Promise - JavaScript MDN

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

To learn about the way promises work and how you can use them, we advise you to read [Using promises]() first.

Description

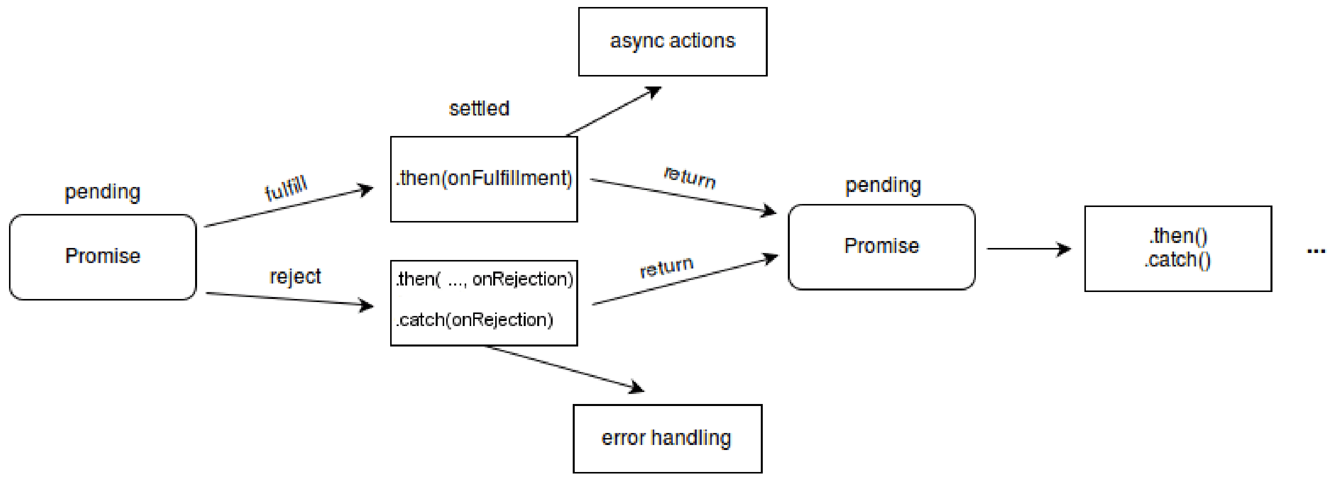
A `Promise` is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the 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.



You will also hear the term *resolved* used with promises — this means that the promise is settled or "locked-in" to match the eventual state of another promise, and further resolving or rejecting it has no effect. The States and fates document from the original Promise proposal contains more details about promise terminology. Colloquially, "resolved" promises are often equivalent to "fulfilled" promises, but as illustrated in "States and fates", resolved promises can be pending or rejected as well. For example:

```
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.

Note: Several other languages have mechanisms for lazy evaluation and deferring a computation, which they also call "promises", e.g., Scheme. Promises in JavaScript represent processes that are already happening, which can be chained with callback functions. If you are looking to lazily evaluate an expression, consider using a function with no arguments e.g., `f = () => expression` to create the lazily-evaluated expression, and `f()` to evaluate the expression immediately.

`Promise` itself has no first-class protocol for cancellation, but you may be able to directly cancel the underlying asynchronous operation, typically using `AbortController`.

Chained Promises

The promise methods `then()`, `catch()`, and `finally()` are used to associate further action with a promise that becomes settled. 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. The `catch()` and `finally()` methods call `then()` internally and make error handling less verbose. For example, a `catch()` is really just a `then()` without passing the fulfillment handler. As these methods return promises, they can be chained. For example:

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("foo");
  }, 300);
});

myPromise
  .then(handleFulfilledA, handleRejectedA)
  .then(handleFulfilledB, handleRejectedB)
  .then(handleFulfilledC, handleRejectedC);
```

We will use the following terminology: *initial promise* is the promise on which `then` is called; *new promise* is the promise returned by `then`. The two callbacks passed to `then` are called *fulfillment handler* and *rejection handler*, respectively.

The settled state of the initial promise determines which handler to execute.

- If the initial promise is fulfilled, the fulfillment handler is called with the fulfillment value.
- If the initial promise is rejected, the rejection handler is called with the rejection reason.

The completion of the handler determines the settled state of the new promise.

- If the handler returns a thenable value, the new promise settles in the same state as the returned value.
- If the handler returns a non-thenable value, the new promise is fulfilled with the returned value.
- If the handler throws an error, the new promise is rejected with the thrown error.
- If the initial promise has no corresponding handler attached, the new promise will settle to the same state as the initial promise — that is, without a rejection handler, a rejected promise stays rejected with the same reason.

For example, in the code above, if `myPromise` rejects, `handleRejectedA` will be called, and if `handleRejectedA` completes normally (without throwing or returning a rejected promise), the promise returned by the first `then` will be fulfilled instead of staying rejected. Therefore, if an error must be handled immediately, but we want to maintain the error state down the chain, we must throw an error of some type in the rejection handler. On the other hand, in the absence of an immediate need, we can leave out error handling until the final `catch()` handler.

```
myPromise
  .then(handleFulfilledA)
  .then(handleFulfilledB)
  .then(handleFulfilledC)
  .catch(handleRejectedAny);
```

Using arrow functions for the callback functions, implementation of the promise chain might look something like this:

```
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.

JavaScript maintains a job queue. Each time, JavaScript picks a job from the queue and executes it to completion. The jobs are defined by the executor of the `Promise()` constructor, the handlers passed to `then`, or any platform API that returns a promise. The promises in a chain represent the dependency relationship between these jobs. When a promise settles, the respective handlers associated with it are added to the back of the job queue.

A promise can participate in more than one chain. For the following code, the fulfillment of `promiseA` will cause both `handleFulfilled1` and `handleFulfilled2` to be added to the job queue. Because `handleFulfilled1` is registered first, it will be invoked first.

```
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 this case, the action is added immediately to the back of the job queue and will be performed when all existing jobs are completed. Therefore, an action for an already "settled" promise will occur only after the current synchronous code completes and at least one loop-tick has passed. This guarantees that promise actions are asynchronous.

```
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

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()` 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` will not only resolve promises, but also trace thenables.

```
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

The `Promise` class offers four static methods to facilitate async task concurrency:

`Promise.all()`

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

`Promise.allSettled()`

Fulfills when **all** promises settle.

`Promise.any()`

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

`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 of promises (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()` 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()` to resolve values to promises.

Note that JavaScript is single-threaded 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 in JavaScript can only be achieved through worker threads.

Constructor

`Promise()`

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

Static properties

`Promise[Symbol.species]`

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

Static methods

`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()`

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()`

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` containing an array of rejection reasons.

`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()`

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

`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.

`Promise.try()`

Takes a callback of any kind (returns or throws, synchronously or asynchronously) and wraps its result in a `Promise`.

`Promise.withResolvers()`

Returns an object containing a new `Promise` object and two functions to resolve or reject it, corresponding to the two parameters passed to the executor of the `Promise()` constructor.

Instance properties

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

`Promise.prototype.constructor`

The constructor function that created the instance object. For `Promise` instances, the initial value is the `Promise` constructor.

`Promise.prototype[Symbol.toStringTag]`

The initial value of the `[Symbol.toStringTag]` property is the string `"Promise"`. This property is used in `Object.prototype.toString()`.

Instance methods

`Promise.prototype.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()`

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()`

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

Basic Example

In this example, we use `setTimeout(...)` to simulate async code. In reality, you will probably be using something like XHR or an HTML API.

```
const myFirstPromise = new Promise((resolve, reject) => {
  // We call resolve(...) when what we were doing asynchronously
  // was successful, and reject(...) when it failed.
  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

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.

```
// 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"));
```

这个then没有传reject handler，所以若.then(promiseGetWord)返回了一个reject的Promise，则会被下面的catch接收并处理。就像上文说的：“ a catch() is really just a then() without passing the fulfillment handler. ” catch()只是一个只有reject handler的then()

Advanced Example

This small example shows the mechanism of a `Promise`. The `testPromise()` method is called each time the `<button>` is clicked. It creates a promise that will be fulfilled, using `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()`. 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

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

JavaScript

```
"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);
```

Result

Another example using `Promise` and `XMLHttpRequest` to load an image is shown below. Each step is commented on and allows you to follow the Promise and XHR architecture closely.

```
function imgLoad(url) {
  // Create new promise with the Promise() constructor;
  // This has as its argument a function with two parameters, resolve and
reject
  return new Promise((resolve, reject) => {
    // XHR to load an image
    const request = new XMLHttpRequest();
    request.open("GET", url);
    request.responseType = "blob";
    // When the request loads, check whether it was successful
    request.onload = () => {
      if (request.status === 200) {
        // If successful, resolve the promise by passing back the request
response
        resolve(request.response);
      } else {
        // If it fails, reject the promise with an error message
        reject(
          Error(
            \`Image didn't load successfully; error code: +
${request.statusText}\`,
          ),
        );
      }
    };
    // Handle network errors
    request.onerror = () => reject(new Error("There was a network error."));
    // Send the request
    request.send();
  });
}

// Get a reference to the body element, and create a new image object
const body = document.querySelector("body");
const myImage = new Image();
const imgUrl =
  "https://mdn.github.io/shared-assets/images/examples/round-balloon.png";

// Call the function with the URL we want to load, then chain the
// promise then() method with two callbacks
imgLoad(imgUrl).then(
  (response) => {
    // The first runs when the promise resolves, with the request.response
    // specified within the resolve() method.
    const imageURL = URL.createObjectURL(response);
    myImage.src = imageURL;
    body.appendChild(myImage);
  },
  (error) => {
    // The second runs when the promise
    // is rejected, and logs the Error specified with the reject() method.
    console.log(error);
  },
);
```

Incumbent settings object tracking

A settings object is an environment 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` and `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>` 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:

```
<!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:

```
<!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:

```
<!-- 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>
```

```
<!-- 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>
```

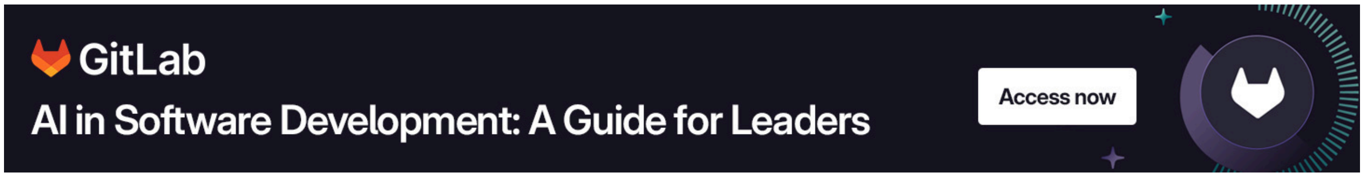
In the above example, the inner text of the `<iframe>` will be updated only if the incumbent settings object is tracked. This is because without tracking the incumbent, we may end up using the wrong environment to send the message.

Note: Currently, incumbent realm tracking is fully implemented in Firefox, and has partial implementations in Chrome and Safari.

Specifications

Specification
ECMAScript® 2026 Language Specification # sec-promise-objects

Browser compatibility



Ad