

# Promise

The **Promise** object is used for asynchronous computations. A Promise represents a value which may be available now, or in the future, or never.

## Syntax

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

## Parameters

### executor

A function that is passed the arguments `resolve` and `reject`. The executor function is executed immediately by the Promise implementation, passing `resolve` and `reject` functions (the executor is called before the Promise constructor even returns the created object). The `resolve` and `reject` functions, when called, resolve or reject the promise respectively. The executor normally initiates some asynchronous work and then, once that completes, calls either the `resolve` or `reject` function to resolve the promise or else reject it if an error occurred.

## Description

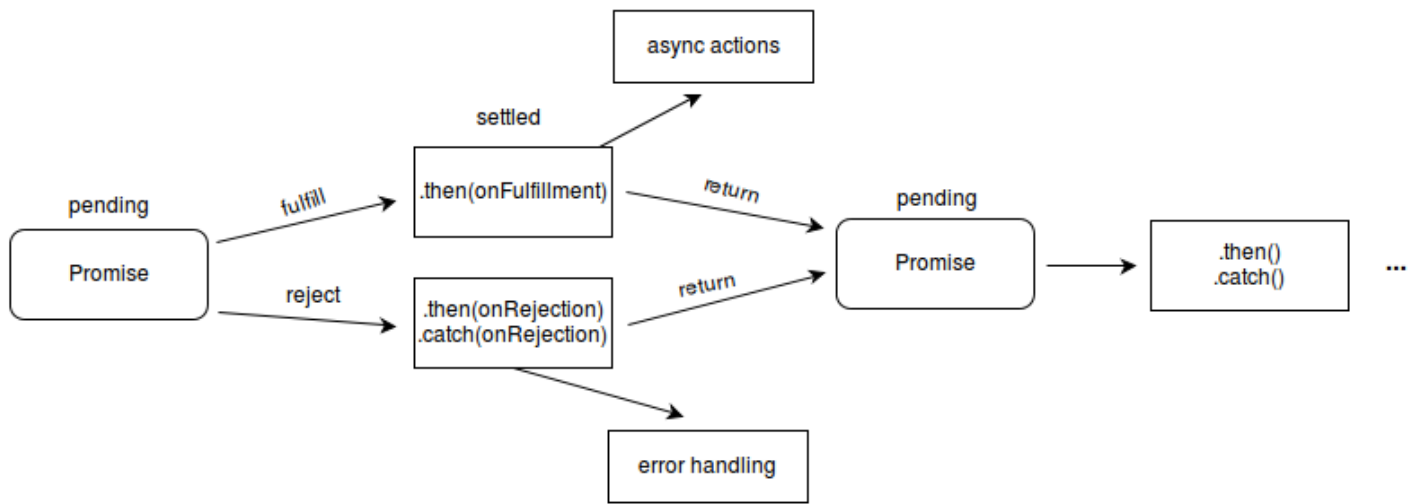
A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers to an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of the final value, the asynchronous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

A Promise is in one of these states:

- *pending*: initial state, not fulfilled or rejected.
- *fulfilled*: meaning that the operation completed successfully.
- *rejected*: meaning that the operation failed.

A pending promise can either be *fulfilled* with a value, or *rejected* with a reason (error). When either of these happens, 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.)

As the `Promise.prototype.then()` and `Promise.prototype.catch()` methods return promises, they can be chained.



📄 **Not to be confused with:** 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 which are already happening, which can be chained with callback functions. If you are looking to lazily evaluate an expression, consider the [arrow function](#) with no arguments: `f = () => expression` to create the lazily-evaluated expression, and `f()` to evaluate.

📄 **Note:** 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 it is locked into a promise chain. Domenic Denicola's [States and fates](#) contains more details about promise terminology.

## Properties

### `Promise.length`

Length property whose value is always 1 (number of constructor arguments).

### `Promise.prototype`

Represents the prototype for the Promise constructor.

## Methods

### `Promise.all(iterable)`

Returns a promise that either fulfills when all of the promises in the iterable argument have fulfilled or rejects as soon as one of the promises in the iterable argument rejects. If the returned promise fulfills, it is fulfilled with an array of the values from the fulfilled promises in same order as defined in the iterable. If the returned promise rejects, it is rejected with the reason from the first promise in the iterable that rejected. This method can be useful for aggregating results of multiple promises.

### `Promise.race(iterable)`

Returns a promise that fulfills or rejects as soon as one of the promises in the iterable fulfills or rejects, with the value or reason from that promise.

### `Promise.reject(reason)`

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

### `Promise.resolve(value)`

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)` it instead and work with the return value as a promise.

## Promise prototype

### Properties

#### `Promise.prototype.constructor`

Returns the function that created an instance's prototype. This is the `Promise` function by default.

### Methods

#### `Promise.prototype.catch(onRejected)`

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.then(onFulfilled, onRejected)`

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

### Creating a Promise

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 fulfill, using `window.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 simply logged, via a fulfill callback set using `p1.then()`. A few logs shows how the synchronous part of the method is decoupled of the asynchronous completion of the promise.

```
1  'use strict';
2  var promiseCount = 0;
3
4  function testPromise() {
5      var thisPromiseCount = ++promiseCount;
6
7      var log = document.getElementById('log');
8      log.insertAdjacentHTML('beforeend', thisPromiseCount +
9          ') Started (<small>Sync code started</small><br/>');
10
11      // We make a new promise: we promise a numeric count of this promise, starting
```

```

12  var p1 = new Promise(
13      // The resolver function is called with the ability to resolve or
14      // reject the promise
15      function(resolve, reject) {
16          log.insertAdjacentHTML('beforeend', thisPromiseCount +
17              ') Promise started (<small>Async code started</small><br/>');
18          // This is only an example to create asynchronism
19          window.setTimeout(
20              function() {
21                  // We fulfill the promise !
22                  resolve(thisPromiseCount);
23                  }, Math.random() * 2000 + 1000);
24      }
25  );
26
27  // We define what to do when the promise is resolved/fulfilled with the then()
28  // and the catch() method defines what to do if the promise is rejected.
29  p1.then(
30      // Log the fulfillment value
31      function(val) {
32          log.insertAdjacentHTML('beforeend', val +
33              ') Promise fulfilled (<small>Async code terminated</small><br/>');
34      })
35  .catch(
36      // Log the rejection reason
37      function(reason) {
38          console.log('Handle rejected promise ('+reason+') here.');

```

This example is executed when clicking the button. You need a browser supporting Promise. By clicking several times the button in a short amount of time, you'll even see the different promises being fulfilled one after the other.

Make a promise!

# Example using XMLHttpRequest

## Creating a Promise

This example shows the implementation of a method which uses a Promise to report the result of (or failure of) an [XMLHttpRequest](#).

```
1  'use strict';
2
3  // A-> $http function is implemented in order to follow the standard Adapter patter
4  function $http(url){
5
6      // A small example of object
7      var core = {
8
9          // Method that performs the ajax request
10         ajax: function (method, url, args) {
11
12             // Creating a promise
13             var promise = new Promise( function (resolve, reject) {
14
15                 // Instantiates the XMLHttpRequest
16                 var client = new XMLHttpRequest();
17                 var uri = url;
18
19                 if (args && (method === 'POST' || method === 'PUT')) {
20                     uri += '?';
21                     var argcount = 0;
22                     for (var key in args) {
23                         if (args.hasOwnProperty(key)) {
24                             if (argcount++) {
25                                 uri += '&';
26                             }
27                             uri += encodeURIComponent(key) + '=' + encodeURIComponent(args[key]);
28                         }
29                     }
30                 }
31
32                 client.open(method, uri);
33                 client.send();
34
35                 client.onload = function () {
36                     if (this.status >= 200 && this.status < 300) {
37                         // Performs the function "resolve" when this.status is equal to 2xx
38                         resolve(this.response);
39                     } else {
40                         // Performs the function "reject" when this.status is different than 2>
41                         reject(this.statusText);
42                     }
43                 }
44             });
45         }
46     };
47 }
```

```

43         };
44         client.onerror = function () {
45             reject(this.statusText);
46         };
47     });
48
49     // Return the promise
50     return promise;
51 }
52 };
53
54 // Adapter pattern
55 return {
56     'get': function(args) {
57         return core.ajax('GET', url, args);
58     },
59     'post': function(args) {
60         return core.ajax('POST', url, args);
61     },
62     'put': function(args) {
63         return core.ajax('PUT', url, args);
64     },
65     'delete': function(args) {
66         return core.ajax('DELETE', url, args);
67     }
68 };
69 };
70 // End A
71
72 // B-> Here you define its functions and its payload
73 var mdnAPI = 'https://developer.mozilla.org/en-US/search.json';
74 var payload = {
75     'topic' : 'js',
76     'q'      : 'Promise'
77 };
78
79 var callback = {
80     success: function(data) {
81         console.log(1, 'success', JSON.parse(data));
82     },
83     error: function(data) {
84         console.log(2, 'error', JSON.parse(data));
85     }
86 };
87 // End B
88
89 // Executes the method call
90 $http(mdnAPI)
91     .get(payload)

```

```
92     .then(callback.success)
93     .catch(callback.error);
94
95 // Executes the method call but an alternative way (1) to handle Promise Reject cas
96 $http(mdnAPI)
97     .get(payload)
98     .then(callback.success, callback.error);
99
100 // Executes the method call but an alternative way (2) to handle Promise Reject cas
101 $http(mdnAPI)
102     .get(payload)
103     .then(callback.success)
104     .then(undefined, callback.error);
```

## Loading an image with XHR

Another simple example using Promise and [XMLHttpRequest](#) to load an image is available at the MDN GitHub [promise-test](#) repository. You can also [see it in action](#). Each step is commented and allows you to follow the Promise and XHR architecture closely.

## Specifications

Specification	Status	Comment
<a href="#">ECMAScript 2015 (6th Edition, ECMA-262)</a> The definition of 'Promise' in that specification.	<b>ST</b> Standard	Initial definition in an ECMA standard.
<a href="#">ECMAScript 2017 Draft (ECMA-262)</a> The definition of 'Promise' in that specification.	<b>D</b> Draft	

## Browser compatibility

Desktop

[Mobile](#)

Feature	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Servo
Promise	32.0	(Yes)	29.0	No support	19	7.1	No support
Constructor requires new	32.0	(Yes)	37.0	No support	19	10	No support
Promise.all	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.prototype	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.prototype.catch	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.prototype.then	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.race	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.reject	32.0	(Yes)	29.0	No support	19	7.1	No support
Promise.resolve	32.0	(Yes)	29.0	No support	19	7.1	No support

## See also

- [Promises/A+ specification](#)
- [Jake Archibald: JavaScript Promises: There and Back Again](#)
- [Domenic Denicola: Callbacks, Promises, and Coroutines – Asynchronous Programming Patterns in JavaScript](#)
- [Matt Greer: JavaScript Promises ... In Wicked Detail](#)
- [Forbes Lindesay: promisejs.org](#)
- [Nolan Lawson: We have a problem with promises — Common mistakes with promises](#)
- [Promise polyfill](#)
- [Udacity: JavaScript Promises](#)