Eric Elliott Follow
Make some magic. #JavaScript
Jan 23, 2017 · 11 min read

Master the JavaScript Interview: What is a Promise?

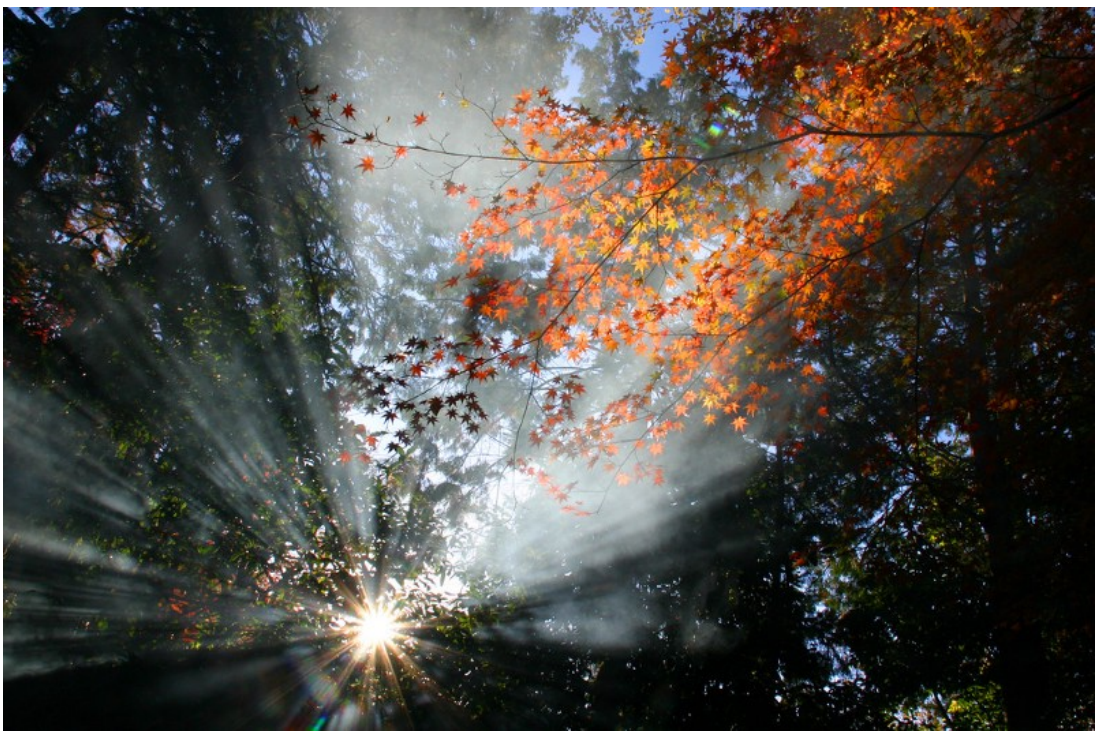


Photo by Kabun (CC BY NC SA 2.0)

> *"Master the JavaScript Interview" is a series of posts designed to prepare candidates for common questions they are likely to encounter when applying for a mid to senior-level JavaScript position. These are questions I frequently use in real interviews.*

What is a Promise?

A promise is an object that may produce a single value some time in the future: either a resolved value, or a reason that it's not resolved (e.g., a network error occurred). A promise may be in one of 3 possible states:

fulfilled, rejected, or pending. Promise users can attach callbacks to handle the fulfilled value or the reason for rejection.

Promises are eager, meaning that a promise will start doing whatever task you give it as soon as the promise constructor is invoked. If you need lazy, check out observables or tasks.

An Incomplete History of Promises

Early implementations of promises and futures (a similar / related idea) began to appear in languages such as MultiLisp and Concurrent Prolog as early as the 1980's. The use of the word "promise" was coined by Barbara Liskov and Liuba Shrira in 1988[1].

The first time I heard about promises in JavaScript, Node was brand new and the community was discussing the best way to handle asynchronous behavior. The community experimented with promises for a while, but eventually settled on the Node-standard error-first callbacks.

Around the same time, Dojo added promises via the Deferred API. Growing interest and activity eventually led to the newly formed Promises/A specification designed to make various promises more interoperable.

jQuery's async behaviors were refactored around promises. jQuery's promise support had remarkable similarities to Dojo's Deferred, and it quickly became the most commonly used promise implementation in JavaScript due to jQuery's immense popularity—for a time. However, it did not support the two channel (fulfilled/rejected) chaining behavior & exception management that people were counting on to build tools on top of promises.

In spite of those weaknesses, jQuery officially made JavaScript promises mainstream, and better stand-alone promise libraries like Q, When, and Bluebird became very popular. jQuery's implementation

incompatibilities motivated some important clarifications in the promise spec, which was rewritten and rebranded as the Promises/A+ specification.

ES6 brought a Promises/A+ compliant `Promise` global, and some very important APIs were built on top of the new standard Promise support: notably the WHATWG Fetch spec and the Async Functions standard (a stage 3 draft at the time of this writing).

The promises described here are those which are compatible with the Promises/A+ specification, with a focus on the ECMAScript standard `Promise` implementation.

How Promises Work

A promise is an object which can be returned synchronously from an asynchronous function. It will be in one of 3 possible states:

- **Fulfilled:** `onFulfilled()` will be called (e.g., `resolve()` was called)
- **Rejected:** `onRejected()` will be called (e.g., `reject()` was called)
- **Pending:** not yet fulfilled or rejected

A promise is **settled** if it's not pending (it has been resolved or rejected). Sometimes people use *resolved* and *settled* to mean the same thing: *not pending*.

Once settled, a promise can not be resettled. Calling `resolve()` or `reject()` again will have no effect. The immutability of a settled promise is an important feature.

Native JavaScript promises don't expose promise states. Instead, you're expected to treat the promise as a black box. Only the function

responsible for creating the promise will have knowledge of the promise status, or access to resolve or reject.

Here is a function that returns a promise which will resolve after a specified time delay:

```
1  const wait = time => new Promise((resolve) => setTimeout(res
2
3  wait(3000).then(() => console.log('Hello!')); // 'Hello!'
```

wait — promise example on CodePen

Our `wait(3000)` call will wait 3000ms (3 seconds), and then log `'Hello!'`. All spec-compatible promises define a `.then()` method which you use to pass handlers which can take the resolved or rejected value.

The ES6 promise constructor takes a function. That function takes two parameters, `resolve()`, and `reject()`. In the example above, we're only using `resolve()`, so I left `reject()` off the parameter list. Then we call `setTimeout()` to create the delay, and call `resolve()` when it's finished.

You can optionally `resolve()` or `reject()` with values, which will be passed to the callback functions attached with `.then()`.

When I `reject()` with a value, I always pass an `Error` object. Generally I want two possible resolution states: the normal happy path, or an exception—anything that stops the normal happy path from happening. Passing an `Error` object makes that explicit.

Important Promise Rules

A standard for promises was defined by the Promises/A+ specification community. There are many implementations which conform to the

standard, including the JavaScript standard ECMAScript promises.

Promises following the spec must follow a specific set of rules:

- A promise or "thenable" is an object that supplies a standard-compliant `.then()` method.
- A pending promise may transition into a fulfilled or rejected state.
- A fulfilled or rejected promise is settled, and must not transition into any other state.
- Once a promise is settled, it must have a value (which may be `undefined`). That value must not change.

Change in this context refers to identity (`===`) comparison. An object may be used as the fulfilled value, and object properties may mutate.

Every promise must supply a `.then()` method with the following signature:

```
promise.then(
  onFulfilled?: Function,
  onRejected?: Function
) => Promise
```

The `.then()` method must comply with these rules:

- Both `onFulfilled()` and `onRejected()` are optional.
- If the arguments supplied are not functions, they must be ignored.
- `onFulfilled()` will be called after the promise is fulfilled, with the promise's value as the first argument.

- `onRejected()` will be called after the promise is rejected, with the reason for rejection as the first argument. The reason may be any valid JavaScript value, but because rejections are essentially synonymous with exceptions, I recommend using Error objects.
- Neither `onFulfilled()` nor `onRejected()` may be called more than once.
- `.then()` may be called many times on the same promise. In other words, a promise can be used to aggregate callbacks.
- `.then()` must return a new promise, `promise2`.
- If `onFulfilled()` or `onRejected()` return a value `x`, and `x` is a promise, `promise2` will lock in with (assume the same state and value as) `x`. Otherwise, `promise2` will be fulfilled with the value of `x`.
- If either `onFulfilled` or `onRejected` throws an exception `e`, `promise2` must be rejected with `e` as the reason.
- If `onFulfilled` is not a function and `promise1` is fulfilled, `promise2` must be fulfilled with the same value as `promise1`.
- If `onRejected` is not a function and `promise1` is rejected, `promise2` must be rejected with the same reason as `promise1`.

Promise Chaining

Because `.then()` always returns a new promise, it's possible to chain promises with precise control over how and where errors are handled. Promises allow you to mimic normal synchronous code's `try` / `catch` behavior.

Like synchronous code, chaining will result in a sequence that runs in serial. In other words, you can do:

```
fetch(url)
  .then(process)
  .then(save)
  .catch(handleErrors)
;
```

Assuming each of the functions, `fetch()`, `process()`, and `save()` return promises, `process()` will wait for `fetch()` to complete before starting, and `save()` will wait for `process()` to complete before starting. `handleErrors()` will only run if any of the previous promises reject.

Here's an example of a complex promise chain with multiple rejections:

```
const wait = time => new Promise(
  res => setTimeout(() => res(), time)
);

wait(200)
  // onFulfilled() can return a new promise, `x`
  .then(() => new Promise(res => res('foo')))
  // the next promise will assume the state of `x`
  .then(a => a)
  // Above we returned the unwrapped value of `x`
  // so `.then()` above returns a fulfilled promise
  // with that value:
  .then(b => console.log(b)) // 'foo'
  // Note that `null` is a valid promise value:
  .then(() => null)
  .then(c => console.log(c)) // null
  // The following error is not reported yet:
  .then(() => {throw new Error('foo');})
  // Instead, the returned promise is rejected
  // with the error as the reason:
  .then(
    // Nothing is logged here due to the error above:
    d => console.log(`d: ${ d }`),
    // Now we handle the error (rejection reason)
```

Promise chaining behavior example on CodePen

Error Handling

Note that promises have both a success and an error handler, and it's very common to see code that does this:

```
save().then(
  handleSuccess,
```

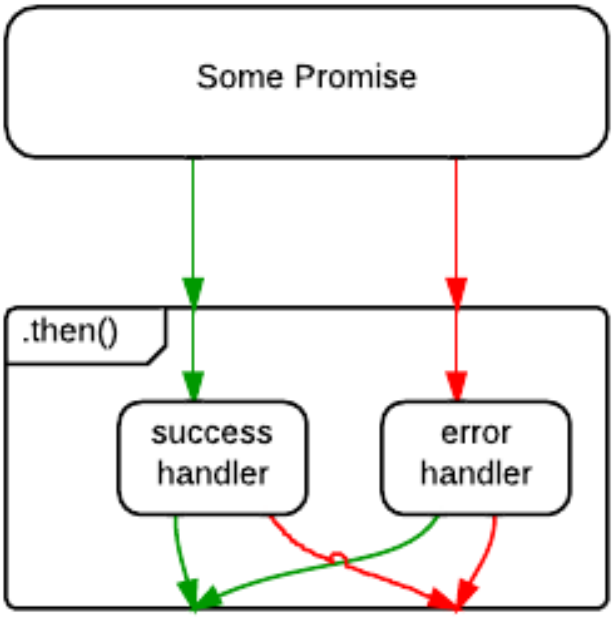
```
    handleError
  );
```

But what happens if `handleSuccess()` throws an error? The promise returned from `.then()` will be rejected, but there's nothing there to catch the rejection—meaning that an error in your app gets swallowed. Oops!

For that reason, some people consider the code above to be an anti-pattern, and recommend the following, instead:

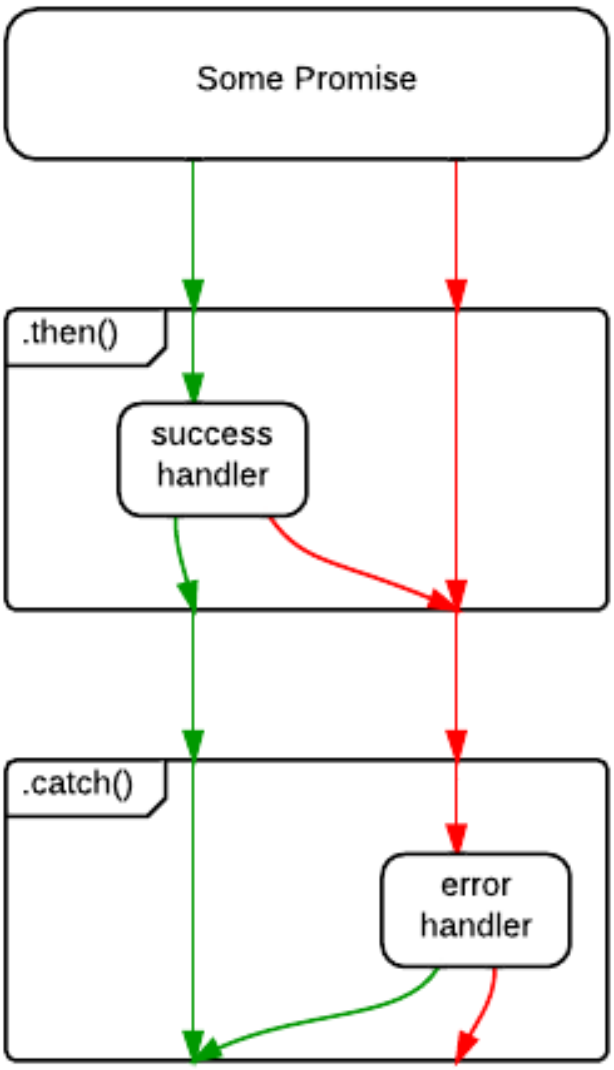
```
save()
  .then(handleSuccess)
  .catch(handleError)
;
```

The difference is subtle, but important. In the first example, an error originating in the `save()` operation will be caught, but an error originating in the `handleSuccess()` function will be swallowed.



Without .catch(), an error in the success handler is uncaught.

In the second example, `.catch()` will handle rejections from either `save()`, or `handleSuccess()`.



With .catch(), both error sources are handled. (diagram source)

Of course, the `save()` error might be a networking error, whereas the `handleSuccess()` error may be because the developer forgot to handle a specific status code. What if you want to handle them differently? You could opt to handle them both:

```
save()
  .then(
    handleSuccess,
    handleNetworkError
  )
```

```
    .catch(handleProgrammerError)
;
```

Whatever you prefer, I recommend ending all promise chains with a `.catch()`. That's worth repeating:

> *I recommend ending all promise chains with a .catch().*

How Do I Cancel a Promise?

One of the first things new promise users often wonder about is how to cancel a promise. Here's an idea: Just reject the promise with "Cancelled" as the reason. If you need to deal with it differently than a "normal" error, do your branching in your error handler.

Here are some common mistakes people make when they roll their own promise cancellation:

Adding .cancel() to the promise

Adding `.cancel()` makes the promise non-standard, but it also violates another rule of promises: Only the function that creates the promise should be able to resolve, reject, or cancel the promise. Exposing it breaks that encapsulation, and encourages people to write code that manipulates the promise in places that shouldn't know about it. Avoid spaghetti and broken promises.

Forgetting to clean up

Some clever people have figured out that there's a way to use `Promise.race()` as a cancellation mechanism. The problem with that is that cancellation control is taken from the function that creates the promise, which is the only place that you can conduct proper cleanup activities, such as clearing timeouts or freeing up memory by clearing references to data, etc...

Forgetting to handle a rejected cancel promise

Did you know that Chrome throws warning messages all over the console when you forget to handle a promise rejection? Oops!

Overly complex

The withdrawn TC39 proposal for cancellation proposed a separate messaging channel for cancellations. It also used a new concept called a cancellation token. In my opinion, the solution would have considerably bloated the promise spec, and the only feature it would have provided that speculations don't directly support is the separation of rejections and cancellations, which, IMO, is not necessary to begin with.

Will you want to do switching depending on whether there is an exception, or a cancellation? Yes, absolutely. Is that the promise's job? In my opinion, no, it's not.

Rethinking Promise Cancellation

Generally, I pass all the information the promise needs to determine how to resolve / reject / cancel at promise creation time. That way, there's no need for a `.cancel()` method on a promise. You might be wondering how you could possibly know whether or not you're going to cancel at promise creation time.

> *"If I don't yet know whether or not to cancel, how will I know what to pass in when I create the promise?"*

If only there were some kind of object that could stand in for a potential value in the future… *oh, wait.*

The value we pass in to represent whether or not to cancel could be a promise itself. Here's how that might look:

```
const wait = (
  time,
  cancel = Promise.reject()
) => new Promise((resolve, reject) => {
  const timer = setTimeout(resolve, time);
  const noop = () => {};

  cancel.then(() => {
    clearTimeout(timer);
    reject(new Error('Cancelled'));
  }, noop);
});

const shouldCancel = Promise.resolve(); // Yes, cancel
```

Cancellable wait—try it on CodePen

We're using default parameter assignment to tell it not to cancel by default. That makes the `cancel` parameter conveniently optional. Then we set the timeout as we did before, but this time we capture the timeout's ID so that we can clear it later.

We use the `cancel.then()` method to handle the cancellation and resource cleanup. This will only run if the promise gets cancelled before it has a chance to resolve. If you cancel too late, you've missed your chance. That train has left the station.

> *Note: You may be wondering what the* `noop()` *function is for. The word noop stands for no-op, meaning a function that does nothing. Without it, V8 will throw warnings:* `UnhandledPromiseRejectionWarning: Unhandled promise rejection`*. It's a good idea to* ***always handle promise rejections,*** *even if your handler is a* `noop()`*.*

Abstracting Promise Cancellation

This is fine for a `wait()` timer, but we can abstract this idea further to encapsulate everything you have to remember:

1. Reject the cancel promise by default—we don't want to cancel or throw errors if no cancel promise gets passed in.
2. Remember to perform cleanup when you reject for cancellations.
3. Remember that the `onCancel` cleanup might itself throw an error, and that error will need handling, too. (Note that error handling is omitted in the wait example above—it's easy to forget!)

Let's create a cancellable promise utility that you can use to wrap any promise. For example, to handle network requests, etc… The signature will look like this:

```
speculation(fn: SpecFunction, shouldCancel: Promise) =>
Promise
```

The SpecFunction is just like the function you would pass into the `Promise` constructor, with one exception—it takes an `onCancel()` handler:

```
SpecFunction(resolve: Function, reject: Function, onCancel:
Function) => Void
```

```
// HOF Wraps the native Promise API
// to add take a shouldCancel promise and add
// an onCancel() callback.
const speculation = (
  fn,
  cancel = Promise.reject() // Don't cancel by default
) => new Promise((resolve, reject) => {
  const noop = () => {};

  const onCancel = (
    handleCancel
  ) => cancel.then(
      handleCancel,
      // Ignore expected cancel rejections:
      noop
```

Note that this example is just an illustration to give you the gist of how it works. There are some other edge cases you need to take into consideration. For example, in this version, `handleCancel` will be called if you cancel the promise after it is already settled.

I've implemented a maintained production version of this with edge cases covered as the open source library, Speculation.

Let's use the improved library abstraction to rewrite the cancellable `wait()` utility from before. First install speculation:

```
npm install --save speculation
```

Now you can import and use it:

```
import speculation from 'speculation';

const wait = (
  time,
  cancel = Promise.reject() // By default, don't cancel
) => speculation((resolve, reject, onCancel) => {
  const timer = setTimeout(resolve, time);

  // Use onCancel to clean up any lingering resources
  // and then call reject(). You can pass a custom reason.
  onCancel(() => {
    clearTimeout(timer);
    reject(new Error('Cancelled'));
  });
}, cancel); // remember to pass in cancel!

wait(200  wait(500)) then(
```

This simplifies things a little, because you don't have to worry about the `noop()`, catching errors in your `onCancel()`, function or other edge cases. Those details have been abstracted away by `speculation()`. Check it out and feel free to use it in real projects.

Extras of the Native JS Promise

The native `Promise` object has some extra stuff you might be interested in:

- `Promise.reject()` returns a rejected promise.
- `Promise.resolve()` returns a resolved promise.
- `Promise.race()` takes an array (or any iterable) and returns a promise that resolves with the value of the first resolved promise in the iterable, or rejects with the reason of the first promise that rejects.

- `Promise.all()` takes an array (or any iterable) and returns a promise that resolves when *all of the promises* in the iterable argument have resolved, or rejects with the reason of the first passed promise that rejects.

Conclusion

Promises have become an integral part of several idioms in JavaScript, including the WHATWG Fetch standard used for most modern ajax requests, and the Async Functions standard used to make asynchronous code look synchronous.

Async functions are stage 3 at the time of this writing, but I predict that they will soon become a very popular, very commonly used solution for asynchronous programming in JavaScript—which means that learning to appreciate promises is going to be even more important to JavaScript developers in the near future.

For instance, if you're using Redux, I suggest that you check out redux-saga: A library used to manage side-effects in Redux which depends on async functions throughout the documentation.

I hope even experienced promise users have a better understanding of what promises are and how they work, and how to use them better after reading this.

Explore the Series

- What is a Closure?
- What is the Difference Between Class and Prototypal Inheritance?
- What is a Pure Function?
- What is Function Composition?
- What is Functional Programming?

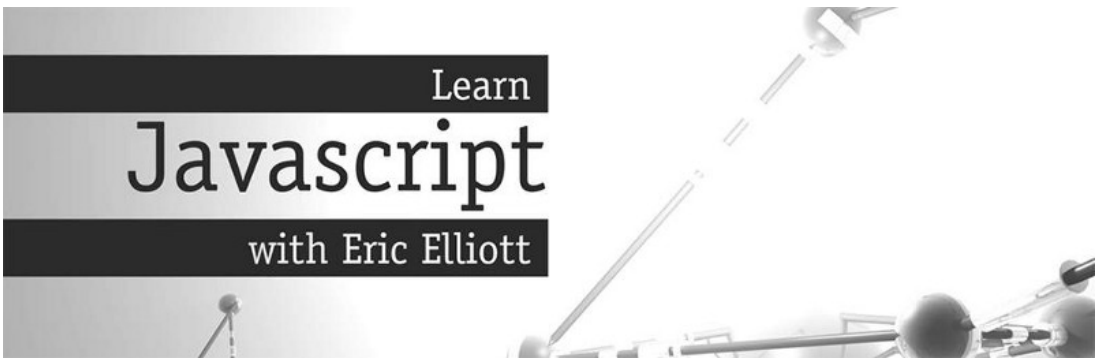
- What is a Promise?
- Soft Skills

. . .

1. *Barbara Liskov; Liuba Shrira (1988). "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems". Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation; Atlanta, Georgia, United States, pp. 260–267. ISBN 0–89791–269–1, published by ACM. Also published in ACM SIGPLAN Notices, Volume 23, Issue 7, July 1988.*

Level Up Your Skills

Learn JavaScript with Eric Elliott. If you're not a member, you're missing out!



. . .

Eric Elliott *is the author of "Programming JavaScript Applications" (O'Reilly), and advanced JavaScript and dev leadership curriculum. He has contributed to software experiences for* ***Adobe Systems****,* ***Zumba Fitness****,* ***The Wall Street Journal****,* ***ESPN****,* ***BBC****, and top recording artists including* ***Usher****,* ***Frank Ocean****,* ***Metallica****, and many more.*

He works anywhere he wants with the most beautiful woman in the world.