Promise

The **`Promise`** object is used for deferred and asynchronous computations. A `Promise` represents an operation that hasn't completed yet, but is expected in the future.

Syntax

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

Parameters

executor

Function object with two arguments `resolve` and `reject`. The first argument is a function that fulfills the promise, the second argument is a function that rejects it. We can call these functions once our operation is completed. The `executor` function is executed immediately. It runs before the `Promise` constructor returns the created object.

Description

A **`Promise`** represents 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* of having a value at some point in the future.

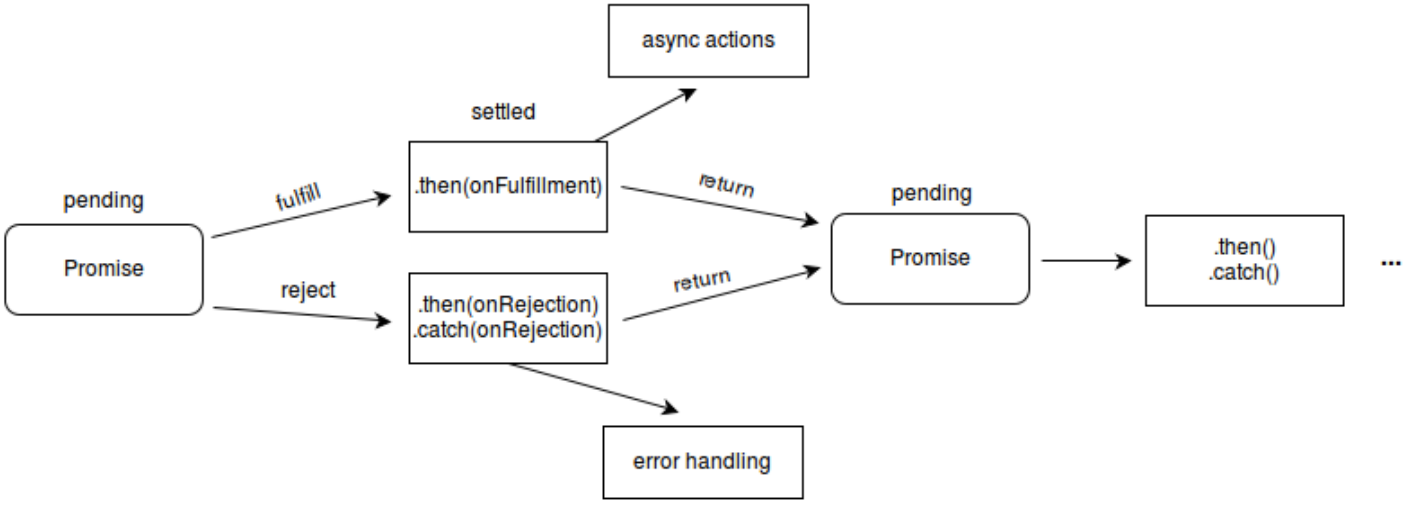
A `Promise` is in one of these states:

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

A pending promise can become either *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—an operation called *composition*.



> **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 1 (number of constructor arguments).

`Promise.prototype`

Represents the prototype for the `Promise` constructor.

Methods

`Promise.all(iterable)`

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

`Promise.race(iterable)`

Returns a promise that resolves or rejects as soon as one of the promises in the iterable resolves 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 want to 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 `undefined`).

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 resolve, 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, starti
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     // We define what to do when the promise is resolved/fulfilled with the then
27     // and the catch() method defines what to do if the promise is rejected.
28     p1.then(
29         // Log the fulfillment value
30         function(val) {
31             log.insertAdjacentHTML('beforeend', val +
32                 ') Promise fulfilled (<small>Async code terminated</small>)<br/>
33         })
34     .catch(
35         // Log the rejection reason
36         function(reason) {
37             console.log('Handle rejected promise ('+reason+') here.');
38         });
39
40     log.insertAdjacentHTML('beforeend', thisPromiseCount +
41         ') Promise made (<small>Sync code terminated</small>)<br/>');
42 }
```

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 new XMLHttpRequest()

Creating a Promise

This example shows the implementation of a method which uses a `Promise` to report the success or failure of an `XMLHttpRequest`.

```
'use strict';

// A-> $http function is implemented in order to follow the standard Adapter pat
function $http(url){

  // A small example of object
  var core = {

    // Method that performs the ajax request
    ajax : function (method, url, args) {

      // Creating a promise
      var promise = new Promise( function (resolve, reject) {

        // Instantiates the XMLHttpRequest
        var client = new XMLHttpRequest();
        var uri = url;

        if (args && (method === 'POST' || method === 'PUT')) {
          uri += '?';
          var argcount = 0;
          for (var key in args) {
            if (args.hasOwnProperty(key)) {
              if (argcount++) {
                uri += '&';
              }
              uri += encodeURIComponent(key) + '=' + encodeURIComponent(args[key
            }
```

```
          }
        }

        client.open(method, uri);
        client.send();

        client.onload = function () {
          if (this.status >= 200 && this.status < 300) {
            // Performs the function "resolve" when this.status is equal to 2xx
            resolve(this.response);
          } else {
            // Performs the function "reject" when this.status is different than
            reject(this.statusText);
          }
        };
        client.onerror = function () {
          reject(this.statusText);
        };
      });

      // Return the promise
      return promise;
    }
  };

  // Adapter pattern
  return {
    'get' : function(args) {
      return core.ajax('GET', url, args);
    },
    'post' : function(args) {
      return core.ajax('POST', url, args);
    },
    'put' : function(args) {
      return core.ajax('PUT', url, args);
    },
    'delete' : function(args) {
      return core.ajax('DELETE', url, args);
    }
  };
};
// End A

// B-> Here you define its functions and its payload
var mdnAPI = 'https://developer.mozilla.org/en-US/search.json';
var payload = {
```

```
  'topic' : 'js',
  'q'     : 'Promise'
};

var callback = {
  success : function(data){
     console.log(1, 'success', JSON.parse(data));
  },
  error : function(data){
     console.log(2, 'error', JSON.parse(data));
  }
};
// End B

// Executes the method call
$http(mdnAPI)
  .get(payload)
  .then(callback.success)
  .catch(callback.error);

// Executes the method call but an alternative way (1) to handle Promise Reject
$http(mdnAPI)
  .get(payload)
  .then(callback.success, callback.error);

// Executes the method call but an alternative way (2) to handle Promise Reject
$http(mdnAPI)
  .get(payload)
  .then(callback.success)
  .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
ECMAScript 2015 (6th Edition, ECMA-262) The definition of 'Promise' in that specification.	ST Standard	Initial definition in an ECMA standard.

ECMAScript 2017 Draft (ECMA-262) The definition of 'Promise' in that specification.	D Draft	

Browser compatibility

Desktop Mobile

Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari
Basic support	32.0	(Yes)	29.0 (29.0)	No support	19	7.1

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