is available at the MDN GitHub [js-examples](https://github.com/mdn/js-examples/tree/master/promises-test) repository. You can also [see it in action](https://mdn.github.io/js-examples/promises-test/). Each step is commented on and allows you to follow the Promise and XHR architecture closely.

[**Incumbent settings object tracking**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#incumbent_settings_object_tracking)

A settings object is an [environment](https://html.spec.whatwg.org/multipage/webappapis.html#environment-settings-object) that provides additional information when JavaScript code is running. This includes the realm and module map, as well as HTML specific information such as the origin. The incumbent settings object is tracked in order to ensure that the browser knows which one to use for a given piece of user code.

To better picture this, we can take a closer look at how the realm might be an issue. A **realm** can be roughly thought of as the global object. What is unique about realms is that they hold all of the necessary information to run JavaScript code. This includes objects like [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) and [Error](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error). Each settings object has its own "copy" of these and they are not shared. That can cause some unexpected behavior in relation to promises. In order to get around this, we track something called the **incumbent settings object**. This represents information specific to the context of the user code responsible for a certain function call.

To illustrate this a bit further we can take a look at how an [<iframe>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe) embedded in a document communicates with its host. Since all web APIs are aware of the incumbent settings object, the following will work in all browsers:

HTMLCopy to Clipboard

<!doctype html> <iframe></iframe>

<!-- we have a realm here -->

<script>

// we have a realm here as well

const bound = frames[0].postMessage.bind(frames[0], "some data", "\*");

// bound is a built-in function — there is no user

// code on the stack, so which realm do we use?

setTimeout(bound);

// this still works, because we use the youngest

// realm (the incumbent) on the stack

</script>

The same concept applies to promises. If we modify the above example a little bit, we get this:

HTMLCopy to Clipboard

<!doctype html> <iframe></iframe>

<!-- we have a realm here -->

<script>

// we have a realm here as well

const bound = frames[0].postMessage.bind(frames[0], "some data", "\*");

// bound is a built in function — there is no user

// code on the stack — which realm do we use?

Promise.resolve(undefined).then(bound);

// this still works, because we use the youngest

// realm (the incumbent) on the stack

</script>

If we change this so that the <iframe> in the document is listening to post messages, we can observe the effect of the incumbent settings object:

HTMLCopy to Clipboard

<!-- y.html -->

<!doctype html>

<iframe src="x.html"></iframe>

<script>

const bound = frames[0].postMessage.bind(frames[0], "some data", "\*");

Promise.resolve(undefined).then(bound);

</script>

HTMLCopy to Clipboard

<!-- x.html -->

<!doctype html>

<script>

window.addEventListener(

"message",

(event) => {

document.querySelector("#text").textContent = "hello";

// this code will only run in browsers that track the incumbent settings object

console.log(event);

},

false,

);

</script>

In the above example, the inner text of the <iframe> will be updated only if the incumbent settings object is tracked. This is because without tracking the incumbent, we may end up using the wrong environment to send the message.